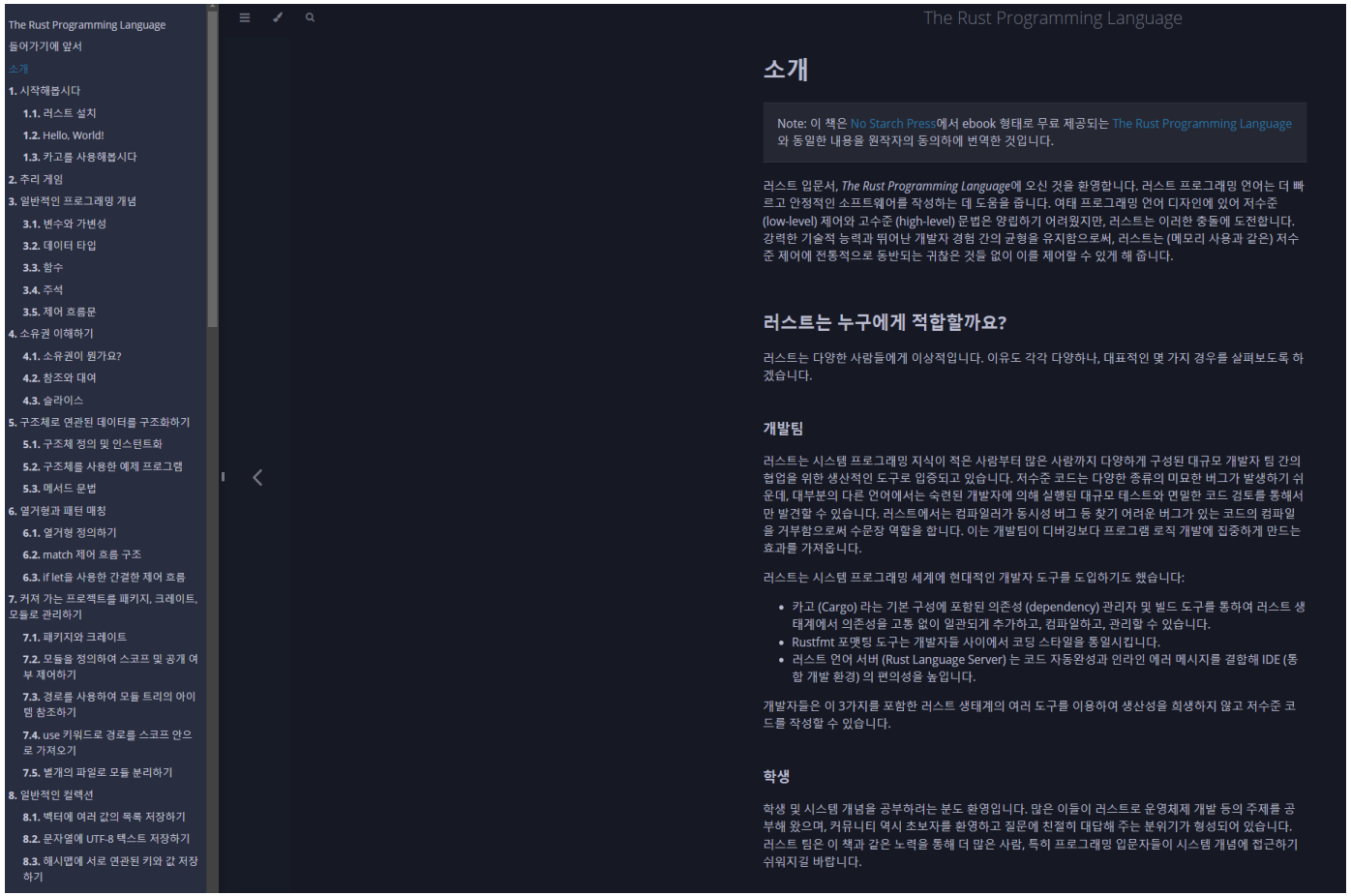
러스트 프로그래밍
- 소유권 -

https://youtu.be/AOlKLNtC6yA

러스트 프로그래밍

- 러스트 프로그래밍을 위한 프로그래밍 입문서를 기반으로 작성
  - https://doc.rust-kr.org/ch00-00-introduction.html

- 단계적으로 러스트 언어에 대해서 학습할 수 있음.



소유권 이해하기

- 러스트에서 가장 독특한 기능이며 러스트 언어의 가장 큰 특징이라고 할 수 있는 개념.

- 소유권은 러스트가 가비지 컬렉터 없이 메모리 안전성을 보장하도록 해주므로, 소유권에 대한 이해는 러스트 프로그래밍의 필수 조건.

- 소유권과 관련하여 대여(borrowing), 슬라이스(slice)의 개념이 있음.

소유권 이해하기 - 기존 언어의 메모리 관리

- C/C++ 메모리 관리
  - 메모리 할당/해제를 프로그래머가 직접 하는 방법
    - Malloc / Free
  - 메모리 할당 문제는 잠재적인 버그와 취약점을 유발. 이 문제는 추적해서 수정하기 까다로움.
  - 메모리 할당 버그 시나리오
    - 할당했던 메모리를 해제하지 않음. 할당된 메모리가 쌓이다 보면 모든 RAM을 사용하게 되면 프로그램 또는 컴퓨터가 멈출 수 있음.
    - 메모리가 해제된 후 포인터를 통해 버퍼를 읽거나 쓰려고 할 때 무작위의 결과가 발생할 수 있음. 이를 댕글링 포인터(Dangling Pointer)라고 한다.
    - 메모리 블록을 이중으로 해제.
  - 이러한 버그는 공격자에게 공격할 기회를 줄 수 있음.

소유권 이해하기 - 기존 언어의 메모리 관리

- 가비지 컬렉터(Garbage Collecor)
  - 자동 메모리 관리 시스템. 프로그램에 의해서 자동으로 메모리가 할당되고 해제됨
  - 자동화된 메모리 관리로 개발자에게는 편리성을 제공하고 프로그램은 안전하고 효율적인 메모리 관리가 가능
  - 하지만, 이러한 기능을 제공하기 위한 추가적인 동작 과정이 필요하기 때문에 기존의 메모리 관리 보다 성능이 저하됨.
    - 중단 시간, 메모리 관리 오버헤드 등

- 결론적으로 기존의 메모리 관리 방법은 성능과 안전성을 동시에 갖기 어려움.

소유권 이해하기

- 소유권의 규칙
  - Rust의 모든 값은 소유자(Owner)가 정해져 있다.
  - 한 값의 소유자는 동시에 여럿 존재할 수 없다.
  - 소유자가 스코프(Scope) 밖으로 벗어날 때, 값은 버려진다(Drop).

- 위 개념들을 설명하기 위해서 String 타입의 변수를 통해서 설명함.
  - String 타입은 일반적인 데이터타입과 다르게 크기가 정해져 있지 않기 때문에 힙(Heap)에 저장됨.
  - Scope : 변수의 범위를 표현하며, 일반적으로 중괄호를 사용하여 이해할 수 있음.

```
{                       // s는 아직 유효하지 않다. 호출시 Error
    let s = "hello";    // 여기서부터 s가 유효하다.
                        // 여기서도 s는 유효하다.
}                       // 이제 s는 유효하지 않다. 호출시 Error
```

소유권 이해하기

- 힙 영역에 저장되기 위해서는 OS를 통해서 메모리를 할당 받고, 사용한 후에 할당받은 메모리를 돌려줘야함.
  - C/C++ 에서는 이 과정을 직접하며, 가비지 컬렉터는 이 과정을 자동으로 해줌.

```
{
    let s = "hello";    // 여기서부터 s가 유효하다.
                        // 여기서도 s는 유효하다.
}                       // s는 메모리를 반환한다.
```

- 변수가 스코프에서 나가게 되면 Rust에서는 drop이라는 함수를 자동으로 호출함

ERROR

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

println!("{}, world!", s1);
```

소유권 이해하기

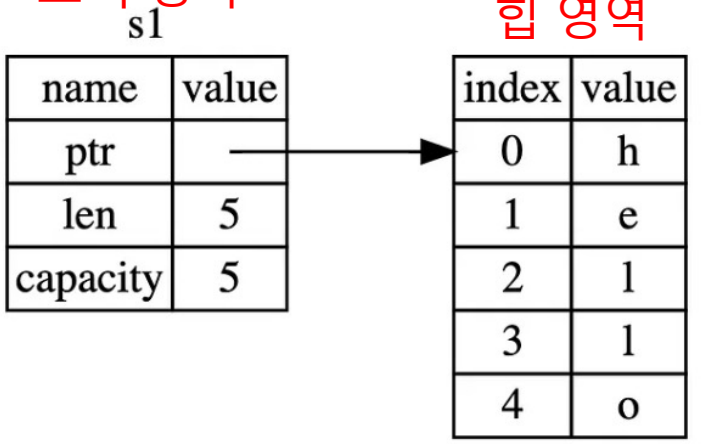
스택 영역

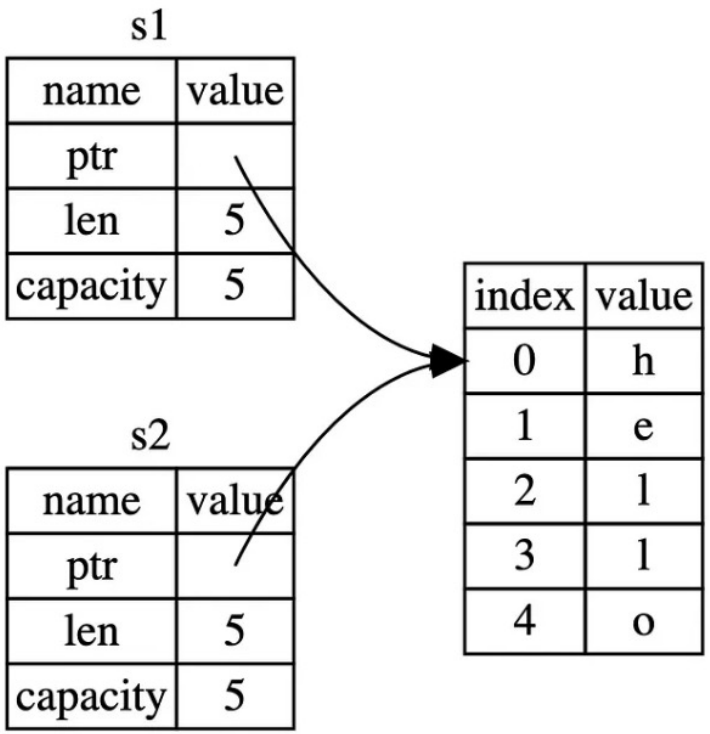
힙 영역

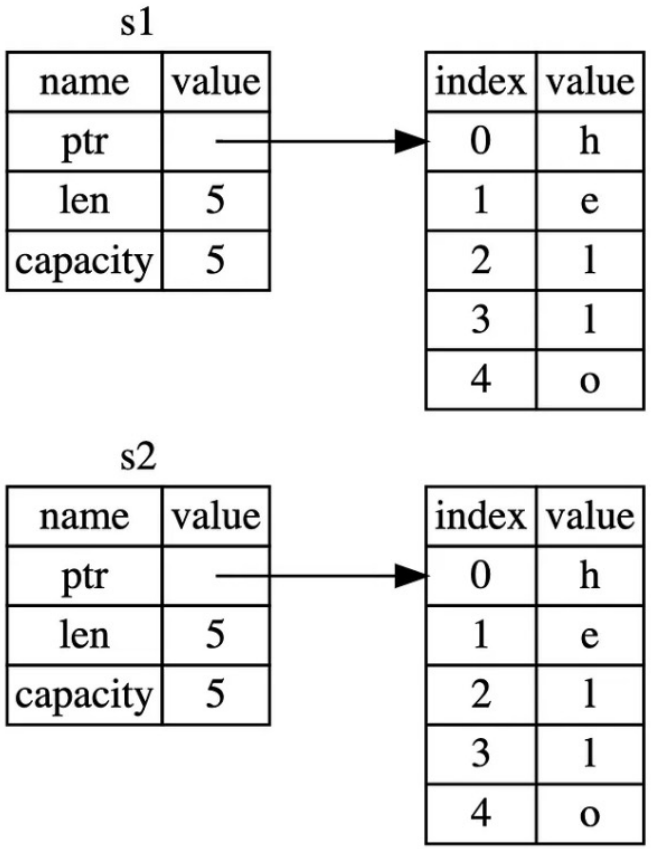
Scope

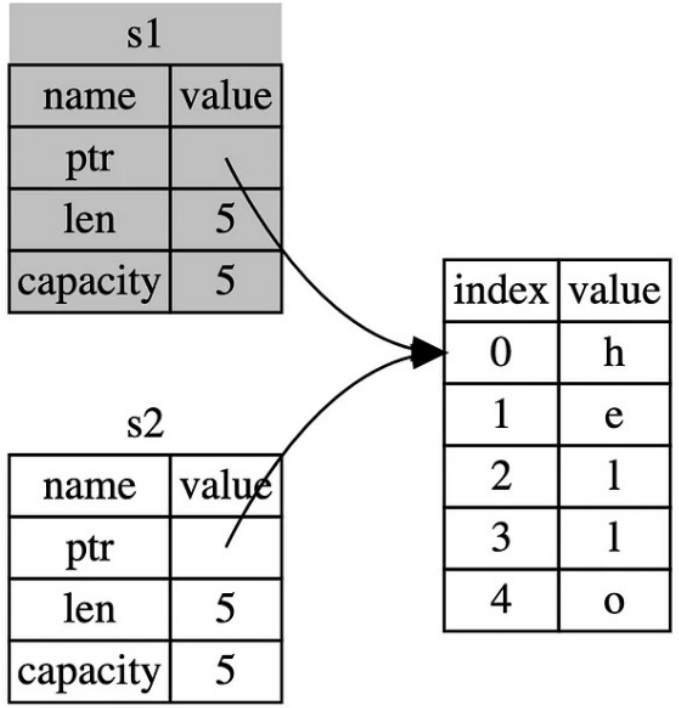
```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```







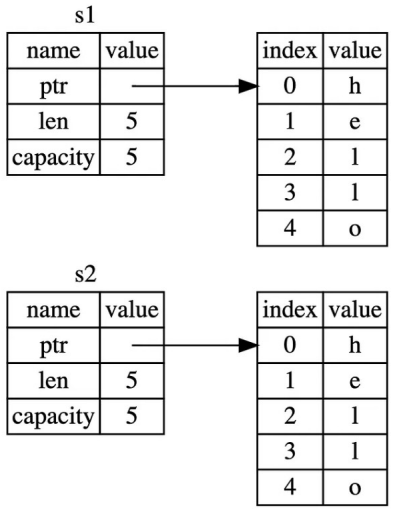


소유권 이해하기

- 변수들의 공통 메서드인 clone 메서드를 사용해서 Deep copy를 할 수 있음.

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

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



문제 없음! Why? 크기가 정해져 있기 때문에 -> Stack에 저장됨.

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

이와 같이 크기가 정해져 있는 변수들은 'Copy' trait이란 것을 가지고 있음.
String 타입은 'Copy' trait이 없고, 'Drop' 사용하며, 두 Trait을 동시에 가질 수 없음.

소유권 이해하기

```
fn main() {
    let s = String::from("hello");  // s가 스코프 안으로 들어옵니다

    takes_ownership(s);             // s의 값이 함수로 이동됩니다...
                                    // ... 따라서 여기서는 더 이상 유효하지 않습니다

    let x = 5;                      // x가 스코프 안으로 들어옵니다

    makes_copy(x);                  // x가 함수로 이동될 것입니다만,
                                    // i32는 Copy이므로 앞으로 계속 x를
                                    // 사용해도 좋습니다

} // 여기서 x가 스코프 밖으로 벗어나고 s도 그렇게 됩니다. 그러나 s의 값이 이동되었으므로
  // 별다른 일이 발생하지 않습니다.

fn takes_ownership(some_string: String) { // some_string이 스코프 안으로 들어옵니다
    println!("{}", some_string);
} // 여기서 some_string이 스코프 밖으로 벗어나고 `drop`이 호출됩니다.
  // 메모리가 해제됩니다.

fn makes_copy(some_integer: i32) { // some_integer가 스코프 안으로 들어옵니다
    println!("{}", some_integer);
} // 여기서 some_integer가 스코프 밖으로 벗어납니다. 별다른 일이 발생하지 않습니다.
```

감 사 합 니 다