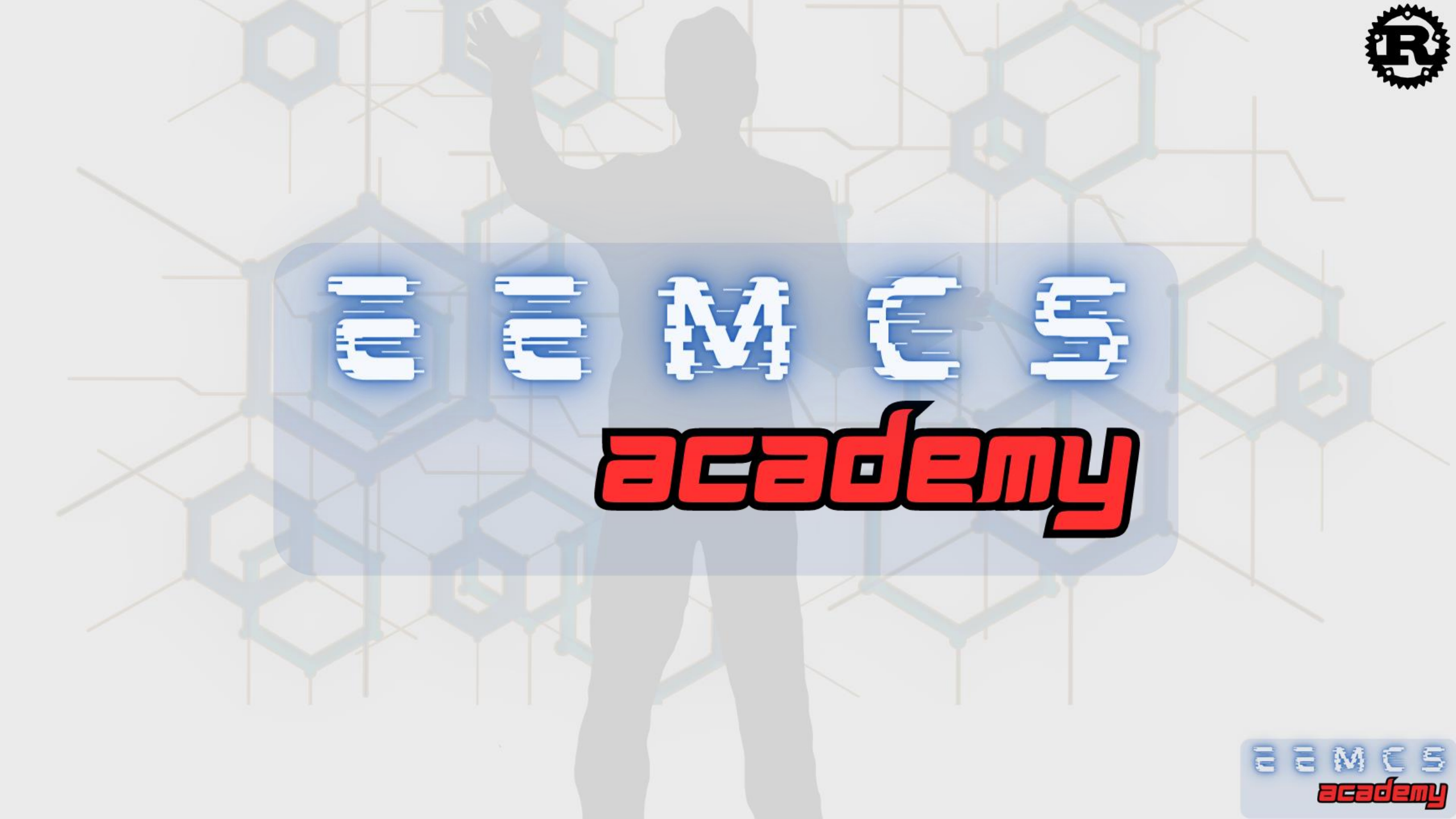

EEMCS
academy

# RUST
# Ownership - Sahiplik
# Borrowing - Ödünç Alma
# Heap Memory
# Stack Memory
# Scope - Kapsam

# Ownership - Sahiplik

Rust'ın en benzersiz özelliğidir

Rust programının belleği nasıl yöneteceğini düzenleyen kuralları kapsar.

Rust'ta bellek, derleyicinin kontrol ettiği **sahiplik-ownership sistemi** aracılığıyla yönetilir.

Rust'ın çöp toplayıcıya ( garbage collector ) ihtiyaç duymadan bellek güvenliğini garanti altına almasını sağlar.

# Ownership kuralları :

- Rust'taki her değerin bir sahibi ( owner ) vardır.
- Bir değerin aynı anda sadece bir sahibi olabilir.
- Veriyi sadece sahibi kullanabilir.
- Sahip kapsam dışına çıktığında, değer RAMden silinir.

Ownership kurallarının ana amacı HEAP memorydeki verilerin yönetimini kontrol etmekdir.

# Verilerin Saklanma Yöntemleri :

STACK memory
HEAP memory

# STACK memory

Sabit boyutlu veriler saklanır

HEAP kısmına veri eklemekten daha hızlıdır.

Veriler sırayla üst üste eklenir.

Veriler bellekten çıkartılırken en üstten sırayla çıkartılır.

Yani son giren eleman ilk çıkartılacak elemandır. ( Last In - First Out )

```
let x = 5;
let y = x;          // x ve y değişkenleri ayrı ayrı değişkenlerdir.
```

let x=5;

| Adress | Değişken adı | Değer |
|---|---|---|
| 0 | X | 5 |
| | | |
| | | |
| | | |
| | | |
| | RAM-STACK | |

let y=x;

| Adres | Değişken adı | Değer |
|---|---|---|
| 1 | Y | 5 |
| 0 | X | 5 |
| | | |
| | | |
| | | |
| | RAM - STACK | |

let z=10;

| Adres | Değişken adı | Değer |
|---|---|---|
| 2 | Z | 10 |
| 1 | Y | 5 |
| 0 | X | 5 |
| | | |
| | | |
| | RAM - STACK | |

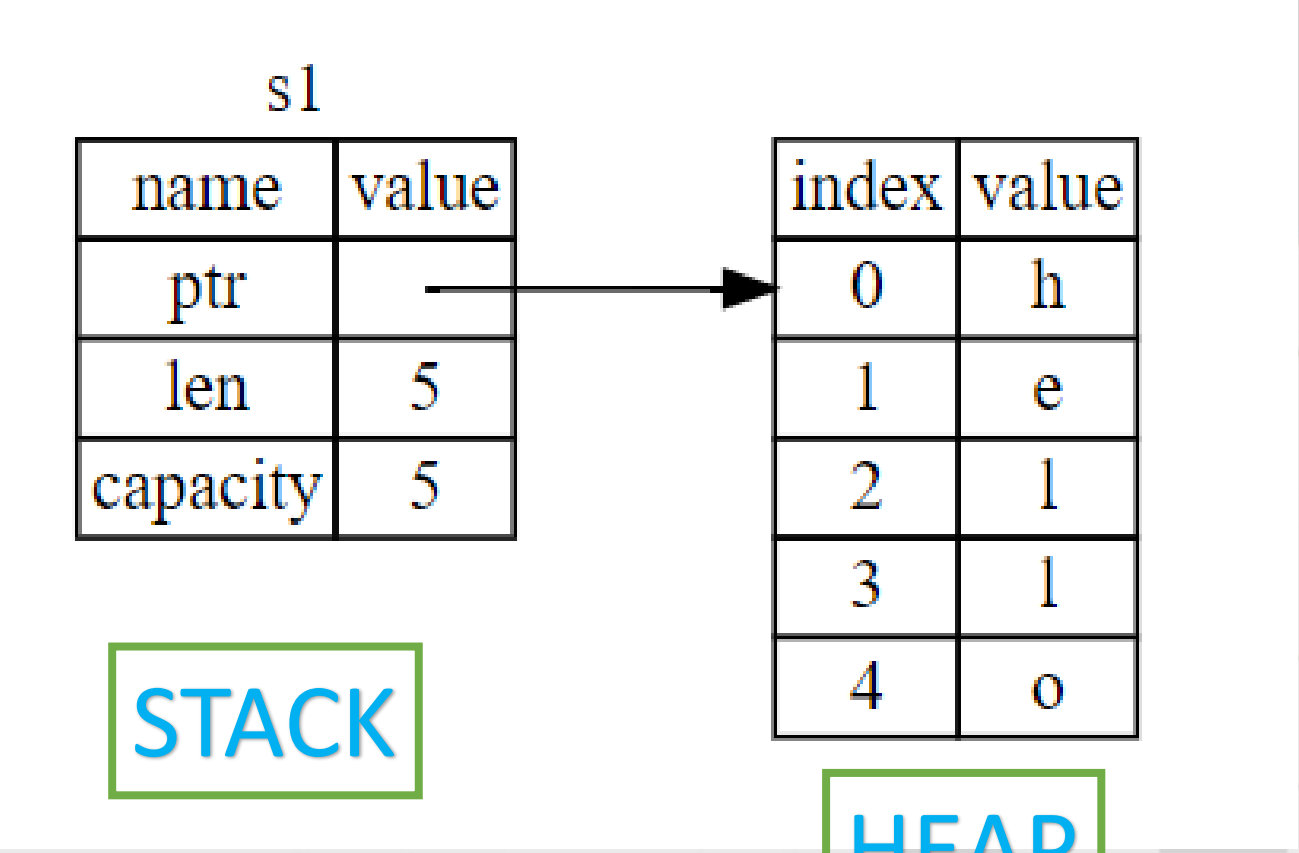
# HEAP memory

Çalışma zamanında boyutu belli olmayan veriler saklanır ( String gibi )
Yani veri boyutu değiştirilebilir olan veriler saklanır

Veri için yer ayrıldığında, bu yerin adresi POINTER da saklanır.

Pointer boyutu sabittir ve STACK ta saklanır.

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



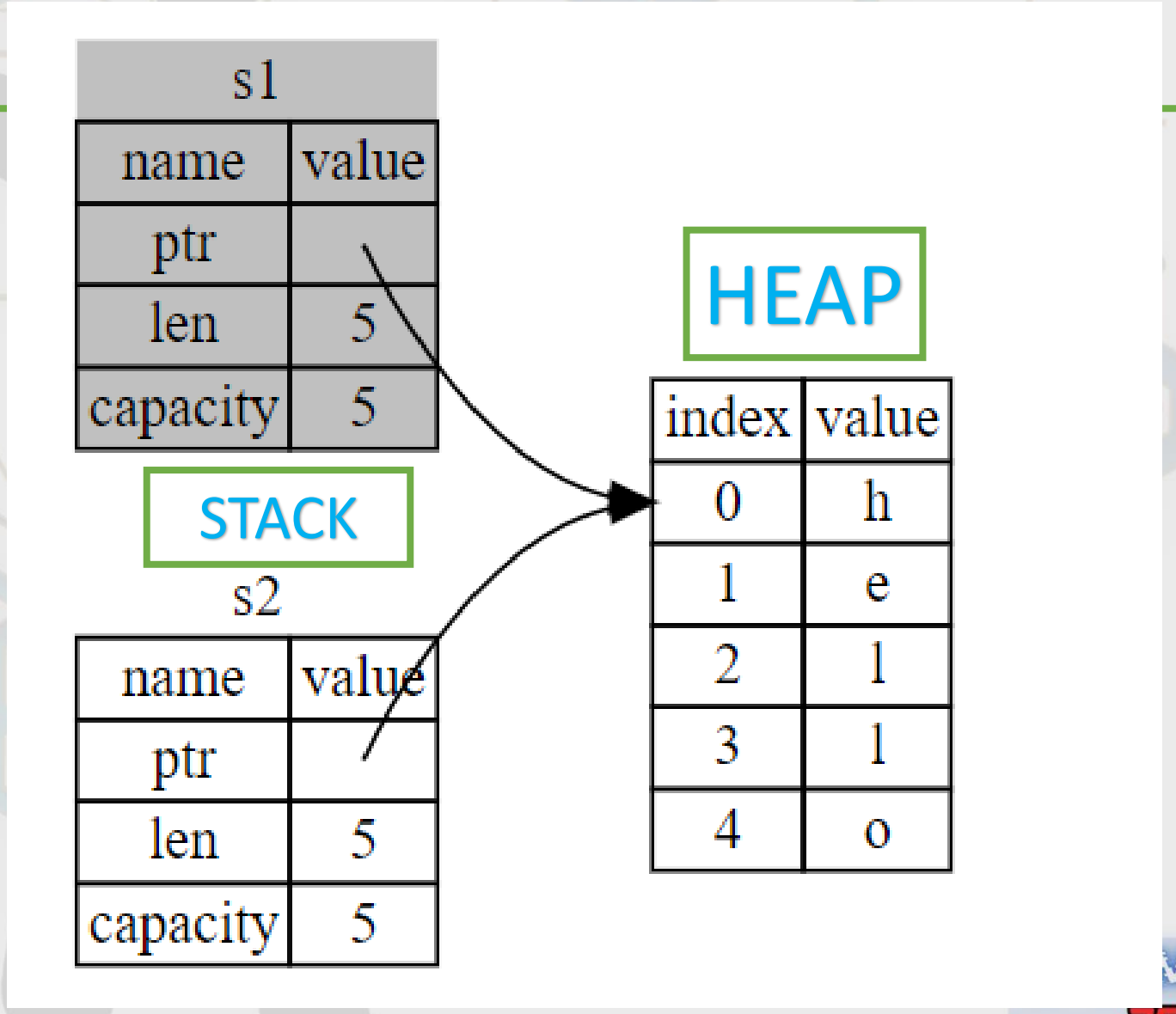

Her değerin bir sahibi vardır

Pointer, verinin heap memorydeki adresini saklar.

```
let s2 = s1;
```

Aynı anda sadece bir sahibi olablir

s1 GEÇERSİZ OLUR



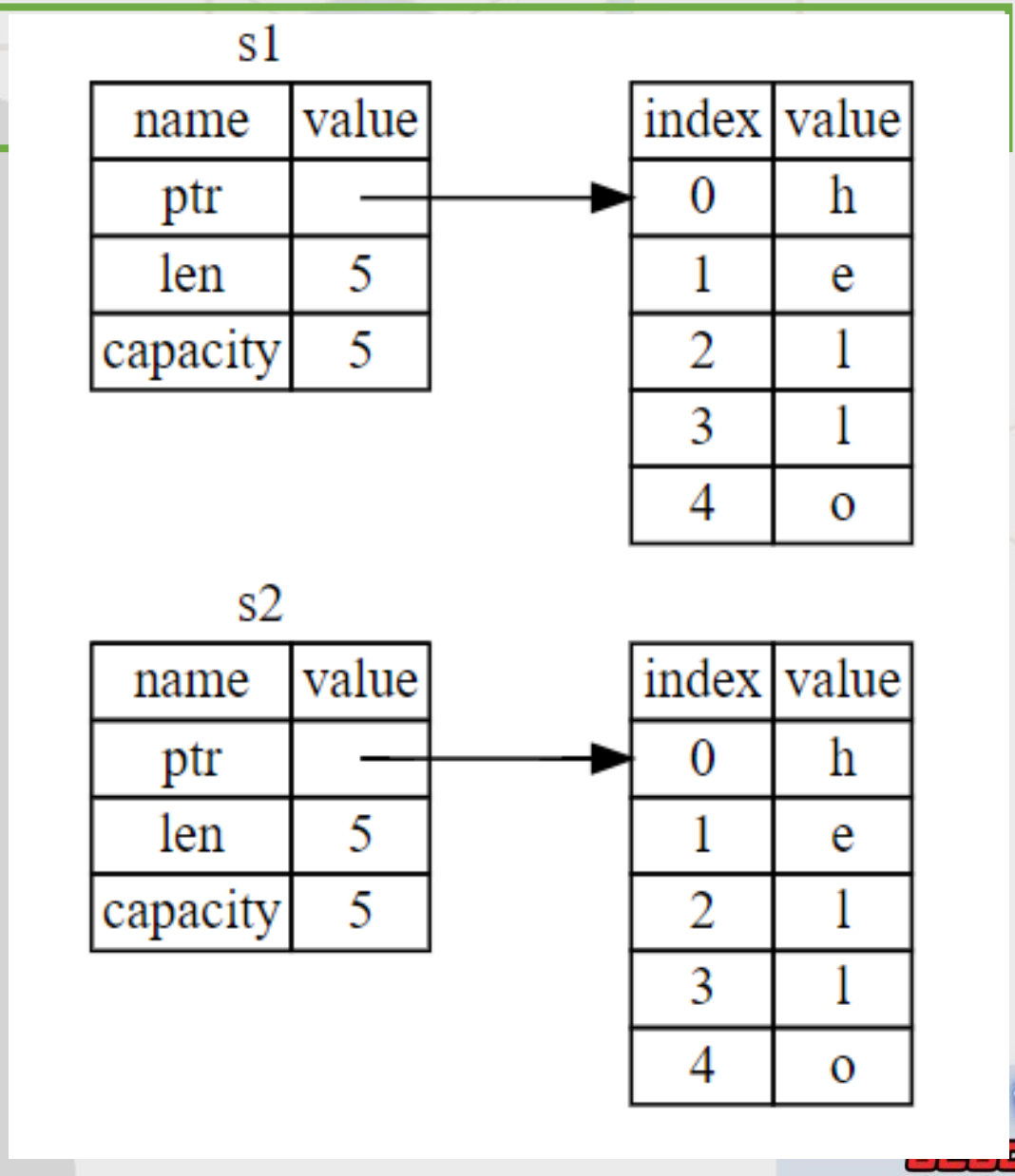
Bir değerin aynı anda sadece bir sahibi olabilir
Gerçekleşen olay "move" olarak isimlendirilir
ve s1, s2'ye taşınmış ( move ) olur.

```
    let s1 = String::from("Merhaba");
    let s2 = s1;
    println!("S1 değişkeninin değeri {}", s1);//
Hata verir
```

Bu yüzden s1 kullanılamaz

# let s2 = s1.clone();

```
let s1 = String::from("Merhaba");
let s2 = s1.clone();
println!("S1 değişkeninin değeri {}", s1);
```



# Scope ( Kapsam ) :

```
{
        let metin = String::from("owership");
}

println!("metin değişkeninin değeri : {}", metin); // bu değişken burada geçerli değil
```

Parantezler dışına çıkıldığında özel bir DROP fonksiyonu çalıştırılır ve veri hafızadan silinir.

# Sahip kapsam dışına çıkınca değer ramden silinir

# AYNI VERİYİ TEKRAR TEKRAR NASIL KULLANABİLİRİZ

**move işlemi yapmadan**
**clone() metodunu kullanmadan**

# Borrowing - Ödünç Alma

HEAP memoryde tanımlanan veriyi **FARKLI BİR DEĞİŞKENE TAŞIMADAN**;
**tekrar kullanabilir miyiz?**

- **POINTER** ile ( **&** ) **referans** göstererek kullanabiliriz.

- Bu işleme BORROWING-ödünç alma denir.

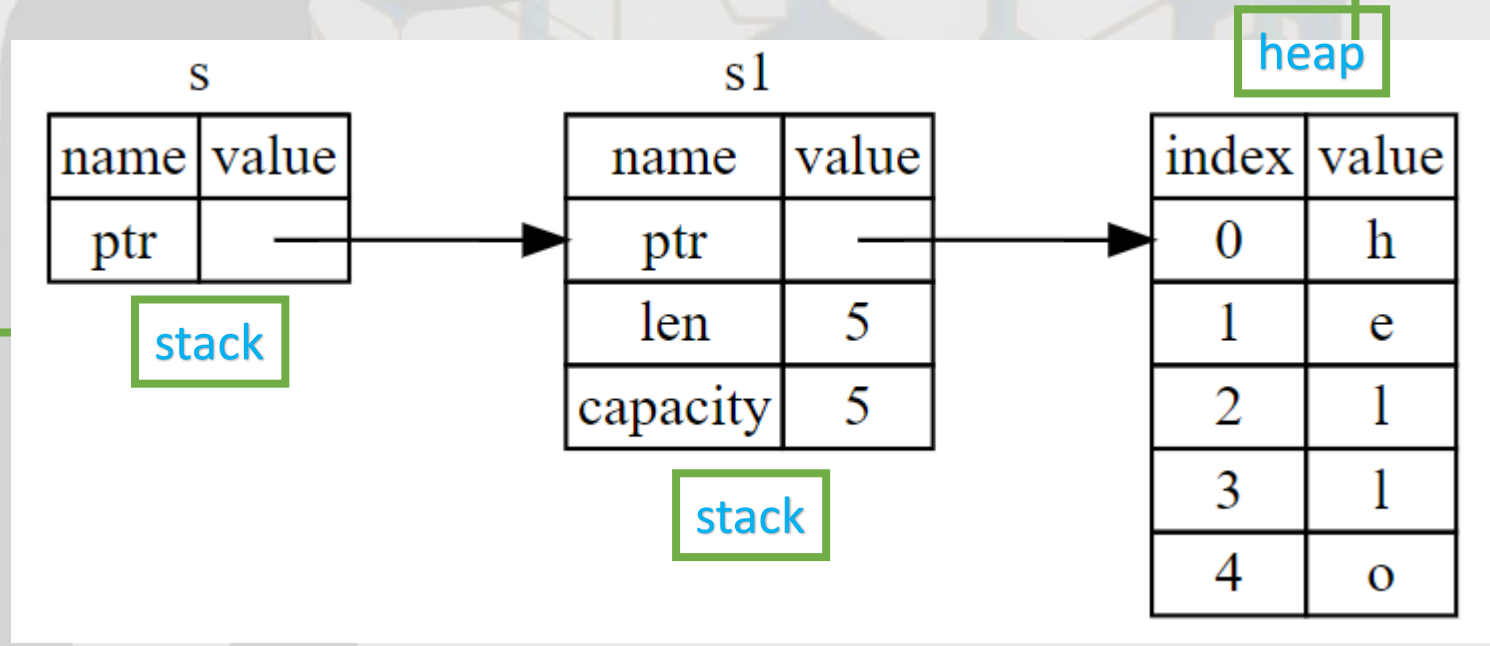
Verinin, mutable-değişebilir ya da immutable-değişmez olmasına dikkat etmeliyiz.

# Immutable Ödünç Alma ( heap memory için )

- **İki veya daha fazla işaretçi** değiştirilemez olan aynı veriye **aynı anda erişir.**

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

let s = &s1;
```



# Immutable Ödünç Alma ( heap memory için )

- **İki veya daha fazla işaretçi** değiştirilemez olan aynı veriye **aynı anda erişebilir.**

```
let mt = String::from("Rust");

let m1 = &mt;

let m2 = &mt;

let m3 = &mt;

println!("{}  {} {}", m1, m2, m3);
```

# Mutable Ödünç Alma ( hesap memory için )

- Ödünç alınmış veri aynı anda kullanılamaz.

```
    let mut m1 = String::from("Borrowing Özelliği");

    let m2 = &mut m1;

    println!("{}  {}", m2, m1); // Ödünç alınmış veri aynı anda
kullanılamaz.
```

# Mutable Ödünç Alma ( heap memory için )

- Ödünç alan değişken veriyi kullandıktan sonra ana değişken verinin sahibi olur.

```
let mut m1 = String::from("Borrowing Özelliği");

let m2 = &mut m1;

println!("m2 {} ", m2);

m2.push_str(" öğreniyorum");

println!("m2 {}", m2);

println!("m1 {}", m1);
```

# Mutable Ödünç Alma ( heap memory için )

- Ödünç alan değişken veriyi kullandıktan sonra başka bir değişken veriyi ödünç alabilir.

```
    .
    .
    .
let m3 = &mut m1;
m3.push_str("Ödünç alan m3");
println!("{}", m3);
println!("{}", m1);
```

# Mutable Ödünç Alma ( heap memory için )

- Veriyi en son ödünç alan değişken kullanabilir, önceden ödünç alanlar kullanamaz.

```
let mut m1 = String::from("Borrowing Özelliği");

let m2 = &mut m1;
m2.push_str(" öğreniyorum");
println!("m2 {}", m2);

let m3 = &mut m1;
m3.push_str(" Rust içinde");
println!("m3 {}", m3);

m2.push_str(" deneme"); // m3 borrowing de hata oluşur
```

Burada unutulmaması gereken;

Ödünç alma işleminde sürekli aynı veri kullanılmaktadır.

Dolayısıyla yapılan değişikliklerin hepsi orijinal veriye uygulanmaktadır ve veri değişmektedir.

Orijinal verinin ne zaman değişmesi ve ne zaman korunması gerektiğine dikkat edilmelidir.

# Ödünç alma ( stack memory için )
# Dereference operatörü "*"

Adres bilgisi yerine ( & ), adresin işaret ettiği veri bilgisini verir. Bütün referans / pointer tiplerini destekler ( Deref trait implement edilerek ).

```
let mut x = 5;

let y = &mut x;
*y = 11;
println!("{}", y);

x = 8;
println!("{}", x);
```