2-3: Ownership Rules and Borrowing (Theory)

Artem Pavlov, TII, Abu Dhabi, 15.05.2025

Stack, heap, and static memory

- The stack stores values in the order it gets them and removes the values in the opposite order (i.e. LIFO)
  - Very efficient
  - Has limited size (usually 2-8 MiB)
  - Does not need any runtime support
  - Size of objects should be known at compile time
- The heap stores objects of arbitrary size and its memory is managed by an “allocator”
  - Less efficient
  - Requires existence of an “allocator”
  - Can store dynamically sized objects
- Static memory is allocated at compile time and often is part of generated binary

Ownership

- Ownership is a set of rules that govern how a Rust program manages memory.
- Ownership rules:
  - Each value in Rust has an owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value will be dropped.

Scopes

```rust
{                           // s is not valid here, it's not yet declared
    let s = "hello";    // s is valid from this point forward

    // do stuff with s
}                           // this scope is now over, and s is no longer valid

{
    let mut s = String::from("hello");
    s.push_str(", world!"); // appends a literal to a String
    println!("{}", s);
    // s gets dropped here
}
```

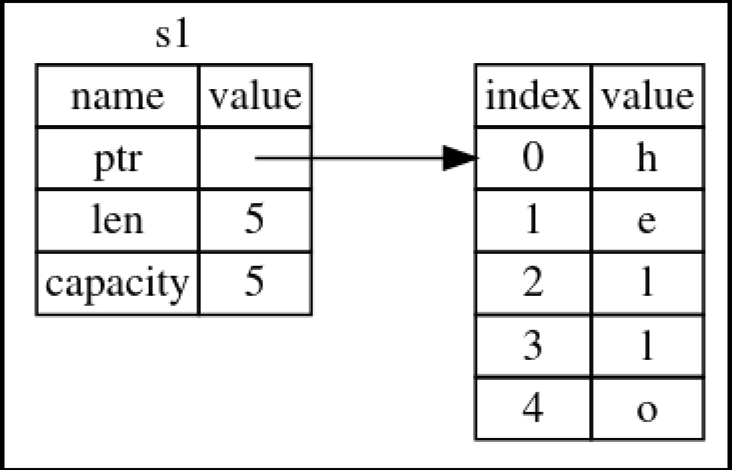
Move and Copy

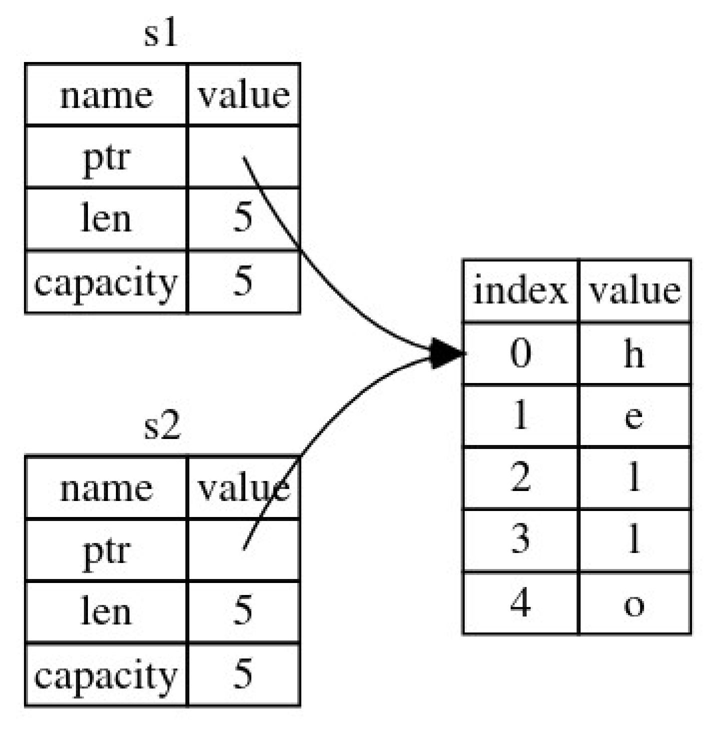
```
#[derive(Debug)]
struct Foo(u32);
let x = Foo(42);
let y = x;

println!("{y:?}");
// x can not be used
// println!("{x:?}");

let x = 42u32;
let y = x;
// both x and y can be used
println!("{x:?} {y:?}");
```

- Moving value performs bitwise copy
- Move usually invalidates old location
- If type implements the Copy trait, then old location stays valid

Move and heap-allocated data

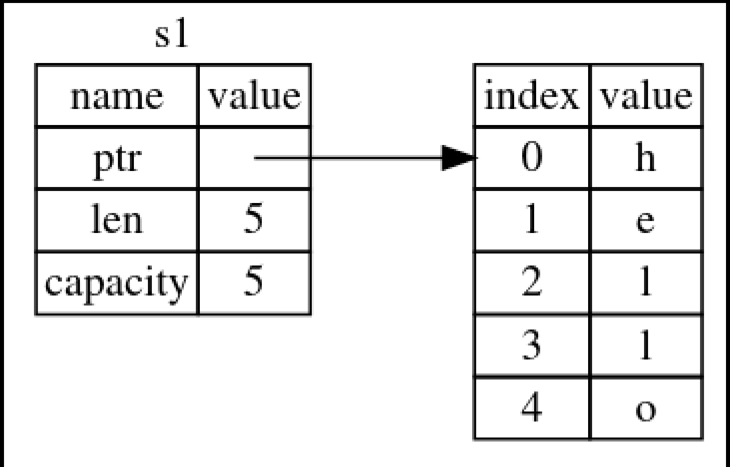
```rust
let s1 = String::from("hello");
let s2 = s1;
```

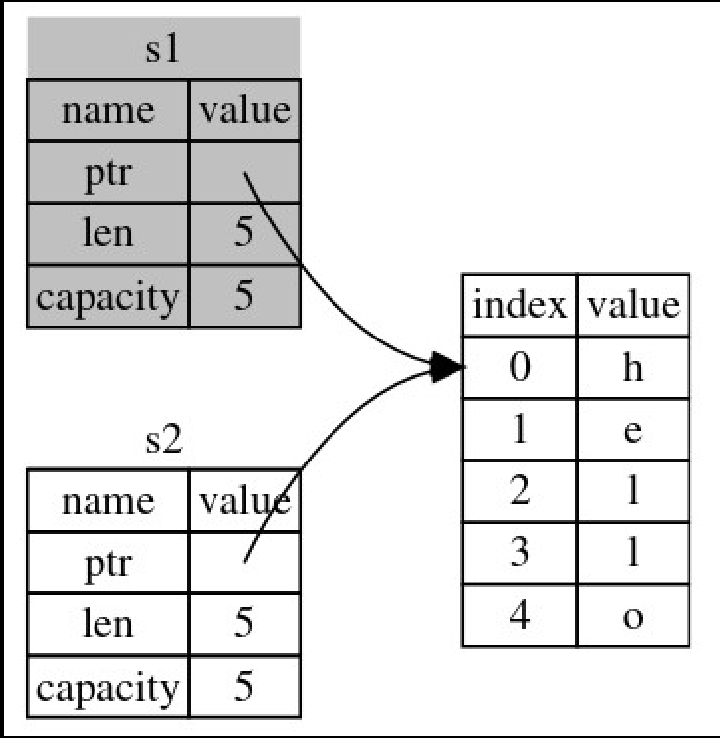




Move and heap-allocated data

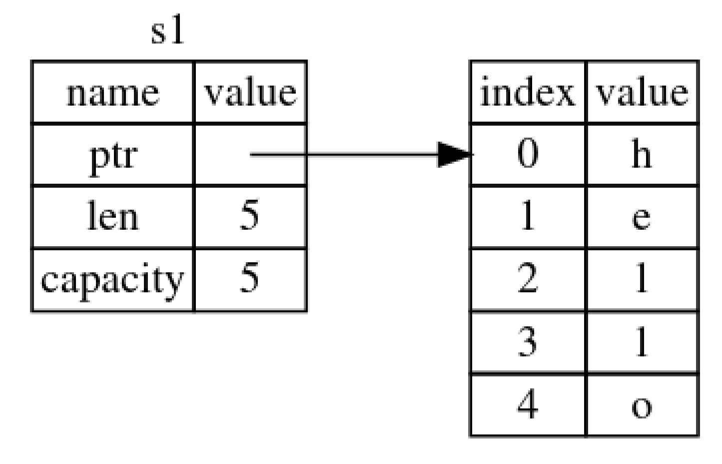
```
let s1 = String::from("hello");
let s2 = s1;
```

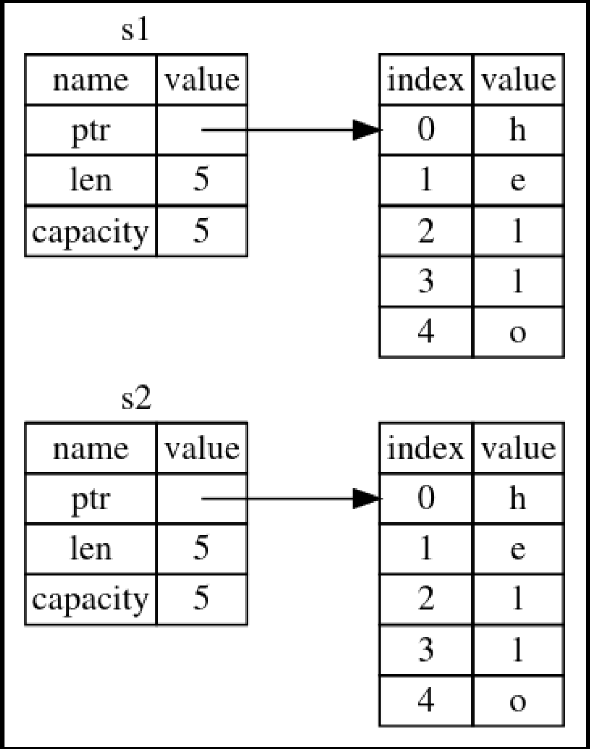




Deep copies

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





Ownership and functions

```rust
fn main() {
    let s = String::from("hello");
    // s comes into scope

    // s's value moves into the function...
    takes_ownership(s);
    // ... and so is no longer valid here

    let x = 5; // x comes into scope

    // x would move into the function,
    // but i32 is Copy, so it's okay to still
    // use x afterward
    makes_copy(x);
} // Here, x goes out of scope, then s.
  // But because s's value was moved, nothing
  // special happens.
```

```rust
fn takes_ownership(some_string: String) {
    // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and
  // `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) {
    // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope.
  // Nothing special happens.
```

Ownership and return values

```rust
fn main() {
    // gives_ownership moves its return
    // value into s1
    let s1 = gives_ownership();


    // s2 comes into scope
    let s2 = String::from("hello");

    // s2 is moved into takes_and_gives_back,
    // which also moves its return value into s3
    let s3 = takes_and_gives_back(s2);

} // Here, s3 goes out of scope and is dropped.
  // s2 was moved, so nothing happens.
  // s1 goes out of scope and is dropped.
```

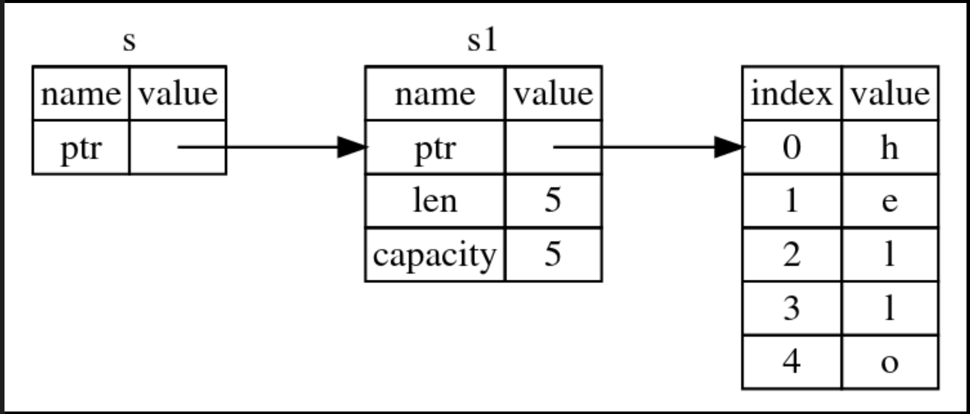
```rust
// gives_ownership will move its
// return value into the function
// that calls it
fn gives_ownership() -> String {
    let some_string = String::from("yours");
    // some_string comes into scope

    // some_string is returned and
    // moves out to the calling function
    some_string
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String {
    // a_string comes into scope

    a_string // a_string is returned and
             // moves out to the calling function
}
```

Passing ownership can be annoying

```rust
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Borrows to the rescue!

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

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



Mutable borrows

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

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

// Results in compilation error:
// println!("{}, {}", r1, r2);
```

```
let mut s = String::from("hello");

{
    let r1 = &mut s;
}
// r1 goes out of scope here,
// so we can make a new reference

let r2 = &mut s;
```

Mutable and shared borrows

```rust
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

```rust
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{} and {}", r1, r2);
// variables r1 and r2 will not be
// used after this point

let r3 = &mut s; // no problem
println!("{}", r3);
```

Dangling References

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

fn main() {
    let reference_to_nothing = dangle();
}
```

Borrow rules

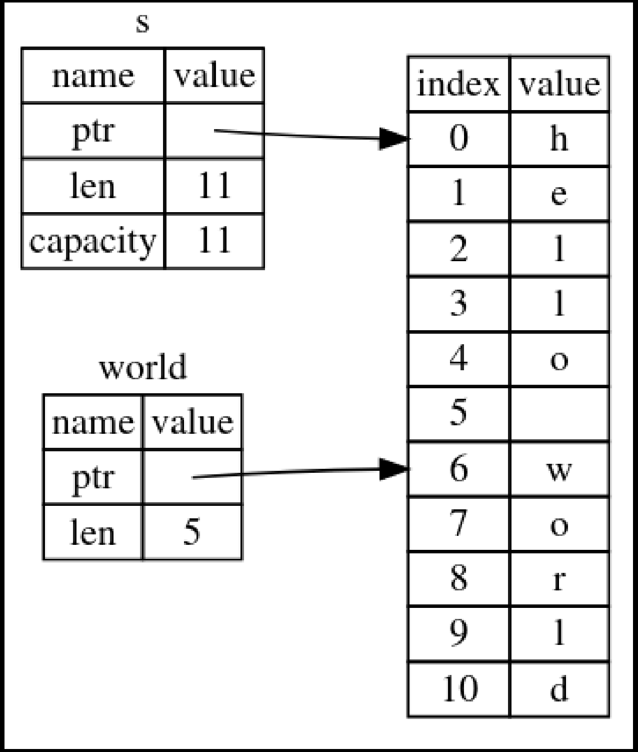
- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

Slices

- Let’s say we want to write function which returns the first word from string
- It accepts &String, but what should it return?
- This can be done using “string slices” &str
- Slices can be created by indexing with range syntax &s[start_idx..end_idx]

String slices

```
let mut s = String::from("hello");

let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```



Slices from arrays and vectors

```rust
fn sum(values: &[u32]) -> u32 {
    let mut sum = 0;
    for value in values {
        sum += value;
    }
    sum
}
```

```rust
let mut v: Vec<u32> = vec![0, 1, 2, 3];
v.push(4);
let a = [0, 1, 2, 3];

let v_sum = sum(&v[1..3]);
let a_sum = sum(&a[1..2]);
```

Lifetimes

- The main aim of lifetimes is to prevent dangling references
- Borrow checker tracks liftetimes and checks that all borrows are valid
- Borrow checker is based on local reasoning about current function

Implicit lifetimes in code

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

Lifetimes in functions

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

Lifetime Annotation Syntax

- &i32 a reference
- &'a i32 a reference with an explicit lifetime
- &'a mut i32 a mutable reference with an explicit lifetime
- &'static i32 a reference with static lifetime

Lifetimes in functions

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

Lifetimes in types

```rust
struct LongestStr<'a> {
    s: &'a str,
}

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

Lifetime ellision

- In simple cases compiler can infer lifetimes
- For example, fn first_word(x: &str) -> &str
- Ellison rules:
  - The compiler assigns a lifetime parameter to each parameter that’s a reference.
  - If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters.
  - If there are multiple input lifetime parameters, but one of them is &self or &mut self (i.e. it is a method), the lifetime of self is assigned to all output lifetime parameters.

Lifetimes in method definitions

```
impl<'a> LongestStr<'a> {
    fn announce_and_return(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

Questions?