Advanced Programming Methods
Lecture 12-13 – Rust
(not requied for the final exam)

Introduction to Rust

Slide Credits

Inspiration from:

https://www.cis.upenn.edu/~cis1905/2024spring/

https://github.com/trifectatechfoundation/teach-rs

https://www.cs.umd.edu/class/fall2021/cmsc388Z/

Rust Quick Links:

"The Rust Book“(https://doc.rust-lang.org/book/ch00-00-introduction.html)

Rust Standard Library Documentation(https://doc.rust-lang.org/std/)

Rust Playground (https://play.rust-lang.org/)

A Brief Rusty History

Early history (2009-2012): personal project by Mozilla employee Graydon Hoare. Sponsored by Mozilla

Pre-release (2012-2015): larger open source community formed, underwent many feature changes trying to find a niche

May 15 2015: Rust 1.0 release, commitment stability

Adoption (2015-2020): Firefox code migrated to Rust, adoption elsewhere in industry

Modern era (2020-present):

- Mozilla lay offs include core Rust contributors
- Rust Foundation started by AWS, Huawei, Google, Microsoft, and Mozilla

Who's using Rust Now?

Developers: StackOverflow's most loved language eight years running

CLI tools: `grep` -> `ripgrep` (~5x faster)

Full rewrites: Dropbox core syncing code fully rewritten in Rust

Partial migrations: Firefox (20% of core codebase)

Rust is a *Systems Programming* Language

One definition: applications that require control of **memory layout** and access to **machine primitives**

Better definition: applications that have strong **correctness** and **performance** requirements.

- OS Kernels
- Databases
- Networking code

But also...

- Scientific computing
- Embedded systems
- Web programming

Why Rust?

We'll primarily be comparing to C/C++, languages that give more control than nearly anything else out there.

Unfortunately, with great power comes great danger:

- read past buffer
- use-after-free
- double free
- memory leaks
- race conditions

Key question: can you keep the control of C/C++ while not having the dangers?

70% of vulnerabilities in Microsoft's codebases are memory safety

Chrome, Firefox have similar numbers

The source? U.S. Homeland Security: *The Urgent Need for Memory Safety in Software Products*

- How can languages address memory safety?

https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

Anatomy of a Rust Program

```
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

Anatomy of a Rust Program

```rust
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

`println!(...)`

Like C-style printf formatting but...

- Type-inferred
- Type-safe
- No run-time cost

Implemented via macros (we'll see more later)

Anatomy of a Rust Program

```rust
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

Function Declaration

- Argument types and return types must be annotated

Numeric types

	signedness		
size	i8	u8	
	i16	u16	f32
	i32	u32	
	i64	u64	f64

Anatomy of a Rust Program

```
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

Pattern Matching

- Very similar to OCaml
- Very flexible, we'll see more in future lectures and homework

Anatomy of a Rust Program

```
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

Hang on, where are the semicolons?
What about `return`??

Anatomy of a Rust Program

```rust
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

```rust
fn fib_imperative(n: u64) -> u64 {
    if n <= 1 {
        return n;
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        return result;
    }
}
```

Anatomy of a Rust Program

```rust
fn main() {
    println!("fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fib(n - 1) + fib(n - 2)
    }
}
```

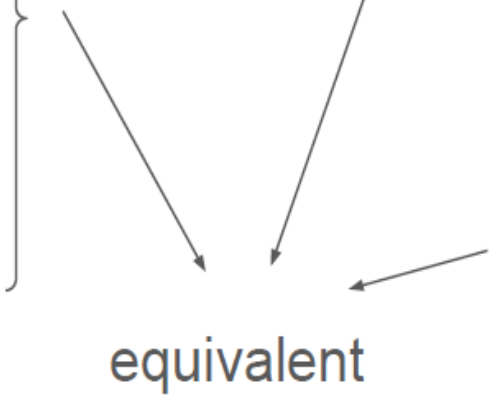
```rust
fn fib_imperative(n: u64) -> u64 {
    if n <= 1 {
        return n;
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        return result;
    }
}
```

Rust borrows ideas from declarative and imperative programming.
Allows you to balance reasoning about code and performance
(In this case, the left is preferable)

Anatomy of a Rust Program

```rust
fn main() {
    println!("fib(6) = {}", fib(6));
}
```

```rust
fn fib_imperative(n: u64) -> u64 {
    if n <= 1 {
        return n;
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        return result;
    }
}
```




```rust
fn fib_imperative(n: u64) -> u64 {
    if n <= 1 {
        n
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        result
    }
}
```

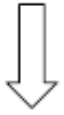
```rust
fn fib_imperative(n: u64) -> u64 {
    return if n <= 1 {
        n
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        result
    };
}
```

Anatomy of a Rust Program

Statements and expressions

Semicolon ➡️ sequence statements

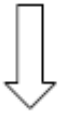
```
fn baz() -> u32 {
    100
}
```



```
fn baz() -> u32 {
    let a = qux();
    let b = buzz();
    a + b
}
```

Braces ➡️ statements in expression context

```
let x = foo();
```



```
let x = { println!("Calling foo..."); foo() };
```

Anatomy of a Rust Program

```
fn main() {
    println!("fib(6) = {}", fib(6));
}


fn fib_imperative(n: u64) -> u64 {
    if n <= 1 {
        return n;
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        return result;
    }
}
```

Type inference

- local variables are statically typed, but type inference allows omitting type annotations

Could write

```
let mut result: u64 = fib(n - 1);
```

Anatomy of a Rust Program

```rust
fn main() {
    println!("fib(6) = {}", fib(6));
}


fn fib_imperative(n: u64) -> u64 {
    if n <= 1 {
        return n;
    } else {
        let mut result = fib(n - 1);
        result += fib(n - 2);
        return result;
    }
}
```

```
error[E0384]: cannot assign twice to immutable variable `result`
 --> fib.rs:17:9
   |
16 |         let result = fib(n - 1);
   |             ------
   |             |
   |             first assignment to `result`
   |             help: consider making this binding mutable: `mut result`
17 |         result += fib(n - 2);
   |         ^^^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0384`.
```

`mut` keyword

- bindings are immutable by default
- Reverse of C/C++ `const` keyword

Rapid fire time

```
fn main() {
    let s = "foobar"; // string literals

    let x = 1.0 + 2.0 / 3.0 * 4.0; // arithmetic

    let b = true || false; // bools

    let s: bool = 1 < 2; // explicit type annotations

    let c = '😻'; // unicode

    let tup = ('🦀', "Ferris"); // tuples

    while false {
        println!("Uh-oh");
    } // looping
}
```

Philosophical Takeaways

Rust emphasizes ~safety~

- immutable by default

Rust emphasizes ~control~

- declarative code for pure functions,
  imperative code for procedural algorithms

Rust emphasizes ~productivity~

- type inference
- helpful error messages

Does it compile? Should it?

```
fn main() {
    x = 5;
    println!("{}", x + 1);
}
```

```
error[E0425]: cannot find value `x` in this scope
--> shadow.rs:2:5
  |
2 |     x = 5;
  |     ^
  |
help: you might have meant to introduce a new binding
  |
2 |     let x = 5;
  |     +++
```

Does it compile? Should it?

```
fn main() {
    let x = 5;
    println!("{}", x + 1);
}
```

Does it compile? Should it?

```rust
fn main() {
    let mut x = 5;
    println!("{}", x + 1);
}
```

```
warning: variable does not need to be mutable
 --> shadow.rs:13:9
  |
13 |     let mut x = 5;
  |         ----^
  |         |
  |         help: remove this `mut`
  |
  = note: `#[warn(unused_mut)]` on by default
```

Does it compile? Should it?

```rust
fn main() {
    let x = 5;
    let x = 6;
    println!("{}", x + 1);
}
```

Does it compile? Should it?

```
fn main() {
    let mut x = 5;
    let x = 6;
    println!("{}", x + 1);
}
```

Does it compile? Should it?

```
fn main() {
    let mut x = 5;
    x = 6;
    println!("{}", x + 1);
}
```

Does it compile? Should it?

```
fn main() {
    let mut x = 5;
    x = "🦀🦀🦀";
    println!("{}", x);
}
```

```
error[E0308]: mismatched types
 --> shadow.rs:37:9
   |
36 |     let mut x = 5;
   |                 - expected due to this value
37 |     x = "🦀🦀🦀";
   |         ^^^^^^^^ expected integer, found `&str`
```

Does it compile? Should it?

```
fn foo() -> u32 {
    let x = 5;
    x
}
```

Does it compile? Should it?

```
fn foo() -> u32 {
    if true {
        1
    } else {
        2
    }
}
```

Does it compile? Should it?

```
fn foo() -> u32 {
    if true {
        1
    } else {
        '🦀'
    }
}
```

```
error[E0308]: mismatched types
 --> func.rs:20:9
   |
16 | fn foo2() -> u32 {
   |              --- expected `u32` because of return type
...
20 |         '🦀'
   |         ^^^^ expected `u32`, found `char`
   |
help: you can cast a `char` to a `u32`, since a `char` always occupies 4 bytes
   |
20 |         '🦀' as u32
   |              ++++++
```

Does it compile? Should it?

```
fn foo() {
    let x = if true {
        1
    } else {
        '🦀'
    };
}
```

```
error[E0308]: `if` and `else` have incompatible types
 --> func.rs:37:9
   |
34 |       let x = if true {
   |  _____________-
35 | |         1
   | |         - expected because of this
36 | |     } else {
37 | |         '🦀'
   | |         ^^^^ expected integer, found `char`
38 | |     };
   | |_____- `if` and `else` have incompatible types
```

Does it compile? Should it?

```rust
fn foo(n: u32) -> u32 {
    match n {
        0 | 1 => 0,
        2 | 3 | 4 | 5 => 1
    }
}
```

```
error[E0004]: non-exhaustive patterns: `6_u32..=u32::MAX` not
covered
 --> func.rs:25:11
   |
25 |     match n {
   |           ^ pattern `6_u32..=u32::MAX` not covered
   |
  = note: the matched value is of type `u32`
help: ensure that all possible cases are being handled by adding a
match arm with a wildcard pattern or an explicit pattern as shown
   |
27 ~        2 | 3 | 4 | 5 => 1,
28 +        6_u32..=u32::MAX => todo!()
   |
```

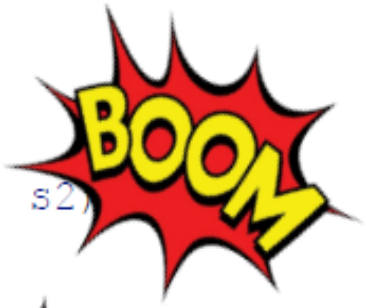
Ownership!

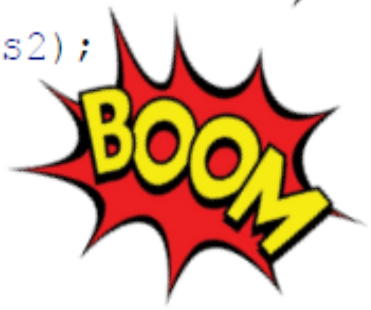
Where can data be allocated?

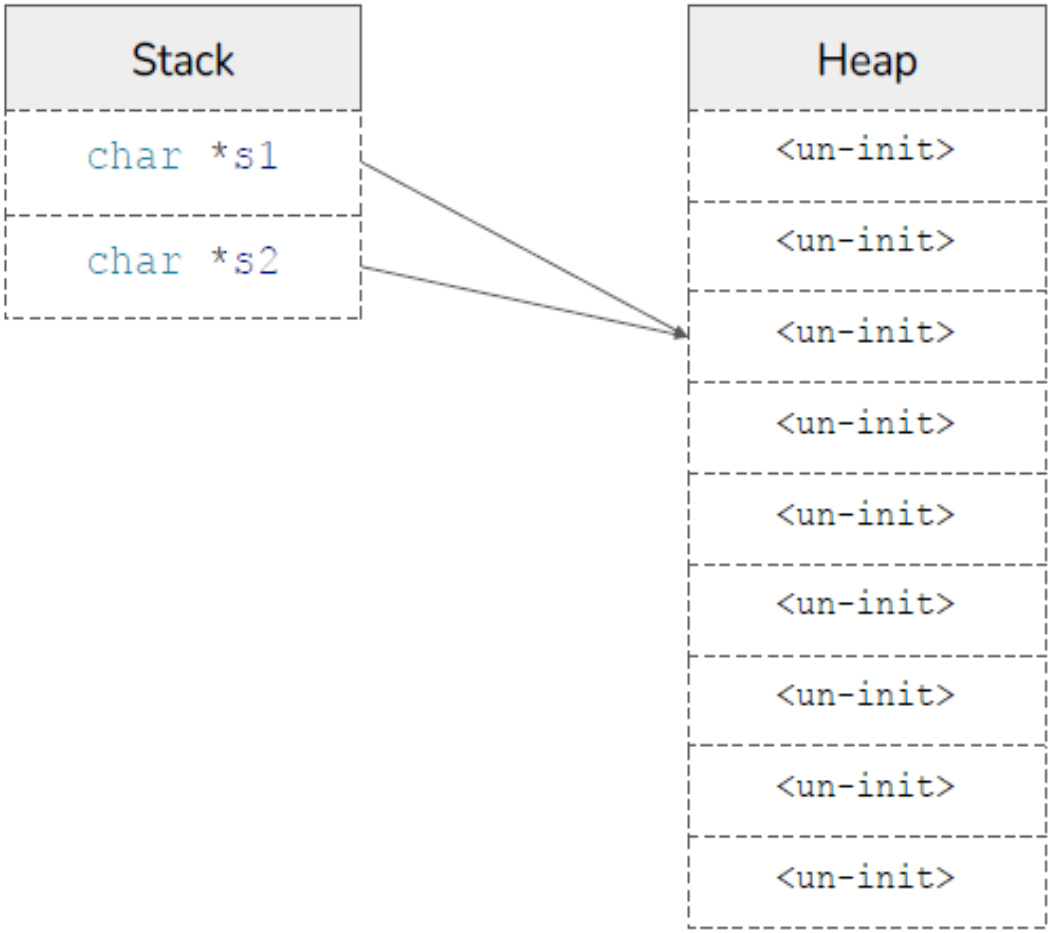
	Static memory	Stack	Heap
Example?	`static float PI = 3.14;` **OR** `char *s = "cis1905";`	`void foo() { Point p = {.x=1,.y=2}; }`	`char *init_username() { char *s; malloc(&s, username_length); ... } void drop_username(char *s) { free(name); }`
How long does it live?	Entire program lifetime	Until end of function	Until explicitly deallocated with `free`
Pros/Cons?	Zero cost Fixed-size	Low performance cost Can't outlive function	Supports allocations of unknown size Error-prone

Heap Programming Challenges

```
int main() {
    char *s1;
    malloc(&s1, 5);
    *s1 = {'p','e','n','n','\0'};

    char *s2 = s1;
    free(s1);
    printf("%s\n", s2);

    free(s2);
}
```







What went wrong here?

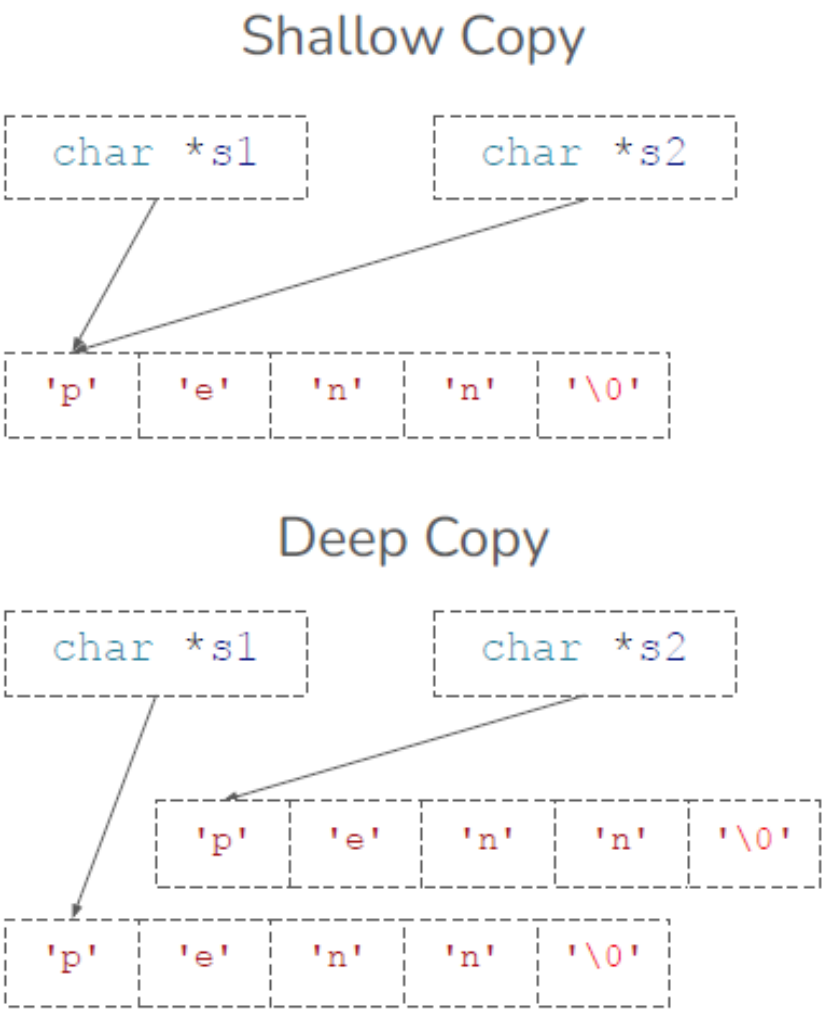
1. Shallow copies vs. deep copies
2. Who is in charge of freeing data?

Recall from lecture 1:

How can we prevent memory safety issues...

- buffer overflow
- use-after-free
- double free

...while still giving the programmer control of heap allocations?



Ownership!

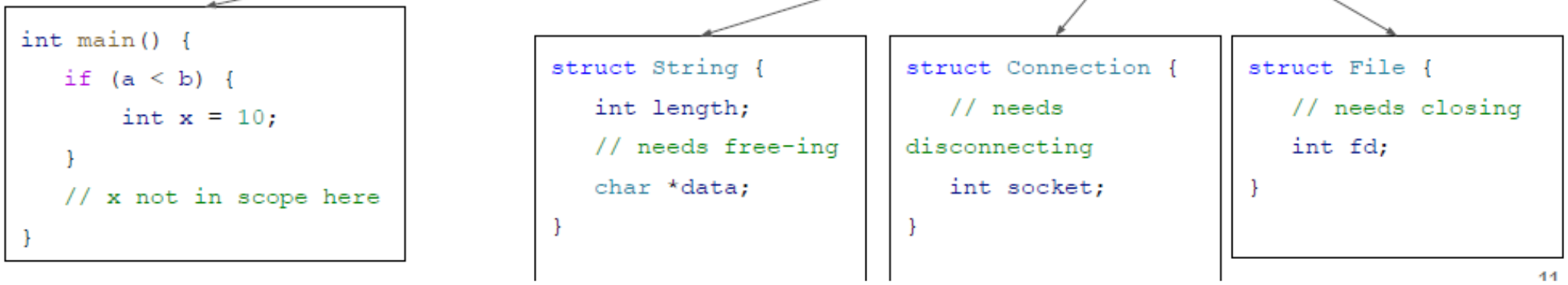
Three golden rules:

1. Each value in Rust has an *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Ownership!

Three golden rules:

1. Each value in Rust has an *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.



```
int main() {
    if (a < b) {
        int x = 10;
    }
    // x not in scope here
}
```

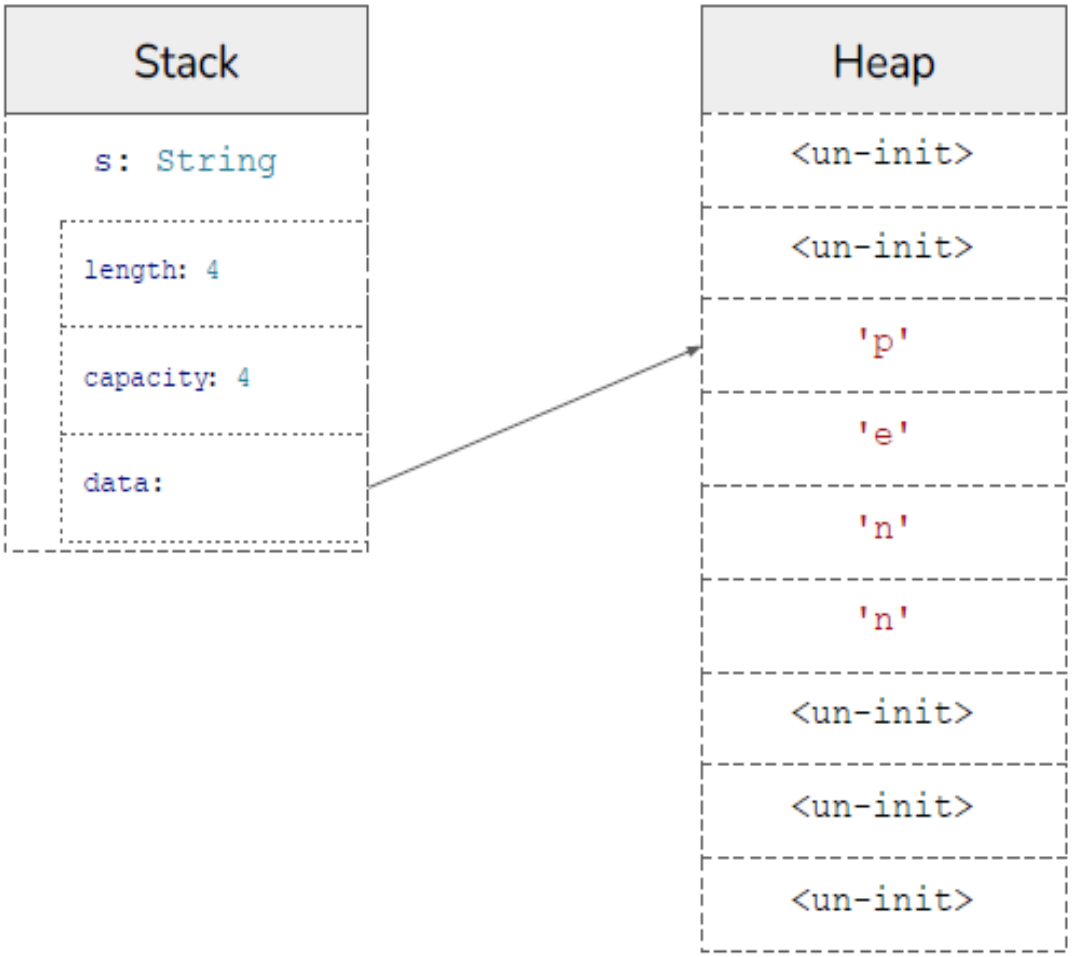
```
struct String {
    int length;
    // needs free-ing
    char *data;
}
```

```
struct Connection {
    // needs
disconnecting
    int socket;
}
```

```
struct File {
    // needs closing
    int fd;
}
```

Examining Ownership with Strings

Numeric types are too simple. Next week we'll talk about defining custom data types, but for now we'll use `std::String`, Rust's built-in String type




```
fn main() {
    let s = String::from("penn");
}
```

Examining Ownership with Strings

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1;
    drop(s1);
    drop(s2);
}
```

generic function to trigger destructor

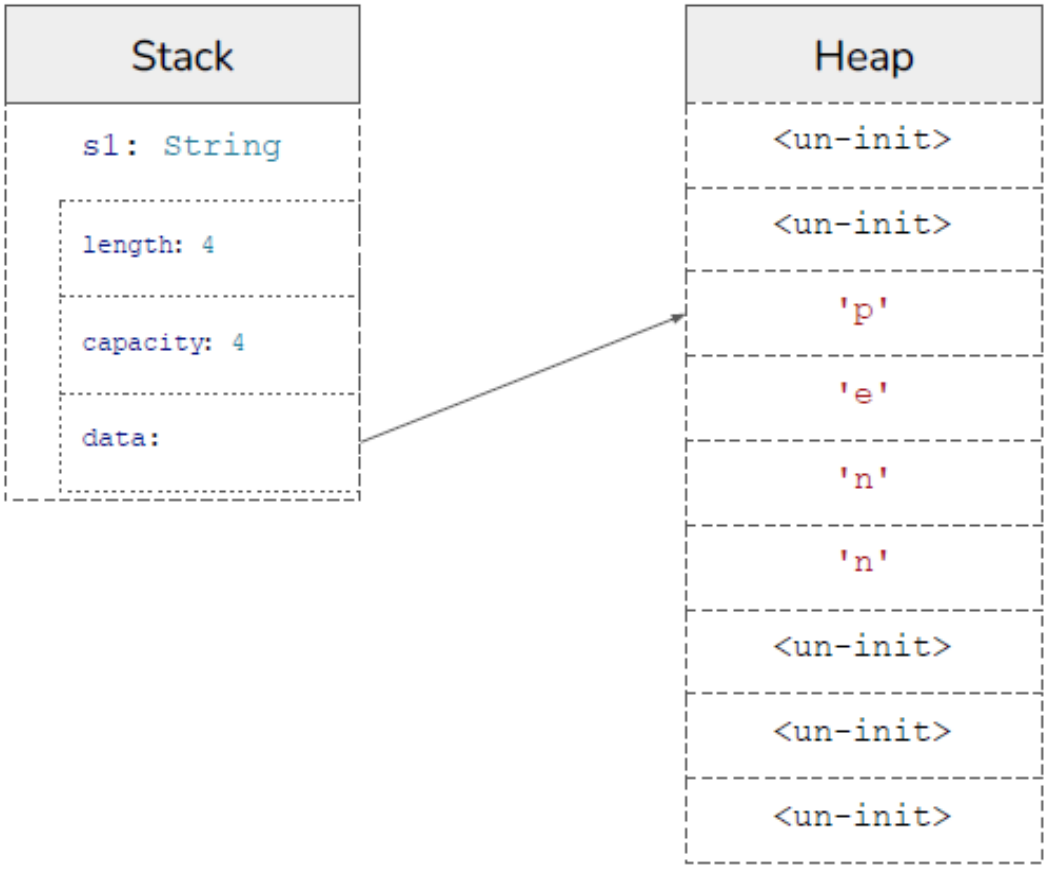
1. Each value in Rust has an *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

No double free !

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

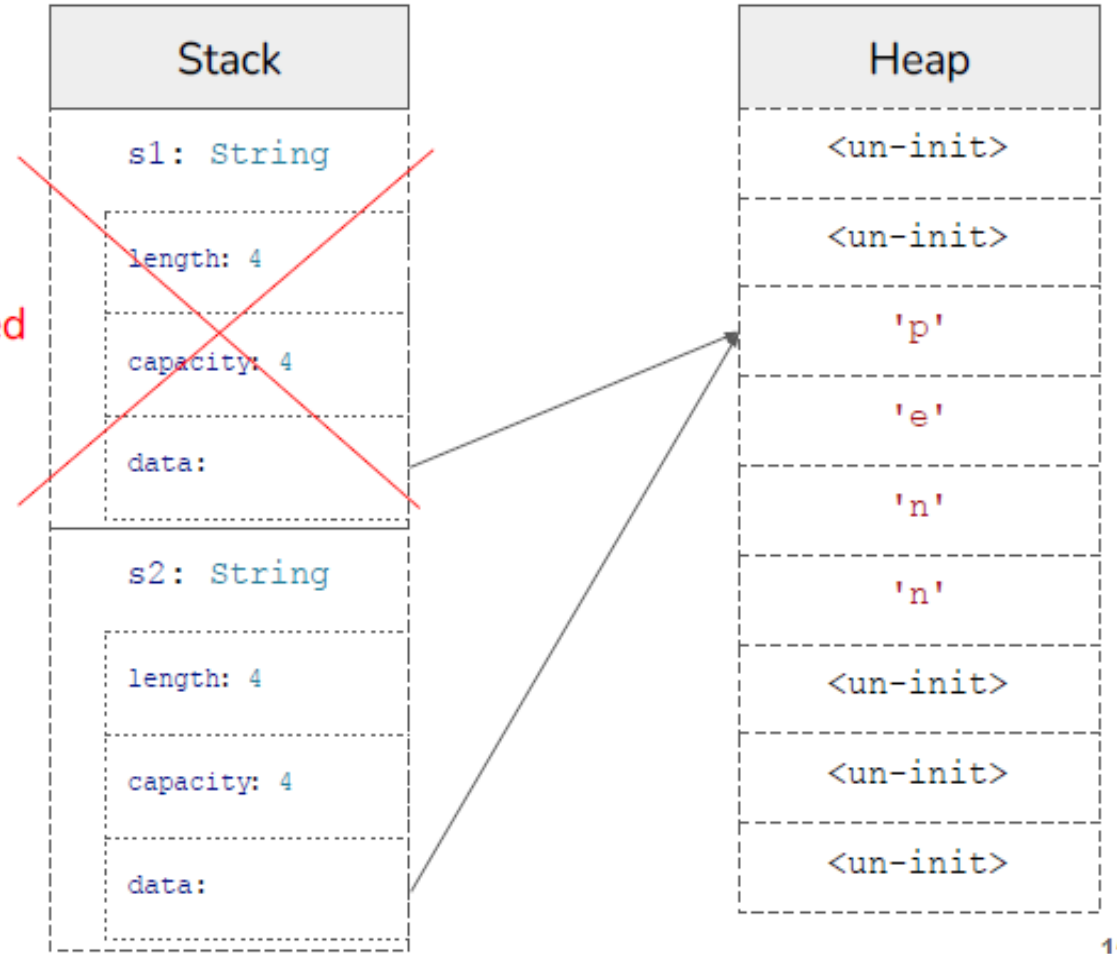
What's in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1;
    drop(s1);
    drop(s2);
}
```



What's in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1;
    drop(s1);
    drop(s2);
}
```



Move ≈ shallow copy + invalidate old owner

Moves are fast! (*O(1)*)

What's in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1;
    drop(s1);
    drop(s2);
}
```

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

What's in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1;
    drop(s1);
    drop(s2);
}
```

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
  |
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                 ++++++++
```

What's in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1.clone();
    drop(s1);
    drop(s2);
}
```

`clone`:

- Deep copy
- Available on most built-in types
- Automatically derive for your own types
- When is a type not cloneable?

What's in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1.clone();
    drop(s1);
    drop(s2);
}
```

```
fn main() {
    let s1: u32 = 1337;
    let s2 = s1;
    drop(s1);
    drop(s2);
}
```

Why no error??

`clone`:

- Deep copy
- Available on most built-in types
- Automatically derive for your own types
- When is a type not cloneable?

`Copy`:

- Types with trivial `clone` functions can be marked copy
- In that case move==clone and you don't have to worry about ownership
- These types often also have trivial destructor functions

What types are `Copy`?

- All numeric types (integers and floats)
- `bool`
- `char`
- Tuples if their members are `Copy` (e.g. `(i32, f64)`)

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

Ways to transfer ownership

1. Assignment (see previous example)
2. Function calls

Hang on... why do you need ownership to print?

```
fn main() {
    let s1 = String::from("penn");
    print_str(s1);
    drop(s1);
}
fn print_str(s: String) {
    println!("{}", s);
}
```

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     print_str(s1);
  |               -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

Borrowing

Need access to a value without owning it?

- Try borrowing
- Defaults to immutable, can also borrow mutably

```
fn main() {
    let s1 = String::from("penn");
    print_str(&s1);
    drop(s1);
}
fn print_str(s: &String) {
    println!("{}", s);
}

fn clear_str(s: &mut String) {
    s.clear();
}
```

✅

What about return values?

Return values can transfer ownership too

```
fn main(){
    let s = String::from("I love Rust");
    let with_ferris = add_ferris(s);
}
```

ownership in

ownership out

```
fn add_ferris(s: String) -> String {
    s + "🦀"
}
```

Other ways to write this function

- Pros and Cons?

```
fn add_ferris1(s: &String) -> String {
    s.clone() + "🦀"
}
fn add_ferris2(s: &mut String) {
    s.push_str("🦀")
}
```

View Types

&str and &[T]

A Motivating Example

```
/// Returns last 4 chars of course name
/// e.g. cis1905 -> 1905
fn course_code(course: &String) -> &str {
    ...
}
```

How would you implement this function based on the signature?

One solution: create a new string and copy bytes from the `course` string to it

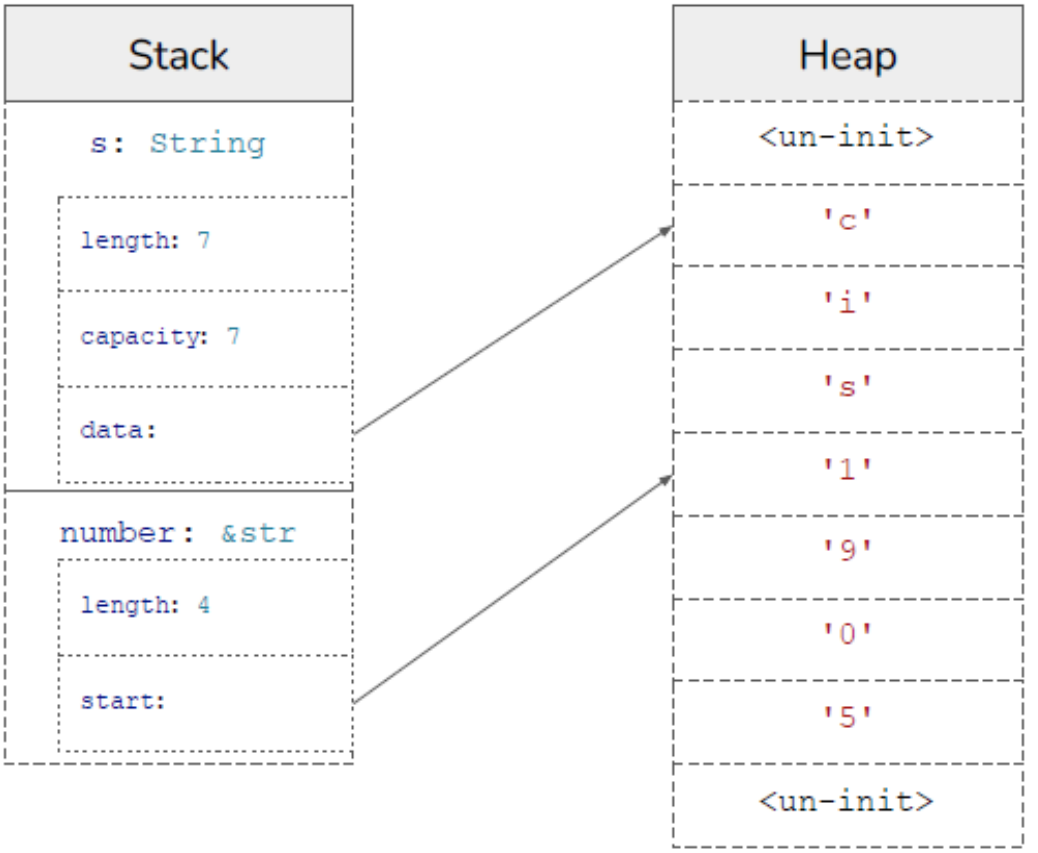
- Inefficient—the bytes already exist in memory so why copy?

A Motivating Example

```
/// Returns last 4 chars of course name
/// e.g. cis1905 -> 1905
fn course_code(course: &String) -> &str {
    course[3..7]
}

fn main() {
    let s = String::new("cis1905");
    let number = course_code(&s);
}
```

"Fat pointer":

- A pointer along with some data
- Never see just `str`, always `&str` or `&mut str`
- Function arg should always be `&str`,

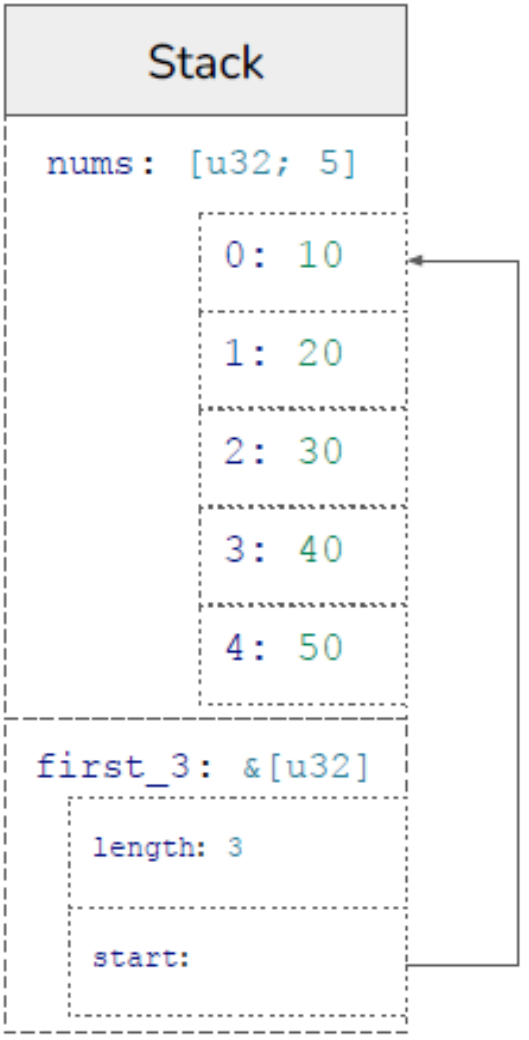


27

Another Fat Pointer: &[T] ("Slice")

Like &str, but for collections of any type

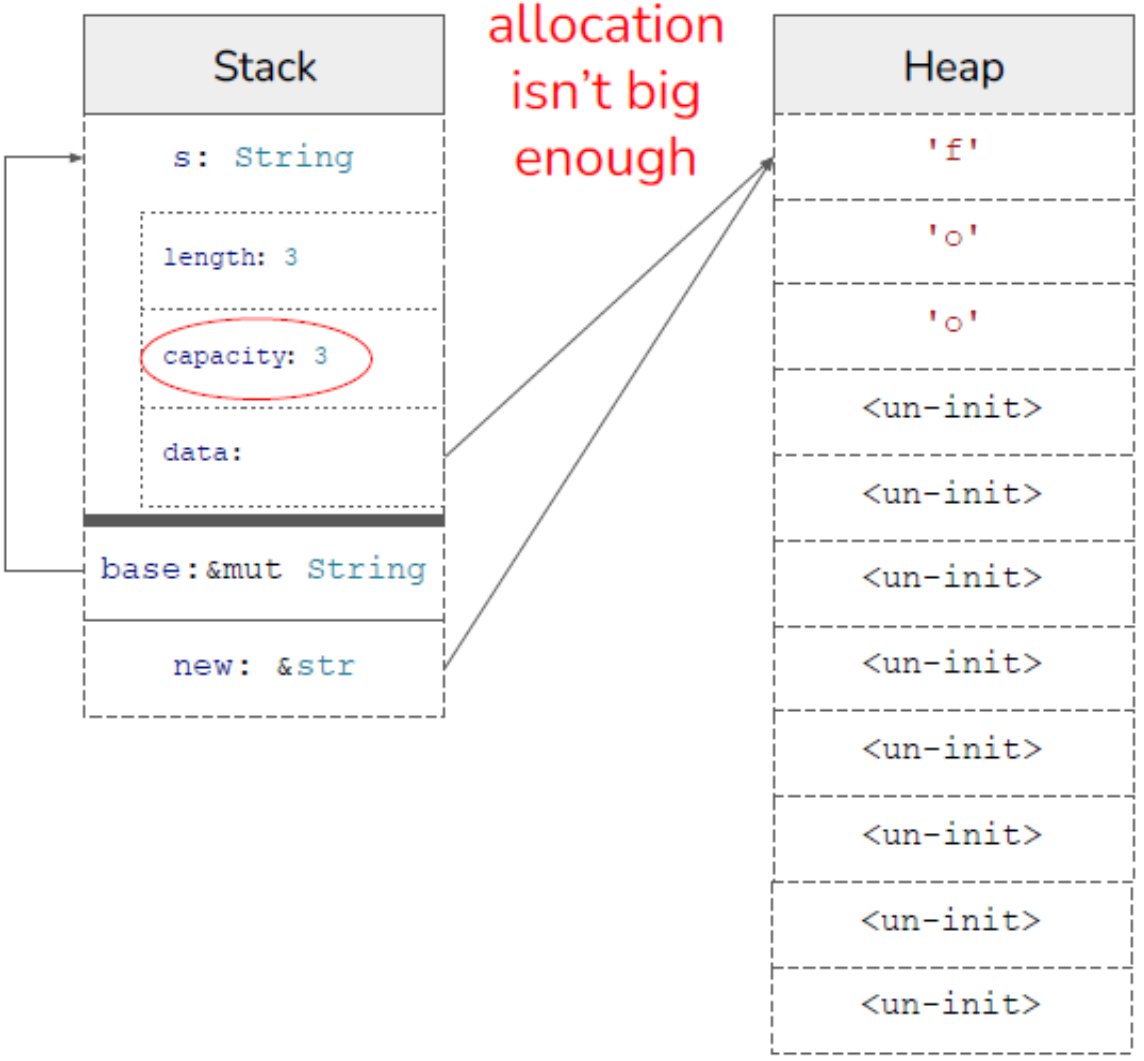
```
fn main() {
    let nums = [10, 20, 30, 40, 50];
    let first_3 = first_3(&nums);
}
fn first_3(arr: &[u32; 5]) -> &[u32] {
    &arr[0..3]
}
```



Controlling mutability

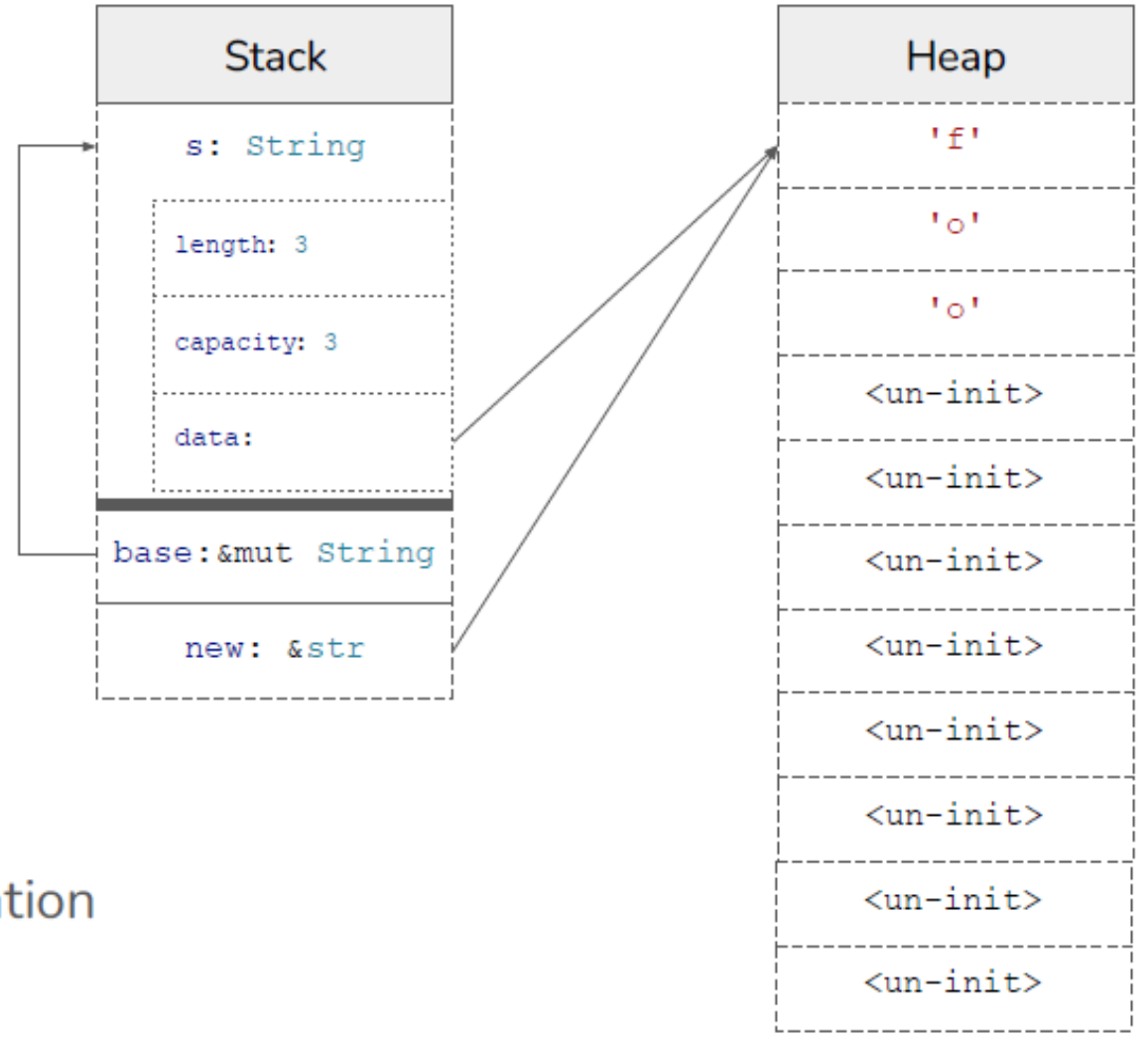
An example

```
fn str_append(
     base: &mut String,
     new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```



An example

```
fn str_append(
      base: &mut String,
      new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```

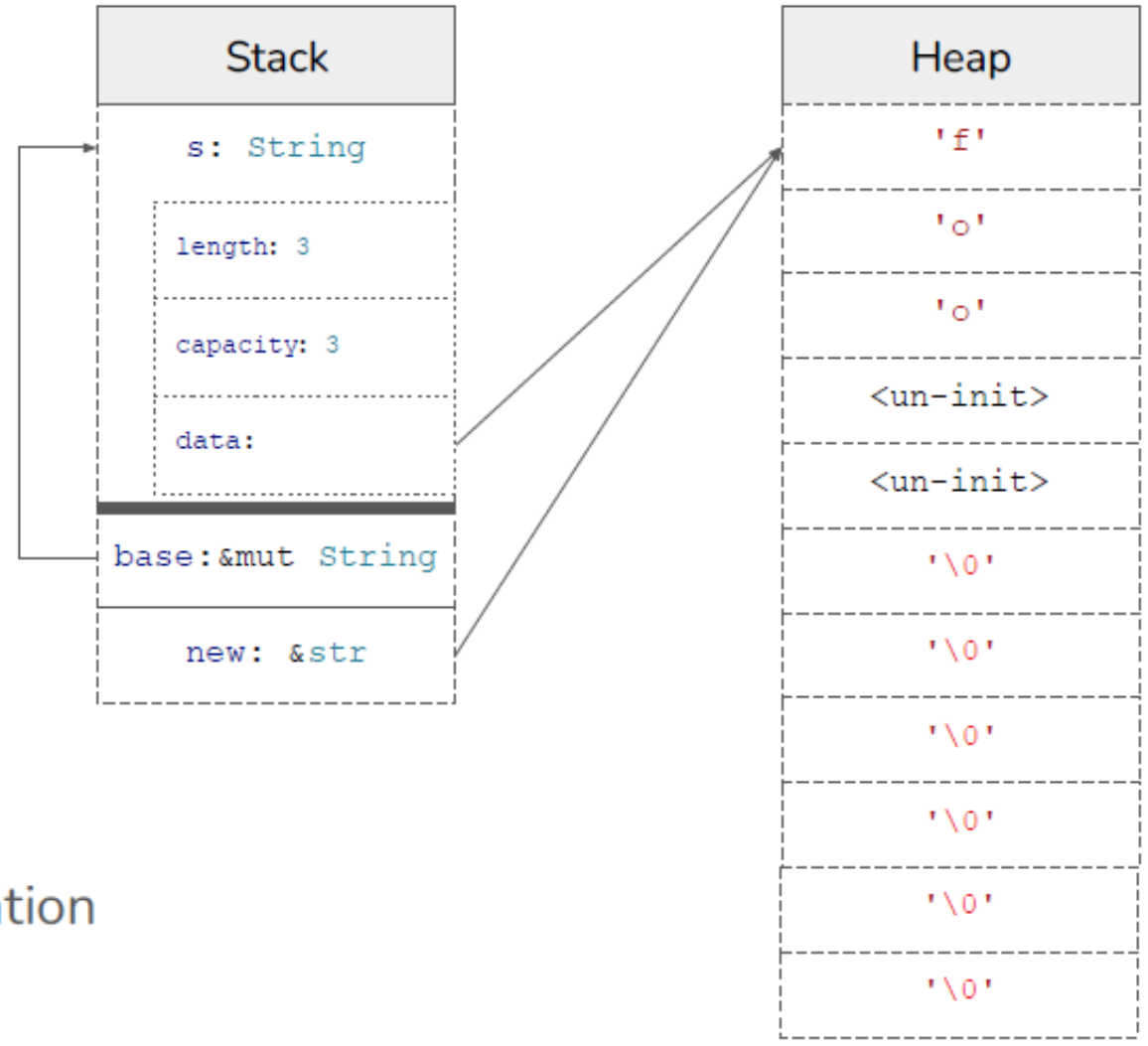


Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

An example

```
fn str_append(
      base: &mut String,
      new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```

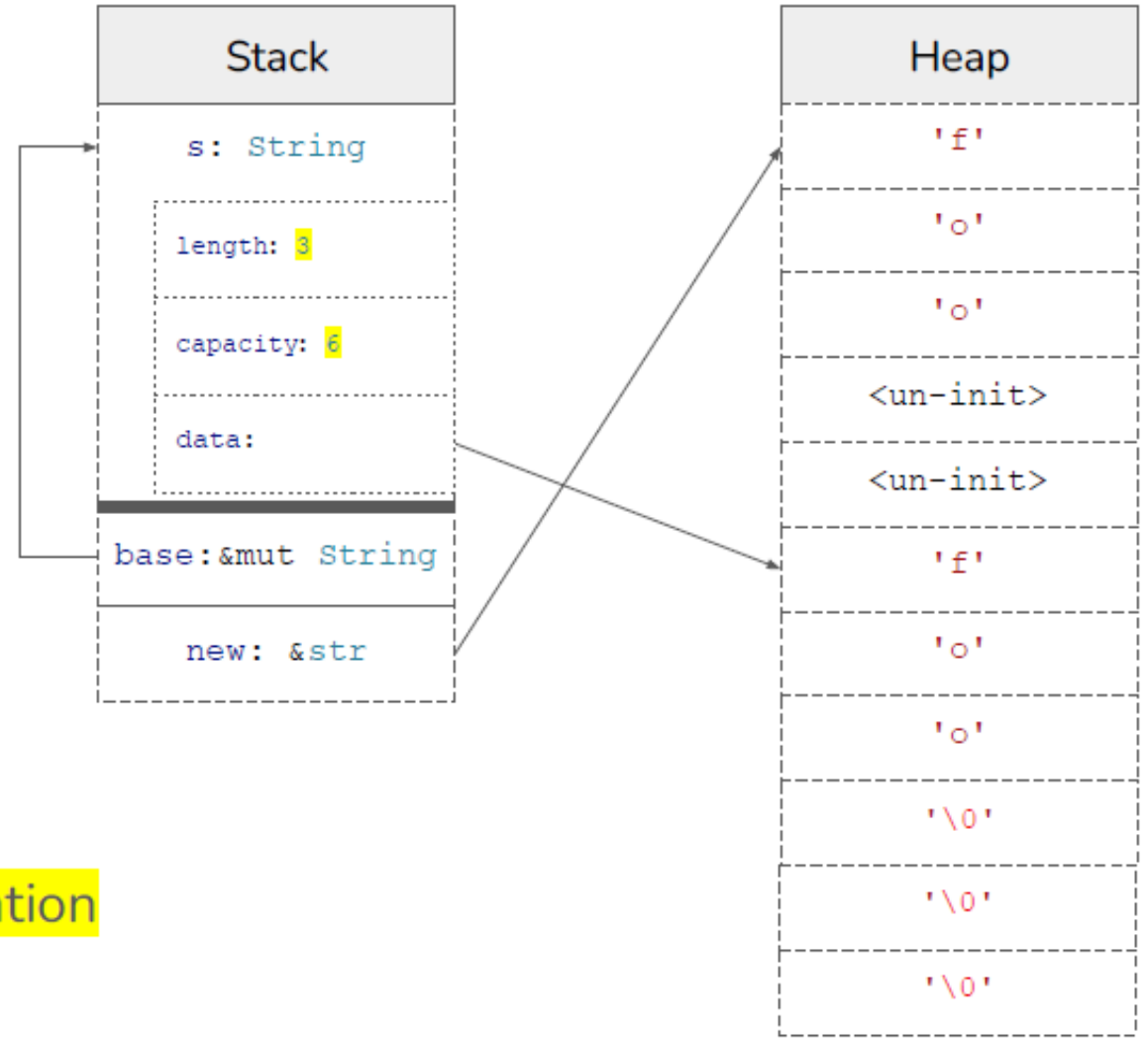


Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

An example

```
fn str_append(
      base: &mut String,
      new: &str) {
    base.push_str(new);
}

fn main() {
    let mut s = String::from("foo");
    str_append(&mut s, &s);
}
```

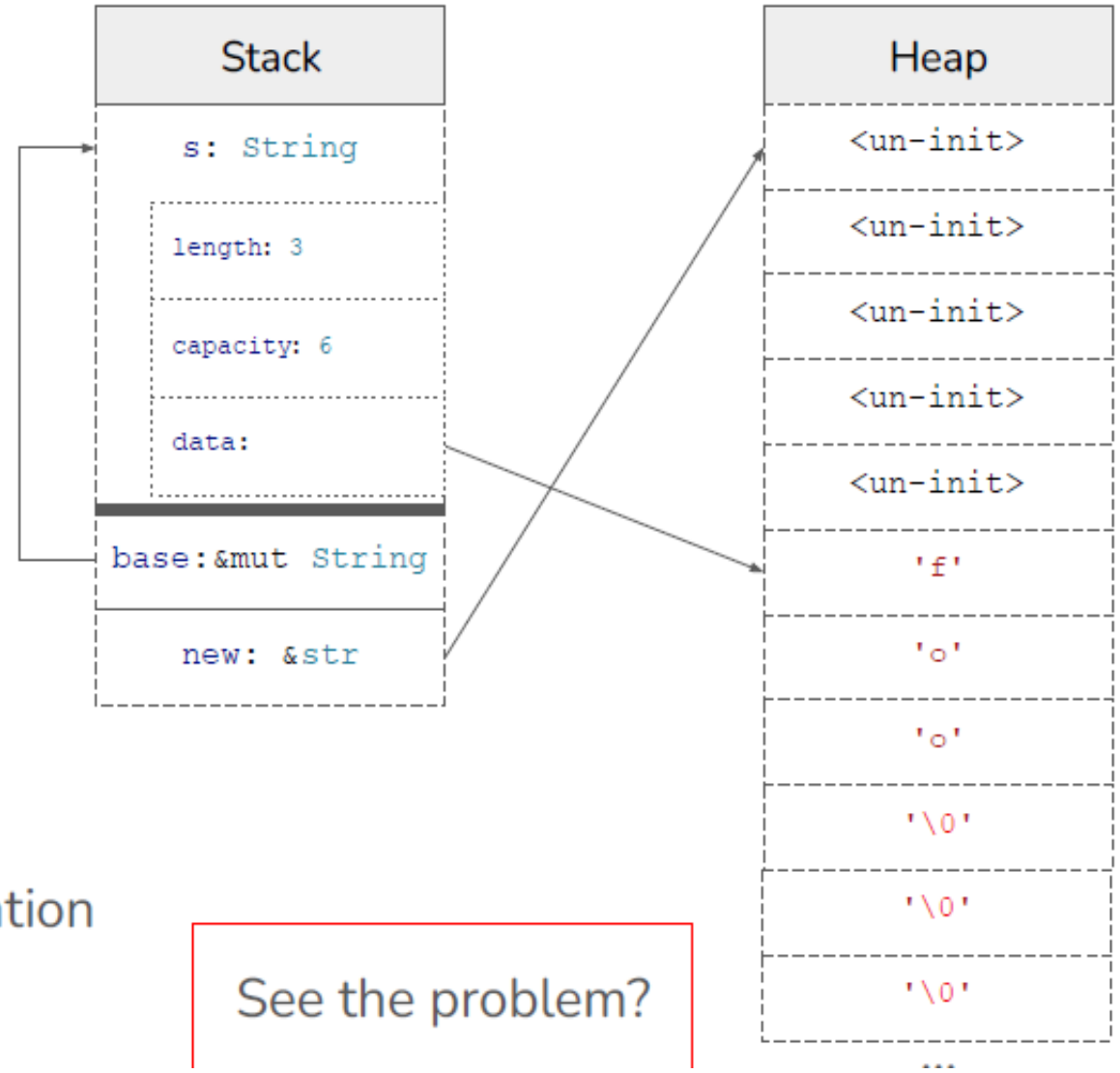


Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

An example

```
fn str_append(
      base: &mut String,
      new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```



Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

See the problem?

The Rule of References:

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

An example

```
fn str_append(
     base: &mut String,
     new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
--> lifetimes.rs:9:24
  |
9 |     str_append(&mut s, &s);
  |     ---------- ------  ^^ immutable borrow occurs here
  |     |          |
  |     |          mutable borrow occurs here
  |     mutable borrow later used by call
```

Quiz

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

let r1 = &s;
let r2 = &s;
println!("{} and {}", r1, r2);

let r3 = &mut s;
println!("{}", r3);
```

Does it compile?

Recap

Ownership:

1. Each value in Rust has an *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Transfer ownership with *move* (like a shallow copy)

- When assigning
- When calling/returning from functions

Opt out of moving by `clone`ing (performance hit)

References:

To avoid transferring ownership, borrow an owned value to get a reference

- Nothing happens when reference goes out of scope

References can be immutable or mutable

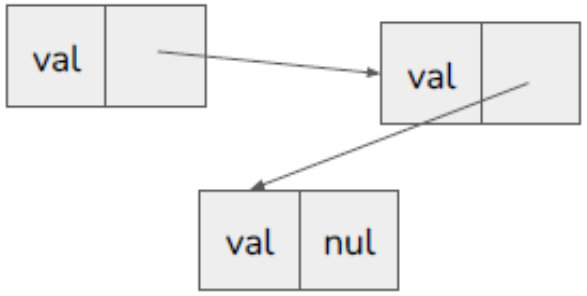
- At any given time, you can have either one mutable reference or any number of immutable references.

References must always be valid.

Defining New Types

Trying to make a linked list

What's a linked list?



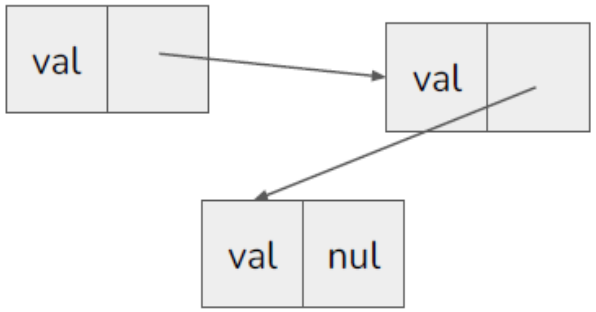
How would you go about implementing a Linked List class in C or C++?

- What structs would you need?
- What kinds of methods would you provide?
- What would your test code look like?
- In terms of memory errors we've been talking about, what could go wrong?

Based on what you know about Rust so far, what do you think will be challenging about implementing a linked list in Rust?

Trying to make a linked list

What's a linked list?




```
struct Node {
    int value;
    Node* next;
}

int main() {
    Node* first = (Node*) malloc(sizeof(Node));
    first->value = 1;
    Node* second = (Node*) malloc(sizeof(Node));
    second->value = 2;
    first->next = second;
    /* do stuff (e.g., print the list) */
    free(first);
    free(second);
}
```

Defining data types in Rust

```
struct Person {
    name: String,
    location: String,
}
fn main() {
    let me = Person {
        name: String::from("paul"),
        location: String::from("Philadelphia")
    };
    println!("{} lives in {}",
        me.name,
        me.location);
}
```

`struct` keyword declares new structs

- each member has a name and a type
- instantiate structs using {}

How to make a Node?

C:

```c
struct Node {
    int value;
    Node* next;
}
```

Rust:

```rust
struct Node {
    value: i32,
    next: Node,
}
```

Infinitely sized struct

```rust
struct Node {
    value: i32,
    next: &Node,
}
```

Borrowing whose data?

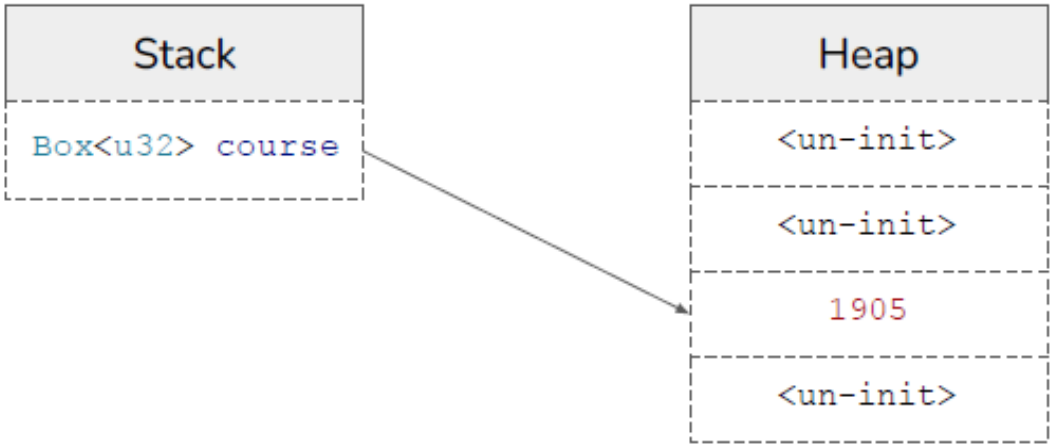
```rust
struct Node {
    value: i32,
    next: /* pointer to a node...? */
}
```

How to make a Node? Box!

```
struct Node {
    value: i32,
    next: Box<Node>
}
```

- Make a Box of some type T
- a T gets put on heap, Box points to that T
- Box owns that T. When Box goes out of scope, the T is destroyed.

```
fn main() {
    let course = Box::new(1905);
}
```



Single Node List

```
struct Node {
    value: i32,
    next: Box<Node>
}
fn main() {
    let list = Box::new(Node {
        value: 1905,
        next: /* null??*/
    });
}
```

Single Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let list = Box::new(Node {
        value: 1905,
        next: None
    });
}
```

An `Option<T>` is either a `T` or `None`.

```
fn main() {
    let student_grade: Option<char> = Some('A');
    let instructor_grade: Option<char> = None;
}
```

Two Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first = Box::new(Node {
        value: 1905,
        next: None
    });
    let second = Box::new(Node {value: 1200,next: None});
    first.next = second;
}
```

What's wrong with this?

Two Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let second = Box::new(Node {value: 1200,next: None});
    first.next = Some(second);
}
```

Three Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let mut second = Box::new(Node {value: 1200,next: None});
    let third = Box::new(Node {value: 4100,next: None});
    first.next = Some(second);
    second.next = Some(third);
}
```

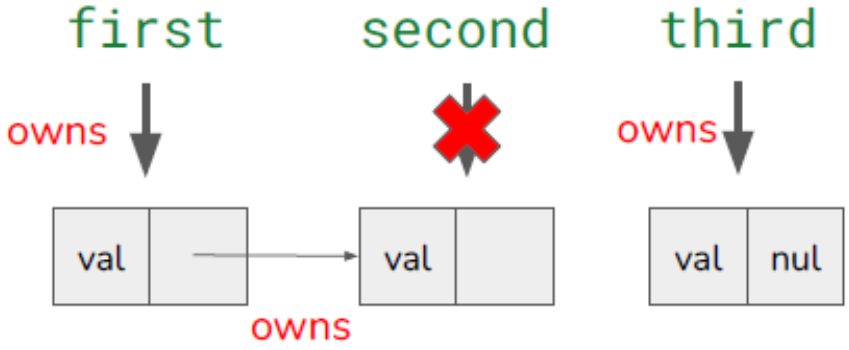
```
error[E0382]: assign to part of moved value: `*second`
 --> list.rs:15:5
   |
13 |     let third = Box::new(Node {value: 4100,next: None});
14 |     first.next = Some(second);
   |                       ------ value moved here
15 |     second.next = Some(third);
   |     ^^^^^^^^^^^ value partially assigned here after move
```

structs own their data

- therefore, assigning to a struct member transfers ownership

Three Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let mut second = Box::new(Node {value: 1200,next: None});
    let third = Box::new(Node {value: 4100,next: None});
    first.next = Some(second);
    second.next = Some(third);
}
```



Implication: when `first` is dropped:

- First node of list is dropped,
- ...so Option (in Node struct) is dropped,
- ...so Box (in Option) is dropped,
- ...so second Node (in Box) is dropped.

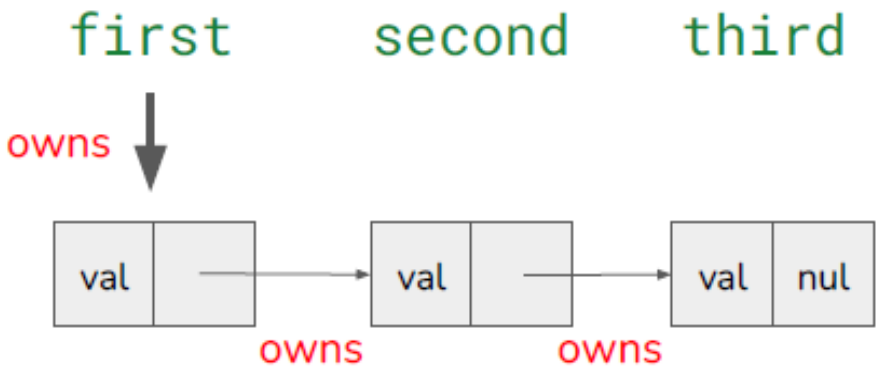
"Chain of ownership"

15

Three Node List Second Attempt

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let mut second = Box::new(Node {value: 1200,next: None});
    let third = Box::new(Node {value: 4100,next: None});
    second.next = Some(third);
    first.next = Some(second);

}
```

swap order



Traversing List

```
struct Node {
    int value;
    Node* next;
}
Node *curr = first;
while (curr != NULL) {
    printf("%d\n", curr->value);
    curr = curr->next;
}
```

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let curr = /* ?? */;
    while curr != /* NULL ? */ {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

What should `curr` be?

- Can't use pointers
- Don't want to take ownership

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

`curr` has type `Option<&Box<Node>`

- contains either a reference to a box containg `Node` or `None`

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

```
error[E0609]: no field `value` on type `Option<&Box<_>>`
 --> list.rs:11:30
   |
11 |         println!("{}", *curr.value);
   |
```

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

```
loop {
    match curr {
        Some(node) => {
            println!("{}",
node.value);
            curr = node.next;
        },
        None => break
    }
}
```

Separating functionality

```
std::list<int> myList;
myList.push_front (200);
myList.push_front (300);
myList.pop_back ();
```

Goal: associate functionality with data by writing methods like `push_front`

Separating functionality

```
struct LinkedList {
    head: Option<Box<Node>>,
    length: usize, // optional
}
impl LinkedList {
    fn new() -> LinkedList {
        LinkedList {
            head: None,
            length: 0,
        }
    }
}
```

`impl` blocks:

- write functions associated with a type
- accessible as `LinkedList::new()`

Constructors:

- don't exist in Rust
- By convention, provide a `new` function to create instances of your type

Separating functionality

```
struct LinkedList {
    head: Option<Box<Node>>,
    length: usize, // optional
}
impl LinkedList {
    fn new() -> LinkedList {
        LinkedList {
            head: None,
            length: 0,
        }
    }
    fn len() -> usize {
        length
    }
}
```

```
error[E0425]: cannot find value `length` in this scope
 --> list.rs:14:9
   |
14 |         length
   |         ^^^^^^
```

Separating functionality

```
struct LinkedList {
    head: Option<Box<Node>>,
    length: usize, // optional
}
impl LinkedList {
    fn new() -> LinkedList {
        LinkedList {
            head: None,
            length: 0,
        }
    }
    fn len(&self) -> usize {
        self.length
    }
}
```

Methods

- Just functions that take a `self` parameter
- Can take `self`, `&self`, or `&mut self`

```
fn main() {
    let list = LinkedList::new();
    let len = list.len();
}
```

Structs

Declared with `struct` keyword

- Can't contain themselves directly, use a Box to break up recursion
- Initialized with brackets (`Node {value:1}`)

Declare functions associated with a struct using an `impl` block

- **associated functions**: don't take a `self` parameter and are called like `Node::new()`
- **methods**: take `self`, `&self`, or `&mut self` and are called like `list.len()`;

By convention, provide a `new` function that acts as a constructor

`Box` owns a value allocated on the heap

- When the box goes out of scope, the value is deallocated
- Auto-deref `Box<T>` into `&T` or `&mut T`

Structs have ownership of their values

- Accessing a struct element can move data out of the struct
- Assigning to a struct element can move data into that struct

Other structs you might see: tuple structs

```
struct Point { x: i32, y: i32 }
fn main() {
    let p = Point { x: 1, y: 2 };
    let x = p.x;
    let y = p.y;
    match p {
        Point { x: x_coord, y: y_coord } =>
{
            println!("{}, {}", x, y);
        }
    }
}
```

```
struct Point(i32, i32)
fn main() {
    let p = Point(1, 2);
    let x = p.0;
    let y = p.1;
    match p {
        Point(x, y) => {
            println!("{}, {}", x, y);
        }
    }
}
```

Struct field names are optional–structs without field names are “tuple structs”

Other structs you might see: wrapper types

```rust
impl f32 {
    fn to_centimeters(self) -> f32 {
        self * 2.54
    }
}
```

```
error[E0390]: cannot define inherent `impl` for
primitive types
--> wrapper.rs:1:1
  |
1 | impl f32 {
  | ^^^^^^^^
```

```rust
struct Inches(f32);
impl Inches {
    fn to_centimeters(self) -> f32 {
        self.0 * 2.54
    }
}
```

Wrap an existing type in a struct

- Separate functionality (e.g. distinguish inches from centimeters at the type leve)
- Add functionality to primitive types

Making our own Option

`Option`: a type that is a value *OR* no value

`structs`: a type that is a value *AND* another value (and another and another...)

No way to implement `Option` with `struct`

- Need a new language construct...

Making our own Option

```
enum NumOption {
    Some(u32),
    None
}

fn main() {
    let id = NumOption::Some(5);
    match id {
        NumOption::Some(i) =>
            println!("{} is some", i),
        NumOption::None =>
            println!("None")
    }
}
```

Enums!

- Better than C enums -> can contain data
- Like OCaml `type` keyword

`NumOption` can be in one of two states:

- `Some`, in which case a value of type `u32` is guaranteed to be present
- None, in which case no values are present

Access different constructors using `::` syntax

Destructure using pattern matching

34

Making our own Option

```
enum NumOption {
    Some(u32),
    None
}

impl NumOption {
    fn subtract_one(&mut self) {
        match self {
            NumOption::Some(i) => *i -= 1,
            NumOption::None => {}
        }
    }
}
```

Enums can have methods/associated functions as well

Option's Cousin: Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` has either a success value of type `T`, or an error value of type `E`.

- `E` contains data (often an error message) that clarifies what the exact error was

Preferred over `Option` when more context for the error is needed

```
fn create(path: String) -> Result<File, IoError>

impl f32 {
    fn from_str(src: &str) -> Result<f32, ParseFloatError>
}
```

Error Handling Woes

```
fn main() -> Result<(), &str> {
    let mut file = match File::create("foo.txt") {
        Ok(file) => file,
        Err(_) => return Err("Failed to create file"),
    };
    match file.write_all(b"Hello, world!") {
        Ok(_) => {},
        Err(_) => return Err("Failed to write to file"),
    };
    match file.flush() {
        Ok(_) => {},
        Err(_) => return Err("Failed to flush file"),
    };
    return Ok(());
}
```

So much code just to open and write to a file!

- Most of it's error handling
- There must be a better way...

Error Handling Woes

```
fn main() -> Result<(), &'static str> {
    let mut file = File::create("foo.txt").unwrap();
    file.write_all(b"Hello, world!").unwrap();
    file.flush().unwrap();
    return Ok(());
}
```

Just `unwrap` it all

- Method available on `Option` and `Result`
- Returns the inner type or aborts the program if not available (i.e. `None` or `Err`)

About print…

So far, we've seen magic printing with `println!` But if we use our own types…

```
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{}", id);
}
```

```
error[E0277]: `Id` doesn't implement `std::fmt::Display`
--> print.rs:7:20
  |
7 |     println!("{}", id);
  |                    ^^ `Id` can't be formatted with the default
formatter
```

But implementing print functions is boilerplate, just print every member right?

Deriving traits

So far, we've seen magic printing with `println!` But if we use our own types...

```
#[derive(Debug)]
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{:?}", id);
}
```

Use `#[derive(...)]` to automatically implement functionality

- e.g. printing, hashing

```
> rustc print.rs && ./print
Id { id: 1905 }
```

Generics: Motivating Example

```
enum NumOption {
    Some(u32),
    None
}
```

Boo! Bad!

- Need to duplicate code for every option type you want
- Need to duplicate functions that take `Option` to work on every option type

```
enum Option<T> {
    Some(T),
    None
}
```

Yes! Better!

- Template for declaring any kind of `Option` you need
- Lets you define functionality for an `Option` of any type

Generics

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Structs or enums can be generic over one or more types

```
fn first<T,U>(x: T, y: U) -> T {
    x
}
```

Correspondingly, functions can be generic over one or more types

Using Generic Types

```
enum Opt<T> {
    Some(T),
    None
}

fn main() {
    // type is inferred, annotation is optional
    let x: Opt<bool> = Opt::Some(true);
}
```

Struct generic types are often inferred

Calling generic functions

```rust
fn first<T,U>(x: T, y: U) -> T {
    x
}

fn main() {
    // automatically infers the type of T and U
    first(1, "hello");
    // Manually specify type of T and U
    first::<u32,&str>(1, "hello");
}
```

Funky syntax to manually specify function generics. Helps resolve a parsing ambiguity that's present in C++

How do `impl` and Generics Interact?

```
enum Opt<T> {
    Some(T),
    None
}

impl Opt<T> {
    fn unwrap(self) -> T {
        match self {
            Opt::Some(v) => v,
            Opt::None => panic!()
        }
    }

}
```

What's wrong with this?

```
error[E0412]: cannot find type `T` in this scope
  --> scratch.rs:337:10
   |
337 | impl Opt<T> {
   |          ^ not found in this scope
   |
```

How do `impl` and Generics Interact?

```
enum Opt<T> {
    Some(T),
    None
}

impl<T> Opt<T> {
    fn unwrap(self) -> T {
        match self {
            Opt::Some(v) => v,
            Opt::None => panic!()
        }
    }

}
```

Generic impl block

"For all T, there is an impl for the type `Opt<T>`"

Why would you ever not `impl<T>`?

```
impl Opt<u32> {
    fn unwrap(self) -> u32 {
        match self {
            Opt::Some(v) => v,
            Opt::None => panic!()
        }
    }
}
```

How do `impl` and Generics Interact?

```
enum Opt<T> {
    Some(T),
    None
}

impl<T> Opt<T> {
    fn swap<U>(&self, value: U) -> Opt<U> {
        ...
    }



}
```

Different levels of generics

- `T` is in scope for the entire `impl` block
- `U` is local to the function scope

More Advanced impls

```
impl<T> Opt<Opt<T>> {
    fn flatten(self) -> Opt<T> {
        match self {
            Opt::Some(Opt::Some(x)) => Opt::Some(x),
            Opt::Some(Opt::None) => Opt::None,
            Opt::None => Opt::None
        }
    }
}
```

Impl blocks can be for arbitrarily complex types, not just simple structs or enums

How are Generics Implemented?

```
enum Opt<T> {
    Some(T),
    None
}
fn main() {
    let x: Opt<bool> = Opt::Some(true);
}
```

Q: What is a generic type?

A: Instructions for how to generate code for a specific `Opt` like `Opt<u32>`. No code generated

Code generated only for case `Opt<bool>`. Exact same code as if you had written `BoolOpt` manually

- No runtime cost
- Some compile time cost
- Some binary size cost
- Same as C++
- “Monomorphization”

Comparing `Options`

How to write a function that takes two `Options` and

1. If both are `Some`, returns true if the first is greater than the second
2. Otherwise, returns `None`

Comparing Opts

```
fn opt_gr<T>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

```
error[E0369]: binary operation `>` cannot be applied to type `T`
--> scratch.rs:3:53
  |
3 |         (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
  |                                                   - ^ - T
  |                                                   |
  |                                                   T
```

Comparing Opts

```
fn opt_gr<T>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

HashMap HashMap ??? HashMap

```
error[E0369]: binary operation `>` cannot be applied to type `T`
--> scratch.rs:3:53
  |
3 |         (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
  |                                                   - ^ - T
  |                                                   |
  |                                                   T
```

Takeaway

`impl<T>` and `fn foo<T>` aren't very useful :(

You can't do much with T:

1. Return it
2. Wrap it in a struct/enum
3. Pass it to another function
4. Not much else

What might we want to do with T?

- Addition/subtraction
- Printing
- Copying
- Checking equality
- Dereference

Bringing Order

```
fn opt_gr<T: Ord>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

Can't call `opt_gr` with just any T

- T has to have an order

`Ord` is a "trait"

Traits

```
trait ToInt {
    fn to_int(&self) -> u32;
}

impl ToInt for f32 {
    fn to_int(&self) -> u32 {
        self.to_bits()
    }
}

fn add<T: ToInt>(a: T, b: T) -> u32 {
    a.to_int() + b.to_int()
}
```

Traits define a set of methods without implementations

Types can provide implementations to "implement" that trait

"`f32` implements `ToInt`"

Generic functions can restrict inputs to only types that implement a certain trait

Yes, they're similar to interfaces

Traits

```
fn foo<T: ToInt + Ord>(a: T, b: T) -> u32 {
    if a > b {
        a.to_int()
    } else {
        b.to_int()
    }
}
```

Without **types,** function **arguments** could be anything. **Types** restrict the domain of **arguments**

Without **traits,** function **generics** could be anything. **Traits** restrict the domain of **generics**

Common Trait Speedrun!

Traits are great for abstracting your own code by organizing shared behavior, but there's also many useful traits built in to the standard library that the language treats specially.

Let's explore some...

Common Trait Speedrun: `Default`

```
trait Default {
    fn default() -> Self;
}

fn main() {
    let x: String = Default::default();
}
```

`Default` is implemented by types that have a default value

- Requires a zero-argument function that returns the type.

What's this?

`Self` refers to the type that the current impl block is for

```
fn main() {
    let x: u32 = Default::default();
    let o: Option<String> =
Default::default();
}
```

Almost like overloading!

Common Trait Speedrun: `Clone`

```
trait Clone {
    fn clone(&self) -> Self;
}

fn main() {
    let s1 = String::new("Hello");
    let s2 = s1.clone()
}
```

`Clone` is implemented by types that can be deep-copied

Common Trait Speedrun: `Display`/`Debug`

```
trait Display {
    fn fmt(&self, f: &mut Formatter)
        -> Result<(), Error>;
}
trait Debug {
    fn fmt(&self, f: &mut Formatter)
        -> Result<(), Error>;
}
fn main() {
    let s = String::new("Test\n");
    // Uses Display
    println!("{}", s);
    // Uses Debug
    println!("{:?}", s);
}
```

`Display` and `Debug` define how a value can be converted to a string for printing

- `Display` for a user-facing representation
- `Debug` for a developer-facing representation

Instead of returning a string directly, work with a `Formatter` object to allow additional configuration (e.g. padding)

Common Trait Speedrun: `PartialEq`

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;




}
impl PartialEq for u32 { ... }

fn main() {
    let x: u32 = 1;
    let y: u32 = 2;
    x == y;
}
```

`PartialEq` is for types that can be compared for equality

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

Compiler uses `PartialEq` impl for `==` operator

Common Trait Speedrun: `PartialEq`

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;

    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
impl PartialEq for u32 { ... }

fn main() {
    let x: u32 = 1;
    let y: u32 = 2;
    x == y;
    x != y;
}
```

`PartialEq` is for types that can be compared for equality

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

What's this?

"Provided method"—implemented for free once you implement the required methods

Compiler uses `PartialEq` impl for `==` operator

Common Trait Speedrun: Eq

```
trait Eq: PartialEq { }

impl Eq for u32 {}
```

Eq is for equality relations

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

reflexive: a == a

What's this?

"Trait inheritance"—*a type can only be `Eq` if it is also `PartialEq`.*

What's this?

"Marker trait"—*note to the compiler that says this type has the properties associated with `Eq`*

Common Trait Speedrun: Copy

```
trait Copy: Clone { }

impl Copy for u32 {}

fn main() {
    let x: u32 = 1;
    let y = x;
    println!("{} and {}", x, y);
}
```

Copy is for types that can be trivially copied

Common Trait Speedrun: `ToString`

```
trait ToString {
    fn to_string(&self) -> String;
}
```

`ToString` types can be converted to a string

```
impl<T: fmt::Display> ToString for T {
  fn to_string(&self) -> String {
      ...
  }
}
```

What's this?

"Blanket Implementation"—*if T implements `Display` then T implements `ToString`*

Takeaways so far

We can use built-in operators on custom types by implementing certain built-in traits

Next up: using built-in arithmetic operators

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);

fn main() {
    let a = Vec3(1.0, 2.0, 3.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use * operator on custom Vec3 type for element-wise product.
Challenge: how to define Mul trait?

```
trait Mul {

    fn mul(self, other: &Self) -> Self;
}
impl Mul for Vec3 {

    fn mul(self, other: &Self) -> Self {
        Vec3(self.0 * other.0,
             self.1 * other.1,
             self.2 * other.2)
    }
}
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use * operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul {

    fn mul(self, other: &Self) -> Self;
}
impl Mul for Scalar {

    fn mul(self, other: &Vec3) -> Self {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

```
error[E0053]: method `mul` has an incompatible
type for trait
 --> scratch.rs:24:25
   |
24 |     fn mul(self, other: &Vec3) -> Self {
   |                         ^^^^^ expected
`Scalar`, found `Vec3`
   |
```

}

Goal: use * operator on `Scalar` and `Vec3`.

Challenge: how to define `Mul` trait?

```
trait Mul {
    fn mul(self, other: &Self) -> Self;
}
impl Mul for Scalar {
    fn mul(self, other: &Vec3) -> Self {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

⚠️Not generic enough!⚠️

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use * operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul<Rhs> {

    fn mul(self, other: &Rhs) -> Self;
}
impl Mul<Vec3> for Scalar {

    fn mul(self, other: &Vec3) -> Self {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

⚠️Not always true!⚠️

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

```
trait Mul<Rhs> {

    fn mul(self, other: &Rhs) -> Self;
}
impl Mul<Vec3> for Scalar {

    fn mul(self, other: &Vec3) -> Self {
* other.0,
* other.1,
* other.2)
```

```
error[E0308]: mismatched types
--> scratch.rs:3:9
  |
2 |       fn mul(self, other: &Vec3) -> Self {
  |                                     ---- expected `Scalar`
3 | /         Vec3(self   * other.0,
4 | |              self   * other.1,
5 | |              self   * other.2)
  | |_____________________________^ expected `Scalar`, found `Vec3`
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
```

"Associated Type"—type associated with a specific trait impl

```
}
```

Goal: use * operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul<Rhs> {
    type Output;
    fn mul(self, other: &Rhs) -> Output;
}

impl Mul<Vec3> for Scalar {
    type Output = Vec3;
    fn mul(self, other: &Vec3) -> Output {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

Final Implementation—fully generic

- Arbitrary left type, right type, and output type

```
trait Mul<Rhs> {
    type Output;
    fn mul(self, other: &Rhs) -> Output;
}
impl Mul<Vec3> for Scalar {
    type Output = Vec3;
    fn mul(self, other: &Vec3) -> Output {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Common Trait Speedrun: `From`

```
trait From<T> {
    fn from(value: T) -> Self;
}
```

`From` is for type to type conversions

```
impl From<A> for B {
    ...
}
```

If you see an impl like this, read it as “B can be made from A”

```
impl From<&String> for String {
    fn from(value: &String) -> String {
        ...
    }
}
```

An example implementation. How can we make a `String` given an `&String`?

Common Trait Speedrun: `Into`

```
trait Into<T> {
    fn into(self) -> T;
}

impl Into<String> for &String {
    fn into(self) -> String {
        ...
    }
}
// In standard library:
impl<T, U: From<T>> Into<U> for T {
    fn into(self) -> U {
        U::from(self)
    }
}
```

`Into` is the opposite of `From`

Can implement very similarly to `From`

- Seems sort of redundant...

Never need to implement `Into`, just implement `From` and `Into` will be implemented automatically.

Common Trait Speedrun: Using `Into`

```rust
trait Into<T> {
    fn into(self) -> T;
}
```

`Into` is the opposite of `From`

Sometimes type mismatches can be frustrating

```rust
fn write_to_file(data: Vec<u8>) {
    ...
}

fn main() {
    write_to_file("Hello");
}
```

```
error[E0308]: mismatched types
--> scratch.rs:6:19
  |
6 | write_to_file("Hello").into();
  | ------------- ^^^^^^^ expected `Vec<u8>`, found `&str`
  | |
  | arguments to this function are incorrect
```

Common Trait Speedrun: Using `Into`

```
trait Into<T> {
    fn into(self) -> T;
}
```

```
fn write_to_file(data: Vec<u8>) {
    ...
}

fn main() {
    write_to_file("Hello".into());
}
```

`Into` is the opposite of `From`

Sometimes type mismatches can be frustrating

Common Trait Speedrun: Fn

```
pub fn map<T, U, F: Fn(T) -> U>(list: &[T], f: F) -> &[U];
```

Fn: trait and special syntax for declaring function types

Common Trait Reference

`Default` types have a default value

`Clone` types can be deep copied

`Copy` types can be cloned by a bit-wise copy

`PartialEq (PartialOrd)` types can be compared with a partial equality (order) relation

`Eq (Ord)` types can be compared with an equality (total order) relation

`ToString` types can be converted to a string

`Debug` types can be converted to a developer-facing string representation

`Display` types can be converted to a user-facing string representation

`Add<T>`, `Mul<T>`, `Sub<T>`, `Div<T>` types can be summed/producted/differenced/divided with a value of type `T`

`From<T>` types can be created from a value of type `T`

`Into<T>` types can be converted to a value of type `T`

`Fn(T, U, ...) -> V` types can be called with the corresponding parameters and return type

Deriving Traits

```
#[derive(Debug)]
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{:?}", id);
}
```

Recall from last time... but now we understand!

`#[derive(...)]` syntax invokes a *macro*, a function that takes the code for your struct (or enum) as input and produces more code as output. In this case, a trait implementation.

Deriving Traits

```
#[derive(Debug, Clone, Copy,
PartialEq)]
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{:?}", id);
}
```

Recall from last time... but now we understand!

`#[derive(...)]` syntax invokes a *macro*, a function that takes the code for your struct (or enum) as input and produces more code as output. In this case, a trait implementation.

How are Traits Implemented?

```
fn write_to_file(data: Vec<u8>) {
    ...
}

fn main() {
    write_to_file("Hello".into());
}
```

Just like generics, no runtime cost

- compiler statically determines which function to call based on type inference

Dowsides of traits: hard to read docs?

Function std::fs::write 1.26.0 · source · [-]

```
pub fn write<P: AsRef<Path>, C: AsRef<[u8]>>(path: P, contents: C) -> Result<()>
```

Upside of traits: high level reasoning

```
pub fn sort_by_key<K, F>(&mut self, f: F)
where
    F: FnMut(&T) -> K,
    K: Ord,
```

Determine the types before writing the implementation

Overview of traits

Functions, structs, and enums can be generic.

To restrict the types that the generic can be instantiated with, use traits

Traits define a set of methods a type must implement

Trait Inheritance: some traits can only be implemented if another trait is implemented as well

Marker traits: traits with no methods

Provided methods: methods that are automatically defined in terms of other trait methods

Generic traits: makes a trait generic over a type

Associated types: types that implement this trait must also specify what the associated type is

Containers

Two bread-and-butter containers

`Vec<T>` ordered collection of values

- `ArrayList<T>` or `std::vector` or `list`

`HashMap<K, V>` unordered collection of key/value pairs

- `HashMap<K, V>` or `std::unordered_map<K, V>` or `dict`

What might be hard about these containers?

- Ownership?
- Mutable/immutable references?
- Trait implementations?

What traits does `Vec` satisfy

"Passes through" many traits

- `PartialEq/Eq`
- `PartialOrd/Ord`
- `Clone`
- `Debug`

Implements some new traits

- `Default`

Vec operations

Getting mutable/immutable references: easy!

```
let mut v = vec![1, 2, 3, 4, 5];
v.push(6);
v.push(7);
v.pop();
println!("{:?}", v);

let x: &u32 = v.get(0).unwrap();

let x: &mut u32 =
v.get_mut(0).unwrap();
*x = 10;
```

Why a macro?

Remove last element

Returns an `Option`

Vec operations

Moving is a “zero-cost” operation

```
let s1: String = String::from("hello");
let s2: String = s1;
// access to `s1` is invalidated by compiler
```

How do we move out of an index?

```
let v: Vec<String> = ...;

let first: String = vec.????(0);
// access to vec[0] is invalidate by compiler??
```

Vec operations

Moving is a “zero-cost” operation

```
let s1: String = String::from("hello");
let s2: String = s1;
// access to `s1` is invalidated by compiler
```

How do we move out of an index?

```
let v: Vec<String> = ...;

let first: String = vec.????(0);
// access to vec[0] is invalidate by compiler??
```

Options:

1. `v.get(0).unwrap().clone();`
2. `v.remove(0);`

Bottom line: can’t move out of vector without modifying data structure

HashMap

```
let mut map = HashMap::new();
map.insert(1905, String::from("Rust Programming"));
map.insert(3200, String::from("Introduction to Algorithms"));

let course_name = map.get(&1905).unwrap();
course_name.push('🦀');
```

Insertion and removal: by *move* (takes ownership)

Lookup: by *reference* (does not take ownership)

Does this program compile?

```
fn main() {
 let mut h = HashMap::new();
 h.insert("k1", 0);
 let v1 = &h["k1"];
 h.insert("k2", 1);
 let v2 = &h["k2"];
 println!("{} {}", v1, v2);
}
```

Recall

- Ownership
- Mutable/Immutable reference rules

Does this program compile?

```
fn main() {
 let mut h = HashMap::new();
 h.insert("k1", 0);
 let v1 = &h["k1"];
 h.insert("k2", 1);
 let v2 = &h["k2"];
 println!("{} {}", v1, v2);
}
```

Recall

- Ownership
- Mutable/Immutable reference rules

Mutate h while immutable reference exists

Looping over a Vec

As we've seen:

```
let v = vec![1, 2, 3];

for element in v {
    println!("{}", element);
}
```

But wait! Looping is complicated

Looping over a Vec

```
for element in v.iter() {
    // element: &T
}

for element in v.iter_mut() {
    // element: &mut T
}

for element in v.into_iter() {
    // element: T
}
```

Iterate by reference

- Read only element access

Iterate by mutable reference

- Read/write element access

Iterate by consumption

- Can't use v afterwards

Iterator adaptors

```
for (i, element) in v.iter().enumerate()
{
    println!("{}: {}", i, element);
}
```

Iterate over values *and indices*

```
for (x, y) in v1.iter().zip(v2.iter()) {
    println!("({}, {})", x, y);
}
```

Iterate over *two vectors simultaneously*

```
for x in v1.iter().chain(v2.iter()) {
    println!("{}", x);
}
```

Iterate over *two vectors sequentially*

Iterators are a trait

```
let s = String::new();
for char in s.chars() {
    // ...
}

let map = HashMap::new();
for (k, v) in map {
    // ...
}

for v in map.values() {
    // ...
}
```

Can loop over anything that implements `Iterator`

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Why care about this?

Performance:

- Spend lots of time running user code
- Run it quickly

Performance is only hard...

If we don't have this?

Safety:

None of

- null pointer deref
- use after free
- double free

if you have to maintain safety

How to make safety easy?

Imagine Rust but without references

- All values are owned
- Every value is
  a. returned from a function, OR
  b. freed at the end of the function

At compile time, know exactly where to insert calls to `malloc()` and `free()`

- impossible to have dangling references

```rust
fn make_list() -> Vec<i32> {
    vec![0, 1, 2, 3]
    // vec is not deallocated, it's returned
}

fn main() {
    let l = make_list();
    // l is deallocated here
}
```

References are what make safety hard!

How to make safety hard?

Now consider references

- Regardless of language!

```c
void main() {
    // "owned" string
    char* name = malloc(8);
    memcpy(name, "cis1905", 8);

    // reference into string
    char* number = name + 3;
}
```

```rust
fn main() {
    let s = String::from("hello");
    let as_ref = &s;
    println!("{}", as_ref);
}
```

```java
public static void main(String[] args) {
    // "Owned" list of strings
    List<String> stringList = new ArrayList<>();
    stringList.add("Hello");
    stringList.add("World");

    // Reference to element of list
    String = stringList.get(0);
}
```

How to make safety hard?

Now consider references

- Regardless of language!

```c
void main() {
    // "owned" string
    char* name = malloc(8);
    memcpy(name, "cis1905", 8);

    // reference into string
    char* number = name + 3;
}
```

What happens when a reference outlives the value it references?

- **Let the reference dangle**
- **Extend the life of the referenced value**

How to deal with reference outliving value?

Let the reference dangle

- Approach taken by C/C++
- Performance but no safety!

Extend the life of the referenced value

- Approach taken by Java/Python
- Safety but poor performance!
- (garbage collection)

Brief primer on garbage collection

Used in all languages that don't have `malloc/free`

Does `clients` get returned in the `db` or can it be freed at the end of setup? Impossible to know

How to know how long values live?

```
public static Database setup() {
    List<String> clients = new ArrayList<>{}
    clients.add("ClientA");
    clients.add("ClientB");

    List<Client> client_list = makeClients(clients);
    List<Orders> orders = makeOrders(client_list, orders);
    List<Invoice> invoices = makeInvoices(clients, orders);

    Database db = makeDatabase(orders, invoices, clients);

    return db;
}
```

Brief primer on garbage collection

Garbage collection:

1. Just put every value on the heap and don't worry about freeing it
2. When you get low on memory, walk through every alive variable to find values that are still reachable.
3. Free any value that isn't reachable

Brief primer on garbage collection

Garbage collection guarantees

- An in-use value will never be freed
- When a value is no longer accessible, it will *eventually* be freed

Developer never needs to free values 😊

Program periodically **stops** and all unused values are freed 😢

Modern garbage collectors are fast...

But manually managing your memory is (usually) faster

How to deal with reference outliving value?

Let the reference dangle

- Approach taken by C/C++
- Performance but no safety!

Extend the life of the referenced value

- Approach taken by Java/Python
- Safety but poor performance!
- (garbage collection)

Disallow compilation of program with dangling reference?

- All programs that compile are performant and safe
- But how?

Why can references dangle?

Where could the returned pointer point to?

- input argument `name`
- input argument `job`
- a local variable created during the function
- a global variable

Some of these make a dangling reference, some don't

- If the compiler is going to detect dangling references, it needs more information...

```
char* foo(char *name, char *job) {
    // implementation omitted
}
```

The same function but in Rust

Where could the returned pointer point to?

- input argument `name`
- input argument `job`
- a local variable created during the function
- a global variable

```
fn foo(name: &str, job: &str) -> &str {
    // implementation omitted
}
```

Need to tell compiler

```
error[E0106]: missing lifetime specifier
--> lecture.rs:1:34
  |
1 | fn foo(name: &str, job: &str) -> &str {
  |              ----      ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `name` or `job`
```

The same function but in Rust

Where could the returned pointer point to?

- input argument `name`
- input argument `job`
- a local variable created during the function
- a global variable

```
fn foo0<'a, 'b>(name: &'a str, job: &'b str) -> &'a str
```

```
fn foo1<'a, 'b>(name: &'a str, job: &'b str) -> &'b str
```

```
fn foo2(name: &str, job: &str) -> &'static str
```

Need to tell compiler

```
fn foo3<'a>(name: &'a str, job: &'a str) -> &'a str
```

Could be from `name` or `job`
(e.g. conditional)

The same function but in Rust

Where could the returned pointer point to?

- input argument `name`
- input argument `job`
- a local variable created during the function
- a global variable

Need to tell compiler

```
fn foo0<'a, 'b>(name: &'a str, job: &'b str) -> &'a str
fn foo1<'a, 'b>(name: &'a str, job: &'b str) -> &'b str
fn foo2(name: &str, job: &str) -> &'static str
fn foo3<'a>(name: &'a str, job: &'a str) -> &'a str
```

Types allow reasoning about how **functions** compose

Lifetimes allow reasoning about how **references** compose

15

How to read lifetimes

```
fn main() {
    let substring;
    if read_from_file {
        let s = read_to_string(file_path);
        substring = find(&s, "fn main");
    } else {
        substring = "no file provided";
    }
    println!("{}", substring)
}
```

```
// find substring `target` in `s`
fn find<'a, 'b>(s: &'a str, target: &'b str)
  -> &'a str
```

`find` takes

- a str `s` that lives for duration `'a`
- a str `target` that lives for duration `'b`

It returns a str that can live for up to duration `'a`

Is this program valid?
How do you know?

How to read lifetimes

```
fn main() {
    let substring;
    if read_from_file {
        let s = read_to_string(file_path);
        substring = find(&s, "fn main");
    } else {
        substring = "no file provided";
    }
    println!("{}", substring)
}
```

```
// find substring `target` in `s`
fn find<'a, 'b>(s: &'a str, target: &'b str)
  -> &'a str
```

`find` takes

- a str `s` that lives for duration `'a`
- a str `target` that lives for duration `'b`

```
error[E0597]: `s` does not live long enough
  |
4 |         let s = read_to_string(file_path);
  |             - binding `s` declared here
5 |         substring = find(&s, "fn main");
  |                          ^^ borrowed value does not live long enough
6 |     } else {
  |     - `s` dropped here while still borrowed
...
9 |     println!("{}", substring)
  |                    --------- borrow later used here
```

17

Appendix: find implementation

```
fn find<'a, 'b>(s: &'a str, target: &'b str) -> &'a str {
    for i in 0..s.len() {
        let snippet = &s[i..(i + target.len())];
        if snippet == target {
            return snippet;
        }
    }
    panic!("Not found");
}
```

Where do lifetimes come from?

```
fn main() {
    let substring;
    if read_from_file {
        let s = read_to_string(file_path);
        substring = find(&s, "fn main");
    } else {
        substring = "no file provided";
    }
    println!("{}", substring)
}
```

Lifetime:

- starts when a value can first be referred to*
- ends when a value can not be referred to*

Lifetimes are implicit

- Never explicitly declared by programmer

*variable lifetimes are complicated and usually you don't need to think too hard. As a rule of thumb, a variable's lifetime is equal to its scope.

Another lifetime example

```
fn main() {
    let s1 = String::from("foobar");
    let mut longest = s1.as_str();
    for i in 0..100 {
        let s2 = i.to_string();
        longest = longer(&s1, &s2);
    }
    println!("{}", longest);
}
```

```
fn longer<'a>(s1: &'a str, s2: &'a str)
  -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

Is it valid to call `longer` with `s1` and `s2` as arguments?

Another lifetime example

```
fn main() {
    let s1 = String::from("foobar");
    let mut longest = s1.as_str();
    for i in 0..100 {
        let s2 = i.to_string();
        longest = longer(&s1, &s2);
    }
    println!("{}", longest);
}
```

```
fn longer<'a>(s1: &'a str, s2: &'a str)
  -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

s1 lives *at least* duration ‘a
s2 lives *at least* duration ‘a
return value lives *at most* duration ‘a

One more lifetime example

```
const program_name: &str = "Theseus";

fn get_program_name() -> &str {
    program_name
}
```

Ok... but we don't have any lifetimes to use

```
error[E0106]: missing lifetime specifier
--> lecture.rs:3:26
  |
3 | fn get program name() -> &str {
  |                          ^ expected named lifetime parameter
```

One more lifetime example

```
const program_name: &str = "Theseus";

fn get_program_name() -> &'static str {
    program_name
}
```

Ok... but we don't have any lifetimes to use

`'static` lifetime: the lifetime of the entire program duration

Back to safety and performance

Safety goal: never have dangling references

Performance goal: avoid using garbage collection

Lifetime annotations enable the compiler to disallow programs that cause dangling references

- avoid garbage collection
- maintain safety

Advanced usage: lifetimes in structs

```
struct BookPage {
    number: u32,
    content: &str,
}
```

```
error[E0106]: missing lifetime specifier
--> lecture.rs:3:14
  |
3 |     content: &str,
  |              ^ expected named lifetime parameter
  |
```

Advanced usage: lifetimes in structs

Structs with references need to expose their lifetime parameter

```
struct BookPage<'a> {
    number: u32,
    content: &'a str,
}
```

```
fn later_page<'a>(p1: BookPage<'a>, p2: BookPage<'a>) -> BookPage<'a>
{
    if p1.number > p2.number {
        return p1;
    } else {
        return p2;
    }
}
```

References in structs are tricky: if you find yourself doing this make sure there isn't a better way

Final notes: lifetime elision

We've previously seen code like this. Why no lifetime annotations required?

```
fn prefix(s: &str) -> &str {
    &s[0..3]
}
```

In simple cases, the Rust compiler will infer lifetimes to make things easier

- If the return type is not a reference
- If the return type is a reference and only one input is a reference

Yes but... what if my code is too complicated?

What if I need mutable and immutable references at the same time?

What if I need to express reference logic but the compiler won't accept my lifetime annotations?

Rust does it's analysis at compile time when possible.
If you can't fit within those bounds, use built-in types that offload safety checks to run-time

Anonymous Functions/Closures

Another side to performance

Can we allow high level programming patterns while maintaining performance?

```
let rainfall nums =
 nums |>
 take_while (fun x -> x != -999) |>
 filter (fun x -> x >= 0) |>
 mean
```

As a case study: higher-order list functions vs. loops

Iterators and lambdas

```rust
fn rainfall(nums: Vec<i32>) -> Option<f64> {
    let valid_nums: Vec<i32> = nums
        .into_iter()
        .take_while(|&x| x != -999)
        .filter(|&x| x >= 0)
        .collect();
    mean(valid_nums);
}
```

```
let rainfall nums =
 nums |>
 take_while (fun x -> x != -999) |>
 filter (fun x -> x >= 0) |>
 mean
```

How do we translate this to Rust?

- need iterator functions
- need anonymous functions

Iterators and lambdas

```
fn rainfall(nums: Vec<i32>) -> Option<f64> {
    let valid_nums: Vec<i32> = nums
        .into_iter()
        .take_while(|&x| x != -999)
        .filter(|&x| x >= 0)
        .collect();
    mean(valid_nums);
}
```

Anonymous functions

Iterator functions

```
let rainfall nums =
 nums |>
 take_while (fun x -> x != -999) |>
 filter (fun x -> x >= 0) |>
 mean
```

How do we translate this to Rust?

- need iterator functions
- need anonymous functions

Iterators and lambdas

```
fn rainfall(nums: Vec<i32>) -> Option<f64> {
    let valid_nums: Vec<i32> = nums
        .into_iter()
        .take_while(|&x| x != -999)
        .filter(|&x| x >= 0)
        .collect();
    mean(valid_nums);
}
```

Anonymous functions

Iterator functions

Once you've called `iter`/`into_iter`/`iter_mut`, many iterator functions are available

- map
- filter
- fold
- take_while
- flat_map
- filter_map
- zip
- https://doc.rust-lang.org/std/iter/trait.Iterator.html

Anonymous Functions

Allows defining short-lived functions

- Type annotations optional
- Abbreviated syntax
- Used frequently in iterator functions

```
fn  add_one_v1    (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

Closures

More than just a function

- can access values that are in scope when they're defined
- function + environment

```
fn add_num(v: &mut Vec<i32>, value: i32) {
  let my_fn = |x| {*x += value};
   v.iter_mut().for_each(my_fn);
}
```

Tricky closures

Three different traits that govern functions

- `Fn` -> immutable access to environment
- `FnMut` -> mutable access to environment
- `FnOnce` -> moves values out of environment

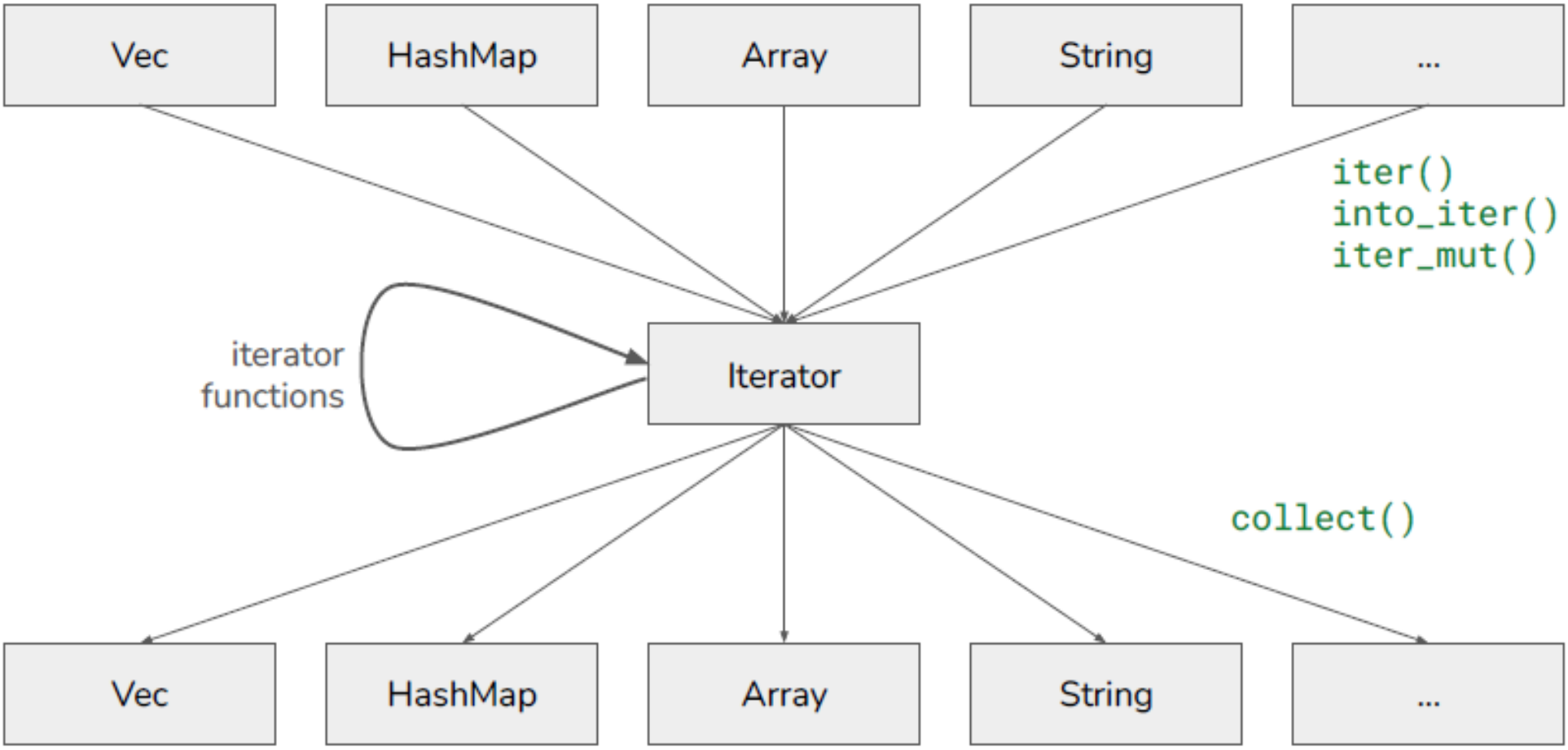
```
fn main() {
    let mut v = Vec::new();
    let impls_fn = || { println!("{}", v)
};

    let impls_fnmut = || { v.push(1)  }

    let impls_fnonce = || { take(v) };
}
```

Noticing a pattern? Behavior changes based on

- reference
- mutable reference
- owned value

Keeping these cases separate gives the Rust compiler enough info to check many things at compile time

Collect: turning iterators back to collections



Iterators and lambdas

```
fn rainfall(nums: Vec<i32>) -> Option<f64> {
    let valid_nums: Vec<i32> = nums
        .into_iter()
        .take_while(|&x| x != -999)
        .filter(|&x| x >= 0)
        .collect();
    mean(valid_nums);
}
```

Loops vs. Iterators: rainfall performance

```rust
fn iter(v: &Vec<i32>) -> f32 {
    let valid_nums: Vec<i32> = v
        .iter()
        .take_while(|&&x| x != -999)
        .cloned()
        .filter(|&x| x >= 0)
        .collect();
    if valid_nums.len() == 0 {
        0.0
    } else {
        valid_nums.iter().fold(0, |n, &a| n + a)
as f32 / valid_nums.len() as f32
    }
}
```

```rust
fn loops(v: &Vec<i32>) -> f32 {
    let mut valid_nums = Vec::new();
    for x in v {
        match x {
            -999 => break,
            &x if x >= 0 => valid_nums.push(x),
            _ => {}
        };
    }
    if valid_nums.len() == 0 {
        0.0
    } else {
        valid_nums.iter().fold(0, |n, &a| n + a)
as f32 / valid_nums.len() as f32
    }
}
```