

Lecture 06

Lifetimes, Closures

Today: safety and performance

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

```
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!

```
void main() {  
    // "owned" string  
    char* name = malloc(8);  
    memcpy(name, "cis1905", 8);  
  
    // reference into string  
    char* number = name + 3;  
}
```



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



```
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!



```
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

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

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

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

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** → `fn foo0<'a, 'b>(name: &'a str, job: &'b str) -> &'a str`
- input argument **job** → `fn foo1<'a, 'b>(name: &'a str, job: &'b str) -> &'b str`
- a local variable created during the function → `fn foo2(name: &str, job: &str) -> &'static str`
- a global variable → `fn foo2(name: &str, job: &str) -> &'static str`

Need to tell compiler

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

The same function but in Rust

Where could the returned pointer point to?

- input argument **name** → `fn foo0<'a, 'b>(name: &'a str, job: &'b str) -> &'a str`
- input argument **job** → `fn foo1<'a, 'b>(name: &'a str, job: &'b str) -> &'b str`
- a local variable created during the function → `fn foo2(name: &str, job: &str) -> &'static str`
- a global variable → `fn foo2(name: &str, job: &str) -> &'static str`

Need to tell compiler

- Could be from **name** or **job** (e.g. conditional) → `fn foo3<'a>(name: &'a str, job: &'a str) -> &'a str`

The same function but in Rust

Where could the returned pointer point to?

- input argument **name** → `fn foo0<'a, 'b>(name: &'a str, job: &'b str) -> &'a str`
- input argument **job** → `fn foo1<'a, 'b>(name: &'a str, job: &'b str) -> &'b str`
- a local variable created during the function → `fn foo2(name: &str, job: &str) -> &'static str`
- a global variable → `fn foo2(name: &str, job: &str) -> &'static str`

Need to tell compiler

```
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

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
```

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?

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

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

Quiz

```
struct Foo<'a> {  
    bar: &'a i32  
}  
fn baz(f: &Foo) -> &i32  
{  
    /* omitted */  
}
```

Will this compile? If so, what lifetime annotations will be inferred?

Quiz

```
struct Foo<'a> {  
    bar: &'a i32  
}  
fn baz<'a, 'b>(f: &'a Foo<'b>) -> &'??? i32  
{  
    /* omitted */  
}
```

Will this compile? If so, what lifetime annotations will be inferred?

Two separate lifetimes in the input

- can't infer output lifetime without ambiguity

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 its 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

Topic of next lecture

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

```
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>

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

What about these?

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);  
}
```


Quiz(?)

Does this code compile?

```
fn take(v: Vec<i32>) {}
```

```
fn main() {  
    let v = Vec::new();  
    let my_fun = || { take(v) };  
    my_fun();  
    my_fun();  
}
```

function



Quiz(?)

Does this code compile?

```
fn take(v: Vec<i32>) {}
```

```
fn main() {  
    let v = Vec::new();  
    let my_fun = || { take(v) };  
    my_fun();  
    my_fun();  
}
```

function



note: closure cannot be invoked more than once because it moves the variable `v` out of its environment

```
5 | let my_fun = || take(v);  
  |
```


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

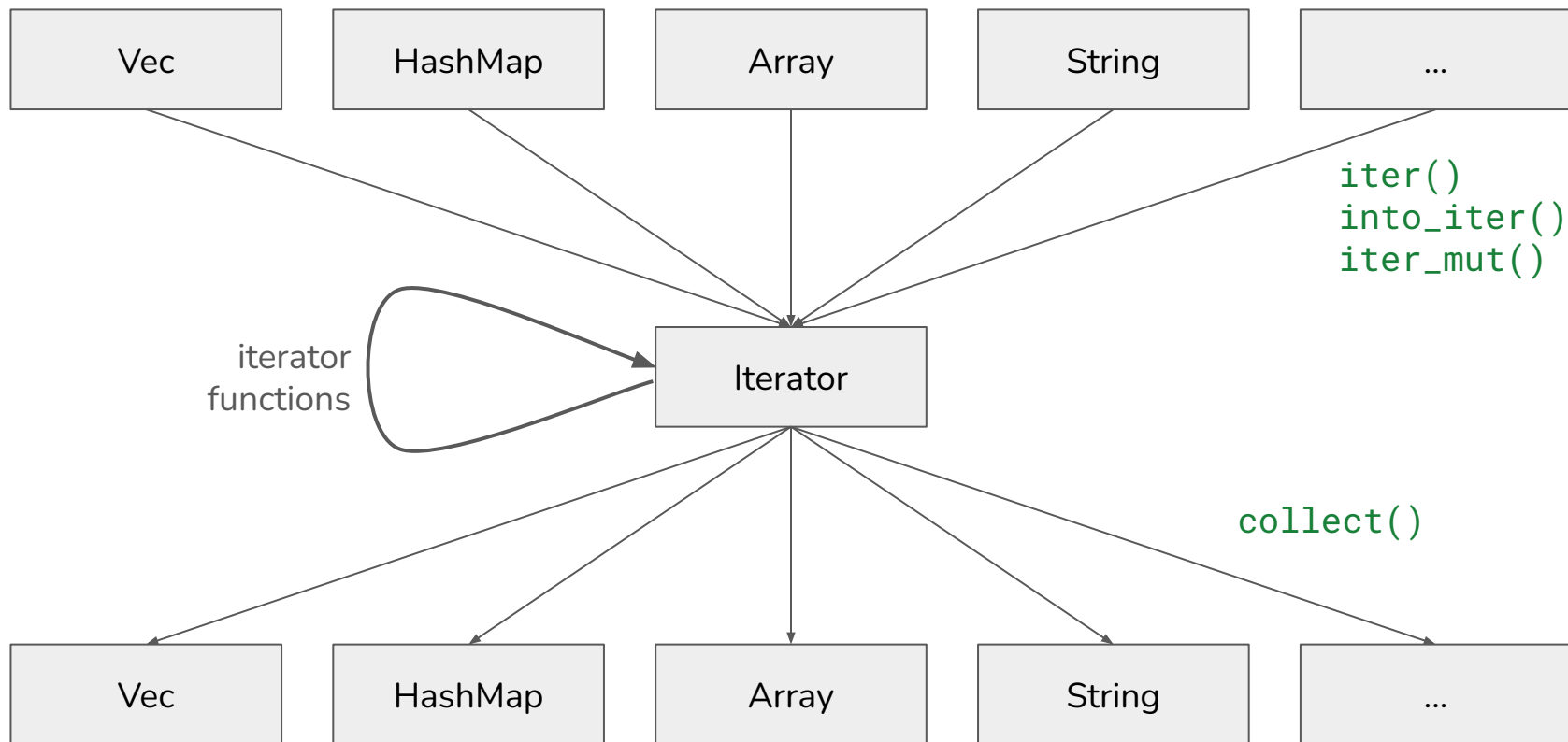
Iterators and lambdas

```
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);  
}
```

What's this?



Collect: turning iterators back to collections



Loops vs. Iterators: rainfall performance

```
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
}
```

```
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
}
```

Appendix: comparing generated code

Comparing generated code when using loops vs iterators

<https://godbolt.org/z/gePhnhzvY>