Ownership and Borrowing and Lifetimes (Oh My!)

EECS 3/496 “Rust”

Spring 2019
Definitions

An *object* is a chunk of memory with a type

Examples:

- The number 4 is a *value*, not an object
- A word of memory containing the number 4 is an object

A *variable* is the name of an object
Ownership

Every object in Rust has an owner. Either:

- a variable, or
- some other object
Ownership

Every object in Rust has an owner. Either:

- a variable, or
- some other object

Ownership comes with rights and responsibilities:

- The owner is allowed to modify the object
- The owner must destroy the object (or transfer it to another owner)
Transferring ownership

Ownership can be transferred:

```rust
pub fn inc_vec(mut v: Vec<usize>, ix: usize) {
    v[ix] += 1;
}
```
Transferring ownership

Ownership can be transferred:

```rust
pub fn inc_vec(mut v: Vec<usize>, ix: usize) {
    v[ix] += 1;
}

#[test]
fn test_inc_vec() {
    let expected = vec![3, 4, 6];
    let actual = vec![3, 4, 5];
    inc_vec(actual, 2);
    assert_eq!(expected, actual);
}
```
Transferring ownership

Ownership can be transferred:

```rust
pub fn inc_vec(mut v: Vec<usize>, ix: usize) {
    v[ix] += 1;
}
```

```rust
#[test]
fn test_inc_vec() {
    let expected = vec![3, 4, 6];
    let actual = vec![3, 4, 5];

    inc_vec(actual, 2);

    assert_eq!(expected, actual); // Error! actual has been moved
}
```
One solution: FP style

```rust
pub fn inc_vec(mut v: Vec<usize>, ix: usize) -> Vec<usize>
    v[ix] += 1;
    v
}

#[test]
fn test_inc_vec() {
    let expected = vec![3, 4, 6];
    let mut actual = vec![3, 4, 5];

    actual = inc_vec(actual, 2);
    assert_eq!(expected, actual);
}
```
The Rust solution: borrowing

```rust
pub fn inc_vec(v: &mut Vec<usize>, ix: usize) {
    v[ix] += 1;
}

#[test]
fn test_inc_vec() {
    let expected = vec![3, 4, 6];
    let mut actual = vec![3, 4, 5];

    inc_vec(&mut actual, 2);

    assert_eq!(expected, actual);
}
```
```rust
pub fn inc_vec(v: &mut [usize], ix: usize) {
    v[ix] += 1;
}

#[test]
fn test_inc_vec() {
    let expected = vec![3, 4, 6];
    let mut actual = vec![3, 4, 5];

    inc_vec(actual.as_mut_slice(), 2);

    assert_eq!(expected, actual);
}
```
## Owned versus borrowed

<table>
<thead>
<tr>
<th>have</th>
<th>want T &amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td>T &amp;T</td>
<td>depends free*</td>
</tr>
<tr>
<td></td>
<td>depends free</td>
</tr>
</tbody>
</table>
## Owned containers versus borrowed views

<table>
<thead>
<tr>
<th>have</th>
<th>T</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>&amp;T</td>
</tr>
<tr>
<td>have</td>
<td>&amp;T</td>
<td></td>
</tr>
<tr>
<td></td>
<td>depends</td>
<td>free*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>have</th>
<th>String</th>
<th>&amp;String</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;str</td>
<td></td>
<td>&amp;str</td>
</tr>
<tr>
<td>have</td>
<td>&amp;str</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;str</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Owned containers versus borrowed views

<table>
<thead>
<tr>
<th>have</th>
<th>T</th>
<th>want T &amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;T</td>
<td></td>
<td>depends</td>
</tr>
<tr>
<td></td>
<td></td>
<td>free*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>have</th>
<th>String</th>
<th>&amp;String</th>
<th>want &amp;str</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;str</td>
<td>$$$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;String</td>
<td>$$$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>$$$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Owned containers versus borrowed views

<table>
<thead>
<tr>
<th>have</th>
<th>T</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>depends free*</td>
</tr>
<tr>
<td>&amp;T</td>
<td></td>
<td>depends free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>have</th>
<th>String</th>
<th>&amp;String</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;str</td>
<td>&amp;str</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>$$ $$</td>
<td>free*</td>
<td>free*</td>
</tr>
<tr>
<td>&amp;String</td>
<td>$$ $$</td>
<td>free*</td>
<td>free*</td>
</tr>
<tr>
<td>&amp;str</td>
<td>$$ $$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Owned containers versus borrowed views

<table>
<thead>
<tr>
<th>have</th>
<th>T</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>depends</td>
<td>free*</td>
</tr>
<tr>
<td>&amp;T</td>
<td>depends</td>
<td>free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>have</th>
<th>String</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>$$$</td>
<td>free*</td>
</tr>
<tr>
<td>&amp;String</td>
<td>$$$</td>
<td>free</td>
</tr>
<tr>
<td>&amp;str</td>
<td>$$$</td>
<td>free</td>
</tr>
</tbody>
</table>
# Owned containers versus borrowed views

<table>
<thead>
<tr>
<th>have</th>
<th>T</th>
<th>&amp;T</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>depends</td>
<td>free*</td>
<td></td>
</tr>
<tr>
<td>&amp;T</td>
<td>depends</td>
<td>free</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>have</th>
<th>String</th>
<th>&amp;String</th>
<th>want</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>$$$</td>
<td>free*</td>
<td>free*</td>
</tr>
<tr>
<td>&amp;String</td>
<td>$$$</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>&amp;str</td>
<td>$$$</td>
<td>$$$**</td>
<td>free</td>
</tr>
</tbody>
</table>
# Owned containers versus borrowed views

<table>
<thead>
<tr>
<th>have</th>
<th>have T &amp; T</th>
<th>want T &amp; T</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>depends</td>
<td>free*</td>
</tr>
<tr>
<td>&amp;T</td>
<td>depends</td>
<td>free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>have</th>
<th>have Vec&lt;T&gt; &amp;Vec&lt;T&gt; &amp;[T]</th>
<th>want Vec&lt;T&gt; &amp;Vec&lt;T&gt; &amp;[T]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vec&lt;T&gt; &amp;[T]</td>
<td>$$$ free* free*</td>
<td>$$$ free free</td>
</tr>
<tr>
<td>&amp;Vec&lt;T&gt; &amp;[T]</td>
<td>$$$ free free</td>
<td>$$$ free free</td>
</tr>
<tr>
<td>&amp;[T]</td>
<td>$$$ $$$** free</td>
<td>$$$ free free</td>
</tr>
</tbody>
</table>
Borrowing implements reader/writer semantics

You can borrow

- as many immutable references as you like, or
- one mutable reference.

```rust
let mut x = SomeObject::new();
{
    let r1 = &x;
    let r2 = &x;
    let r3 = r1;
    let r4 = &mut x;  // error!
}
{
    let r5 = &mut x;  // ok
    let r6 = &x;     // error!
}
```
Hidden borrows

Methods calls may (mutable) borrow `self`:

```rust
impl SomeObject {
    pub fn f(&mut self) {
    }
}

let x = SomeObject::new();
x.f(); // error: x isn’t mutable
```
When borrowing won’t do

- The **Copy** trait for cheap copies
- The **Clone** trait for expensive copies
The Copy trait

Types implementing the `Copy` trait are copied implicitly rather than moved:

- `usize` and other built-in numeric types
- `&str` and other immutable reference types
- In general, types that
  - are cheap to copy (small), and
  - don’t own a resource (e.g., heap allocations)

```rust
let a = 5;
let b = a;
f(a);
let c = a + b;
```
The Clone trait supports explicitly copying:

- String, Vec, HashMap, etc.
- In general, types that may be expensive to copy, and don’t involve a unique resource (e.g., a file handle)

```rust
let v = vec![3, 4, 5];
let u = v.clone();
f(v);
g(u);
```
Lifetimes

Object have lifetimes (or more precisely, death times)

```rust
{
    let mut r: &str;

    {
        let s = "hello".to_owned();

        r = &s; // error because r outlives s
    }
    // s dies here

    println!("{{}}", r);
}
    // r dies here
```
Lifetimes

Object have lifetimes (or more precisely, death times)

```rust
{
    let mut r: &str;
    {
        let s = "hello".to_owned();
        r = &s;       // error because r outlives s
    } // s dies here

    println!("{}", r);
} // r dies here
```

A reference must die before its referent!
The static lifetime

The only named lifetime is 'static—the lifetime of the whole program

String slice literals have static lifetime. That is,

```rust
let s: &str = "hello";
```

means

```rust
let s: &'static str = "hello";
```
Lifetime variables

All other lifetimes are relative:

```rust
fn choose<'a>(x: &'a usize, y: &'a usize) -> &'a usize
```

Why does `&'a 0` work? How does that have lifetime `'a`?

Subtyping: `&'static T` < `&'a T`.
Lifetime variables

All other lifetimes are relative:

```rust
fn choose<'a>(x: &'a usize, y: &'a usize) -> &'a usize {
    if is_even(*x) {x}
    else if is_even(*y) {y}
    else {&0}
}
```

Why does `&0` work? How does that have lifetime `'a`?

Subtyping: `&'static T` <: `&'a T`. 

Lifetime variables

All other lifetimes are relative:

```rust
fn choose<'a>(x: &'a usize, y: &'a usize) -> &'a usize {
    if is_even(*x) {x}
    else if is_even(*y) {y}
    else {&0}
}
```

Why does &0 work? How does that have lifetime 'a?
Lifetime variables

All other lifetimes are relative:

```rust
fn choose<'a>(x: &'a usize, y: &'a usize) -> &'a usize {
    if is_even(*x) { x }
    else if is_even(*y) { y }
    else { &0 }
}
```

Why does &0 work? How does that have lifetime 'a?

Subtyping: &'static T <: &'a T.
Be careful, because it’s fragile

```rust
def ref_even(n: &usize) -> &usize {
    if is_even(*n) { n }
    else {
        let zero = 0;
        &zero
    }
}

def ref_even(n: &usize) -> &usize {
    if is_even(*n) { n }
    else {
        let zero = &0;
        zero
    }
}
```