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:

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

Transferring ownership

Ownership can be transferred:

```
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:

```
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); // Error! actual has been moved
}
```

One solution: FP style

```
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

```
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);
}
```

More idiomatic Rust: take a slice

```
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

		want T	want &T
have	T	depends	free*
	&T	depends	free

Owned containers versus borrowed views

		want T	want &T
have	T	depends	free*
	&T	depends	free

		want String	want &String	want &str
have	String			
	&String			
	&str			

Owned containers versus borrowed views

		want T	want &T
have	T	depends	free*
	&T	depends	free

		want String	want &String	want &str
have	String	$$$		
	&String	$$$		
	&str	$$$		

Owned containers versus borrowed views

		want T	want &T
have	T	depends	free*
	&T	depends	free

		want String	want &String	want &str
have	String	$$$	free*	free*
	&String	$$$		
	&str	$$$		

Owned containers versus borrowed views

		want T	want &T
have	T	depends	free*
	&T	depends	free

		want String	want &String	want &str
have	String	$$$	free*	free*
	&String	$$$	free	
	&str	$$$		free

Owned containers versus borrowed views

		want T	want &T
have	T	depends	free*
	&T	depends	free

		want String	want &String	want &str
have	String	$$$	free*	free*
	&String	$$$	free	free
	&str	$$$	$$$**	free

Owned containers versus borrowed views

		want T	want &T
have	T	depends	free*
	&T	depends	free

		want Vec<T>	want &Vec<T>	want &[T]
have	Vec<T>	$$$	free*	free*
	&Vec<T>	$$$	free	free
	&[T]	$$$	$$$**	free

Borrowing implements reader/writer semantics

You can borrow

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

```
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`:

```
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*)

```
let a = 5;
let b = a;
f(a);
let c = a + b;
```

The Clone trait

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)

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

Lifetimes

Object have lifetimes (or more precisely, death times)

```
{
    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)

```
{
    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,

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

means

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

Lifetime variables

All other lifetimes are relative:

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

Lifetime variables

All other lifetimes are relative:

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

Lifetime variables

All other lifetimes are relative:

```
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:

```
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

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

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