Ownership

Schedule

	M	W
Sept 14	Intro/Background	Basics/ideas
Sept 21	Allocation/layout	GGGGC
Sept 28	Mark/Sweep	Copying GC
Octo 5	Details	Ref C
Octo 12	Thanksgiving	Mark/Compact
Octo 19	Partitioning/Gen	Generational
Octo 26	Other part	Runtime
Nove 2	Final/weak	Conservative
Nove 9	Ownership	Adv topics
Nove 16	Adv topics	Adv topics
Nove 23	Presentations	Presentations
Nove 30	Presentations	Presentations

Automatic memory management

- Any form of memory management that does not require an explicit `free` is automatic
- Garbage collection does it with no foreknowledge of object lifetimes
- Ownership types put object lifetime in the language itself

Ownership types

- Ownership types are used in many contexts, for many things
- We only care about using them for memory management
- Note: This will be our most compiler lecture

Rust

- Rust is the flag-bearer for ownership memory management
- Ownership is mostly implicit in Rust
- As such, I'll start by showing ownership in pseudo-Java

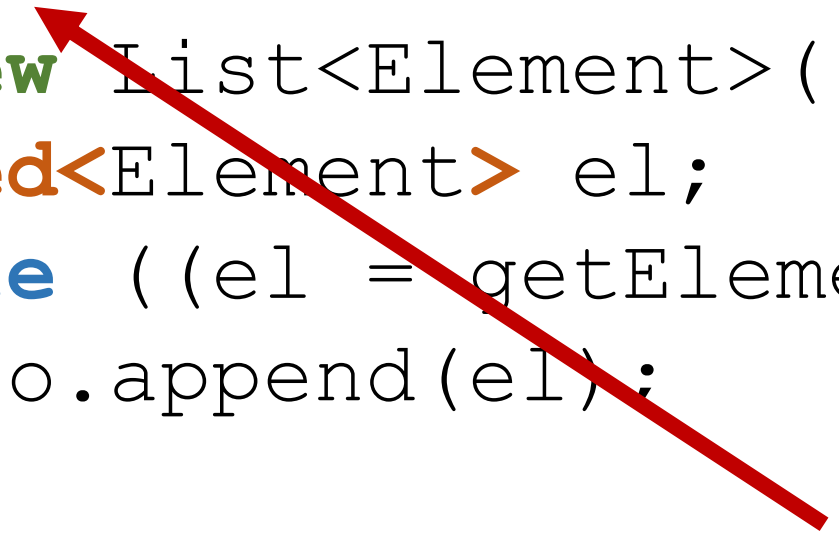
```
void handleElements() {
  List<Element> foo = new List<Element>();
  Element el;
  while ((el = getElement()) != null) {
    foo.append(el);
  }
  for (el : foo) {
    handle(el);
  }
}
```

```
void handleElements() {
  List<Element> foo = new List<Element>();
  Element el;
  while ((el = getElement()) != null) {
    foo.append(el);
  }
  for (el : foo) {
    handle(el);
  }
}
```

Programmer knows that `foo` and all its elements are dead here, but GC does not

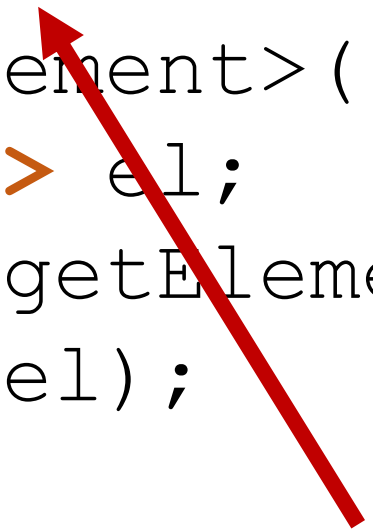
```
void handleElements() {
  Owned<List<Owned<Element>>> foo =
    new List<Element>();
  Owned<Element> el;
  while ((el = getElement()) != null) {
    foo.append(el);
  }
  for (el : foo) {
    handle(el);
  }
}
```

```
void handleElements() {
  Owned<List<Owned<Element>>> foo =
    new List<Element>();
  Owned<Element> el;
  while ((el = getElement()) != null) {
    foo.append(el);
  }
  for (el : foo) {
    handle(el);
  }
}
```



This reference owns this `List`. When this reference goes away, the `List` should be deleted.

```
void handleElements() {
  Owned<List<Owned<Element>>> foo =
    new List<Element>();
  Owned<Element> el;
  while ((el = getElement()) != null) {
    foo.append(el);
  }
  for (el : foo) {
    handle(el);
  }
}
```



This `List` owns its elements. When the `List` is deleted, its elements should be deleted as well.

Ownership

- Compiler knows lifetime of root references: Local and global variables
- Ownership ties lifetime of objects to lifetime of their references
- When an owning *reference* dies, its owned *object* dies

```
void handleElements() {
  Owned<List<Owned<Element>>> foo =
    new List<Element>();
  Owned<Element> el;
  while ((el = getElement()) != null) {
    foo.append(el);
  }
  for (el : foo) {
    handle(el);
  }
  // Compiler-generated:
  for (Element el : foo) delete el;
  delete foo;
}
```

Owned object deletion

- Deleting an object is only safe when its last reference dies
- Thus, an object may have only one owner
- This is also assured by the language by *ownership transfer*

```
void handleElements() {
  Owned<List<Owned<Element>>> foo =
    new List<Element>();
  Owned<Element> el;
  while ((el = getElement()) != null) {
    foo.append(el);
    print(el.value);
  }
  for (el : foo) {
    handle(el);
  }
}
```

```
void handleElements() {
  Owned<List<Owned<Element>>> foo =
    new List<Element>();
  Owned<Element> el;
  while ((el = getElement()) != null) {
    foo.append(el);
    print(el.value);
  }
  for (el : foo) {
    handle(el);
  }
}
```

This line will not compile. `el`'s ownership has been transferred away, so it is no longer a valid reference.

Ownership transfer

- Ownership is transferred by assignment, including function arguments
- From-reference becomes *invalid*
- From-reference can only become valid again by transferring a new object
- A reference is invalid at a program point if *any* path to that point leaves it invalid

```
void handler() {
  Owned<Element> el = new Element();
  if (el.mustUpdate) {
    update(el);
  }
  print(el.value);
}
```

```
void handler() {
  Owned<Element> el = new Element();
  if (el.mustUpdate) {
    update(el);
  }
  print(el.value);
}
```

`update` takes ownership of `el`

```
void handler() {
  Owned<Element> el = new Element();
  if (el.mustUpdate) {
    update(el);
  }
  print(el.value);
}
```

`update` takes ownership of `el`

This line is invalid, as `el` might have been transferred away.

Ownership destruction

- If an object *x* is transferred into a reference's ownership,
- and that reference already owned an object *y*,
- then object *y* must be destroyed.

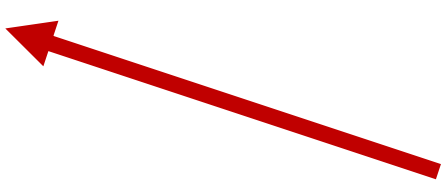
Borrowing

- Having only one reference is impractical
- Need a way to have a reference to an object without owning it
- This concept is *borrowing*
- Borrowing must assure that owned object deletion still works!

```
void handler() {
  Owned<Element> el = new Element();
  if (el.mustUpdate)
    update(el);
  print(el.value);
}

void update(Borrowed<Element> el) {
  el.value++;
  el.mustUpdate = false;
}
```

```
void handler() {
  Owned<Element> el = new Element();
  if (el.mustUpdate)
    update(el);
  print(el.value);
}

static Borrowed<Element> lastUpdated;

void update(Borrowed<Element> el) {
  lastUpdated = el;
  el.value++;
  el.mustUpdate = false;
}
```

```
void handler() {
  Owned<Element> el = new Element();
  if (el.mustUpdate)
    update(el);
  print(el.value);
}

static Borrowed<Element> lastUpdated;

void update(Borrowed<Element> el) {
  lastUpdated = el;
  el.value++;
  el.mustUpdate = false;
}
```



lastUpdated might live longer than whatever we borrowed el from, so this is illegal.

Lifetimes

- Concept of borrowing is based on lifetime
- Lifetime is dynamic, but with static relationships: Some reference cannot outlive others
- For instance, arguments of callee function cannot outlive variables of caller[1]

[1]Ignoring closures, which complicate but preclude ownership types.

Lifetimes

Object owned by reference *x*

Object owned by reference *x.y*

Borrowed reference to *x.y*

Object owned by reference *x.y* (second)

Borrowed reference to *x.y*

Compiler must guarantee:

- If *a* owns *b*, *a* lives longer than *b*
- If *b* borrows *a*, *a* lives longer than *b*

Lifetimes

Object owned by reference *x*

Object owned by reference *x.y*

Borrowed reference to *x.y*

Object owned by reference *x.y* (second)

This case is invalid. Compiler must assure that x.y cannot be changed while it is borrowed.

This particularly complicates concurrency.

Lifetimes

- Borrowing is legal if borrower cannot outlive borrowee
- This must be guaranteed statically
  - For local variables: Owning reference immutable during borrow
  - For object fields: Can only borrow a reference if we borrow or own its owner

```
class ElementWrapper {
  Owned<Element> el;
}

void handler(Borrowed<ElementWrapper> ew) {
  Borrowed<Element> el = ew.el;
  update(el);
  updateWithWrapper(ew, el);
}

…
```

```
class ElementWrapper {
  Owned<Element> el;
}

void handler(Borrowed<ElementWrapper> ew) {
  Borrowed<Element> el = ew.el;
  update(el);
  updateWithWrapper(ew, el);
}

…
```

Owning reference is immutable because owner (caller) is paused while `handler` executes.

```
class ElementWrapper {
  Owned<Element> el;
}

void handler(Borrowed<ElementWrapper> ew) {
  Borrowed<Element> el = ew.el;
  update(el);
  updateWithWrapper(ew, el);
}

…
```

Owning reference is immutable because owner (caller) is paused while `handler` executes.

This is fine: `el` is guaranteed a shorter lifetime than `ew`

```
class ElementWrapper {
  Owned<Element> el;
}

void handler(Borrowed<ElementWrapper> ew) {
  Borrowed<Element> el = ew.el;
  update(el);
  updateWithWrapper(ew, el);
}
```

Owning reference is immutable because owner (caller) is paused while `handler` executes.

This is fine: `el` is guaranteed a shorter lifetime than `ew`

… This argument invalidates `el`: `updateWithWrapper` could change `ew.el`, destroying our lifetime guarantee

```
class ElementWrapper {
  Owned<Element> el;
}

void handler(Borrowed<ElementWrapper> ew) {
  Borrowed<Element> el = ew.el;
  update(el);
  updateWithWrapper(ew, el);
}
```

Owning reference is immutable because owner (caller) is paused while `handler` executes.

This is fine: `el` is guaranteed a shorter lifetime than `ew`

… This argument invalidates `el`: `updateWithWrapper` could change `ew.el`, destroying our lifetime guarantee

Note: Rust has immutable references and a way of explicitly specifying lifetime relationships. With these, it is possible to work around these situations.

Ownership and mutability

- A mutable borrowed reference complicates things
- If I borrow *y* from *x*, then someone else borrows *x*, my reference *y* becomes invalid
- Avoidable with immutable references
- Rust avoids with explicit lifetime relationships

Rust

- Rust is the current flagship language with ownership for memory management
- ScopeJ introduced the concept academically
- Now we'll go into Rust syntax

```rust
fn foo() {
    let v = vec![1, 2, 3];
}
```

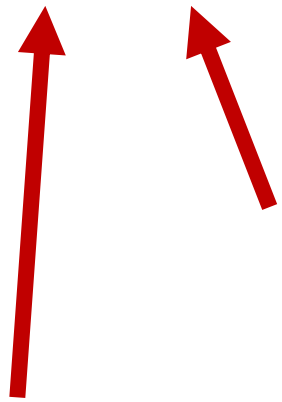
Examples taken from the Rust Book

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

Rust references are owners by default

Examples taken from the Rust Book

```
fn foo() {
    let v = vec![1, 2, 3];
}
```



Rust references are owners by default

Rust references are immutable by default.
In this case, nothing can ever mutate this vector: It has no mutable references and it is impossible to cast an immutable reference to a mutable one.

Examples taken from the Rust Book

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

let v2 = v;

println!("v[0] is: {}", v[0]);
```

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

Examples taken from the Rust Book

```
fn take(v: Vec<i32>) {
    // what happens here isn't important.
}

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

take(v);

println!("v[0] is: {}", v[0]);
```

```
 error: use of moved value: `v`
 println!("v[0] is: {}", v[0]);
                          ^
```

Examples taken from the Rust Book

```rust
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2
    // we cannot assign v1 to a global variable

    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// we can use v1 and v2 here!
```

Examples taken from the Rust Book

Rust and mutability

- A Rust object may:
  - Have zero or more immutable borrows (&T), *xor*
  - Have exactly one mutable borrow (&mut T)
- This prevents most forms of borrow mutability errors

```
let mut x = 5;
let y = &mut x;

*y += 1;

println!("{}", x);
```

```
 error: cannot borrow `x` as immutable because
 it is also borrowed as mutable
     println!("{}", x);
                    ^
```

Examples taken from the Rust Book

```
let mut x = 5;
let y = &mut x;

*y += 1;

println!("{}", x);
```

Remember: Assigning to a function argument is either ownership transfer or borrowing!

```
error: cannot borrow `x` as immutable because
it is also borrowed as mutable
    println!("{}", x);
                   ^
```

Examples taken from the Rust Book

Opinion corner

Personally, I suspect that the complicated rules about ownership, borrowing and mutability will effectively kill Rust.

… and we haven’t even gotten to the complicated part yet.

Borrowing and lifetimes

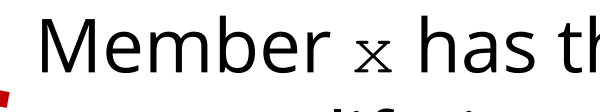
- Usually, implicit lifetimes from function scopes is enough
- Sometimes, explicitly specifying lifetimes is valuable

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
   if (condition) {
      x
   } else {
      y
   }
}
```

Illegal! We promised a return with the same lifetime as `x`

Examples taken from the Rust Book

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = 5;
    let f = Foo { x: y };

    println!("{}", f.x);
}
```



Member `x` has the same (or greater) lifetime as the structure, but is not owned by it.

Allowed: `y` and `f` have the same lifetime.

Examples taken from the Rust Book

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                          // -+ x goes into scope
                                    //  |
    {                               //  |
        let y = 5;                  // ---+ y goes into scope
        let f = Foo { x: y };       // ---+ f goes into scope
        x = &f.x;                   //  | | error here
    }                               // ---+ f and y go out of scope
                                    //  |
    println!("{}", x);              //  |
}                                   // -+ x goes out of scope
```

Examples taken from the Rust Book

More complicated ownership

- Sometimes, ownership is easy to define
- Sometimes, not so (e.g. circular list)
- Ownership types for memory management are still under development
- Current answer: Use GC and hope they cover your case later

Summary

- Automatic memory management need not mean garbage collection
- Ownership types can essentially generate delete statements
- Language complication, compiler smarts, but no runtime cost