Introduction to Rust: Ownership and Lifetimes

A brief background to the Rust project (~ 1 minute)

- Designed as a research language by Mozilla
- Used to implement their new experimental browser, Servo
- Designed to be appropriate for any applications for which C++ to be appropriate.
- It has been described as an *opinionated* language.
- List of supported features (from `www.rust-lang.org`):
    - zero-cost abstractions
    - move semantics
    - guaranteed memory safety
    - threads without data races
    - trait-based generics
    - pattern matching
    - type inference
    - minimal runtime
    - efficient C bindings

State Memory Model Guarantees (~ 0.5 minutes)

One of its main goals is safe use of pointers and safe (mostly) manual memory management. (No garbage collector in core language.)

RAII in C++ (~ 1 minute)

To understand how rust gives these guarantees, we start with a review of C++ RAII stack-based memory management. (Give a brief explanation about implicit heap allocation/deallocation within a C++ scope.)

```
#include <vector>

int main() {
  std::vector<int> vec (3, 100);  // Stack-allocated data structure points to
                                  // three heap-allocated ints, each set to
                                  // 100.
}  // This is the end of `vec`'s "lifetime".
```

The object's destructor is called at the end of its lifetime. Any associated memory is freed in the process. If the data structure is recursively defined, then those links should be freed as well.

If we stick to this discipline, we can greatly mitigate potential errors. However, Rust is *opinionated*, so it tries to enforce something like RAII, but more strongly.

Motivating Problem (~ 0.5 Minutes)

```
#include <vector>
#include <foobar>

int main() {
  std::vector<int> vec (3, 100);
  foo(vec);  // What might `foo()` do with the memory which `vec` encapsulates?
}
```

Rust Does Something Like Stack Discipline using Lifetimes (~ 0.5 minutes)

```
fn main() {
    let vec = vec![100, 100, 100];  // Stack-allocated data structure points to
                                    // three heap-allocated ints, each set to
                                    // 100.
}  // This is the end of `vec`'s lifetime.
```

The variable is dropped at the end of its lifetime. Any associated memory is freed in the process. If the data structure is recursively defined, then those links are freed as well.

Rust Makes Borrowing Explicit (~ 2 minutes)

```
fn foo_borrow(vec: &Vector<u32>)) {  // Notice the &.
    // ...
}

fn main() {
    let vec = vec![100, 100, 100];  // Stack-allocated data structure points to
                                    // three heap-allocated ints, each set to
                                    // 100.
    foo(vec);
}
```

Because of the type signature of `foo_borrow()`, the function body is guaranteed to not stash a reference any memory encapsulated by `vec`. The callee is guaranteed to give back all ownership of the object to the caller.

Rust Makes Moves Explicit (~ 2 minutes)

```
fn foo_move(vec: Vector<u32>)) {    // Notice the lack of an &.
    // ...
}

fn main() {
    let vec = vec![100, 100, 100];  // Stack-allocated data structure points to
                                    // three heap-allocated ints, each set to
                                    // 100.
    foo_move(vec);
}
```

Because of the type signature of `foo_move()`, ownership of the object is given over to the callee. Once `main()` passes `vec` to `foo_move()`, `main()` can no longer use that object because it no longer owns it. The ownership of the object has moved.

Rust Makes Lifetimes Explicit, but Usually Implied (~ 2 minutes)

```
struct Pair {
    x: u32,
    y: u32
}

// This wouldn't compile without lifetime elision:
fn fst(pair: &Pair) -> &u32 {
    &(pair.x)
}
```

Lifetime inference is happening here. If we were to write it all out explicitly, then `fst` would be defined with lifetime parameters:

```
fn fst<'a>(pair: &'a Pair) -> &'a u32 {
    &(pair.x)
}
```

This tells the complier that the reference to which the return value will point needs to have the same lifetime as the argument's lifetime.

Explicit Ownership and Lifetimes with Threads (~ 0.5 minutes)

I haven't worked-out how to present this connection yet.