Rust: Systems Programming for Everyone

Leo Testard, Mozilla
Why Rust...?
Why use Rust?

Fast code, low memory footprint

Go from bare metal (assembly; C FFI) ...

... to high-level (collections, closures, generic containers) ...

with zero cost (no GC, unboxed closures, monomorphization of generics)

Safety and Parallelism
Safety and Parallelism

Safety

No segmentation faults
No undefined behavior
No data races
Why would Mozilla sponsor Rust?

Hard to prototype research-y browser changes atop C++ code base

Rust → Servo, WebRender

Want Rust for next-gen infrastructure (services, IoT)
Where is Rust now?

1.0 release was back in May 2015

Rolling release cycle (up to Rust 1.8 as of May 2nd 2016)

Open source from the begining
https://github.com/rust-lang/rust/

Open model for future change (RFC process)
https://github.com/rust-lang/rfcs/

Awesome developer community (~1,000 people in #rust, ~250 people in #rust-internals, ~1,300 unique committers to rust.git)
Talk plan

"Why Rust"

How we build safe abstractions in Rust:

ownership & borrowing

Example 1: Pointers and allocation

Example 2: Concurrency
Ownership: a metaphor
"Ownership is intuitive"

Let's buy a car

```javascript
let money: Money = bank.withdraw_cash();
let my_new_car: Car = dealership.buy_car(money);

let second_car = dealership.buy_car(money); // <-- cannot reuse
```

money transferred into dealership, and car transferred to us.
"Ownership is intuitive"

Let's buy a car

```javascript
let money: Money = bank.withdraw_cash();
let my_new_car: Car = dealership.buy_car(money);
// let second_car = dealership.buy_car(money); // <-- cannot reuse
```

money transferred into `dealership`, and car transferred to us.

```javascript
my_new_car.drive_to(home);
garage.park(my_new_car);
```

```javascript
my_new_car.drive_to(...); // now doesn't work
```

(can't drive car without access to it, e.g. taking it out of the garage)
"Ownership is intuitive"

Let's buy a car

```javascript
let money: Money = bank.withdraw_cash();
let my_new_car: Car = dealership.buy_car(money);
// let second_car = dealership.buy_car(money); // <-- cannot reuse
```

money transferred into **dealership**, and car transferred to us.

```javascript
my_new_car.drive_to(home);
garage.park(my_new_car);
// my_new_car.drive_to(...) // now doesn't work
```

(can't drive car without access to it, e.g. taking it out of the garage)

```javascript
let my_car = garage.unpark();
my_car.drive_to(work);
```

...reflection time...
Ownership is important

<table>
<thead>
<tr>
<th>Ownership enables:</th>
<th>which removes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RsAll-style destructors</td>
<td>a source of memory leaks (or fd leaks, etc)</td>
</tr>
<tr>
<td>no dangling pointers</td>
<td>many resource management bugs</td>
</tr>
<tr>
<td>no data races</td>
<td>many multithreading heisenbugs</td>
</tr>
</tbody>
</table>

Do I need to take ownership here, accepting the associated resource management responsibility? Would temporary access suffice?

Good system developers ask this already!

“The pointer may subsequently be used as an argument to the function free(3).” STRDUP(2)

Rust forces function signatures to encode the answers, and they are checked by the compiler.
Problem: Ownership is intuitive, except for programmers ...

(copies data like integers, and characters, and .mp3's, is "free")

... and anyone else who names things

If ownership were all we had, car-purchase slide seems nonsensical

```java
my_new_car.drive_to(home);
```

Does this transfer `home` into the car?

Do I lose access to my home, just because I drive to it?

We must distinguish an object itself from ways to name that object

`home` must be some kind of `reference` to a `Home`
So we will need references

We can solve any problem by introducing an extra level of indirection

-- David J. Wheeler
Sharing Data: Ownership and References
Rust types

<table>
<thead>
<tr>
<th>Move</th>
<th>Copy</th>
<th>Copy if $T$:Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vec&lt;$T$&gt;, String, ...</td>
<td>i32, char, ...</td>
<td>$[T; n]$, ($T_1, T_2, T_3$), ...</td>
</tr>
</tbody>
</table>

```rust
define structure Car {
    color: Color,
    engine: Engine
}

fn demo_ownership() {
    let mut used_car: Car = Car {
        color: Color::Red,
        engine: Engine::BrokenV8
    };
    let apartments = ApartmentBuilding::new();

    let my_home: &Home; // <-- an "immutable" borrow
    let christine: &mut Car; // <-- a "mutable" borrow
    my_home = &apartments[6]; // (read `mut` as "exclusive")
    let neighbors_home = &apartments[5];
    christine = &mut used_car;
    christine.engine = Engine::VintageV8;
}
```

references to data ($\&mut$ $T$, $\&T$):
Why multiple &-reference types?

Distinguish *exclusive* access from *shared* access

Enables *safe*, *parallel* API's
Borrowing: A Metaphor (continued)
(reminder: metaphors never work 100%)
let christine = Car::new();

This is "Christine"

pristine unborrowed car

(apologies to Stephen King)
let read_only_borrow = &christine;

single inspector (immutable borrow)
read_only_borrows[2] = &christine;
read_only_borrows[3] = &christine;
read_only_borrows[4] = &christine;

many inspectors (immutable borrows)
When inspectors are finished, we are left again with:

pristine unborrowed car
let mutable_borrow = &mut christine; // like taking keys ...
give_arnie(mutable_borrow); // ... and giving them to someone

driven car (mutably borrowed)
Can't mix the two in safe code!

Otherwise: (data) races!
read_only_borrows[2] = &christine;
let mutable_borrow = &mut christine;
read_only_borrows[3] = &christine;
// ⇒ CHAOS!

mixing mutable and immutable is illegal
Mixing mutable and immutable is illegal

Reminder: this does not apply only to concurrency (iterator invalidation, etc.)

```cpp
std::vector<int> v = {1};
int &i = v[0];
std::cout << i << std::endl; // prints 1

v.push_back(2);

std::vector<int> v2 = {2};
std::cout << i << std::endl; // prints 2
```
Ownership  
Exclusive access  
Shared access

Now let's see how we can apply that to build safe abstractions
Pointers, Smart and Otherwise
Stack allocation

```rust
let b = B::new();
```
let b = B::new();

let r1: &B = &b;
let r2: &B = &b;

stack allocation and immutable borrows

(b has lost write capability)
let mut b = B::new();

let w: &mut B = &mut b;

stack allocation and mutable borrows

(b has temporarily lost both read and write capabilities)
Heap allocation: \textbf{Box}<B>

```rust
let a = Box::new(B::new());
```

Pristine boxed B

\texttt{a} (as owner) has both read and write capabilities
Immutably borrowing a box

```rust
default
let a = Box::new(B::new());
let r_of_box: &Box<B> = &a; // (not directly a ref of B)
default
let r1: &B = &*a;
default
let r2: &B = &a; // <-- coercion!
```

Immutable borrows of heap-allocated B; `a` retains read capabilities (has temporarily lost write)
Mutably borrowing a box

```rust
let mut a = Box::new(B::new());
let w: &mut B = &mut a; // (again, coercion happening here)
```

mutable borrow of heap-allocated B

a has temporarily lost both read and write capabilities
Heap allocation: \texttt{Vec\langle B\rangle}

```rust
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
```

vec, filled to capacity
... 

```rust
a.push(B::new());
```

**Vec Reallocation**

**before**

**after**
Slices: borrowing \textit{parts} of an array
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }

pristine unborrowed vec
(a has read and write capabilities)
Immutable borrowed slices

```rust
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
let r1 = &a[0..3];
let r2 = &a[7..n-4];
```

Multiple borrowed slices vec

(a has only read capability now; shares it with r1 and r2)
Safe overlap between &[..]

```rust
let mut a = Vec::new();
for i in 0..n { a.push(B::new()); }
let r1 = &a[0..7];
let r2 = &a[3..n-4];
```

overlapping slices
Basic $\text{Vec}<\text{B}>$ again

pristine unborrowed vec

(a has read and write capabilities)
Mutable slice of whole vec

```rust
let w = &mut a[0..n];
```

Mutable slice of vec

(a has no capabilities; w now has read and write capability)
Mutable disjoint slices

```rust
let (w1, w2) = a.split_at_mut(n-4);
```

disjoint mutable borrows

(w1 and w2 share read and write capabilities for disjoint portions)
Sharing Work: Parallelism / Concurrency
Big Idea

3rd parties identify (and provide) new abstractions for (safe) concurrency and parallelism unanticipated in std lib.
Example: rayon's scoped parallelism
rayon demo 1: map reduce

Sequential

```rust
fn demo_map_reduce_seq(stores: &[Store], list: Groceries) -> u32 {
    let total_price = stores.iter()
        .map(|store| store.compute_price(&list))
        .sum();
    return total_price;
}
```

Parallel (potentially)

```rust
fn demo_map_reduce_par(stores: &[Store], list: Groceries) -> u32 {
    let total_price = stores.par_iter()
        .map(|store| store.compute_price(&list))
        .sum();
    return total_price;
}
```
rayon demo 2: quicksort

```rust
fn quick_sort<T: PartialOrd + Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                  || quick_sort(hi));
    }
}

fn partition<T: PartialOrd + Send>(v: &mut [T]) -> usize {
    // see https://en.wikipedia.org/wiki/
    // Quicksort#Lomuto_partition_scheme
    ...
}
```
**rayon demo 3: buggy quicksort**

```rust
def quick_sort<T: PartialOrd + Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                    || quick_sort(hi));
    }
}
```

(See blog post "Rayon: Data Parallelism in Rust" [bit.ly/1IZcku4])
Threading APIs (plural!)

`std::thread`

`dispatch`: OS X-specific "Grand Central Dispatch"

`crossbeam`: Lock-Free Abstractions, Scoped "Must-be" Concurrency

`rayon`: Scoped Fork-join "Maybe" Parallelism (inspired by Cilk)

(Only the first comes with Rust out of the box)
Final Words
Thanks

www.rust-lang.org

Hack Without Fear