Aaron Turon
Mozilla Research
**Rust** is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety.

- [https://www.rust-lang.org/](https://www.rust-lang.org/)
Safety

Control

“Low-level”

C  C++

Go

Java  Scala

Haskell

Safety
The Essence of Rust

Low-level ⇔ Safe
Unsafe
Why Rust?

- You’re already doing systems programming, want safety or expressiveness.

- You wish you could do some systems work
  - Maybe as an embedded piece in your Java, Python, JS, Ruby, …
Why Mozilla?

Browsers need **control**.

Browsers need **safety**.

**Rust**: New language for safe systems programming.

**Servo**: Next-generation browser built in Rust.
What is control?

```cpp
void example() {
    vector<string> vector;  // Stack and inline layout.
    ...
    auto& elem = vector[0];  // Interior references
    ...
}
```

Stack and inline layout.

Interior references

Deterministic destruction
Zero-cost abstraction

Ability to define abstractions that optimize away to nothing.

Not just memory layout:
- Static dispatch
- Template expansion
- …
What is safety?

```cpp
template <typename T>
class vector
{
  int length;
  int capacity;
  T* data;
  size_t n;

  T& operator[](size_t idx);
};

void example()
{
  vector<string> vector;
  ...
  auto& elem = vector[0];
  vector.push_back(some_string);
  cout << elem;
}
```

Mutating the vector freed old contents.

Dangling pointer: the pointer to freed memory.

Aliasing: more than one pointer to same memory.
What about GC?

No control.

Requires a runtime.

**Insufficient** to prevent related problems: iterator invalidation, data races, many others.
Ownership & Borrowing

No need for a runtime

Memory safety

Data-race freedom (and more)

C++

GC
... Plus lots of goodies

- Pattern matching
- Traits
- “Smart” pointers
- Metaprogramming
- Package management (think Bundler)

**TL;DR: Rust is a modern language**
Ownership

n. The act, state, or right of possessing something.
Ownership (T)
fn give() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    take(vec);
}

fn take(vec: Vec<int>) {
    // ...
}

Take ownership of a Vec<int>
fn give() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    take(vec);
}

fn take(vec: Vec<int>) {
    // …
}

Error: vec has been moved

Prevents:
- use after free
- double moves
- …
Borrow

v. To receive something with the promise of returning it.
Aliasing + Mutation

Shared borrow (\&T)
Mutable borrow (&mut T)
fn lender() {
    let mut vec = Vec::new(); // ...
    vec.push(1);
    vec.push(2);
    use(&vec);
    ...
}

fn use(vec: &Vec<int>) {
    "Shared reference to Vec<int>"
}
Shared references are **immutable**: *

```rust
fn use(vec: &Vec<int>) {
    vec.push(3);
    vec[1] += 2;
}
```

**Error:** cannot mutate shared reference

* Actually: mutation only in controlled circumstances
**Mutable references**

```rust
def push_all(from: &Vec<int>, to: &mut Vec<int>) {
    for elem in from {
        to.push(*elem);
    }
}
```

`push()` is legal
```rust
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {
    for elem in from {
        to.push(*elem);
    }
}
```
What if **from** and **to** are equal?

```rust
def push_all(from: &Vec<int>, to: &mut Vec<int>) {
    for elem in from {
        to.push(*elem);
    }
}
```

dangling pointer
fn push_all(from: &Vec<int>, to: &mut Vec<int>) {...}

fn caller() {
    let mut vec = ...;
    push_all(&vec, &mut vec);
}

Error: cannot have both shared and mutable reference at same time

A &mut T is the only way to access the memory it points at
```rust
let mut vec = Vec::new();
...
for i in 0 .. vec.len() {
    let elem: &int = &vec[i];
    ...
    vec.push(...);
}
...
vec.push(...);
```
Concurrency

*n.* several computations executing simultaneously, and potentially interacting with each other.
Rust’s vision for concurrency

**Originally:** only isolated message passing

**Now:** libraries for many paradigms, using ownership to avoid footguns, guaranteeing no data races
Data race

Two unsynchronized threads accessing same data where at least one writes.
Aliasing

Mutation

No ordering

Data race

Sound familiar?
No data races =
No accidentally-shared state.

All sharing is explicit!

*some_value = 5;
return *some_value == 5;  // ALWAYS true
Messaging
(ownership)
fn parent() {
    let (tx, rx) = channel();
    spawn(move || {...});
    let m = rx.recv();
    tx.send(m);
}

fn parent() {
    let m = Vec::new();
    ...
    tx.send(m);
}
Locked mutable access
(ownership, borrowing)
fn sync_inc(mutex: &Mutex<int>) {
    let mut data = mutex.lock();
    *data += 1;
}

Destructor releases lock
Yields a mutable reference to data
Destructor runs here, releasing lock
Disjoint, scoped access (borrowing)
fn qsort(vec: &mut [int]) {
    if vec.len() <= 1 { return; }
    let pivot = vec[random(vec.len())];
    let mid = vec.partition(vec, pivot);
    let (less, greater) = vec.split_at_mut(mid);
    qsort(less);
    qsort(greater);
}

let vec: &mut [int] = ...;
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T])
fn parallel_qsort(vec: &mut [int]) {
    if vec.len() <= 1 { return; }
    let pivot = vec[random(vec.len())];
    let mid = vec.partition(vec, pivot);
    let (less, greater) = vec.split_at_mut(mid);
    parallel::join(|| parallel_qsort(less), || parallel_qsort(greater));
}

let vec: &mut [int] = …;
Static checking for thread safety

```rust
fn send<T: Send>(&self, t: T)

Only “sendable” types

Arc<Vec<int>>: Send
Rc<Vec<int>>: !Send
```
And beyond…

Concurrency is an area of active development.

Either already have or have plans for:
- Atomic primitives
- Non-blocking queues
- Concurrent hashtables
- Lightweight thread pools
- Futures
- CILK-style fork-join concurrency
- etc.

Always data-race free
Unsafe

*adj.* not safe; hazardous
Safe abstractions

```rust
defn something_safe(...) {
    unsafe {
        ...
    }
}
```

Useful for:

- Bending mutation/aliasing rules (split_at_mut)
- Interfacing with C code

Ownership enables **safe** abstraction boundaries.
Community

*n.* A feeling of fellowship with others sharing similar goals.
“The Rust community seems to be populated entirely by human beings. I have no idea how this was done.”

— Jamie Brandon
It takes a village…

**Community focus** from the start:
Rust 1.0 had > 1,000 contributors
Welcoming, pragmatic culture

Developed “in the open”
Much iteration; humility is key!

**Clear leadership**
Mix of academic and engineering backgrounds
“Keepers of the vision”
This RFC extends traits with associated items, which make generic programming more convenient, scalable, and powerful. In particular, traits will consist of a set of methods, together with:

- Associated functions (already present as "static" functions)
- Associated statics
- Associated types
- Associated lifetimes

These additions make it much easier to group together a set of related types, functions, and constants into a single package.

This RFC also provides a mechanism for multidispatch traits, where the impl is selected based on multiple types. The connection to associated items will become clear in the detailed text below.
Articulating the vision

Memory safety
Concurrency
Abstraction
Stability

without
garbage collection
data races
overhead
stagnation

Hack without fear!