

Rust for C++ Programmers

Vinzent Steinberg

C++ User Group, FIAS

April 29, 2015

Motivation

- ▶ C++ has a lot of problems
- ▶ C++ cannot be fixed
(because of backwards compatibility)
- ▶ *Rust to the rescue!*

What is Rust?



Source: <https://github.com/rust-lang/rust-www/tree/gh-pages/logos>

What is Rust?

- ▶ systems programming language
- ▶ developed by Mozilla, Samsung and the Rust community
- ▶ designed to replace C++ in Mozilla's next-generation browser engine

What is Rust?

- ▶ systems programming language
- ▶ developed by Mozilla, Samsung and the Rust community
- ▶ designed to replace C++ in Mozilla's next-generation browser engine

Promises

- ▶ blazingly fast
- ▶ memory safety without garbage collection
- ▶ concurrency without data races

Popularity

A list of programming languages that are actively developed on GitHub.

Sort ▾

[jashkenas/coffeescript](#)

CoffeeScript ★ 11,136 ↗ 1,490

Unfancy JavaScript

[rust-lang/rust](#)

Rust ★ 10,485 ↗ 2,040

a safe, concurrent, practical language

[golang/go](#)

Go ★ 7,717 ↗ 631

The Go programming language

[ruby/ruby](#)

Ruby ★ 7,303 ↗ 2,191

The Ruby Programming Language

[php/php-src](#)

C ★ 5,748 ↗ 2,078

The PHP Interpreter

[JuliaLang/julia](#)

Julia ★ 5,419 ↗ 1,142

The Julia Language: A fresh approach to technical computing.

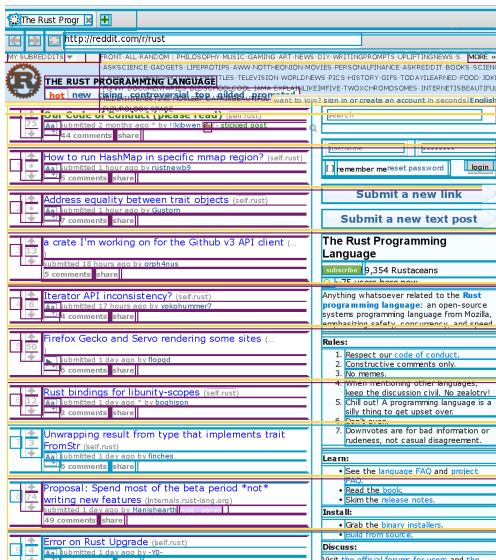
Source: <https://github.com/showcases/programming-languages?s=stars>

Motivating Application: Servo

Our goal is nothing less than building the fastest and most secure browser engine, and we aim to succeed by miles, not inches. We aim for double the performance of current engines, with no crashes.

– Servo job opening

Servo: Parallelized Rendering



The screenshot shows the Reddit interface for the **r/rust** subreddit. The main content area displays a list of posts, each with a title, author, submission time, and comment count. The posts include:

- Our Code of Conduct (please read)** by [kubrow](#) (locked post), 44 comments
- How to run HashMap in specific mmap region?** (self.rust) by [rustnewb9](#), 5 comments
- Address equality between trait objects** (self.rust) by [Custom](#), 7 comments
- a crate I'm working on for the Github v3 API client (...)** by [erph4ns](#), 5 comments
- Iterator API inconsistency?** (self.rust) by [evkahurmer7](#), 4 comments
- Firefox Gecko and Servo rendering some sites (...)** by [flopod](#), 5 comments
- Rust bindings for libunity-scopes** (self.rust) by [boahison](#), 2 comments
- Unwrapping result from type that implements trait FromStr** (self.rust) by [finches](#), 5 comments
- Proposal: Spend most of the beta period *not* writing new features** (internal: rust-lang.org) by [Rahnsheer1](#), 49 comments
- Error on Rust Upgrade** (self.rust) by [y0-](#), 1 comment

On the right side, there is a sidebar for the **The Rust Programming Language** subreddit, showing 9,354 subscribers and 1,576 users online. It includes a **Rules** section with 7 items:

1. Respect our code of conduct.
2. Constructive comments only.
3. No memes.
4. When mentioning other languages, keep the discussion civil. No zealotry!
5. Chill out! A programming language is a silly thing to get upset over.
6. Stay sane.
7. Downvotes are for bad information or rudeness, not casual disagreement.

Below the rules is a **Learn:** section with links to the language FAQ and project FAQ, and a **Read the book** section with a link to skim release notes. There is also an **Install:** section with links to grab binary installers and build from source. At the bottom, there is a **Discuss:** section with a link to the official forum.

Source: <http://blog.servo.org/2015/04/02/twis-29/>

Hello World

```
fn main() {  
    // This is a comment.  
    println!("Hello, World!");  
}
```

- ▶ syntax similar to C++
- ▶ part of STL automatically used => println! already in scope
- ▶ ! means macro
(necessary because no function overloading in Rust)

Functions and Variables

```
fn add(a: i32, b: i32) -> i32 {  
    a + b // equivalent to `return a + b;`  
}
```

```
fn main() {  
    // Create a constant variable of type `i32`.  
    let answer: i32 = add(39, 3);  
    println!("The answer is {}.", answer);  
}
```

Functions and Variables

```
fn add(a: i32, b: i32) -> i32 {  
    a + b  
}
```

```
fn main() {  
    // Create a constant variable of type `i32`.  
    let answer = add(39, 3);  
    // ^ type inferred from context,  
    // like `auto` in C++  
    println!("The answer is {}.", answer);  
}
```

Loops and Mutability

```
/// Sum all integers from 1 to 100.  
fn solve_gauss_homework() -> i32 {  
    let mut sum = 0;  
    for i in 1..101 {  
        sum += i;  
    }  
    sum  
}
```

- ▶ `const` by default
- ▶ like C++11/Python loops
- ▶ generated code as efficient as a C loop

if and else

```
fn signum(x: i32) -> i32 {
    let mut sign;
    if x < 0 {
        sign = -1;
    } else if x > 0 {
        sign = 1;
    } else {
        sign = 0;
    }
    sign
}
```

- ▶ compiler checks `sign` is always initialized

if and else

```
fn signum(x: i32) -> i32 {  
    let sign =  
        if x < 0 {  
            -1  
        } else if x > 0 {  
            1  
        } else {  
            0  
        };  
    sign  
}
```

- ▶ `if {...} else {...}` is an expression
(replaces ternary operator)

enum and match

```
enum Ordering { Less, Equal, Greater }

fn cmp(a: i32, b: i32) -> Ordering;

fn guess_secret_number(guessed_number: i32) {
    let secret_number = 42;
    match cmp(secret_number, guessed_number) {
        Ordering::Less
            => println!("My number is smaller."),
        Ordering::Greater
            => println!("My number is bigger."),
        Ordering::Equal
            => println!("You guessed my number!"),
    }
}
```

- ▶ similar to C++'s switch (but without fallthrough)
- ▶ matches have to be exhaustive

Generalized enums

```
enum Option<T> {  
    Some(T), // Some value `T`  
    None     // No value  
}
```

```
fn print_optional_integer(x: Option<i32>) {  
    match x {  
        Some(i) => println!("Got an integer: {}", i),  
        None => println!("Got nothing."),  
    }  
}
```

- ▶ can be used for pointers, better than NULL:
 - ▶ None has to be considered in match (cannot be forgotten)
 - ▶ pointers are always valid (no redundant checks)

Ownership

```
fn make_vec() {  
    let mut vec = Vec::new();  
    // ^ owned by `make_vec`'s scope  
    vec.push(0);  
    vec.push(1);  
    // scope ends, `vec` is destroyed  
}
```

Source: <http://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html>

- ▶ like `std::vector` in C++
- ▶ `auto` cannot do this
(`vec`'s type inferred from elements being pushed onto it)

Passing Ownership

```
fn make_vec() -> Vec<i32> {  
    let mut vec = Vec::new();  
    vec.push(0);  
    vec.push(1);  
    vec // transfer ownership to the caller  
}
```

- ▶ move semantics by default
(similar to C++'s `unique_ptr`, but without runtime overhead)
- ▶ primitive types have copy semantics (i.e. numbers)

Passing Ownership

```
fn print_vec(vec: Vec<i32>) {  
    // the `vec` parameter is part of this scope,  
    // so it's owned by `print_vec`  
    for i in vec {  
        println!("{}", i)  
    }  
    // now, `vec` is deallocated  
}  
  
fn use_vec() {  
    let vec = make_vec();  
    //^ take ownership of the vector  
    print_vec(vec);  
    //^ pass ownership to `print_vec`  
}
```

Passing Ownership

```
fn use_vec() {  
    let vec = make_vec();  
    // ^ take ownership of the vector  
    print_vec(vec);  
    // ^ pass ownership to `print_vec`  
    // (`vec` is moved, not copied)  
  
    for i in vec { // continue using `vec`  
        println!("{}", i * 2)  
    }  
}
```

- ▶ results in compiler error:

error: use of moved value: `vec`

```
for i in vec {  
    ^~~
```

Borrowing

- ▶ `print_vec` destroys vectors, instead of temporary access
- ▶ introduce concept of *borrowing*
- ▶ **compiler checks that leases do not outlive objects being borrowed**

Borrowing

```
fn print_vec(vec: &Vec<i32>) {  
    // the `vec` parameter is borrowed for this scope  
    for i in vec {  
        println!("{}", i)  
    }  
    // now, the borrow ends  
}  
  
fn use_vec() {  
    let vec = make_vec(); // take ownership of the vector  
    print_vec(&vec);      // lend access to `print_vec`  
    for i in vec {        // continue using `vec`  
        println!("{}", i * 2)  
    }  
    // `vec` is destroyed here  
}
```

Pointers in Rust

- ▶ references valid for limited scope (checked by compiler)
- ▶ two kinds of reference:
 - ▶ *immutable* &T: sharing but no mutation
 - ▶ *mutable* &mut T: mutation but no sharing

Pointers in Rust

- ▶ references valid for limited scope (checked by compiler)
- ▶ two kinds of reference:
 - ▶ *immutable* &T: sharing but no mutation
 - ▶ *mutable* &mut T: mutation but no sharing

- ▶ “raw” pointers in unsafe Rust
 - ▶ *T: anything goes, like in C++
 - ▶ not memory-safe
 - ▶ can only be used inside `unsafe { ... }` blocks

Mutable Borrowing

```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {  
    for i in from {  
        to.push(*i);  
    }  
}
```

- ▶ what if `push_all(&vec, &mut vec)`?

Mutable Borrowing

- ▶ what if `push_all(&vec, &mut vec)`?
 - ▶ elements pushed onto vector
 - ▶ vector grows, copying elements over
 - ▶ iterator with dangling pointer into old memory
 - ▶ Disaster!

Mutable Borrowing

- ▶ what if `push_all(&vec, &mut vec)`?
 - ▶ elements pushed onto vector
 - ▶ vector grows, copying elements over
 - ▶ iterator with dangling pointer into old memory
 - ▶ Disaster!

- ▶ **compiler checks that whenever a mutable borrow is active, no other borrows of the object are active**

error: cannot borrow `vec` as mutable because it is also
borrowed as immutable

```
push_all(&vec, &mut vec);  
          ^~~
```

Thread Safety

Thread Safety

- ▶ data races prevented by ownership semantics
- ▶ compiler rejects programs violating thread safety (cannot share data that is not thread-safe)
- ▶ safely shared memory by using special types (like C++'s `shared_ptr`)
- ▶ safe communication between threads

Summary

- ▶ sharing without mutation, mutation without sharing
⇒ memory safety, no data races
- ▶ performance comparable to C++
- ▶ strong type system (*Let the compiler do the work for you.*)
- ▶ first stable release in 16 days

Resources

- ▶ The Rust Book
<http://doc.rust-lang.org/book/>
- ▶ Rust by Example
<http://rustbyexample.com>
- ▶ The Rust Programming Language Blog:
<https://blog.rust-lang.org>
See “Fearless Concurrency with Rust” for the section on ownership.

Features

- ▶ modern systems programming: designed ground up instead of evolved over decades
- ▶ explicit ownership (lifetimes checked by the compiler)
 - ▶ **no data races**
 - ▶ **no segfaults**
- ▶ modules (with automatic dependencies and online repository), no headers
- ▶ built-in testing/benchmarking
- ▶ built-in documentation generator
- ▶ excellent first-party documentation
- ▶ traits (better than classes, constrained generics)
- ▶ closures (like lambdas in C++)
- ▶ channels (threading by message passing)

Differences to C++

- ▶ move by default (less unnecessary copies)
- ▶ containers use views (much less copies of strings)
- ▶ private by default
- ▶ no overloading (of functions)
- ▶ no implicit conversions (affects numbers and references, see gotchas)
- ▶ implicit dereference
- ▶ explicitly declared unsafe code
- ▶ uninitialized variables are impossible
- ▶ warning if violating conventions (snake_case vs CamelCase)

Gotchas for C++ Programmers

- ▶ `std::mem::swap(a, b)` vs `std::mem::swap(&mut a, &mut b)`
- ▶ references in Rust have to be explicit!

Error Handling

```
enum Result {
    Ok(File),
    Err(String),
}

fn open(path: &String) -> Result;

fn main() {
    let result = open("data.txt");
    match result {
        Ok(_)
            => println!("Opened file successfully!"),
        Err(why)
            => println!("Could not open file: {}", why),
    }
}
```

Message Passing

```
fn send<T: Send>(chan: &Channel<T>, t: T);
fn recv<T: Send>(chan: &Channel<T>) -> T;

fn main() {
    let mut chan: Channel<Vec<i32>> = ...;
    let mut vec = Vec::new();

    do_some_computation(&mut vec);

    send(&chan, vec);
    print_vec(&vec);
}
```

- ▶ results in compiler error:

Error: use of moved value `vec`