

# Rust for Rubyists

Jan-Erik / @badboy\_

FrOSCon, RedFrogConf, 24.08.2014

[1] 20756 segmentation fault (core dumped) ./awesome-prog

---

*From Simple Machines to Impossible Programs*

# Understanding Computation



O'REILLY®

*Tom Stuart*

## git.io/smallstep-rust

---

```
let mut env = HashMap::new();
env.insert("y".to_string(), number!(1));

let mut m = Machine::new(
    sequence!(
        assign!("x", number!(3)),
        assign!("res", add!(add!(number!(38), variable!("x")), variable!("y")))
    ), env);

m.run();
```

# Rust what?

---

*Rust is a systems programming language that runs blazingly fast, prevents almost all crashes\*, and eliminates data races.*

\* In theory. Rust is a work-in-progress and may do anything it likes up to and including eating your laundry.

# Facts

- developed by Mozilla Research
- very young (v0.11.0 atm)
- in active development, things change fast
- docs miiiight be a little bit unfinished

---

```
["Grace", "Sophie", "Ada"].each do |name|  
  puts "Hello #{name}!"  
end
```

---

```
fn main() {  
  for name in ["Grace", "Sophie", "Ada"].iter() {  
    println!("Hello {}!", name);  
  }  
}
```

---



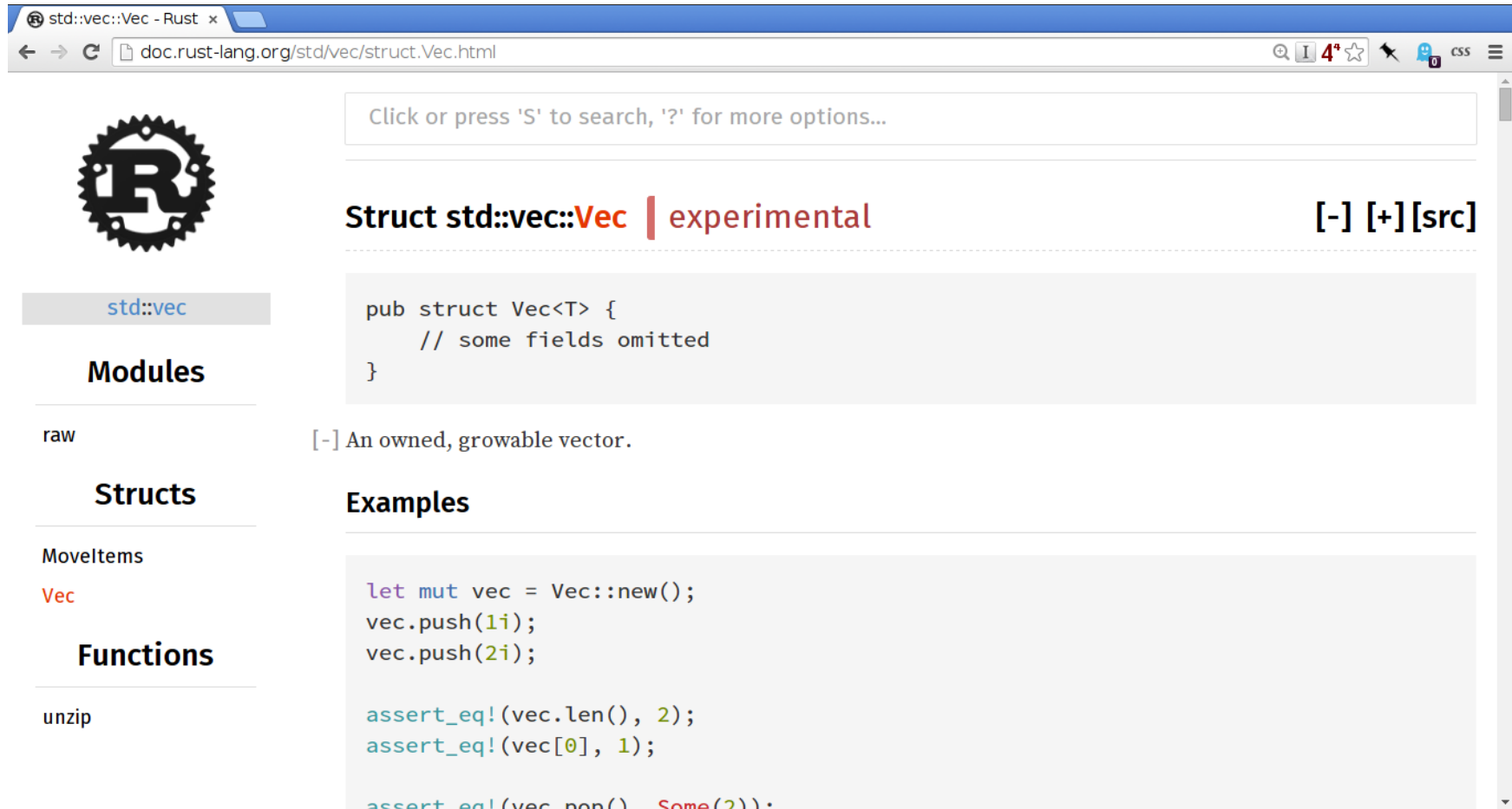
---

```
$ rustc hello.rs
$ ./hello
Hello Grace!
Hello Sophie!
Hello Ada!
```

---



# Use the docs



The screenshot shows a web browser window displaying the Rust documentation for `std::vec::Vec`. The browser's address bar shows the URL `doc.rust-lang.org/std/vec/struct.Vec.html`. The page features a search bar at the top with the placeholder text "Click or press 'S' to search, '?' for more options...". Below the search bar, the title "Struct `std::vec::Vec` | experimental" is displayed, along with navigation links `[-]`, `[+]`, and `[src]`. The left sidebar contains a navigation menu with the Rust logo at the top, followed by "std::vec" (highlighted), "Modules", "raw", "Structs", "MoveItems", "Vec" (highlighted), "Functions", and "unzip". The main content area shows the Rust code definition for the `Vec` struct: 

```
pub struct Vec<T> {  
    // some fields omitted  
}
```

 Below the code, there is a description: `[-]` An owned, growable vector. The "Examples" section contains the following code: 

```
let mut vec = Vec::new();  
vec.push(1i);  
vec.push(2i);  
  
assert_eq!(vec.len(), 2);  
assert_eq!(vec[0], 1);  
  
assert_eq!(vec.pop(), Some(2));
```

# Document your code

---

```
//! This is a documentation comment  
//! `rustdoc` will generate the beautiful HTML API doc you saw before
```

# Don't fear the errors

---

```
# hello.rs
fn main() {
    let foo = "bar";
    println!("{}", fooz);
}
```

---

```
$ rustc hello.rs
hello.rs:3:20: 3:24 error: unresolved name `fooz`. Did you mean `foo`?
hello.rs:3     println!("{}", fooz);
```

# Test your stuff

---

```
#[test]
fn this_tests_code() {
    fail!("Fail!");
}
```

---

```
$ rustc --test my_test.rs && ./my_test
running 1 test
test this_tests_code ... FAILED
```

failures:

```
---- this_tests_code stdout ----
task 'this_tests_code' failed at 'Fail!', my_test.rs:3
```

failures:  
  this\_tests\_code

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

# Use language features

- immutable by default
- pattern matching → think switch/case but better
- closures → known as blocks/lambdas as well
- type inference → save some typing, still be correct
- zero-cost abstractions → the compiler can do its magic
- guaranteed memory safety → no more null-pointer-hell
- lightweight green tasks → concurrency, but the easy way

# Things I didn't mention yet

---

Traits Cargo  
unsafe {} Channels  
enum struct object orientation  
inheritance mutability  
ffi functional programming  
memory ownership



# Sources

- [rust-lang.org](http://rust-lang.org) (official site)
- [Rust for Rubyists](#) (book)
- [A 30 minute introduction to Rust](#) (blog post)
- [Understanding Computation](#)
- [git.io/smallstep-rust](https://git.io/smallstep-rust)



# Backup

---

```
int *dangling(void)
{
    int i = 1234;
    return &i;
}
```

```
int add_one(void)
{
    int *num = dangling();
    return *num + 1;
}
```

---

```
fn dangling() -> &int {
    let i = 1234;
    return &i;
}

fn add_one() -> int {
    let num = dangling();
    return *num + 1;
}

fn main() {
    add_one();
}
```

---

```
fn dangling() -> &int {  
    let i = 1234;  
    return &i;  
}
```

```
fn add_one() -> int {  
    let num = dangling();  
    return *num + 1;  
}
```

```
fn main() {  
    add_one();  
}
```

---

```
dangling_broken.rs:1:18: 1:22 error: missing lifetime specifier [E0106]  
dangling_broken.rs:1 fn dangling() -> &int {  
                                ^~~~~
```

```
error: aborting due to previous error
```

---

```
fn dangling<'a>() -> &'a int {  
    let i = 1234;  
    return &i;  
}
```

```
fn add_one() -> int {  
    let num = dangling();  
    return *num + 1;  
}
```

```
fn main() {  
    add_one();  
}
```

---

```
dangling_lifetime.rs:3:13: 3:14 error: `i` does not live long enough  
dangling_lifetime.rs:3     return &i;  
                             ^
```

```
dangling_lifetime.rs:1:30: 4:2 note: reference must be valid for the  
lifetime 'a as defined on the block at 1:29...
```

```
...  
dangling_lifetime.rs:1:30: 4:2 note: ...but borrowed value is only valid for the  
block at 1:29
```

---

```
fn dangling() -> Box<int> {
    let i = box 1234i;
    return i;
}

fn add_one() -> int {
    let num = dangling();
    return *num + 1;
}

fn main() {
    println!("{}", add_one());
}
```