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

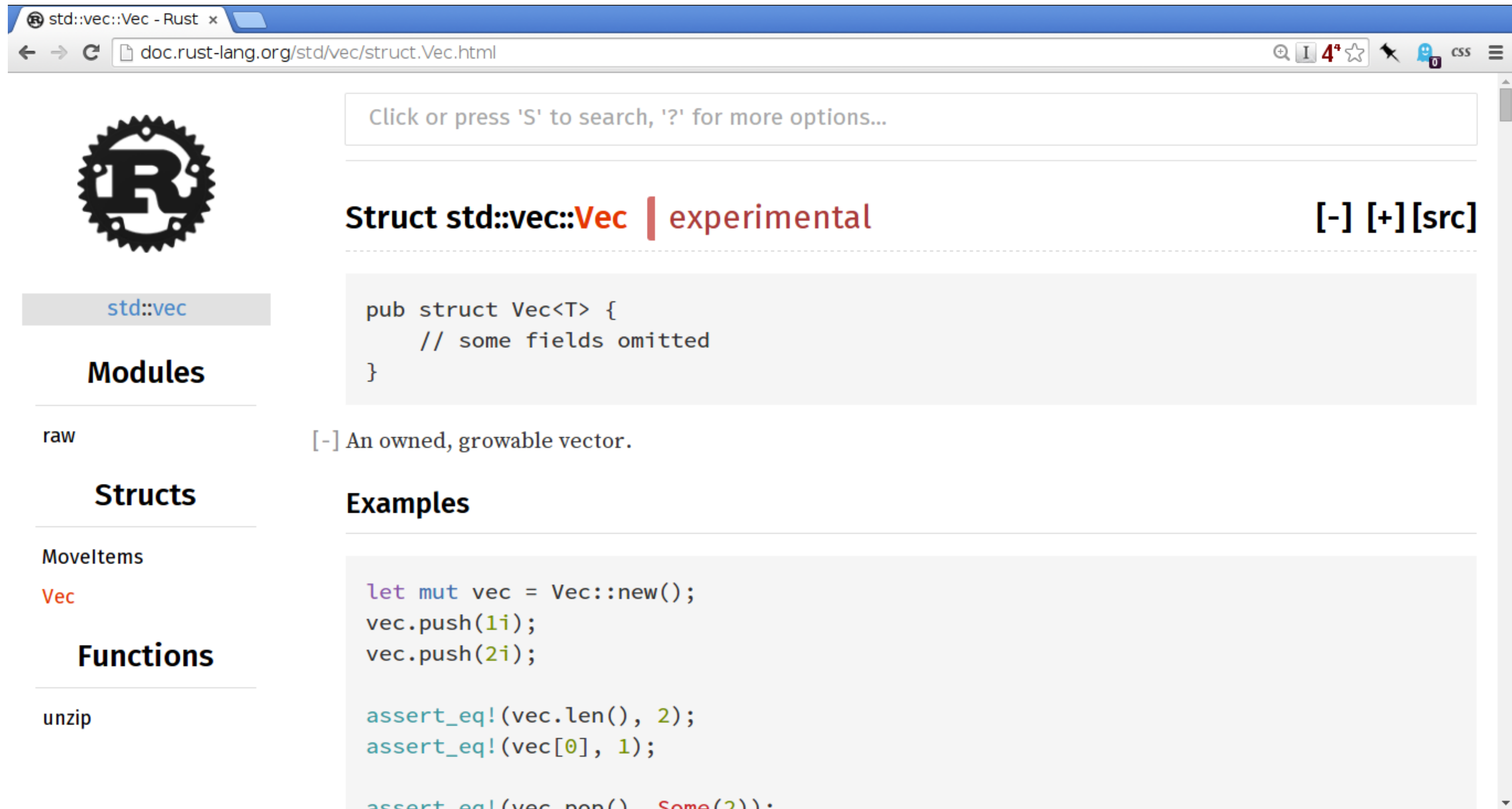
- developed by Mozilla Research
- very young (v0.11.0 atm)
- in active development, things change fast
- docs miiiight be a little bit unfinished

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

```rust
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



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

```rust
#[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
object orientation
struct
inheritance
mutability
ffi
functional programming
memory ownership

Sources

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

Backup

```c
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());
}
```