A Friendly Introduction to Rust

Jens Nockert

jens.nockert@topvisible.se

December 3, 2013

Overview

1. Rust Skåne
2. Language Design
3. Let's Learn Some Rust
4. Four kinds of pointers?
5. Let's Write Some Rust
6. Real simple programs

Rust Skåne

- Somewhere to learn Rust

Rust Skåne

- Somewhere to learn Rust
- Somewhere to learn how to present

Rust Skåne

- Somewhere to learn Rust
- Somewhere to learn how to present
- Somewhere to learn things they don't teach in school

Rust Skåne

- Somewhere to learn Rust
- Somewhere to learn how to present
- Somewhere to learn things they don’t teach in school

Rust Skåne

- Somewhere to learn Rust
- Somewhere to learn how to present
- Somewhere to learn things they don’t teach in school

And all in a friendly environment.

No one knows Rust

No one here works with Rust, and we're not likely to start using Rust at work soon.

No one knows Rust

No one here works with Rust, and we're not likely to start using Rust at work soon.

- Presentations about projects in Rust.

No one knows Rust

No one here works with Rust, and we're not likely to start using Rust at work soon.

- Presentations about projects in Rust.
- Presentations about something you know.

No one knows Rust

No one here works with Rust, and we're not likely to start using Rust at work soon.

- Presentations about projects in Rust.
- Presentations about something you know.
- Presentations about how to write better code.

No one knows Rust

No one here works with Rust, and we're not likely to start using Rust at work soon.

- Presentations about projects in Rust.
- Presentations about something you know.
- Presentations about how to write better code.
- Presentations about techniques and paradigms.

No one knows Rust

No one here works with Rust, and we're not likely to start using Rust at work soon.

- Presentations about projects in Rust.
- Presentations about something you know.
- Presentations about how to write better code.
- Presentations about techniques and paradigms.
- Presentations about how Rust differs from your favorite language.

Why replace C?

- Buffer overflows.
- Dangling pointers.
- Out-of-bounds array accesses.
- Format string errors.
- Stack overflows.
- Memory leaks.
- Double free errors.

Rust is like C, but . . .

- . . . with memory safety.

Rust is like C, but . . .

- . . . with memory safety.
- . . . with mutability control.

Rust is like C, but . . .

- . . . with memory safety.
- . . . with mutability control.
- . . . with a powerful type system.

Rust is like C, but . . .

- . . . with memory safety.
- . . . with mutability control.
- . . . with a powerful type system.
- . . . with generics.

Rust is like C, but . . .

- . . . with memory safety.
- . . . with mutability control.
- . . . with a powerful type system.
- . . . with generics.
- . . . with lightweight tasks and fast asynchronous, copyless message passing.

Rust is like C, but . . .

- . . . with memory safety.
- . . . with mutability control.
- . . . with a powerful type system.
- . . . with generics.
- . . . with lightweight tasks and fast asynchronous, copyless message passing.
- . . . with macros.

Rust is like C, but . . .

- . . . with memory safety.
- . . . with mutability control.
- . . . with a powerful type system.
- . . . with generics.
- . . . with lightweight tasks and fast asynchronous, copyless message passing.
- . . . with macros.

Rust is like C, but ...

- ... with memory safety.
- ... with mutability control.
- ... with a powerful type system.
- ... with generics.
- ... with lightweight tasks and fast asynchronous, copyless message passing.
- ... with macros.
- ... without runtime checks.

So what is like C?

- The close relationship between written code and generated assembly.
- The ability to run with close to no runtime.
- No costly abstractions.
- You'll still have to think about allocations and memory layout.
- Can call (and be called from) C without any special treatment.
- Aims to be as fast as C or Fortran.

What about Go?

Rust is significantly different in philosophy.

- No shared mutable state.
- Minimal GC impact, individual tasks can completely avoid GC.
- No null pointers.
- Type parametric code.
- Greater type safety.

Memory safety

Memory safety has many meanings, but for example, outside of 'unsafe' blocks, Rust programs ...

- ... cannot have null pointers. (use-before-initialize or use-after-move)
- ... can only dereference previously allocated pointers that have not yet been freed.
- ... cannot have dangling pointers.
- ... cannot overflow the stack.
- ... cannot have invalid format strings.
- ... cannot free a block twice.

Memory safety in C or C++

In C, you cannot see if a function is memory safe just by looking at its implementation.

Memory safety in C or C++

In C, you cannot see if a function is memory safe just by looking at its implementation. You have to check if the implementation of each function it calls, in the context of the caller.

```
class foo {
public:
    std::vector<int> indices;
    int counter;

    foo() : indices(), counter(0) {
        indices.push_back(1);
        indices.push_back(2);
        indices.push_back(3);
    }

    void increment_counter();

    int &get_first_index() {
        assert(indices.size() > 0);
        return indices[0];
    }

    void munge() {
        int &first = get_first_index();
        increment_counter();
        std::cout << first << std::endl;
        first = 20;
    }
};
```

Immutability

Variables are immutable by default,

```
let a = 1.0f64;
a += 1.0; // Type error

let mut b = 2.0f64;
b *= 2.0; // OK
```

Functions

Semicolon at the end of a line turns an expression into a statement,

```
fn f(x:f64) -> f64 {
  return x.exp() + 5.0;
}

fn f(x:f64) -> f64 {
  x.exp() + 5.0
}
```

Ps. Some styles use function return, some always use the semicolon-less style.

Structures

Structs are binary compatible with C, so you can pass and receive them from foreign functions.

```
struct Complex {
  r:f64, i:f64
}
```

Enumerations

Enums are not binary compatible though, but have gained some new super powers!

```
enum Plans {
  Trial, Silver, Gold
}
```

and they can have fields,

```
enum Option<T> {
  None, Some(T)
}
```

Pattern Matching

Enums are not binary compatible though, but have gained some new super powers!

```
fn cost_per_month(p:Plan) -> f64 {
  match p {
    Trial => 0.0,
    Silver => 5000.0,
    Gold   => 50000.0
  }
}
```

If we forget an entry, that's a exhaustiveness error. This means we can just add a Bronze plan and let the compiler spit out all the places we need to modify.
Values on the LHS are type-checked, unlike in C switches.

Destructuring

```
let option = Some(~"something")
match option {
  None => ~"It's nothing",
  Some(something) => fmt!("It's %s", something)
}
```

This is the only way to access 'something'.

```
let x = [1, 2, 3];
match x {
  [1, ..tail] => tail,
  [_, ..tail] => []
}
```

And it works for vectors and structs as well.

Loops

```
for i in uint::range(0, 10) {
  printfln!("Hello, %?" , i);
};
```

```
uint::range(0, 10, |i| {
  printfln!("Hello, %?" , i);
});
```

```
let mut i = 0;
while i < 10 {
  printfln!("Hello, %?" , i); j += 1;
}
```

So what is this thing called memory safety?

- The easiest way to be ‘safe’ is to use a GC for all allocations, and not allow pointers.
- But Rust’s performance requirements doesn’t really allow for an all-GC solution, yet we want automatic memory management.

The two interesting kinds, ~and &

- A ~pointer is a unique pointer, there can only be one.
- A & pointer is a borrowed pointer, they get created when you lend your unique pointer to someone. You cannot use your unique pointer again until you get it back. Kind of like a book.

Borrowed pointers

- It's safe to borrow a pointer to data in a stack frame that will outlive your own.
- It's also efficient: the compiler proves statically that lifetimes nest properly, so borrowed pointers need no automatic memory management. (Unlike ARC)
- Rust accomplishes both of these goals without making the programmer responsible for reasoning about pointer lifetimes. The compiler checks your assumptions.

Project Euler: 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

A function that finds multiples of 3 and 5

```
fn f(x:int) -> bool {
  return match x {
    i if i % 5 == 0 => true,
    i if i % 3 == 0 => true,
    _ => false
  };
}
```

And compose that

```
std::iter::range(0, 1001).filter(f).fold(0, |sum, i| {
  sum + i
})
```

Fibonacci

```
fn fib(n:int) -> int {
  if n < 2 {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

Or if we want to win benchmarks

```
use std::f64;
use std::libc;

fn factorial(n:f64) {
  let mut sign:libc::c_int = 0;

  return f64::exp(f64::lgamma(n - 1, &mut sign));
}
```

```rust
#[no_mangle]
pub extern "C" fn main() {
  unsafe {
    init();
    delay(1);
    pinMode(LED, OUTPUT);

    loop {
      digitalWrite(LED, HIGH);
      delay(1000);
      digitalWrite(LED, LOW);
      delay(100);
    }
  }
}
```

```rust
extern mod extra;

use std::os;
use std::str;
use std::libc;
use extra::getopts::groups;

struct c_passwd {
    pw_name: *libc::c_char,
    // Maybe I should put here others struct members, but...Well, maybe.
}

extern {
    pub fn geteuid() -> libc::c_int;
    pub fn getpwuid(uid: libc::c_int) -> *c_passwd;
}

unsafe fn getusername() -> ~str {
    let passwd: *c_passwd = getpwuid(geteuid());

    let pw_name: *libc::c_char = (*passwd).pw_name;
    let name = str::raw::from_c_str(pw_name);

    name
}

fn main() {
    let args = os::args();
    let program = args[0].as_slice();
    let opts = ~[
        groups::optflag("h", "help", "display this help and exit"),
        groups::optflag("V", "version", "output version information and exit"),
    ];
    let matches = match groups::getopts(args.tail(), opts) {
        Ok(m) => m,
        Err(f) => fail!(f.to_err_msg()),
    };
    if matches.opt_present("help") {
        println("whoami 1.0.0");
        println("");
        println("Usage:");
        println!(" {:s}", program);
        println("");
        print(groups::usage("print effective userid", opts));
        return;
    }
    if matches.opt_present("version") {
        println("whoami 1.0.0");
        return;
    }

    exec();
}

pub fn exec() {
    unsafe {
        let username = getusername();
        println!("{:s}", username);
    }
}
```

```
extern mod extra;

use std::os;
use std::io::stderr;
use extra::getopts::groups;

fn main() {
    let args = os::args();
    let program = args[0].clone();
    let opts = ~[
        groups::optflag("h", "help", "display this help and exit"),
        groups::optflag("V", "version", "output version information and exit"),
    ];
    let matches = match groups::getopts(args.tail(), opts) {
        Ok(m) => m,
        Err(f) => {
            writeln!(&mut stderr() as &mut Writer,
                     "Invalid options\n{}", f.to_err_msg());
            os::set_exit_status(1);
            return
        }
    };
    if matches.opt_present("help") {
        println("yes 1.0.0");
        println("");
        println("Usage:");
        println!(" {0:s} [STRING]... [OPTION]...", program);
        println("");
        print(groups::usage("Repeatedly output a line with all specified STRING(s), or 'y'.", opts));
        return;
    }
    if matches.opt_present("version") {
        println("yes 1.0.0");
        return;
    }
    let mut string = ~"y";
    if !matches.free.is_empty() {
        string = matches.free.connect(" ");
    }

    exec(string);
}

pub fn exec(string: ~str) {
    loop {
        println(string);
    }
}
```