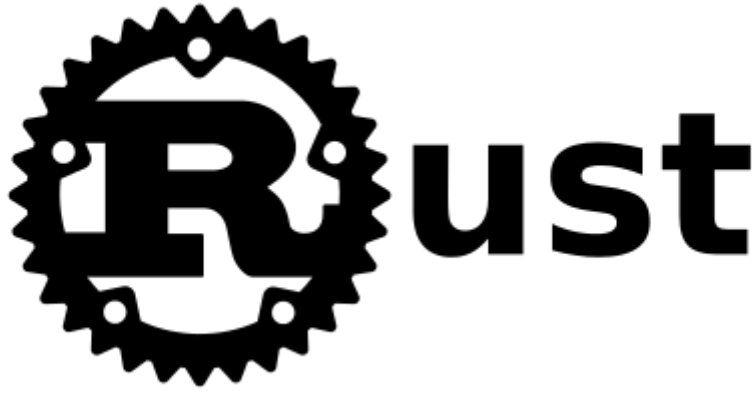


schupbach@fnal.gov

November 20, 2018

Rust

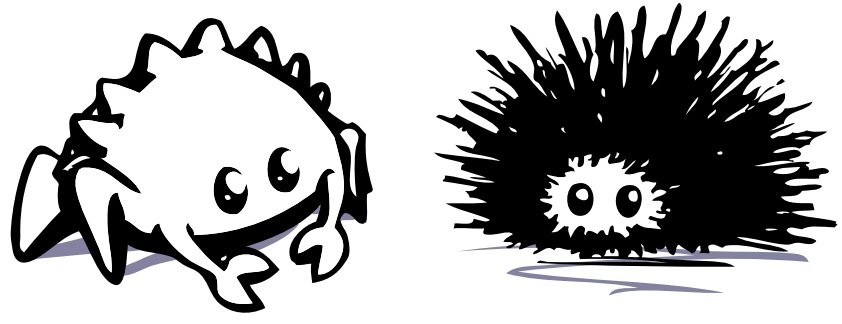
a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

a bold claim indeed, let's investigate

Systems programming

what is it, and in what context

- compiles to native binary
- hardware interfaces
- emphasis on speed and memory consumption
- operating system functionality
- device drivers
- networking basics
- databases



Key language features

- **zero-cost abstractions**
- **move semantics**
- **guaranteed memory safety**
- **threads without data races**
- **trait-based generics**
- **pattern matching**
- **type inference**
- **minimal runtime**
- **efficient C bindings**

Why

Memory management is hard and C/C++ take full advantage of undefined behavior.

...it's all up to you, so good luck with that, for example;

```
int main(int argc, char **argv) {
    unsigned long a[1];
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

This *will* complile and sets the return address of main() to somewhere in libc.

Consider

...this contrived example to print "Hello" using C++17.

```
#include "required_headers.hpp"

void print(std::vector<char> seq)
{
    auto newend = std::remove_if( std::begin(seq), std::end(seq), [](char ch) { return !std::isalpha(ch); });

    std::sort(std::begin(seq), newend);
    std::for_each( std::cbegin(seq), std::vector<char>::const_iterator(newend), std::putchar);
}

auto greeting()
{
    return std::tuple<std::vector<char>,int>({'e','&','.','l','l','{','H','o'},42);
}

int main()
{
    if (auto [str, answer] = greeting(); answer == 42)
    print(str);
}
```

Research project

Supported by

which inlcudes the following

- web resources
- conferences and events
- paid developer support
- directly benefits Firefox browser with Servo & Stylo

Resources to learn you some Rust

- Rust-lang homepage
- std lib
- crates.io
- Rust playground
- Compiler Explorer
- docs.rs
- New Rustacean podcast
- #rust-beginners irc channel

There's a lot of stuff out there, canonical texts include

Klabnik, S. & Nichols, C. *The Rust Programming Language*, on-line 2018.
Blandy, J. & Orendorf, J. *Programming Rust*, O'Reilly 2018.

Events & Media

a thriving communinty exists to move Rust forward

- Rust conf
- Chicago meetup
- Youtube channel

First class tooling

learn from the mistakes of others.

- rustup
- cargo

...ever tried to build a moderately large c++ project
with multiple dependencies on Windows?

rustup.rs

installs & updates the build environment

```
$ curl https://sh.rustup.rs -sSf | sh
$ export PATH="$HOME/.cargo/bin:$PATH"
$ rustc --version
```

but I want the newest stable version, or not

```
$ rustup update
$ rustup self uninstall
```

cargo.rs

the Rust build tool and package manager

```
$ cargo new hello_rust --bin
$ cd hello_rust
```

creates a new binary exe with the following directory structure

```
$ ls
.
├── Cargo.lock
├── Cargo.toml
├── .gitignore
├── .git/
├── src/
│   └── main.rs
└── target/
        ├── debug/
        │   └── hello_rust.exe
        └── rls/
```

Cargo.toml

project info, configuration, and dependency declarations

```
[package]
name = "hello_rust"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
description = "I'm new to Rust"
license = "MIT/Apache-2.0"

[dependencies]
rand = "0.3.14"
```

Cargo.lock

rebuild the same artifact each time, i.e. fix your dependencies

initial versions are written to the lock file for safe keeping

```
$ cargo update           # updates all dependencies

$ cargo update -p rand   # updates just “rand”
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating rand v0.3.14 -> v0.3<.15
```

cargo.rs

will your code compile

```
$ cd hello_rust
$ cargo check
    Checking hello_rust v0.1.0 (file:///C:/Users/schupbach/Documents/code/rust/hello_rust)
    Finished dev [unoptimized + debuginfo] target(s) in 0.52s
```

build and execute

```
$ cargo build
    Compiling hello_rust v0.1.0 (file:///C:/Users/schupbach/Documents/code/rust/hello_rust)
    Finished dev [unoptimized + debuginfo] target(s) in 1.03s

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target\debug\hello_rust.exe`
    Hello World!
    ["target\\debug\\hello_rust.exe"]

$ cargo build --release
    Compiling hello_rust v0.1.0 (file:///C:/Users/schupbach/Documents/code/rust/hello_rust)
    Finished release [optimized] target(s) in 1.02s
```

cargo.rs

write test functions as you develop your application

```
//! ```
//! assert_eq!(4, 2 + 2);
//! ```

#[test]
fn it_works() {
    assert_eq!(4,myadd(2,2));
}
```

tests written in the comments are visible in the documentation!

```
$ cargo test
    Compiling rand v0.1.0 (https://github.com/rust-lang-nursery/rand.git#9f35b8e)
    Compiling hello_rust v0.1.0 (file:///path/to/project/hello_rust)
    Running target/test/hello_rust-9c2b65bbb79eabce

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

docs

automagically document your Rust project

```
$ cargo doc
    Documenting hello_rust v0.1.0 (file:///C:/Users/schupbach/Documents/code/rust/hello_rust)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
```

the Serde library on docs.rs

IDEs

the following provide Rust lang support

- Visual Studio, via plugin
- Visual Studio Code, via extensions
- IntelliJ / CLion
- Syntax highlighting for Emacs, vim, etc...

community supported, open source, you can check it all out on github

Language specifics

influenced by, ML (lisp with data types), C/C++, Haskell

memory safety and concurrency

no null pointers or shared mutable state

these guarantees are enforced at compile time

Getting started

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);         // debug format

    let greeting = "Hello World!";
    println!("{}",greeting);
}
```

kinda looks like C, but with type inference

Primitive types

the usual suspects

```
// singular value scalar types
let x: u32 = 0xdeadbeef;                    // 32 bit unsigned integer

let y: i32 = 16;
let y = y + 1;                              // shadowing, transforms y with immutability

let z: f32 = 3.14159;                       // default type is f64

let f: bool = false;

let tup: (i32, f64, u8) = (500, 6.4, 1);    // tup.0, tup.1, tup.2
let (a, b, c) = tup;

// static collections
let r: [i32;3] = [1,2,3];                   // r[0], r[1], r[2], fixed size array

let t = 'c';                                // a char is a UTF-8 Unicode Scalar Value
let data = "a string literal";              // stored in the binary output of the program
```

work horses

Strings, Vectors, and HashMaps

```
// dynamic (heap)
let s = String::from("Hello!");                 // a String is just a Vector of chars

let v1 = vec![1, 2, 3];                         // macro initialization, or
let v2: Vec<i32> = Vec::new();
v2.push(42);                                    // add items to the vector

use std::collections::HashMap;
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);        // adding items

let team_name = String::from("Blue");
let score = scores.get(&team_name);             //accessing items
```

user defined types

```
struct Square (i32, i32);                          // tuple struct
let s = Square(6,4);                               // s.0, s.1, anonymous fields

impl Square {                                      // Square method
    fn area(&self) -> i32 {
        self.0 * self.1
    }
}
assert_eq!(24,s.area());

struct User {
    username: String,                              // named fields
    email: String,
}

let client = User {
    email: String::from("thedude@fnal.gov"),       // client.email
    username: String::from("urbanAchiever"),       // client.username
};
```

you can extend a data types default implementation!

enums for the win

```
// can be one of three variants
enum Colors {
    Red,
    Green,
    Blue
}

let c = Colors::Red;

// pattern match on all variants of Colors
match c {
    Colors::Red => { println!("it's Red!") }
    _ => { println!("it's not Red!") }
}
```

proper sum types

pattern matching

```
// from the std lib
enum Option<T> {
Some(T),
None
}

let x = Some(42);

fn inc(i: i32) -> Option<i32> {
if (i < 0) {
    return None;
} else {
    Some(i+1)
}
}

let result = inc(1);    // with Option<i32> return type

match result {
    Some(i) => { println!("{},{},i,i+1") }
    None => { println!("invalid input!") }
}
```

sane error handling in plain sight, without using exceptions

control flow

```
if true && true {
    // do something
}

for i in 0..10 {                    // range based for loops, nice
    println!("{}",i);
}

let mut x = 0;
loop {                              // while true { }
    x = x + 1;
    println!("{}",x);               // prints 1 thru 5
    if x % 5 == 0 { break; }
}
```

more cf

```
fn main() {

    // returns a valid Enum
    fn foo() -> Option<i32> {
        Some(3)
    }

    if let Some(3) = foo() {          // just landed in C++17
        println!("three");
    }
}
```

immutability

```
let x = 5;                      // immutable by default
x = 10;                         // compile time error

let mut y = 6;                  // we can now change y
y =  2;
```

Rust's concurrency model relies on this feature

ownership

variable bindings ***have ownership*** of what they're bound to

```
let v = vec![1,2,3];                    // intialize a vector of i64
let v2 = v;

println!("where'd you go? {}", v[0]);    // compile time error
```

move semantics ensure that there is ***exactly one*** binding to any given resource

this guarantees memory safety at compile time

borrowing

but I want what you have, if only temporarily

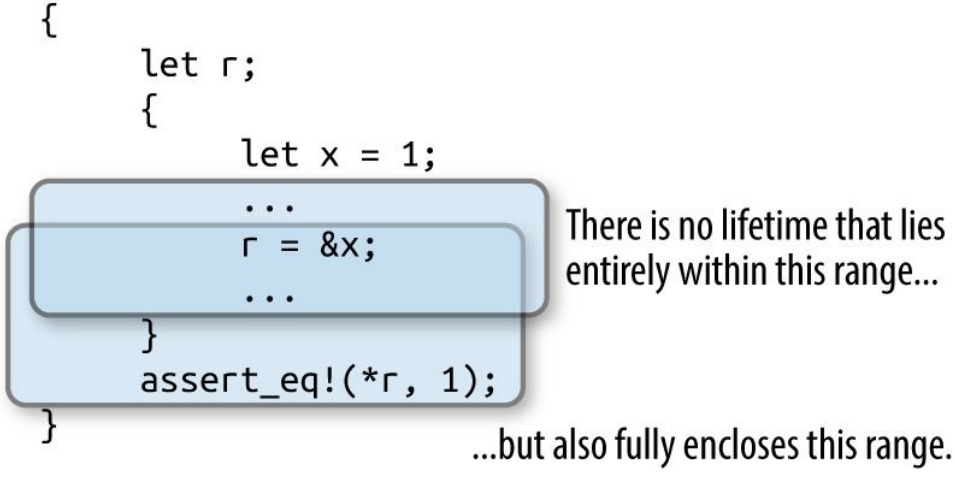
```
let v = vec![1,2,3];

fn foo(v: &Vec<i32>) {
    v.push(4);                  // compile time error
}

foo(&v);                        // the ref is immutable by default

let mut v = vec![1,2,3];        // the vector should be mutable

fn foo(v: &mut Vec<i32>) {
    v.push(4);
}

foo(&mut v);                    // as well as the ref
```

you need to be specific, there are rules, so pick one

- **many** *immutable* references **&T** to a resource
- exactly **one** *mutable* reference **&mut T**

lifetimes

how to prevent dangling pointers, or '*use after free*'



explicitly ensures the object ***pointed to*** outlives any references to it

example

more lifetimes

```
fn main() {
    let i = 3; // Lifetime for `i` starts. ──────────────────────┐
    //                                                          │
    { //                                                        │
        let borrow1 = &i; // `borrow1` lifetime starts. ──────┐ │
    //                                                        │ │
        println!("borrow1: {}", borrow1); //                  │ │
    } // `borrow1 ends.  ──────────────────────────────────────┘ │
    //                                                          │
    //                                                          │
    { //                                                        │
        let borrow2 = &i; // `borrow2` lifetime starts. ──────┐ │
    //                                                        │ │
        println!("borrow2: {}", borrow2); //                  │ │
    } // `borrow2` ends. ──────────────────────────────────────┘ │
    //                                                          │
}   // Lifetime ends.  ──────────────────────────────────────────┘
```

lifetime syntax

the **borrow checker** compares the scopes of resources and references

```
// Multiple elements with different lifetimes. In this case, it
// would be fine for both to have the same lifetime `'a`, but
// in more complex cases, different lifetimes may be required.

fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x is {}, y is {}", x, y);
}

fn substr<'a>(s: &'a str, until: u32) -> &'a str {
    // do something and a return string slice
}
```

valid for *input* and *output* params

putting it together

```
fn main() {
    // annotate reference lifetimes
    fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
        if x.len() > y.len() {
            x
        } else {
            y
        }
    }

    let string1 = String::from("long string is long");          // ref 1
    let result;
    {
        let string2 = String::from("xyz");                      // ref 2
        result = longest(string1.as_str(), string2.as_str());  // returns ref 1 or 2
    }

    println!("The longest string is {}", result);
}
```

the **borrow checker** is the *most* difficult aspect of learning Rust

modules

enables the re-use of code in an organized fashion
functions, types, constants, and modules are private by default

```
// external module
extern crate mycommunicator;

// internal module
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
    mycommunicator::client::connect();

    // bring into local scope
    use a::series::of::*;

    nested_modules();
}
```

memory safety in Rust

it all boils down to

- ownership
- borrowing
- lifetimes

Concurrency

the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Rust leverages ownership semantics, immutability, and thread safe data types

threads

the pitfalls of executing code simultaneously

- race conditions, ...who writes first
- deadlocks, ...you have what I need & I have what you need
- nondeterministic ordering, ...what just happened

os threads *vs* green threads

Rust uses a ***1:1*** threading model

- smaller runtime, smaller binaries
- crates available for ***M:N*** threading model

Shared-State

you've got two choices

- communicate by sharing memory
- share memory by communicating

smart pointer types **Mutex<T>** and **Arc<T>**
or
channels for message passing

brute force threading example

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

channels

asynchronous mechanism for communication between threads
ie: passing messages of **Some<T>**

allows for the flow of information between a *Sender* and a *Receiver*

example

think one or more producer and single consumer

closure

a function and the lexical environment within which that function was declared

functions are first class objects in Rust
they can be passed around and returned like primitive types

```
fn main() {
    let num = 5;
    let plus_num = |x: i32| x + num;

    assert_eq!(10, plus_num(5));
}
```

generics

Lift algorithms and data structures from concrete examples to their most general and abstract form.

— Bjarne Stroustrup

Traits

a language feature that tells the Rust compiler about functionality a type ***must*** provide

useful because they allow a type to make certain promises about its behavior

traits can constrain or bound generic functions

implementation for *compile-time* polymorphism

some Traits

```
// specify the interface
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

// the implementation
impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

using Traits

```
// roughly equivalent to template<T>
// where T must implement area()
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    print_area(c);
}
```

Trait Objects

used to implement a heterogeneous collection of objects that implement an interface

where the precise type can only be known at runtime

overhead to manage function pointers to vtable, no inlining, no code bloat

implementation for *runtime* polymorphism

example

Rusts added value

- simlarity of syntax
- can be added to existing types
- allows for multiple independant implementations

Comparing

C++

- Templates are expressive compile time comuptations
- errors pile up during template expansion
- no type checking
- base classes and virtual methods

Rust

- Traits lack compile time computations
- errors pile up at the ***call*** site
- type checking
- Trait Objects

Unsafe

writing safe code in C/C++ ***is*** possible

compile-time static analysis is conservative, the following are allowed

- dereference raw pointer
- call an unsafe function or method
- access or mutate static variable
- implement an unsafe trait

the programmer ensures memory is handled in a valid manner

example

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    // raw pointers
    let address = 0x01234usize;
    let r = address as *mut i32;
    let slice: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };

    // ffi
    unsafe { println!("Absolute value of -3 according to C: {}", abs(-3)); };

    // static vars
    static HELLO_WORLD: &str = "Hello, world!";
    println!("name is: {}", HELLO_WORLD);

    // traits
    unsafe trait Foo { /* methods go here */ };
    unsafe impl Foo for i32 { /* method implementations go here */ };
}
```

misc

- hygenic or declaritive macros to extend the syntax of the language
  - type checked
  - similar to pattern matching
  - vec!
- procedureal macros
  - similar to functions, they take and return Rust code
- a **fn** type, pass and return functions
- foreign function interface, calling C↔Rust code without hastle

OK Fan Boy

slow your roll

Hurdles

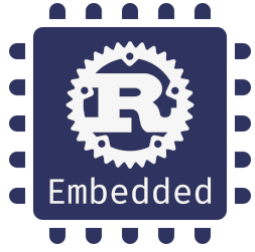
commonly voiced criticisms

- needlessly foreign syntax
  - lifetimes
  - casting to another datatype
- compile times can be quite high
  - generics
  - modules
- modules on crates.io
  - separating quality from the garbage
- the language is evolving
  - requires effort to keep up, breaking changes
- steep learning curve
  - ownership semantics
  - borrow checker

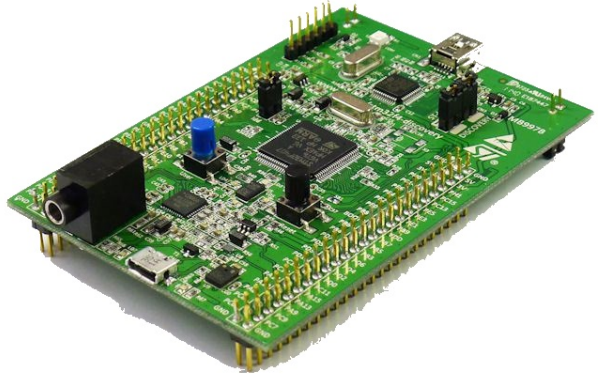
IMO

Rust is a compelling addition to the CS landscape for the following;

- ecosystem
- tooling
- language features



Rust on your microcontroller



- 168 MHz
- 1 MB flash
- 192 kB RAM
- ADC's, DAC's, and interrupts

does Rust's memory model facilitate bare metal

Conclusion

I will push on with it, looks promising.

- Learn more
- Give it a try!

Thanks!