Intro To Rust
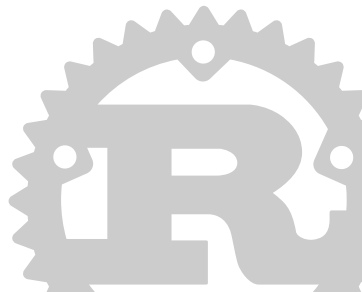
Danilo Bargen (@dbrgn), Raphael Nestler (@rnestler)
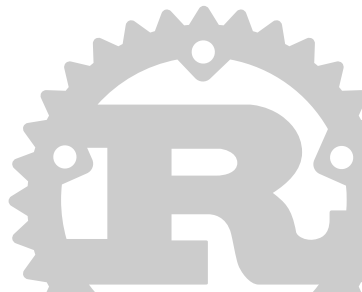July 15, 2016

Coredump Rapperswil

Outline

What is Rust?

What is Rust?

*«Rust is a systems programming language
that runs blazingly fast, prevents nearly all segfaults,
and guarantees thread safety.»*
— www.rust-lang.org

What's wrong with systems languages?

- It's difficult to write secure code.
- It's very difficult to write multithreaded code.

These are the problems Rust was made to address.

Quick Facts about Rust

(As of June 2016)

- Started by Mozilla employee Graydon Hoare
- First announced by Mozilla in 2010
- Community driven development
- First stable release: 1.0 in May 2015
- Latest stable release: 1.9
- More than 54'000 commits on Github
- Largest well-known project written in Rust: Servo[1]

---

[1]https://servo.org/

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait based generics
- Pattern matching
- Type inference
- Minimal runtime, no GC
- Efficient C bindings

Getting Started

Getting Started

Installing Rust

*«rustup is an installer for
the systems programming language Rust»*
— www.rustup.rs

Rustup.rs

- Makes it easy to install different Rust versions
- Successor of multirust
- Written in Rust itself

Rustup.rs

- Makes it easy to install different Rust versions
- Successor of multirust
- Written in Rust itself
- Installing is easy:
  ```
  $ curl https://sh.rustup.rs -sSf | sh
  ```

Rustup.rs

- Makes it easy to install different Rust versions
- Successor of multirust
- Written in Rust itself
- Installing is easy:
  $ curl https://sh.rustup.rs -sSf | sh
  Obviously you shouldn't do that ;)
- Alternatively you can use https://play.rust-lang.org/

Getting Started

Cargo, Rust's Package Manager

Cargo

- Project and package manager
- Fetches and builds your project's dependencies
- Invokes rustc or another build tool with the correct parameters to build your project

Cargo – Create a New Project

```
$ cargo new hello_world --bin
$ cd hello_world
$ tree
.
   Cargo.toml
   src
       main.rs

1 directory, 2 files
```

Cargo – Compile and Run

```
$ cargo build
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)

$ ./target/debug/hello_world
Hello, world!

$ cargo run
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
   Running `target/debug/hello_world`
Hello, world!
```

Cargo – Dependencies

- Cargo generated a manifest for us:

  ```
  [package]
  name = "hello_world"
  version = "0.1.0"
  authors = ["Your Name <you@example.com>"]
  ```

- To add a dependency (from https://crates.io or github) we add it to the manifest:

  ```
  [dependencies]
  time = "0.1"
  ```

- Cargo uses semantic versioning[2] $\rightarrow$ we get the latest 0.1.x version

---

Cargo – Dependencies

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading winapi v0.2.7
   Compiling winapi v0.2.7
   Compiling winapi-build v0.1.1
   Compiling libc v0.2.11
   Compiling kernel32-sys v0.2.2
   Compiling time v0.1.35
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
```

Cargo – Testing

Rust has integrated unit testing[3]

```
#[test]
fn it_works() {
    assert_eq!(1, 1);
}

#[test]
fn it_fails() {
    assert_eq!(1, 2);
}
```

---

[3]https://doc.rust-lang.org/book/testing.html

Cargo – Testing

```
$ cargo test

running 2 tests
test it_fails ... FAILED
test it_works ... ok

...

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

Reading Rust

Basic Syntax

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```rust
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```rust
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```rust
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```rust
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Basic Syntax

```rust
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Structs

```rust
pub struct Point {
  x: f64,
  y: f64,
}
```

```rust
pub struct Point {
  x: f64,
  y: f64,
}

impl Point {
  /// static method ("constructor" per convention)
  pub fn new() -> Point {
    Point { x: 0.0, y: 0.0 }
  }
  pub fn distance_from_zero(&self) -> f64 {
    (self.x*self.x + self.y*self.y).sqrt()
  }
}
```

```
let p = Point::new();
assert_eq!(p.x, 0.0);
assert_eq!(p.y, 0.0);

let p = Point { x: 1.0, y: 2.0 };
assert_eq!(p.x, 1.0);
assert_eq!(p.y, 2.0);

let p = Point { x: 3.0, y: 4.0 };
assert_eq!(p.distance_from_zero(), 5.0);
```

Enumerations

```rust
pub enum Color {          let color = Color::Red;
  Red,                    match color {
  Green,                    Color::Red => println!("Got red"),
  Blue,                     Color::Green => println!("Got green"),
}                           Color::Blue => println!("Got blue"),
                          }
```

- In its simplest form similar to C/C++ (but type safe)
- Data that is one of several possible variants

Enumerations

```rust
pub enum Color {          let color = Color::Red;
  Red,                    match color {
  Green,                    Color::Red => println!("Got red"),
  Blue,                     Color::Green => println!("Got green"),
}                           Color::Blue => println!("Got blue"),
                          }
```

- In its simplest form similar to C/C++ (but type safe)
- Data that is one of several possible variants
- We can match (switch) on enums
- We *must* match all variants

```rust
pub enum Shape {
  Circle{radius: f64},
  Square{side: f64},
}
```

```rust
pub enum Shape {                let shape = Shape::Circle{radius: 1.0};
  Circle{radius: f64},          match shape {
  Square{side: f64},              Shape::Circle{radius} => {
}                                   println!("Circle with radius {}", radius)
                                  }
                                  _ => panic!("Must be a circle!"),
                                }
```

- A Shape is Either a Circle with radius or a Square with side
- We can use match to "extract" data

Traits

```rust
trait HasArea {
  fn area(&self) -> f64;
}
```

- Like interfaces
- Defines functionality a type must provide

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

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

Trait Based Generics

```rust
fn min<T: Ord>(a: T, b: T) -> T {
  if a <= b { a } else { b }
}
```

Trait Based Generics

```
fn min<T: Ord>(a: T, b: T) -> T {
  if a <= b { a } else { b }
}

...

min(10i8,  20)    == 10;    // T is i8
min(10,    20u32) == 10;    // T is u32
min("abc", "xyz") == "abc"; // Strings are Ord

min(10i32, "xyz"); // error: mismatched types
```

Generic Types

```
struct Range<Idx> {
  start: Idx,
  end: Idx,
}
```

Generic Types

```rust
struct Range<Idx> {
  start: Idx,
  end: Idx,
}

...

Range { start: 200, end: 800 } // OK
Range { start: 1.3, end: 4.7 } // Also OK
```

Generic Enumerations

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

```rust
fn safe_div(n: i32, d: i32) -> Option<i32> {
  if d == 0 {
    return None;
  }
  Some(n / d)
}
```

Matching an Option

```
match safe_div(num, denom) {
  None => println!("No quotient."),
  Some(v) => println!("Quotient is {}.", v)
}
```

What is Type Safety?

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

A C Program

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

According to C99, undefined behavior. Output:

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

Definitions

- If a program has been written so that no possible execution can exhibit undefined behavior, we say that program is **well defined**.

- If a language's type system ensures that every program is well defined, we say that language is **type safe**.

Type Safe Languages

- C and C++ are not type safe.

- Python is type safe:
  ```
  >>> a = [0]
  >>> a[3] = 0x7ffff7b36ceb
  Traceback (most recent call last):
  File "", line 1, in <module>
  IndexError: list assignment index out of range
  >>>
  ```

- Java, JavaScript, Ruby, and Haskell are also type safe.

It's Ironic.

- C and C++ are not type safe.
- Yet they are being used to implement the foundations of a system.
- Rust tries to resolve that tension

Memory Safety in Rust

Three Key Promises

To guarantee memory safety, Rust gives us three key promises:

- No null pointer dereferences

Three Key Promises

To guarantee memory safety, Rust gives us three key promises:

- No null pointer dereferences
    - There are no null pointers in safe Rust
    - For error handling and control flow, `Option` and `Result` types are used.

Three Key Promises

To guarantee memory safety, Rust gives us three key promises:

- No null pointer dereferences
    - There are no null pointers in safe Rust
    - For error handling and control flow, `Option` and `Result` types are used.

- No dangling pointers

Three Key Promises

To guarantee memory safety, Rust gives us three key promises:

- No null pointer dereferences
    - There are no null pointers in safe Rust
    - For error handling and control flow, `Option` and `Result` types are used.

- No dangling pointers
    - The concepts of "ownership", "borrowing" and "lifetimes" prevent the use of uninitialized or freed pointers

Three Key Promises

To guarantee memory safety, Rust gives us three key promises:

- No null pointer dereferences
    - There are no null pointers in safe Rust
    - For error handling and control flow, `Option` and `Result` types are used.
- No dangling pointers
    - The concepts of "ownership", "borrowing" and "lifetimes" prevent the use of uninitialized or freed pointers
- No buffer overruns

Three Key Promises

To guarantee memory safety, Rust gives us three key promises:

- No null pointer dereferences
    - There are no null pointers in safe Rust
    - For error handling and control flow, `Option` and `Result` types are used.

- No dangling pointers
    - The concepts of "ownership", "borrowing" and "lifetimes" prevent the use of uninitialized or freed pointers

- No buffer overruns
    - There's no pointer arithmetic in safe Rust
    - Arrays in Rust are not just pointers
    - There are runtime bounds checks for indexing
    - But most stdlib functions use iterators, which are checked at compile time

Memory Safety in Rust

Promise 1: No null pointer dereferences

Promise 1: No null pointer dereferences

Null pointers are useful.

They can indicate the absence of optional information.

They can indicate failures.

Promise 1: No null pointer dereferences

Null pointers are useful.

They can indicate the absence of optional information.

They can indicate failures.

But they can introduce severe bugs.

Promise 1: No null pointer dereferences

Null pointers are useful.

They can indicate the absence of optional information.

They can indicate failures.

But they can introduce severe bugs.

Rust separates the concept of a pointer from the concept of an optional or error value.

Optional values are handled by Option<T>.

Error values are handled by Result<T, E>.

Many helpful tools to do error handling.

```rust
fn safe_div(n: i32, d: i32) -> Option<i32> {
    if d == 0 {
        return None;
    }
    Some(n / d)
}
```

You already saw `Option<T>`

```rust
fn safe_div(n: i32, d: i32) -> Option<i32> {
    if d == 0 {
        return None;
    }
    Some(n / d)
}
```

But what if you want to return an error, not just None?

There's also `Result<T, E>`

```rust
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

How to use `Results`:

```rust
enum Error {
    DivisionByZero,
}

fn safe_div(n: i32, d: i32) -> Result<i32, Error> {
    if d == 0 {
        return Err(Error::DivisionByZero);
    }
    Ok(n / d)
}
```

How to use `Results`:

```rust
enum Error {
    DivisionByZero,
}

fn safe_div(n: i32, d: i32) -> Result<i32, Error> {
    if d == 0 {
        return Err(Error::DivisionByZero);
    }
    Ok(n / d)
}
```

It's good practice to define your own error types instead of using strings.

But `Result` can get tedious...

```rust
fn do_calc() -> Result<i32, String> {
    let a = match do_subcalc1() {
        Ok(val) => val,
        Err(msg) => return Err(msg),
    };
    let b = match do_subcalc2() {
        Ok(val) => val,
        Err(msg) => return Err(msg),
    };
    Ok(a + b)
}
```

Ergonomic error handling with the `try!` macro

```rust
fn do_calc() -> Result<i32, String> {
    let a = try!(do_subcalc1());
    let b = try!(do_subcalc2());
    Ok(a + b)
}
```

Ergonomic error handling with the try! macro

```
fn do_calc() -> Result<i32, String> {
    let a = try!(do_subcalc1());
    let b = try!(do_subcalc2());
    Ok(a + b)
}
```

Note: Error signature must match!

Mapping Errors

What if the signature does not match?

Mapping Errors

What if the signature does not match? Then we can use map_err():

```rust
fn do_subcalc() -> Result<i32, String> { ... }
fn do_calc() -> Result<i32, Error> {
    let res = do_subcalc();
    let mapped = res.map_err(|msg| {
        println!("Error: {}", msg);
        Error::CalcFailed
    });
    let val = try!(mapped);
    Ok(val + 1)
}
```

Mapping Errors: A closer look

```rust
let mapped = res.map_err(|msg| Error::CalcFailed);
```

...is the same as...

```rust
let mapped = match res {
    Ok(val) => Ok(val),
    Err(msg) => Err(Error::CalcFailed),
}
```

Memory Safety in Rust

Promise 2: No dangling pointers

Promise 2: No dangling pointers

- Rust programs never try to access a heap-allocated value after it has been freed.
- By default, no garbage collection or reference counting involved!
- Everything is enforced at compile-time.

Three Rules

Rule 1
Every value has a single owner at any given time.

Three Rules

Rule 1
Every value has a single owner at any given time.

Rule 2
You can borrow a reference to a value, so long as the reference doesn't outlive the value.

Three Rules

Rule 1
Every value has a single owner at any given time.

Rule 2
You can borrow a reference to a value, so long as the reference doesn't outlive the value.

Rule 3
You can only modify a value when you have exclusive access to it.

Ownership

- Variable bindings own their values
- A struct owns its fields
- An enum owns its values
- Every heap-allocated value has a single pointer that owns it
- All values are dropped when their owner is dropped

Ownership: Scoping

If a value goes out of scope, the corresponding memory is automatically freed.

```
{
    let s = "Chuchichästli".to_string();
} // s goes out of scope, memory is freed
```

Ownership: Move Semantics

Ownership is moved by default.

```
let s = "Chuchichästli".to_string();

// t1 takes ownership from s
let t1 = s;

// compile-time error: use of moved value s
let t2 = s;
```

Ownership: Opt-in Implicit Copy Semantics

Types that implement the Copy marker trait (more about traits later) are copied
instead of moved. The stdlib implements Copy for all primitive types.

```
let pi = 3.1415926f32;
let foo = pi;
let bar = pi; // This is fine!
```

Ownership: Opt-in Explicit Copy Semantics

If you prefer copies to be explicit, you can implement the Clone trait instead.

```
let s = "Chuchichästli".to_string();
let t1 = s.clone();
let t2 = s.clone();
```

Ownership: Deriving Copy / Clone

The compiler can automatically derive implementations of Copy and Clone for us.

```
#[derive(Copy, Clone)]
struct Color {
    r: u8,
    g: u8,
    b: u8
}
```

Ownership: Function Parameters

But what about this?

```rust
fn print_loud(text: String) { println!("{}!!!!!", text); }
let s = "Hello, Mozillians".to_string();
print_loud(s);
println!("{}", s);
```

Ownership: Function Parameters

But what about this?

```rust
fn print_loud(text: String) { println!("{}!!!!!", text); }
let s = "Hello, Mozillians".to_string();
print_loud(s);
println!("{}", s);
```

```
error: use of moved value: `s`
println!("{}", s);
               ^
note: `s` moved here because it has type `collections::string::String`,
which is non-copyable
print_loud(s);
           ^
```

Borrowing

Instead of moving a value, it can also be borrowed.

```rust
fn print_loud(text: &String) { println!("{}!!!!!", text); }
let s = "Hello, Mozillians".to_string();
print_loud(&s);
println!("Original value was {}", s);
```

Many functions can borrow at the same time, because they cannot modify.

Mutable Borrowing

If you need exclusive (=write) access, you can use mutable borrows.

```
fn make_loud(text: &mut String) { text.push_str("!!!!!"); };
let mut s = "Hello, Mozillians".to_string();
make_loud(&mut s);
println!("New value is {}", s);
```

While borrow a mutable reference to a value, that refrence is the only way to access that value at all.

Borrowing prevents moving

While borrowed, a move must be prevented. Otherwise you might end up with a dangling pointer.

```
let x = String::new();
let borrow = &x;
let y = x;
```

```
error: cannot move out of `x` because it is borrowed [E0505]
    let y = x;
            ^
note: borrow of `x` occurs here
    let borrow = &x;
                 ^
```

Lifetimes

What's the problem here?

```
let borrow;
let x = String::new();
borrow = &x;
```

```
error: `x` does not live long enough
    borrow = &x;
              ^
```

Lifetimes

The lifetime of the borrow is longer than the lifetime of 'x'.

```
let borrow;
let x = String::new();
borrow = &x;
```

This can also be visualized differently:

```
{
    let borrow;
    {
        let x = String::new();
        borrow = &x;
    }
}
```

Using lifetime checking, the compiler guarantees that there are no dangling pointers.

Memory Safety in Rust

Promise 3: No buffer overruns

No buffer overruns: Recap

- There's no pointer arithmetic in safe Rust
- Arrays in Rust are not just pointers
- There are runtime bounds checks for indexing
- But most stdlib functions use iterators, which are checked at compile time

Multithreaded Programming

We'll make this short

- The Rust compiler does not know about concurrency
- Everything works based on the three rules[4]
- We'll step through an example

---

[4]Slide 47

Threads

```rust
let t1 = std::thread::spawn(|| { return 23; });
let t2 = std::thread::spawn(|| { return 19; });

let v1 = t1.join().unwrap();
let v2 = t2.join().unwrap();

println!("{} + {} = {}", v1, v2, v1 + v2);
```

Shared Data

```rust
let mut data = vec![0];
let t1 = thread::spawn(|| { data.push(19); });
```

```
error: closure may outlive the current function, but it borrows `data`,
which is owned by the current function [E0373]
    let t1 = thread::spawn(|| {
        data.push(19);
    });
note: `data` is borrowed here
    data.push(19);
    ^~~~
help: to force the closure to take ownership of `data` (and any other
referenced variables), use the `move` keyword, as shown:
    let t1 = thread::spawn(move || {
        data.push(19);
    });
```

Shared Data (2) – Move data

Let's move the data into the Thread.

```
let mut data = vec![0];
let t1 = thread::spawn(move || { data.push(19); });
```

Shared Data (3) – Outside Access

But now we can't access it anymore..

```
let mut data = vec![0];
let t1 = thread::spawn(move || { data.push(19); });
t1.join().unwrap();
println!("Data: {:?}", data);
```

```
error: use of moved value: `data` [E0382]
    println!("Data: {:?}", data);
                           ^~~~
note: `data` moved into closure environment here because it has
type `collections::vec::Vec<i32>`, which is non-copyable
    let t1 = thread::spawn(move || { data.push(19); });
                           ^~~~~~~~~~~~~~~~~~~~~~~~~~~
help: perhaps you meant to use `clone()`?
```

Shared Data (4) – Arcs

Atomic reference counting to the rescue!

```
let data = Arc::new(vec![0]);

let data2 = data.clone();
let t1 = thread::spawn(move || {
    println!("Data2: {:?}", data2);
});

t1.join().unwrap();
println!("Data: {:?}", data);
```

```
Data2: [0]
Data: [0]
```

Shared Data (5) – Mutate?

```
let data = Arc::new(vec![0]);

let mut data2 = data.clone();
let t1 = thread::spawn(move || {
    data2.push(1);
});

t1.join().unwrap();
println!("Data: {:?}", data);
```

```
error: cannot borrow immutable borrowed content as mutable
    data2.push(1);
    ^~~~~
```

Shared Data (6) – Arc + Mutex

```rust
let data = Arc::new(Mutex::new(vec![0]));

let data2 = data.clone();
let t1 = thread::spawn(move || {
    let mut guard = data2.lock().unwrap();
    guard.push(1);
});

t1.join().unwrap();
println!("Data: {:?}", *data.lock().unwrap());
```

```
Data: [0, 1]
```

Shared Data (7) – Multiple Threads

Now we can also create multiple threads.

```
...
let data2 = data.clone();
let t1 = thread::spawn(move || {
    let mut guard = data2.lock().unwrap();
    guard.push(1);
});

let data3 = data.clone();
let t2 = thread::spawn(move || {
    let mut guard = data3.lock().unwrap();
    guard.push(2);
});
...
```

Data: [0, 1, 2]

Channels

Besides threading, you can also use channels:

```rust
use std::sync::mpsc::channel;
```

Signature:

```rust
fn channel<T>() -> (Sender<T>, Receiver<T>)
```

Rust Community

Projects Using Rust[6]

- Rust / Cargo itself :)
- Servo, the Parallel Browser Engine
  https://servo.org
- Dropbox[5]
- Maidsafe — The New Decentralized Internet
  http://maidsafe.net
- Parity — Next Generation Ethereum Client
  https://ethcore.io/parity.html

---

[5]https://www.reddit.com/r/rust/comments/4adabk/the_epic_story_of_dropboxs_exodus_from_the_amazon/

[6]https://www.rust-lang.org/friends.html

Rust Community Considered Helpful[8]

- The Rust Community is really friendly and welcoming
- You can get help on:
    - Reddit https://www.reddit.com/r/rust/
    - IRC[7]
    - User Forum https://users.rust-lang.org/
    - Stackoverflow http://stackoverflow.com/questions/tagged/rust
- Discussions about the language
    - Forum https://internals.rust-lang.org/
    - GitHub RFCs https://github.com/rust-lang/rfcs/

---

[7]https://client00.chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust
[8]https://www.rust-lang.org/community.html

Rust Projects — Looking for a side project?

- SpaceAPI[9] implementation:
  https://github.com/coredump-ch/spaceapi-rs
  https://github.com/coredump-ch/spaceapi-server-rs
  https://github.com/coredump-ch/status

- rpsrtsrs:
  https://github.com/coredump-ch/rpsrtsrs

---

[9] http://spaceapi.net/

Rust Projects — Looking for a side project?

- SpaceAPI[9] implementation:
  https://github.com/coredump-ch/spaceapi-rs
  https://github.com/coredump-ch/spaceapi-server-rs
  https://github.com/coredump-ch/status
- rpsrtsrs:
  https://github.com/coredump-ch/rpsrtsrs
- Easy issues:
  https://github.com/servo/servo/labels/E-easy
  https://this-week-in-rust.org/

---

[9]http://spaceapi.net/

Thank you!

www.coredump.ch

Slides: https://github.com/coredump-ch/intro-to-rust

Examples: https://github.com/coredump-ch/intro-to-rust/tree/master/examples