An Introduction
to
Rust Programming Language

Haozhong Zhang

Jun 1, 2015

Acknowledgment

Parts of contents in the following slides may use contents from following sources.

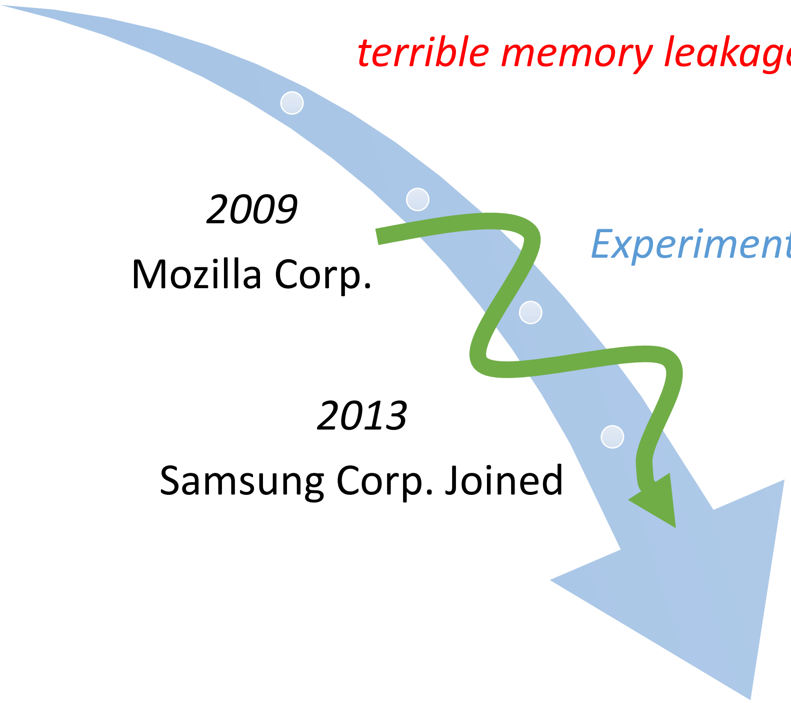
- Aaron Turon, *The Rust Programming Language,* Colloquium on Computer Systems Seminar Series (EE380) , Stanford University, 2015.
- Alex Crichton, *Intro to the Rust programming language*, http://people.mozilla.org/~acrichton/rust-talk-2014-12-10/
- *The Rust Programming Language*, https://doc.rust-lang.org/stable/book/

What is Rust?

From the official website (http://rust-lang.org):

“Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety. ”

A brief history



Pre-2009

Graydone Hoare

terrible memory leakages/bugs in Firefox

2009

Mozilla Corp.

Experimental web browser layout engine:

Servo

2013

Samsung Corp. Joined

2015/05/15

v1.0 Stable Released!

Who are using Rust?

- rustc: Rust compiler
  - https://github.com/rust-lang/rust
- Cargo: Rust’s package manager
  - https://github.com/rust-lang/cargo
- Servo: Experimental web browser layout engine
  - https://github.com/servo/servo
- Piston: A user friendly game engine
  - https://github.com/PistonDevelopers/piston
- Iron: An extensible, concurrent web framework
  - https://github.com/iron/iron
- …

Control & Safety

Things make Rust Rust.

In the real world …

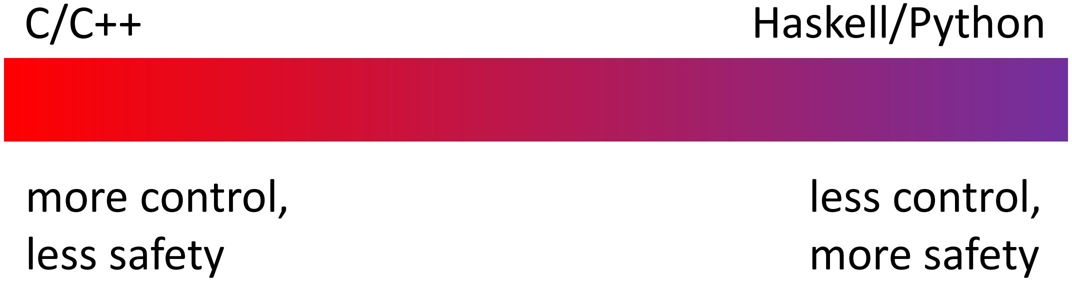
- **Rust** is the coating *closest* to the *bare metal*.

As a programming language …

```
fn main() {
    println!(“Hello, world!”);
}
```

- **Rust** is a *system programming language* barely on the *hardware*.
    - No *runtime* requirement (*eg*. GC/Dynamic Type/…)
    - More *control* (*over* memory allocation/destruction/…)
    - …

More than that …



C/C++

Haskell/Python

more control,
less safety

less control,
more safety

Rust

*more control,
more safety*

What is control?

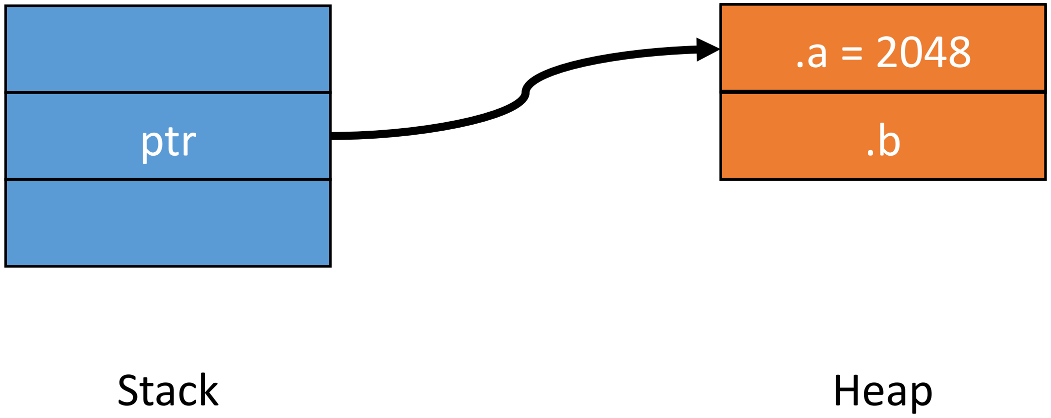
```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    ptr->a = 2048;
    free(ptr);
}
```

Precise memory layout

Lightweight reference

Deterministic destruction



Rust’s Solution: Zero-cost Abstraction

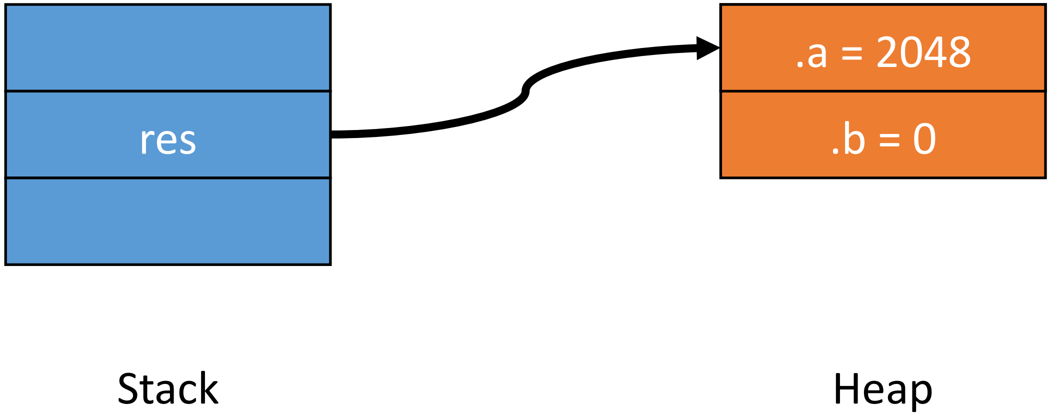
```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res: Box<Dummy> = Box::new(Dummy {
                                         a: 0,
                                         b: 0
                                     });
    res.a = 2048;
}
```

Memory allocation

Variable binding

Resource owned by res *is freed automatically*



Side Slide: Type Inference

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res~~: Box<Dummy> ~~= Box::new(Dummy {
                                a: 0,
                                b: 0
                            });
    res.a = 2048;
}
```

What is safety?

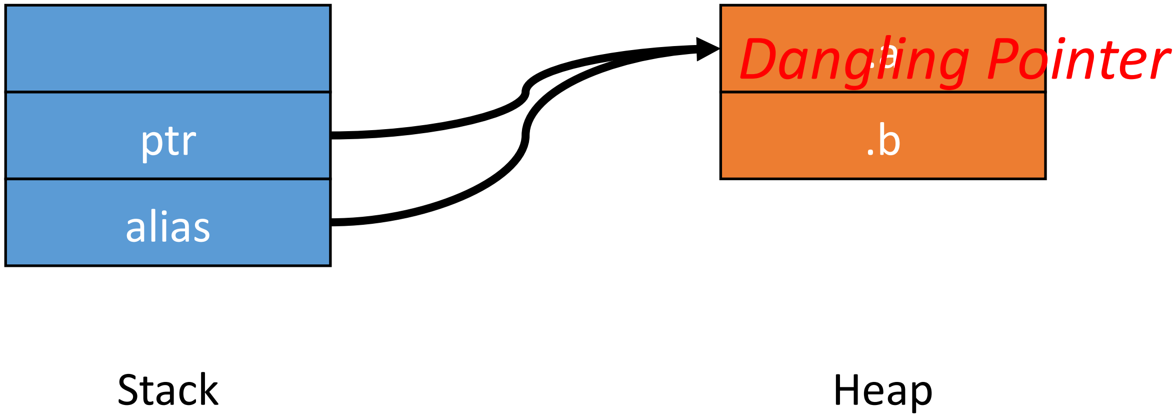
```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    Dummy *alias = ptr;
    free(ptr);
    int a = alias.a;
    free(alias);
}
```

Use after free

Double free

Aliasing + Mutation

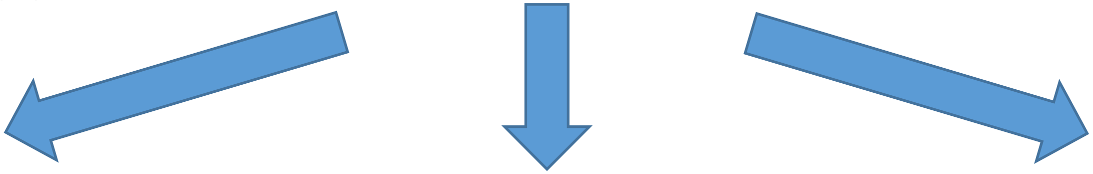


Rust’s Solution: Ownership & Borrowing

~~Aliasing~~ + *~~Mutation~~*

Compiler enforces:

- Every resource has a unique *owner*.
- Others can *borrow* the resource from its owner.
- Owner *cannot* free or mutate its resource while it is borrowed.



No need for runtime

Memory safety

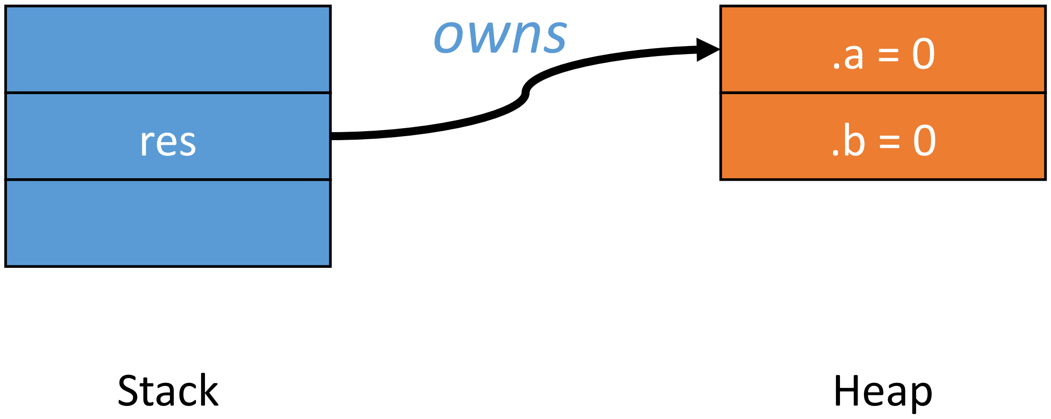
Data-race freedom

Ownership

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                      a: 0,
                      b: 0
                  });
}
```

res is out of scope and its resource is ***freed*** *automatically*



Ownership: Lifetime

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res: Box<Dummy>;
    {
        res = Box::new(Dummy {a: 0, b: 0});
    }
    res.a = 2048;
}
```

Lifetime that res owns the resource.

Compiling Error: res no longer owns the resource

- Lifetime is determined and checked statically.

Ownership: Unique Owner

~~Aliasing~~ + *Mutation*

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                    a: 0,
                    b: 0
                 });
    take(res);
    println!(“res.a = {}”, res.a);
}
```

Compiling Error!

Ownership is moved from res to arg

```
fn take(arg: Box<Dummy>) {
}
```

arg is out of scope and the resource is freed automatically

Immutable/Shared Borrowing (&)

Aliasing ✚ ~~Mutation~~

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{
                      a: 0,
                      b: 0
                  });
    take(&res);
    res.a = 2048;
}
```

Resource is returned from arg to res

Resource is immutably borrowed by arg from res

```
fn take(arg: &Box<Dummy>) {
    arg.a = 2048;
}
```

Compiling Error: Cannot mutate via an immutable reference

Resource is still owned by res. No free here.

Immutable/Shared Borrowing (&)

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{a: 0, b: 0});
    {
        let alias1 = &res;
        let alias2 = &res;
        let alias3 = alias2;
        ~~res.a = 2048;~~
    }
    res.a = 2048;
}
```

- Read-only sharing

Mutable Borrowing (&mut)

~~Aliasing~~ ✚ *Mutation*

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{a: 0, b: 0});

    take(&mut res);
    res.a = 4096;

    let borrower = &mut res;
    ~~let alias    = &mut res;~~
}

fn take(arg: &mut Box<Dummy>) {
    arg.a = 2048;
}
```

Mutably borrowed by arg from res

Multiple mutable borrowings are disallowed

Returned from arg to res

Side Slide: Mutability

- Every resource in Rust is immutable by default.
- mut is used to declare a resource as mutable.

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let res = Box::new(Dummy{a: 0, b: 0});

    res.a = 2048;

    let borrower = &mut res;
}
```

Error: Resource is immutable

Error: Cannot get a mutable borrowing of an immutable resource

Concurrency & Data-race Freedom

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {a: 0, b: 0});

    std::thread::spawn(move || {
        let borrower = &mut res;
        borrower.a += 1;
    });

    res.a += 1;
}
```

Spawn a new thread

res is mutably borrowed

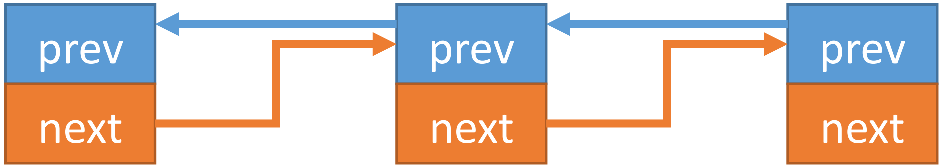
Error: res is being mutably borrowed

Unsafe

Life is hard.

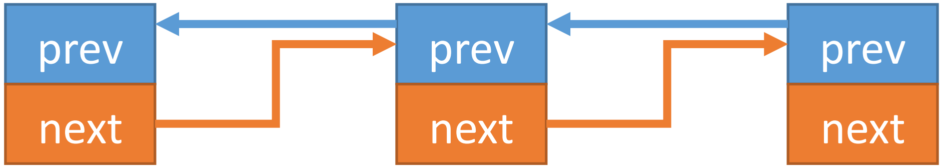
Mutably Sharing

- Mutably sharing is *inevitable* in the real world.
- Example: mutable doubly linked list



```
struct Node {
    prev: option<Box<Node>>,
    next: option<Box<Node>>
}
```

Rust’s Solution: Raw Pointers



```
struct Node {
    prev: option<Box<Node>>,
    next: *mut Node
}
```

Raw pointer

- Compiler does *NOT* check the memory safety of most operations *wrt.* raw pointers.
- Most operations *wrt.* raw pointers should be encapsulated in a *unsafe* {} syntactic structure.

Rust’s Solution: Raw Pointers

```
let a = 3;

unsafe {
    let b = &a as *const u32 as *mut u32;
    *b = 4;
}

println!(“a = {}”, a);
```

I know what I’m doing

Print “a = 4”

Foreign Function Interface (FFI)

- All foreign functions are unsafe.

```
extern {
    fn write(fd: i32, data: *const u8, len: u32) -> i32;
}

fn main() {
    let msg = b”Hello, world!\n”;
    unsafe {
        write(1, &msg[0], msg.len());
    }
}
```

Inline Assembly

```
#![feature(asm)]
fn outl(port: u16, data: u32) {
    unsafe {
        asm!(“outl %0, %1”
             :
             : “a” (data), “d” (port)
             :
             : “volatile”);
    }
}
```

Other Goodies

Enums, Pattern Match, Generic, Traits, Tests, …

Enums

- First-class
  - Instead of integers (C/C++)

- Structural
  - Parameters
  - Replacement of **union** in C/C++

Enums

```
enum RetInt {
    Fail(u32),
    Succ(u32)
}

fn foo_may_fail(arg: u32) -> RetInt {
    let fail = false;
    let errno: u32;
    let result: u32;
    ...
    if fail {
        RetInt::Fail(errno)
    } else {
        RetInt::Succ(result)
    }
}
```

Enums: No Null Pointers

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

struct SLStack {
    top: Option<Box<Slot>>
}

struct Slot {
    data: Box<u32>,
    prev: Option<Box<Slot>>
}
```

Pattern Match

```
let x = 5;

match x {
    1              => println!(“one”),
    2              => println!(“two”),
    3|4             => println!(“three or four”),
    5 ... 10        => println!(“five to ten”),
    e @ 11 ... 20 => println!(“{}”, e);
    _              => println!(“others”),
}
```

Compiler enforces the matching is complete

Pattern Match

```
let x = Dummy{ a: 2048, b: 4096 };

match x {
    Dummy{ a: va, b: vb } => va + vb,
}

match x {
    Dummy{ a: va, .. } => println!(“a={}”, va),
}
```

Pattern Match

```
enum RetInt {
    Fail(u32),
    Succ(u32)
}

fn foo_may_fail(arg: u32) -> RetInt {
    ...
}

fn main() {
    match foo_may_fail(2048) {
        Fail(errno) => println!(“Failed w/ err={}”,
                                 errno),

        Succ(result) => println!(“Result={}”, result),
    }
}
```

Pattern Match

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

struct SLStack {
    top: Option<Box<Slot>>
}

fn is_empty(stk: &SLStack) -> bool {
    match stk.top {
        None     => true,
        Some(..) => false,
    }
}
```

Generic

```
struct SLStack<T> {
    top: Option<Box<Slot<T>>>
}

struct Slot<T> {
    data: Box<T>,
    prev: Option<Box<Slot<T>>>
}

fn is_empty<T>(stk: &SLStack<T>) -> bool {
    match stk.top {
        None     => true,
        Some(..) => false,
    }
}
```

Traits

- More generic
- Typeclass in Haskell

Traits

Type implemented this trait

Object of the type implementing this trait

```
trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
    fn pop(&mut self) -> Option<Box<T>>;
}

impl<T> Stack<T> for SLStack<T> {
    fn new() -> SLStack<T> {
        SLStack{ top: None }
    }

    fn is_empty(&self) -> bool {
        match self.top {
            None     => true,
            Some(..) => false,
        }
    }
}
```

Traits

```
trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
    fn pop(&mut self) -> Option<Box<T>>;
}

fn generic_push<T, S: Stack<T>>(stk: &mut S,
                     data: Box<T>) {
    stk.push(data);
}

fn main() {
    let mut stk = SLStack::<u32>::new();
    let data = Box::new(2048);
    generic_push(&mut stk, data);
}
```

Traits

```
trait Clone {
    fn clone(&self) -> Self;
}

impl<T> Clone for SLStack<T> {
    ...
}

fn immut_push<T, S: Stack<T>+Clone>(stk: &S, data: Box<T>) -> S {
    let mut dup = stk.clone();
    dup.push(data);
    dup
}

fn main() {
    let stk = SLStack::<u32>::new();
    let data = Box::new(2048);
    let stk = immut_push(&stk, data);
}
```

Tests

- Rust provides a builtin test system.

Tests

Testing annotation

```
#[test]
fn test_pop_empty_stack() {
    let stk = SLStack::<u32>::new();
    assert!(stk.pop() == None);
}
```

Passed

```
$ rustc --test slstack.rs; ./slstack
running 1 test
test test_pop_empty_stack ... ok

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

Tests

Testing annotation

```
#[test]
fn test_pop_empty_stack() {
    let stk = SLStack::<u32>::new();
    assert!(stk.pop() == None);
}
```

Failed

```
$ rustc --test slstack.rs; ./slstack
running 1 test
test test_pop_empty_stack … FAILED

--- test_pop_empty_stack stdout ---
        thread ‘test_pop_empty_stack’ panicked at ‘assertion failed: stk.pop() == None’,
slstack.rs: 4

failures:
    test_pop_empty_stack

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

Documentation Tests

````
/// # Examples
/// ```
/// let stk = SLStack::<u32>::new();
/// assert!(stk.pop() == None);
/// ```
fn pop(&mut self) -> Option<Box<T>> {
    ...
}
````

Passed

```
$ rustdoc --test slstack.rs; ./slstack
running 1 test
test test_pop_empty_stack_0 ... ok

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

Others

- Closures
- Concurrency
- Comments as documentations
- Hygienic macro
- Crates and modules
- Cargo: Rust’s package manager
- …

Learning & Development Resourc es

Official Resources

- Rust website: http://rust-lang.org/
- Playground: https://play.rust-lang.org/
- Guide: https://doc.rust-lang.org/stable/book/
- Documents: https://doc.rust-lang.org/stable/
- User forum: https://users.rust-lang.org/
- Dev forum: https://internals.rust-lang.org/
- Source code: https://github.com/rust-lang/rust
- IRC: server: *irc.mozilla.org*, channel: *rust*
- Cargo: https://crates.io/

3rd Party Resources

- Rust by example: http://rustbyexample.com/
- Reddit: https://reddit.com/r/rust
- Stack Overflow: https://stackoverflow.com/questions/tagged/rust

Academic Research

- https://doc.rust-lang.org/stable/book/academic-research.html

Projects

- rustc: Rust compiler
  - https://github.com/rust-lang/rust
- Cargo: Rust's package manager
  - https://github.com/rust-lang/cargo
- Servo: Experimental web browser layout engine
  - https://github.com/servo/servo
- Piston: A user friendly game engine
  - https://github.com/PistonDevelopers/piston
- Iron: An extensible, concurrent web framework
  - https://github.com/iron/iron
- On Github
  - https://github.com/trending?l=rust

Development Environment

- Microsoft Visual Studio
  - Rust plugin: https://visualstudiogallery.msdn.microsoft.com/c6075d2f-8864-47c0-8333-92f183d3e640

- Emacs
  - rust-mode: https://github.com/rust-lang/rust-mode
  - racer: https://github.com/phildawes/racer
  - flycheck-rust: https://github.com/flycheck/flycheck-rust

- Vim
  - rust.vim: https://github.com/rust-lang/rust.vim
  - racer: https://github.com/rust-lang/rust.vim

Questions?