Intro to Rust

Florian “Florob” Zeitz

2025-08-06

What is Rust

- systems programming language
- compiled
- strongly, statically typed
- low-level access, high-level abstractions
- "curly-braced" (C-like)
- large community
- multi-paradigm
- inspired by: C++, Erlang, Haskell, OCaml, Swift, ...

History (up to 1.0)

from 2006	personal project of Graydon Hoare, compiler in OCaml
since 2009	development supported by Mozilla, as part of Mozilla Research
since 2011	self-hosting
since 2014	language changes through RFC process
May 2015	release of Rust 1.0

History (after 1.0)

2019	async-await syntax and futures
2020	Mozilla layoffs affecting Rust
since 2021	Rust Foundation
2018, 2021, 2024	new Rust “editions”

Philosophy

- memory-safe
  - no use-after-free
  - no out-of-bounds accesses
- no data races
- no runtime or mandatory garbage collector
- explicit costs
- enforce handling error conditions
- immutable by default
- zero-cost abstractions

Fields of Application

- CLI tools
- Web Services
- WebAssembly
- Embedded Systems
- ...

Nullable Types


```
let matches = some_string.match(/spam/)[0]
// Might throw if `match()` returns `null`
let match = matches[0]
```

```
int *x = malloc(sizeof(*x));
// Undefined behavior if `malloc()` returned `NULL`
*x = 12,
```

Option


```
1 enum Option<T> {
2     None,
3     Some(T)
4 }
```

- used instead of NULL-pointers, nil-objects, etc.
- cf. Haskell's Maybe monad

Option


```
1 match some_string.find("spam") {
2     Some(idx) => println!("Spam at {idx}"),
3     None => println!("No spam")
4 }
```

- forces explicit handling of the `None` case

Rust Enums


```rust
// C-like
enum Dir {
    North,
    East,
    South,
    West
}
let d = Dir::East;
```

```rust
// with associated data
enum Shape {
    Rect { x: f32, y: f32 },
    Circle(f32),
}
let c = Shape::Circle(23.0);
```

- sum type
- similar to a tagged union in C

Hindley-Milner-Style Type Inference


```cpp
auto v = std::vector<double>();
v.push(3.0);
```

```rust
let mut v = Vec::new();
v.push(3.0);
```

- type information can flow backwards
- types can be derived from future calls
- allows leaving of most type annotations
- only functions have to be fully annotated

match


```rust
let x = 2u32;

match x {
    1 => "One",
    2 | 3 => "Twree",
    5..9 => "Large small number",
    _ => "Fallthrough"
}
```

match


```rust
let d = Dir::East;

match d {
    Dir::North => println!("Northwards!")
    Dir::East => println!("Go East!")
    Dir::South => println!("Southwards!")
    Dir::West => println!("Go West!")
}
```

match


```rust
let c = Shape::Rect { x: 3.0, y: 4.0 };

match c {
    Shape::Rect { x, y } => println!("{x} x {y}"),
    Shape::Circle(r) => println!("{r}"),
}
```

`let` *and assignment*


```rust
// Extracting fields
let Person { age, name } = marv;
let (x, y) = point;

// With renaming
let Person { age: edad, name: nombre } = marv;

// Swap variables
(x, y) = (y, x);
```

if let


```
1 if let Ok(dir) = std::env::var("HOME") {
2     println!("Home directory is {}", dir);
3 }
```

Iterators

- implemented with an associated function
  ```
  fn next(&mut self) -> Option<Item>
  ```
- `for`-loops are syntactic sugar for repeatedly calling `next()` until it returns `None`
- lots of "adapters" for functional-style programming

Example: Iterator Adaptors


```rust
fn main() {
    let mult = (1u32..)
        .filter(|x| (1..=10).all(|y| x.is_multiple_of(y)))
        .next()
        .unwrap();
    println!("{}", mult);
}
```

- finds the smallest number evenly divisible by every number from 1 through 10

Why?

- systems programming can be scary
- a lot of bugs concern memory safety and data races
- (most) systems language do not protect against them
- Rust's ownership model rules out these classes of bugs

C++: Realloc


```cpp
#include <iostream>
#include <vector>
#include <string>

int main() {
    auto v = std::vector<std::string> { "Foo" };

    std::cout << "Capacity: " << v.capacity() << '\n';
    auto const &x = v[0];
    v.emplace_back("Bar");
    std::cout << x << '\n';
}
```

C++: Realloc


```
$ clang++ -Wall -O1 vector.cc -o vector-cc
$ ./vector-cc
Capacity: 1
Segmentation fault (core dumped)
```

C++: Iterator Invalidation


```cpp
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::vector<std::string> v = { "F", "o", "o" };

    for (auto const &it : v) {
        v.push_back(it + it);
    }
    for (auto const &it : v) {
        std::cout << it << '\n';
    }
}
```

C++: Iterator Invalidation


```
$ clang++ -Wall iter.cc -o iter-cc
$ ./iter-cc
F
o
o
FF


$
```

C++: Use After Free


```cpp
#include <iostream>
#include <memory>

int& f() {
    auto i = std::make_unique<int>(42);
    auto &i_ref = *i;
    return i_ref;
}

int main() {
    int &i = f();
    std::cout << i << '\n';
}
```

C++: Use After Free


```
$ clang++ -Wall after-free.cc -o after-free
$ ./after-free
-662549570
$
```

C++: Type punning


```
3  struct Foo {
4      int field;
5  };
6
7  void f(Foo &foo, float const *x) {
8      auto a = *x + 42.0;
9      foo.field = 0x7fffffff;
10     auto b = *x + 42.0;
11     std::cout << a << ' ' << b << '\n';
12 }
13
14 int main() {
15     Foo foo { 12 };
16     f(foo, reinterpret_cast<float*>(&foo.field));
17 }
```

C++: Type punning


```
$ clang++ -Wall field.cc -o field-cc
$ ./field-cc
42 nan
$ clang++ -Wall -O1 field.cc -o field-cc
$ ./field-cc
42 42
```

Observation

Problems arise when combining:

Mutability

```
v.emplace_back(...)
v.push_back(...)
foo.field = 0x7fffffff;
```

+

Aliasing

```
auto const &x = v[0]
auto const &it : v
foo, &foo.field
```

Rust prevents this by ensuring an object is never mutable and aliased.

Ownership: Bindings


```
1  struct Crop;
2
3  fn main() {
4      let c = Crop;
5
6      // moves c to _miller1
7      let _miller1 = c;
8
9      // error: use of moved value: `c`
10     let _miller2 = c;
11 }
```

Ownership: Functions


```
1  struct Crop;
2  struct Flour;
3
4  fn grind(_c: Crop) -> Flour {
5      Flour
6      // _c is freed here
7  }
8
9  fn main() {
10     let c = Crop;
11
12     grind(c); // c moves into grind()
13     // error: use of moved value: `c`
14     grind(c);
15 }
```

Returning Ownership


```
1  struct Book { page: u32 }
2
3  fn read(b: Book) -> Book {
4      println!("I read page {}", b.page);
5      b
6  }
7
8  fn main() {
9      let b = Book { page: 1 };
10     // b moves into `read()`
11     let b1 = read(b);
12     // error: use of moved value: `b`
13     // let b2 = read(b);
14     let _b2 = read(b1);
15 }
```

Shared Borrow


```
1  struct Book { page: u32 }
2
3  fn read(b: &Book) {
4      println!("I read page {}", b.page);
5  }
6
7  fn main() {
8      let b = Book { page: 1 };
9      let l = &b;
10
11     read(&b);
12     read(l);
13     read(&b);
14 }
```

Mutable Borrow


```
7  fn turn_page(b: &mut Book) { b.page += 1; }
8
9  fn main() {
10     let mut b = Book { page: 1 };
11
12     read(&b);
13     turn_page(&mut b);
14     read(&b);
15
16     let l = &b;
17     // turn_page(&mut b); // error: cannot borrow `b` as
18                           // mutable because it is also
19                           // borrowed as immutable
20     read(l);
21 }
```

Exception: Copy Types


```
1  struct Dress;
2  #[derive(Copy, Clone)]
3  struct Mp3;
4
5  fn main() {
6      let shop_dress = Dress;
7      let _your_dress = shop_dress;
8      // error: use of moved value: `shop_dress`
9      let _their_dress = shop_dress;
10
11     let shop_mp3 = Mp3;
12     let _your_mp3 = shop_mp3;
13     // This is fine
14     let _their_mp3 = shop_mp3;
15 }
```

Summary

- Ownership: `T`
  - one owner
  - readable
  - mutable[†]
  - can be moved or borrowed

Summary

- Ownership: `T`
  - one owner
  - readable
  - mutable†
  - can be moved or borrowed
- Shared borrow: `&T`
  - arbitrarily shareable (aliasing)
  - readable
  - immutable

Summary

- Ownership: `T`
  - one owner
  - readable
  - mutable†
  - can be moved or borrowed
- Shared borrow: `&T`
  - arbitrarily shareable (aliasing)
  - readable
  - immutable
- Mutable borrow: `&mut T`
  - only one at a time
  - readable
  - mutable

1 What is Rust

2 Features

3 Ownership & Borrowing

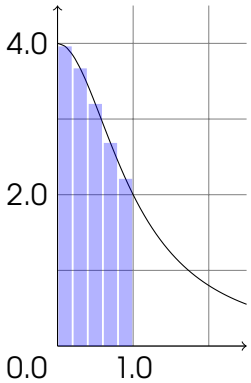
4 Data Races
- Free-Threaded Python Example
- C++ Example
- Rust Example

5 Unsafe Rust

Example: Calculating π

$$\pi = 4 \arctan(1) = \int_0^1 \frac{4}{1+x^2} \, \mathrm{d}x$$

- calculate π by Riemann integration
- approximate the area with thin rectangles
- embarrassingly parallel



Free-Threaded Python: Data Race


```python
from threading import Thread

NUM_THREADS = 4
NUM_STEPS = 100_000_000
THREAD_STEPS = NUM_STEPS // NUM_THREADS
STEP = 1.0 / NUM_STEPS

pi = 0.0

def threadFunc(lower, upper):
    global pi
    for j in range(lower, upper):
        x = (j + 0.5) * STEP
        pi += 4.0 / (1.0 + x * x) * STEP
```

Free-Threaded Python: Data Race


```
16 threads = []
17 for i in range(NUM_THREADS):
18     lower = THREAD_STEPS * i
19     upper = THREAD_STEPS * (i + 1)
20     t = Thread(target=threadFunc, args=(lower, upper))
21     threads.append(t)
22
23 for t in threads:
24     t.start()
25
26 for t in threads:
27     t.join()
28
29 print(f"Pi = {pi}")
```

Free-Threaded Python: Data Race


```
1 $ python3.13t pi.py
2 Pi = 1.0351948517209917
3 $ python3.13t pi.py
4 Pi = 1.0590036387978847
```

- thread A reads `pi = 0.1423`
- thread B reads `pi = 0.1423`
- thread A writes `pi = 0.7609`
- thread B writes `pi = 0.5768`
- `pi = 0.5768`, thread A's calculation is lost
- this is a classical data race

C++: Data Race


```cpp
#include <cstdint>
#include <print>
#include <thread>
#include <vector>

constexpr uint64_t NUM_THREADS = 4;
constexpr uint64_t NUM_STEPS = 100'000'000;
constexpr uint64_t THREAD_STEPS = NUM_STEPS / NUM_THREADS;
constexpr double STEP = 1.0 / NUM_STEPS;

void thread_fn(double *pi, uint64_t lower, uint64_t upper) {
  for (uint64_t j = lower; j < upper; ++j) {
    double x = (j + 0.5) * STEP;
    *pi += 4.0 / (1.0 + x * x) * STEP;
  }
}
```

C++: Data Race


```
18  int main() {
19    double pi = 0;
20
21    std::vector<std::thread> threads;
22
23    for (uint64_t i = 0; i < NUM_THREADS; ++i) {
24      uint64_t lower = THREAD_STEPS * i;
25      uint64_t upper = THREAD_STEPS * (i + 1);
26      threads.emplace_back(thread_fn, &pi, lower, upper);
27    }
```

C++: Data Race


```
29    for (auto &t : threads)
30      t.join();
31
32    std::println("Pi = {:.10}", pi);
33
34    return 0;
35  }
```

C++: Data Race


```
$ clang++ -Wall -std=c++23 pi.cc -o pi-cc
$ ./pi-cc
Pi = 1.156130797
$ ./pi-cc
Pi = 1.099799814
```

- thread A reads `pi = 0.1423`
- thread B reads `pi = 0.1423`
- thread A writes `pi = 0.7609`
- thread B writes `pi = 0.5768`
- `pi = 0.5768`, thread A's calculation is lost
- this is a classical data race

A Naïve Port


```rust
use std::thread;

const NUM_THREADS: u32 = 4;
const NUM_STEPS: u32 = 100_000_000;
const THREAD_STEPS: u32 = NUM_STEPS / NUM_THREADS;
const STEP: f64 = 1.0 / NUM_STEPS as f64;

fn thread_fn(pi_ref: &mut f64, lower: u32, upper: u32) {
    for x in lower..upper {
        let x = (f64::from(x) + 0.5) * STEP;
        *pi_ref += 4.0 / (1.0 + x * x) * STEP;
    }
}

```

A Naïve Port


```
15  fn main() {
16      let mut pi: f64 = 0.0;
17
18      let mut guards = Vec::new();
19      for i in 0..NUM_THREADS {
20          let lower = THREAD_STEPS * i;
21          let upper = THREAD_STEPS * (i + 1);
22          let pi_ref = &mut pi;
23          guards.push(thread::spawn(
24              move || thread_fn(pi_ref, lower, upper)
25          ));
26      }
27
```

A Naïve Port


```
28          for g in guards {
29              g.join().unwrap();
30          }
31
32          println!("Pi = {:.10}", pi);
33      }
```

A Naïve Port


```
error[E0597]: `pi` does not live long enough
  --> code/pi/pi.rs:22:22
   |
16 |       let mut pi: f64 = 0.0;
   |           ------ binding `pi` declared here
...
22 |           let pi_ref = &mut pi;
   |                        ^^^^^^^ borrowed value does not live long enough
23 |           guards.push(thread::spawn(
   |  _____________________-
24 | |             move || thread_fn(pi_ref, lower, upper)
25 | |         ));
   | |_________- argument requires that `pi` is borrowed for `'static`
...
33 |   }
```

Spawned threads could live past `main()`.
The typesystem does not know they are joined.
Therefore, borrowed data needs to live indefinitely.

Adding Some Scope


```
16  let mut pi: f64 = 0.0;
17
18  thread::scope(|scope| {
19      for i in 0..NUM_THREADS {
20          let lower = THREAD_STEPS * i;
21          let upper = THREAD_STEPS * (i + 1);
22          let pi_ref = &mut pi;
23          scope.spawn(move || thread_fn(pi_ref, lower, upper));
24      }
25  });
```

Adding Some Scope


```
error[E0499]: cannot borrow `pi` as mutable more than once at a time
  --> code/pi/pi-scoped.rs:22:36
   |
18 |     thread::scope(|scope| {
   |                    ----- has type `&'1 Scope<'1, '_>`
...
22 |             let pi_ref: &mut f64 = &mut pi;
   |                                    ^^^^^^^ `pi` was mutably borrowed here
   |                                            in the previous iteration of the loop
23 |             scope.spawn(move || thread_fn(pi_ref, lower, upper));
   |             ------------------------------------------------- argument requires that `pi`
   |                                                                 is borrowed for `'1`
```

We can only have one mutable borrow at a time, not one per thread.
This effectively makes the data race impossible.

Adding a Mutex


```
9  fn thread_fn(pi_ref: &Mutex<f64>, lower: u32, upper: u32) {
10     for x in lower..upper {
11         let x = (f64::from(x) + 0.5) * STEP;
12         *pi_ref.lock().unwrap()
13             += 4.0 / (1.0 + x * x) * STEP;
14     }
15 }
16
```

Adding a Mutex


```
17  fn main() {
18      let pi: Mutex<f64> = Mutex::new(0.0);
19
20      thread::scope(|scope| {
21          for i in 0..NUM_THREADS {
22              let lower = THREAD_STEPS * i;
23              let upper = THREAD_STEPS * (i + 1);
24              let pi_ref = &pi;
25              scope.spawn(
26                  move || thread_fn(pi_ref, lower, upper)
27              );
28          }
29      });
```

Adding a Mutex


```
31      println!("Pi = {:.10}", pi.lock().unwrap());
32  }
```

```
1   $ ./pi-locked
2   Pi = 3.1415926536
```

A Nicer Solution


```
8  fn thread_fn(lower: u32, upper: u32) -> f64 {
9      (lower..upper)
10         .map(|x| {
11             let x = (f64::from(x) + 0.5) * STEP;
12             4.0 / (1.0 + x * x) * STEP
13         })
14         .sum()
15 }
16
```

A Nicer Solution


```
17  fn main() {
18      let mut guards = Vec::new();
19      for i in 0..NUM_THREADS {
20          let lower: u32 = THREAD_STEPS * i;
21          let upper: u32 = THREAD_STEPS * (i + 1);
22          guards.push(thread::spawn(move || thread_fn(lower, upper)));
23      }
24      let pi: f64 = guards.into_iter().map(|t| t.join().unwrap()).sum();
25
26      println!("Pi = {:.10}", pi);
27  }
```

Unsafe Rust

- borrowing rules impose restrictions making some things impossible to express
- **`unsafe`** allows some additional things
  - calling functions marked **`unsafe`**
  - FFI calls
  - dereference arbitrary pointers
- keeps regular language semantics in place
- used to create safe abstractions

Unsafe Rust


```rust
use std::mem::{self, MaybeUninit};

let data = {
    let mut data: [MaybeUninit<Vec<u32>>; 1000] = unsafe {
        MaybeUninit::uninit().assume_init()
    };

    for elem in &mut data[..] {
        elem.write(vec![42]);
    }

    unsafe { mem::transmute::<_, [Vec<u32>; 1000]>(data) }
};
```

Resources

- The Book
- Rustlings
- Rust by Example

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2025-08-06-rust-intro.pdf