Intro to Rust

Florian “Florob” Zeitz

2024-06-04

What is Rust

- systems programming language
- compiled
- strongly, statically typed
- affine type system
- low-level access, high-level abstractions
- C-like syntax
- 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	new Rust “editions”

Philosophy

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

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


```
1 $ clang++ -Wall vector.cc -o vector-cc
2 $ ./vector-cc
3 Capacity: 1
4 Segmentation fault (core dumped)
```

C++: Iterator Invalidation


```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  int main() {
6      std::vector<std::string> v = { "F", "o", "o" };
7
8      for (auto const &it : v) {
9          v.push_back(it + it);
10     }
11     for (auto const &it : v) {
12         std::cout << it << '\n';
13     }
14 }
```

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


```
1 $ clang++ -Wall field.cc -o field-cc
2 $ ./field-cc
3 42 nan
4 $ clang++ -Wall -O field.cc -o field-cc
5 $ ./field-cc
6 42 42
```

Observation

Problems arise when combining:

Mutability + **Aliasing**

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

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

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

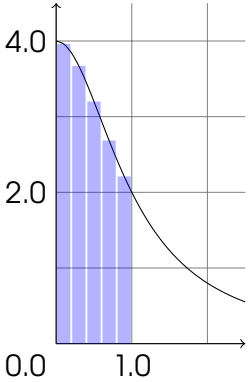
Summary

- 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

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



C++: Data Race


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

int main() {
  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;

```

C++: Data Race


```
12  double pi = 0;
13
14  std::vector<std::thread> threads;
15
16  for (int i = 0; i < NUM_THREADS; ++i) {
17    uint64_t lower = THREAD_STEPS * i;
18    uint64_t upper = THREAD_STEPS * (i + 1);
19    threads.emplace_back([=, &pi]() {
20      for (uint64_t j = lower; j < upper; ++j) {
21        double x = (j + 0.5) * STEP;
22        pi += 4.0 / (1.0 + x * x) * STEP;
23      }
24    });
25  }
```

C++: Data Race


```
26
27      for (auto &t : threads)
28        t.join();
29
30      std::cout.precision(10);
31      std::cout << "Pi = " << pi << '\n';
32
33      return 0;
34  }
```

C++: Data Race


```
1 $ clang++ -lpthread -Wall pi.cc -o pi-cc
2 $ ./pi-cc
3 Pi = 1.156130797
4 $ ./pi-cc
5 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


```
1  use std::thread;
2
3  fn main() {
4      const NUM_THREADS: u32 = 4;
5      const NUM_STEPS: u32 = 100_000_000;
6      const THREAD_STEPS: u32 = NUM_STEPS / NUM_THREADS;
7      const STEP: f64 = 1.0 / NUM_STEPS as f64;
8
```

A Naïve Port


```
9  let mut pi: f64 = 0.0;
10
11 let guards: Vec<_> = (0..NUM_THREADS)
12     .map(|i| {
13         let lower: u32 = THREAD_STEPS * i;
14         let upper: u32 = THREAD_STEPS * (i + 1);
15         let pi_ref: &mut f64 = &mut pi;
16         thread::spawn(move || {
17             for j in lower..upper {
18                 let x: f64 = (f64::from(j) + 0.5) * STEP;
19                 *pi_ref += 4.0 / (1.0 + x * x) * STEP;
20             }
21         })
22     })
23     .collect();
```

A Naïve Port


```
24
25      for g in guards {
26          g.join().unwrap();
27      }
28
29      println!("Pi = {:.10}", pi);
30  }
```

A Naïve Port


```
error[E0597]: `pi` does not live long enough
  --> pi.rs:15:41
   |
12 |             .map(|i| {
   |                  --- value captured here
...
15 |                 let pi_ref: &mut f64 = &mut pi;
   |                                        ^^ borrowed value does not live long enough
16 | /               thread::spawn(move || {
17 | |                   for j in lower..upper {
18 | |                       let x: f64 = (f64::from(j) + 0.5) * STEP;
19 | |                       *pi_ref += 4.0 / (1.0 + x * x) * STEP;
20 | |                   }
21 | |               })
   | |_______________- argument requires that `pi` is borrowed for `'static`
...
30 |   }
   |   - `pi` dropped here while still borrowed
```

Spawned threads could live past `main()`.
Therefore, borrowed data needs to live indefinitely.

Adding Some Scope


```
9   let mut pi: f64 = 0.0;
10
11  thread::scope(|scope| {
12      for i in 0..NUM_THREADS {
13          let lower: u32 = THREAD_STEPS * i;
14          let upper: u32 = THREAD_STEPS * (i + 1);
15          let pi_ref: &mut f64 = &mut pi;
16          scope.spawn(move || {
17              for j in lower..upper {
18                  let x: f64 = (f64::from(j) + 0.5) * STEP;
19                  *pi_ref += 4.0 / (1.0 + x * x) * STEP;
20              }
21          });
22      }
23  });
```

Adding Some Scope


```
error[E0499]: cannot borrow `pi` as mutable more than once at a time
  --> pi-scoped.rs:15:36
   |
11 |       thread::scope(|scope| {
   |                      ----- has type `&'1 Scope<'1, '_>`
...
15 |               let pi_ref: &mut f64 = &mut pi;
   |                                      ^^^^^^^ `pi` was mutably borrowed here in the previous
   ↪  iteration of the loop
16 | /             scope.spawn(move || {
17 | |                 for j in lower..upper {
18 | |                     let x: f64 = (f64::from(j) + 0.5) * STEP;
19 | |                     *pi_ref += 4.0 / (1.0 + x * x) * STEP;
20 | |                 }
21 | |             });
   | |_____________- 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


```
1  use std::sync::Mutex;
2  use std::thread;
3
4  fn main() {
5      const NUM_THREADS: u32 = 4;
6      const NUM_STEPS: u32 = 100_000_000;
7      const THREAD_STEPS: u32 = NUM_STEPS / NUM_THREADS;
8      const STEP: f64 = 1.0 / NUM_STEPS as f64;
9
```

Adding a Mutex


```
10  let pi: Mutex<f64> = Mutex::new(0.0);
11  thread::scope(|scope| {
12      for i in 0..NUM_THREADS {
13          let lower: u32 = THREAD_STEPS * i;
14          let upper: u32 = THREAD_STEPS * (i + 1);
15          let pi_ref: &Mutex<f64> = &pi;
16          scope.spawn(move || {
17              for j in lower..upper {
18                  let x: f64 = (f64::from(j) + 0.5) * STEP;
19                  *pi_ref.lock().unwrap()
20                      += 4.0 / (1.0 + x * x) * STEP;
21              }
22          });
23      }
24  });
```

Adding a Mutex


```
25
26      println!("Pi = {:.10}", pi.lock().unwrap());
27  }
```

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

A Nicer Solution


```
9  let pi: f64 = thread::scope(|scope| {
10     let guards: Vec<_> = (0..NUM_THREADS)
11         .map(|i| {
12             let lower: u32 = THREAD_STEPS * i;
13             let upper: u32 = THREAD_STEPS * (i + 1);
14             scope.spawn(move || {
15                 (lower..upper)
16                     .map(|j| {
17                         let x: f64 = (f64::from(j) + 0.5) * STEP;
18                         4.0 / (1.0 + x * x) * STEP
19                     })
20                     .sum::<f64>()
21             })
22         })
23         .collect();
24
25     guards.into_iter().map(|t| t.join().unwrap()).sum()
26 });
```

Enums


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

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

- sum type
- similar to a tagged union

Example: `Option`


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

- equivalent to Haskell's Maybe monad
  (`return = Some, bind = and_then`)
- used instead of NULL-pointers, nil-objects, etc.

Example: Option


```
1  let v: Vec<u32> = ...;
2
3  v.last()
4      .and_then(|l| if l < 3 { Some(l + 4) } else { None })
5      .and_then(|l| if l > 12 { Some(l - 4) } else { None })
6      .unwrap_or(42);
```

match


```
1  let x = 2u32;
2
3  match x {
4      1 => "One",
5      2 | 3 => "Twree",
6      5..9 => "Large small number",
7      _ => "Fallthrough"
8  }
```

match


```
1  let d = Dir::East;
2
3  match d {
4      Dir::North => println!("Northwards!")
5      Dir::East => println!("Go East!")
6      Dir::South => println!("Southwards!")
7      Dir::West => println!("Go West!")
8  }
```

match


```
1  let c = Shape::Circle { r: 1.0 };
2
3  match c {
4      Shape::Rect { x, y } => println!("{} x {}", x, y),
5      Shape::Circle { r } => println!("{}", r)
6  }
```

let


```
1 let Person { age, name } = marv;
2 let (x, y) = point;
3 let Person { age: edad, name: nombre } = marv;
```

if let


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

Function parameters


```
1 fn print_person(&Person { age, name }: &Person) {
2     println!("{}: {}", name, age);
3 }
```

Iterators

- implemented with an associated function
  `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


```
1  fn main() {
2      let mult = (1..)
3          .filter(|x| (1..11).all(|y| x % y == 0))
4          .next()
5          .unwrap();
6      println!("{}", mult);
7  }
```

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

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) }
};
```

Materials

- The Book
- Rustlings
- Rust by Example

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2024-06-05-rust-intro.pdf