Rust's Ownership Model

Florian "Florob" Zeitz

2022-08-24

Why?

- systems programming can be scary
- a lot of bugs in 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';

  return 0;
}
```

C++: Realloc


```
1 $ clang++ -std=c++17 -Wall vector.cc -o vector-cc
2 $ ./vector-cc
3 Capacity: 1
4 zsh: segmentation fault (core dumped)  ./vector-cc
```

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
15     return 0;
16 }
```

C++: Iterator Invalidation


```
$ clang++ -std=c++17 -Wall iter.cc -o iter-cc
$ ./iter-cc
F
o
o
FF


$
```

C++: Use After Free


```cpp
#include <iostream>

int& f() {
    int i = 42;
    return i;
}

void print(int &x) {
    std::cout << x << '\n';
}

int main() {
    int &i = f();
    print(i);
    return 0;
}
```

C++: Use After Free


```
$ clang++ -Wall after-free.cc -o after-free
after-free.cc:5:16: warning: reference to stack memory
↪  associated with local variable 'i' returned
↪  [-Wreturn-stack-address]
        return i;
               ^
1 warning generated.
$ ./after-free
32766
$
```

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     return 0;
18 }
```

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      let _grinder1 = c; // moves c to _grinder1
7
8      let _grinder2 = c; // error: use of moved value: `c`
9  }
```

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     grind(c); // error: use of moved value: `c`
14 }
```

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     let b1 = read(b); // b moves into `read()`
11     // let b2 = read(b); // error: use of moved value: `b`
12     let _b2 = read(b1);
13 }
```

Returning Ownership (Mutable)


```
8  fn turn_page(mut b: Book) -> Book {
9      b.page += 1;
10     b
11 }
12
13 fn main() {
14     let b = Book { page: 1 };
15
16     let b1 = read(b);
17     let b2 = turn_page(b1);
18     let _b3 = read(b2);
19 }
```

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 }
```

Intermission: 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      let _their_dress = shop_dress; // error:
9                                     // use of moved value:
10                                    // `shop_dress`
11
12     let shop_mp3 = Mp3;
13     let _your_mp3 = shop_mp3;
14     let _their_mp3 = shop_mp3; // This is fine
15 }
```

Basic lifetimes


```
1  fn main() {
2      let r;
3
4      {
5          let x = 5;
6          r = &x; // error: `x` does not live long enough
7      }
8
9      println!("r: {}", r);
10 }
```

Basic lifetimes


```
1  fn main() {
2      let r;                // --------+-- 'a
3                            //         |
4      {                     //         |
5          let x = 5;        // -+-- 'b |
6          r = &x;           //  |      |
7      }                     // -+      |
8                            //         |
9      println!("r: {}", r); //         |
10 }                         // --------+
```

- `x` must live until the last use of `r`

Returning References


```
1  struct Page;
2  struct Book {
3      page: u32,
4      content: Vec<Page>,
5  }
6
7  fn get_page<'a>(b: &'a Book, page: usize) -> &'a Page {
8      &b.content[page]
9  }
```

Lifetime Elision


```
1  struct Page;
2  struct Book {
3      page: u32,
4      content: Vec<Page>,
5  }
6
7  fn get_page(b: &Book, page: usize) -> &Page {
8      &b.content[page]
9  }
```

- if there is only one input lifetime all outputs get it
- in methods all outputs get `self`'s lifetime

Multiple Input Lifetimes


```
1  struct Book { page: u32 }
2
3  fn longer<'a>(a: &'a Book, b: &'a Book) -> &'a Book {
4      if a.page > b.page {
5          a
6      } else {
7          b
8      }
9  }
```

Rust: Realloc


```
1  fn main() {
2      let mut v: Vec<String> = Vec::new();
3
4      v.push("Foo".to_string());
5      let x: &String = &v[0];
6      v.push("Bar".to_string());
7
8      println!("{}", x);
9  }
```

Rust: Realloc


```
$ rustc vector.rs
error[E0502]: cannot borrow `v` as mutable because it is also
↪  borrowed as immutable
 --> vector.rs:6:5
  |
5 |     let x: &String = &v[0];
  |                      - immutable borrow occurs here
6 |     v.push("Bar".to_string());
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}", x);
  |                    - immutable borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.
```

Rust: Iterator Invalidation


```
1  fn main() {
2      let mut v: Vec<String> = vec!["F".into(), "o".into(),
3                                    "o".into()];
4
5      for it in &v {
6          v.push(it.clone() + &it);
7      }
8      for it in &v {
9          println!("{}", it);
10     }
11 }
```

Rust: Iterator Invalidation


```
$ rustc iter.rs
error[E0502]: cannot borrow `v` as mutable because it is also
↪  borrowed as immutable
 --> iter.rs:6:9
  |
5 |     for it in &v {
  |               --
  |               |
  |               immutable borrow occurs here
  |               immutable borrow later used here
6 |         v.push(it.clone() + &it);
  |         ^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.
```

Rust: Type punning


```
3  struct Foo {
4      field: i32,
5  }
6
7  fn f(foo: &mut Foo, x: &f32) {
8      let a = *x + 42.0;
9      foo.field = 0x7fffffff;
10     let b = *x + 42.0;
11     println!("{a} {b}");
12 }
13
14 fn main() {
15     let mut foo = Foo { field: 12 };
16     f(&mut foo, unsafe { mem::transmute(&foo.field) });
17 }
```

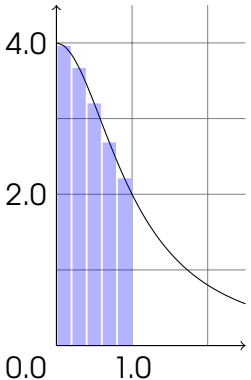
Rust: Type punning


```
$ rustc field.rs
error[E0502]: cannot borrow `foo.field` as immutable because it is
              also borrowed as mutable
  --> field.rs:16:41
   |
16 |     f(&mut foo, unsafe { mem::transmute(&foo.field) });
   |     - --------                           ^^^^^^^^^^ immutable
   |     | |                                    borrow occurs here
   |     | |
   |     | mutable borrow occurs here
   |     mutable borrow later used by call

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.
```

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 by thin rectangles
- embarrassingly parallel



C++: Data Race


```
1  #include <cstdint>
2  #include <iostream>
3  #include <thread>
4  #include <vector>
5
6  int main() {
7    constexpr uint64_t NUM_THREADS = 4;
8    constexpr uint64_t NUM_STEPS = 100000;
9    constexpr uint64_t THREAD_STEPS = NUM_STEPS / NUM_THREADS;
10   constexpr double STEP = 1.0 / NUM_STEPS;
11
```

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

C++: Data Race


```
1 $ clang++ -std=c++17 -lpthread -Wall pi.cc -o pi-cc
2 $ ./pi-cc
3 Pi = 1.156130797
4 $ ./pi-cc
5 Pi = 1.099799814
```

- classical data race
- 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

Calculating π

Lifecoding

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2022-08-24-rust-ownership.pdf