Secure Software Development

Defensive III (Rust)

Lukas Prokop

02.12.2020

Winter 2020/21, www.iaik.tugraz.at/ssd

Lukas Prokop

- PhD student at IAIK
- post-quantum cryptography
- speaker at RustGraz

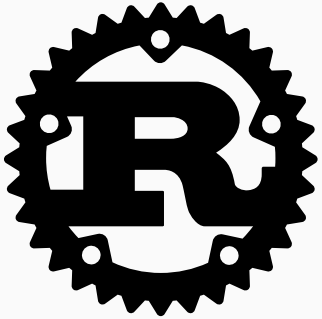
1. Introduction to rust
2. Pattern matching and error handling
3. Arrays
4. References
5. Mutation xor aliasing
6. Ownership model and borrowing
7. unsafe superpowers
8. Does undefined behavior exist in rust?
9. Anonymous functions
10. Macros
11. typestate pattern
12. Resources

We discuss only few topics today

1. Introduction to rust
2. Pattern matching and error handling
3. Arrays
4. References
5. Mutation xor aliasing
6. Ownership model and borrowing
7. unsafe superpowers
8. Does undefined behavior exist in rust?
9. Anonymous functions
10. Macros
11. typestate pattern
12. Resources

Introduction to rust

- multi-paradigmatic
  (imperative, functional)
- systems programming language
  (easy interop with C, no GC)
- focus on memory safety and concurrency
- uses the LLVM infrastructure
- syntax similar to C++, immutability by default
- Modern competitors: Nim, Crystal, D, Zig, Go?
- 1.0 (May 2015), 1.31 (2018), 1.48 (current)



"Most loved programming language"
(Stack Overflow Developer Survey, 2016–2020)

Bjarne Stroustrup defines the concept of so-called *zero-cost abstraction*:

> *"What you don't use, you don't pay for.*
> *And further: What you do use, you couldn't hand code any better."*
> *—"The Design and Evolution of C++" (1994)*

```
u8   u16   u32   u64   u128
i8   i16   i32   i64   i128
 isize   usize   f32   f64
       bool   char
```

→ type suffix notation: `42u8`

```
42   42_000   0xFF   0o777   0b0010_1010   std::u32::MAX
1.   1e6   -4e-4f64   std::f64::INFINITY   std::f64::NAN
1usize   true   false   'c'
```

→ type inference to determine data type
→ default integer type is i32

```rust
fn square(arg: f64) -> f64 {
  arg * arg
}

fn main() {
  let a: f64 = 2.1;
  println!("the square of {} is {}", a, square(a));
}
```

```
trait Submission {
  fn len(&self) -> u32;
  fn summary(&self) -> String;
}
```

- `trait`s are inspired by Haskell typeclasses (no subtyping); like interfaces
- Nominal type system, not structural type system (like Go)
- default implementations and constant attributes can be provided
- `struct`s, `enum`s, and `union`s can implement traits
- first parameter is not `&self`? Then static method
- Implementing `std::ops::Add`? Enables `+` (operator overloading)

trait implementation

```
struct Talk {
  desc: String,
  duration: u16,
}
impl Submission for Talk {
  fn len(&self) -> u32 { self.duration as u32 }
  fn summary(&self) -> String {
    let dot = self.desc.find('.');
    match dot {
      Some(idx) => {
        let mut s = String::new();
        s.push_str(&self.desc[0..idx]);
        s.push_str(" …");
        s
      },
      None => self.desc.clone(),
} } }
```

Pattern matching and error handling

```
1  fn main() {
2    let course = "ssd";
3    println!("{} ({})",
4      match course {
5        "ssd" => "Secure Software Development",
6        _ => "unknown"
7      },
8      course.to_uppercase()
9    );
10 }
```

In computer programming, especially functional programming and type theory, an ***algebraic data type*** *(ADT) is a kind of composite type, i.e., a type formed by combining other types.*
—Wikipedia

Haskell's data and rust's enum

```haskell
data List a = Nil | Cons a (List a)
```

```haskell
data List a = Nil | Cons a (List a)
```

```rust
enum List {
  Nil,
  Cons(Box<List>, u32),
}
```

Haskell's data and rust's enum

```
data Tree = Empty
          | Leaf Int
          | Node Tree Tree
```

```haskell
data Tree = Empty
          | Leaf Int
          | Node Tree Tree
```

```rust
enum Tree {
  Empty,
  Leaf(u32),
  Node(Box<Tree>, Box<Tree>),
}
```

enum introduces an algebraic data type

- Algebraic? Sums and products.
  - `List` is the sum of `Nil` and `Cons(_)`.
  - `Cons` is the product of `Box<List>` and `u32`.
- Boxing? Avoids `recursive type `List` has infinite size`.
- Article: Algebraic data types in four languages (namely Haskell, Scala, rust, and TypeScript)

```
impl fmt::Display for List {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    match self {
      List::Cons(inner, item)
        => write!(f, "(cons {} {})", item, inner),
      List::Nil
        => write!(f, "nil"),
    }
  }
}
```

Recognize that `Cons` is addressed by `List::Cons`.

Contrived error handling


```
enum Result {
  Okay(Digest),
  Error(String),
}
```

Contrived error handling


```
enum Result {
  Okay(Digest),
  Error(String),
}

fn generate_digest() -> Result {
  Result::Okay([42u8; 32])
}
```

Contrived error handling

```
enum Result {
  Okay(Digest),
  Error(String),
}

fn generate_digest() -> Result {
  Result::Okay([42u8; 32])
}

fn main() {
  match generate_digest() {
    Result::Okay(d) => {
      for byte in d.iter() { print!("{:02X}", byte); }
      println!("");
    },
    Result::Error(msg) => eprintln!("error: {}", msg),
  }
}
```

`std::result::Result<T, E>`

- `Ok(T)`
- `Err(E)`

No exceptions, no error codes.

`std::result::Result<T, E>`

- `Ok(T)`
- `Err(E)`

No exceptions, no error codes.

`std::option::Option<T>`

- `None`
- `Some(T)`

If we fetch one element from a container data structure,
we get *some* value or *none*.

```rust
// Example for Result
match File::open("foo.txt") {
  Ok(fd) => { /* ... */ },
  Err(e) => panic!(e),
}

// Example for Some
let fetched = Some(value);
fetched.unwrap();  // return Some value or panic
fetched.unwrap_or(default_value);  // … or default
```

You usually implement error types (SyntaxError, InvalidArgError, …) on your own in your library.

```rust
// Result API (excerpt)
fn is_ok(&self) -> bool;
fn is_err(&self) -> bool;
fn ok(self) -> Option<T>;
fn err(self) -> Option<E>;
fn and_then<U, F>(self, op: F) -> Result<U, E>;

// Option API (excerpt)
fn is_some(&self) -> bool;
fn is_none(&self) -> bool;
fn unwrap(self) -> T;
fn unwrap_or(self, default: T) -> T;
fn ok_or<E>(self, err: E) -> Result<T, E>;
```

rust has a unique error handling operator

The question mark operator exits early in case of `Err` or returns the value otherwise.

```
fn compile(src: &str) -> Result<(), Error> {
  let tokens = tokenize(&src)?;
  let ast = parse(&tokens)?;
  // …
  Ok(())
}
```

Return type of function must be a corresponding `Result`.

It is can be rewritten with a match expression:

```
1  fn compile(src: &str) -> Result<(), Error> {
2    let tokens = match tokenize(&src) {
3      Err(E) => return Err(E),
4      Ok(ts) => ts,
5    };
6    let ast = parse(&tokens)?;
7    // …
8    Ok(())
9  }
```

Arrays

```rust
// declaration and initialization
let mut array: [u32; 3] = [0; 3]; // [{init-value}; {length}]

// iterate over an array
for x in array.iter() { }

// indexing and assignment
array[1] = 1;
array[2] = 2;
```

- Arrays have a known, fixed size
- Arrays need to be initialized
  - compile time checks
  - exceptions via MaybeUninit
- Memory layout: like a pointer and a length
- Few API limitations for arrays of length >32

```
1 // iterate over a slice
2 for x in array[..].iter() { }
```

- Slices are memory views into an array
- Unknown size
- Memory layout: only a pointer
- Barely useful because they cannot be passed as fn argument or return value

SSD array example

The following snippet can trigger an overflow. Where?

```
char buffer[128];
int bytesToCopy = packet.length;
if (bytesToCopy < 128) {
    strncpy(buffer, packet.data, bytesToCopy);
}
```

Example via CS 110L, Ryan Eberhardt and Armin Namavari

The following snippet can trigger an overflow. Where?

```
char buffer[128];
int bytesToCopy = packet.length;
if (bytesToCopy < 128) {
    strncpy(buffer, packet.data, bytesToCopy);
}
```

Example via CS 110L, Ryan Eberhardt and Armin Namavari

- Proper bounds check (yay!)
- `strncpy`, not `strcpy` (yay!)

Overflow snippet solved

The issue:

1. As declared, `bytesToCopy` is an `int`
2. Third argument of `strncpy` is a `size_t`
3. `bytesToCopy < 128` is true if `bytesToCopy` is negative
4. `bytesToCopy` is cast to an unsigned type and becomes huge

How is this prevented in rust?

- Types contain length (`String` is `Vec<u8>`, a `Vec` carries a `len`)
- No implicit casts (explicit casts via `as`)
- Bounds checks: arrays are sized anyways, but other container per default use bound checks

References

```
fn main() {
  let mut a = 42u32;
  let b: &u32 = &a;

  println!("value of ref: {}", *b);
}
```

- b is a [shared] reference (&u32) to a.
- Reference operator &
- Dereference operator *

```rust
fn main() {
  let mut a = 42u32;
  let b: &mut u32 = &mut a;

  *b = 2;
  println!("value of ref: {}", *b);
}
```

- `b` is a [*mutable*] reference (`&u32`) to `a`.
- Reference operator `&mut`
- Dereference operator `*`

```rust
fn main() {
  let mut a = 42u32;
  let b: &u32 = &a;

  println!("value of ref: {}", b);
}
```

Recognize that **b** does not need the dereference operator. Rust implements *auto-dereferencing*. Best practice: write derefs explicitly.

Mutation xor aliasing

Rules:

- one or more *shared* references (`&T`) to a resource
- exactly one *mutable* reference (`&mut T`)
- either or, not both! (“aliasing xor mutation”)

Benefits of reference limitations for memory safety:

- one writer XOR n readers in concurrent context
- prevents data races

Ownership and borrowing

C++ uses the notion of RAII:

```
void WriteToFile(const std::string& message) {
  static std::mutex mutex;
  std::lock_guard<std::mutex> lock(mutex);
  std::ofstream file("example.txt");
  if (!file.is_open()) {
    throw std::runtime_error("unable to open file");
  }
  file << message << std::endl;
}
```

- Each value in Rust has a variable that's called its *owner*
- There can only be one owner at a time
- Ownership can *move* from one variable to another
- When the owner goes out of scope, the value will be "dropped"

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(mut s: Stats) {
  s.score += 1;
}

fn main() {
  let a = Stats { score: 8 };
  sub(a);

}
```

Ownership example

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(mut s: Stats) {
  s.score += 1;
}

fn main() {
  let a = Stats { score: 8 };
  sub(a);
  println!("{:?}", a);
}
```

Ownership example

```
error[E0382]: borrow of moved value: `a`
  --> src/main.rs:10:20
   |
8  |     let a = Stats { score: 8 };
   |         - move occurs because `a` has type `Stats`,
   |           which does not implement the `Copy` trait
9  |     sub(a);
   |         - value moved here
10 |     println!("{}", a);
   |                    ^
   |         value borrowed here after move
```

Ownership example

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(mut s: Stats) {
  // owner of Stats instance = `s`
  s.score += 1;
  // `s` goes out of scope → Stats instance is dropped
}

fn main() {
  let a = Stats { score: 8 };
  // owner of Stats instance = `a`
  sub(a); // move Stats instance: `a` → `s`
  println!("{:?}", a); // has been dropped already → error
}
```

Solutions:

- Use `#[derive(Debug,Copy,Clone)]`. Then `sub` uses copied instance. Results in `Stats { score: 8 }`
- Return `Stats` instance and assign it again in `main`.
- Use references (*borrowing* ownership)

Benefits of ownership for memory safety:

- we can pin-point when a variable is dropped (across threads!)

Ownership example with borrowing

```rust
#[derive(Debug)]
struct Stats { score: u32 }

fn sub(s: &mut Stats) {
  s.score += 1;
}

fn main() {
  let mut a = Stats { score: 8 };
  // ownership of `a` is borrowed to `s`
  sub(&mut a);
  // ownership of `s` is returned back to `a`
  println!("{:?}", a);
}
```

unsafe superpowers

```
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn rdtsc {
    let (mut eax, mut ebx, mut ecx) = (0, 0, 0);
    unsafe {
        asm!("rdtscp",
             out("eax") eax,
             out("ecx") ecx,
             out("edx") edx);
    }
}
```

Superpowers:

1. Dereference a raw pointer (`const *`)
2. Call an `unsafe` function or method
3. Access or modify a mutable static variable
4. Implement an `unsafe` trait
5. Access fields of `unions`

Does UB exist in rust?

- Not all bugs can be caught with the type system
- A type system needs to be relaxed to be pragmatic
- A type system needs to be strict to be able to reason about it

Does undefined behavior (UB) exist in rust? Yes.

- See Behavior considered undefined for a non-exhaustive list
- Corner cases are still subject to academic debate

Anonymous functions

```
fn named(name1: T1, name2: T2) -> T_RETURN {}
let unnamed = |name1: T1, name2: T2| -> T_RETURN { };
let short   = |name1    , name2    |             { };
```

Example of anonymous function usage:

```
use std::thread;
let handler = thread::spawn(|| {
    println!("Hello World!");
});
handler.join().unwrap();
```

Macros

- Three kinds of macros
  1. function-like macros (`println!("hi")`)
  2. derive macros (`derive(Debug)`)
  3. attribute-like macros (`cfg(target_arch = "x86")`)

Important differences from C:

- They operate on tokens, not the lexical level
- Macro hygiene (variables are not visible outside)

```
macro_rules! shake {
  (update $base:ident with $($elem:expr, )*)
    => { $( $base.update($elem); )* };
}
```

Input:

```
shake!(update h with &data, &[b' '], &data2,);
```

Output:

```
h.update(&data);
h.update(&[b' ']);
h.update(&data2);
```

typestate pattern

Idea: Encode the state in the type

Example:

- `fopen` returns `FileOpened`
- `fwrite` returns `FileNonEmpty`
- `fclose` returns `FileClosed`

More details, for example, in The Typestate Pattern in Rust (blog post).

Why? Can increase security.

> *Both from a design point of view as from an implementation perspective the entire scope can be considered of exceptionally high standard. Using the type system to* ***statically encode properties such as the TLS state transition function*** *is one just one example of great defense-in-depth design decisions.*
> *—rustls formal audit report*

Resources

I mostly used the rust book.

- Learning Rust via Advent of Code
- Small exercises to get you used to reading and writing Rust code
- Rust by example
- Rust official doc
- stdlib
- Idiomatic rust
- A half hour to learn rust

clippy detects common mistakes and unidiomatic code

rustfmt allows you to reformat/normalize rust source code

There are many UNIX utilities rewritten in rust (xsv, ripgrep, etc.)

University courses on Rust:

- Rust course by Lukas Kalbertodt
- CS196 at Illinois
- CS110L at Stanford

Academia:

- RustBelt: academic project for formal verification of the Rust compiler

Thank you! Q/A?