Considering Rust for Scientific Software

Max Orok

RustFest Global 2020

Who am I?

Mechanical engineering master’s student at the University of Ottawa

Radiation modelling researcher

Contractor at Mevex, a linear accelerator manufacturer

https://github.com/mxxo

Scientific software

Written by a small team with limited time and resources

Correctness is extremely important

Performance is usually also very important

“Developers usually have other jobs”

Scientific software is usually written by people may not identify as developers first (physicists, chemists, engineers)

Programs are a means to an end

Compilation is sometimes the first and last unit test

“If it works, don’t touch it”

The Therac-25 Legacy

The Therac-25 was a radiation therapy device created by Atomic Energy of Canada

The Therac-25 was part of 6 major accidents between 1985 and 1987

Investigators found a variety of problems, but data races in the control software were part of the failure

Software bugs have real-world consequences

Scientific computing and its discontents

Python as *lingua franca*

C and C++ as bedrock supporting Python

(With apologies to Fortran, Julia and others)

An issue with the current landscape

Going from Python to C++ should be a natural step:

- Many Python libraries build on top of C++ implementations
- Researcher time is precious, many Python scripts run too slowly

Unfortunately this is often a very difficult transition

Rust is a viable alternative to C++ in this context

Why not Rust?

A relatively young language (Python is 30, C++ is 40)

Steep learning curve

You have a large codebase written in another language

An important library is missing from the ecosystem

Concerns about a single vendor

Rust aligns with my goals as a researcher

I want to write the fastest code I can, with as few bugs as possible.

How Rust helps:

1. Entire classes of bugs are eliminated
2. Speed without sacrificing productivity
3. A language explicitly designed for non-expert users
4. Built-in documentation and tests

No implicit conversions between primitive types

```rust
let x: f64 = 5 / 3;
println!("{:?}", x);
```

```
error[E0308]: mismatched types
 --> src/main.rs:2:18
  |
2 |     let x: f64 = 5 / 3;
  |            ---   ^^^^^ expected `f64`, found integer
  |            |
  |            expected due to this

error: aborting due to previous error
```

… this can be noisy but is better than bugs later on

```rust
let x: u32 = 10;
let y: usize = x;
```

```
error[E0308]: mismatched types
 --> src/main.rs:3:20
  |
3 |     let y: usize = x;
  |            -----   ^ expected `usize`, found `u32`
…
help: you can convert an `u32` to `usize` and panic if
the converted value wouldn't fit
  |
3 |     let y: usize = x.try_into().unwrap();
  |                    ^^^^^^^^^^^^^^^^^^^^^
```

Safe defaults...

```
let xs = vec![1, 2, 3];
println!("{:?}", xs[10]);
```

```
thread 'main' panicked at
'index out of bounds: the len is 3 but the index is 10'
```

Safe defaults… with opt-in low-level control

```
let xs = vec![1, 2, 3];
println!("{:?}",
  unsafe { xs.get_unchecked(10) }
);
```

```
-537096032
```

This is generally not recommended, use with caution! Calling this method with an out-of-bounds index is *undefined behavior* even if the resulting reference is not used. For a safe alternative see `get`.

Floating point numbers are treated with caution

```
match 0.1 + 0.1 + 0.1 {
    0.3 => println!("got 0.3"),
    _ => println!("got something else"),
}
```

```
got something else
```

```
warning: floating-point types cannot be used in patterns
 --> src/main.rs:3:9
  |
3 |         0.3 => println!("got 0.3"),
  |         ^^^
  |
```

… which can be a little annoying sometimes

```rust
let mut xs: Vec<f64> = vec![1.0, 12.0, 3.0, 100.0];
xs.sort();
```

```
error[E0277]: the trait bound `f64: std::cmp::Ord` is not satisfied
 --> src/main.rs:3:8
  |
3 |     xs.sort();
  |        ^^^^ the trait `std::cmp::Ord` is not implemented for `f64`
```

```rust
let mut xs: Vec<f64> = vec![1.0, 12.0, 3.0, 100.0];
xs.sort_by(|a, b| a.partial_cmp(b).unwrap());
```

Debugging and prototyping features

```rust
#[derive(Debug)]
struct CoolData {
    xs: Vec<f64>,
    data: Vec<f64>,
}

pub fn main() {
    let cd = CoolData {
        xs: vec![0.0, 1.0, 2.0],
        data: vec![10.0, 20.0, 30.0],
    };
    dbg!(cd);
}
```

```
[src/main.rs:12] cd = CoolData {
    xs: [
        0.0,
        1.0,
        2.0,
    ],
    data: [
        10.0,
        20.0,
        30.9,
    ],
}
```

Integrated testing means tests are much more likely to be written

```rust
fn some_math_expr(x: f64) -> f64 {
    16.0 * x*x + 100_000.0 * x.sin() * x.cos()
}

#[test]
fn some_math_test() {
    assert!((some_math_expr(10.0) - 47247.262536).abs() < 1e-6);
}
```

Doctests are a killer feature for scientific code

```rust
/// A very interesting math expression.
/// ```
/// assert!((some_math_expr(10.0) - 47247.262536).abs() < 1e-6);
/// ```
pub fn some_math_expr(x: f64) -> f64 {
        16.0 * x*x + 100_000.0 * x.sin() * x.cos()
}
```

```rust
pub fn some_math_expr(x: f64) -> f64
```

[-] A very interesting math expression.

```rust
assert!((some_math_expr(10.0) - 47247.262536).abs() < 1e-6);
```

Rust’s safety guarantees and solid fundamentals have a large qualitative impact on what kind of code we’re capable of writing

Take for instance, data races in multithreaded code…

From the Rustonomicon:

Data Races and Race Conditions

Safe Rust guarantees an absence of data races, which are defined as:

- two or more threads concurrently accessing a location of memory
- one of them is a write
- one of them is unsynchronized

... compared to the C++ Core Guidelines

CP.2: Avoid data races

Reason Unless you do, nothing is guaranteed to work and subtle errors will persist.

Enforcement Some is possible, do at least something. There are commercial and open-source tools that try to address this problem, but be aware that solutions have costs and blind spots. Static tools often have many false positives and run-time tools often have a significant cost. We hope for better tools. Using multiple tools can catch more problems than a single one.

The thing that sets Rust apart is that software engineering best practices are built into the language and core tools

Choosing Rust will have the biggest impact for small, resource-constrained teams who don’t identify as expert software developers

Rust's place in scientific computing

The speed and power of C++ without the sharp edges

A systems language explicitly designed to lower barriers

Companion and complement to C and C++

- There are many tradeoffs between these languages, and no "correct" choice

Rust's foundational values help us to write good software