Pawsey

Rust for Scientific Computing

Dr Emily Kahl, Pawsey Supercomputing Research Centre

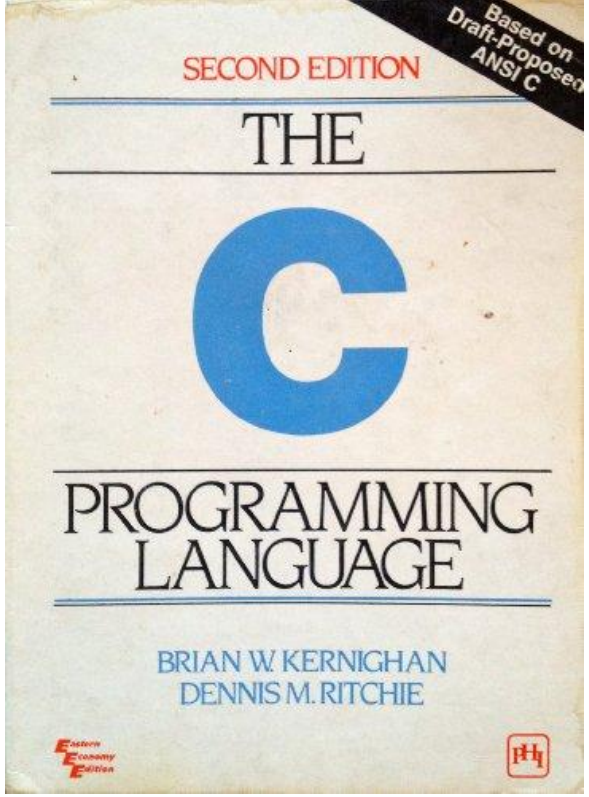
Principles of scientific code?

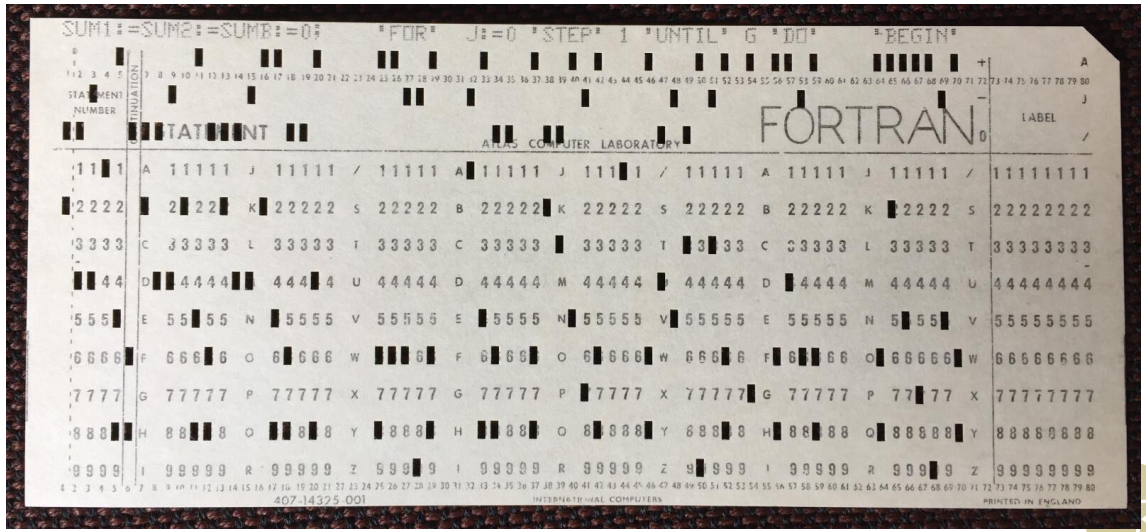
Important to think about what our values are as a field:

1. Correctness
2. Maintainability
3. Usability/fitness for purpose
4. Performance

C and Fortran

- Old programming languages, monomaniacal about backwards-compatibility with legacy code
  - Double-edged sword
  - Lots of old, battle-tested code still works (LAPACK)
  - So much cruft! Spaghetti!
- Weak, limited abstractions
  - Hard to maintain very large codebases
  - Many (most) large scientific codes either reinvent complex language features (badly) or use pre-processing frontends (e.g. python, ruby)
- Limited safety/correctness guarantees
  - No enforcement of interface contracts
  - Memory-safety? Thread-safety? Skill issue!





Python

- Relatively modern programming language
- Super popular in data science
- Memory-safe!
  - Automatic memory management is very complicated!
- Interpreted language
  - Portable
  - Flexible
  - Performance overhead
- Permissive duck typing: really flexible, but enforced at runtime
  - Addendum: Julia is even worse for this. Unsafe at any speed

pawsey

Rust!

Whither Rust?

- Compiled (but has a REPL now!)
- Sophisticated type system
  - Complex, user-defined types
  - Can manipulate types and interfaces as first-class objects
  - Compiler can reason about and enforce composition, edge-cases
- Sophisticated memory semantics
  - Statically-enforced memory safety
  - Predictable semantics -> reduced performance overhead
  - Statically-enforced thread-safety
- Really good C FFI, mature ecosystem (it's in the Linux Kernel!), package management
- Fantastic community

Great documentation:

- The Rust Programming Language: https://doc.rust-lang.org/book/
  - Good explanations
  - Great worked examples
  - Interactive

Type-safety

- Encode the *semantics* of your data in a way that's legible to the toolchain
  - Handle logically distinct types with the same underlying representation
  - Encode constraints on e.g. domains, memory semantics
  - Enforce interface contracts!
- Compiler enforces completeness
  - Forces you to handle edge-cases
  - Extremely useful for error-handling, complicated parsing logic
  - Compiler errors are actually useful!

```
>> enum Atom {
Hydrogen,
Helium,
Lithium,
Uranium,
}

>>
>> fn categorise(atom: Atom) -> u8 {
match atom {
Atom::Hydrogen => 1,
Atom::Helium => 2,
Atom::Lithium => 3
}
}
[E0004] Error: non-exhaustive patterns: `Atom::Uranium` not covered
    ┌─[ command:1:1 ]
    │
 2  │  match atom {
    │        ──┬─
    │          └── pattern `Atom::Uranium` not covered
    │
 5  │  Atom::Lithium => 3
    │                   ┬
    │                   └── help: ensure that all possible cases are being handled by adding a
match arm with a wildcard pattern or an explicit pattern as shown: `,
Atom::Uranium => todo!()`
    │
```

Memory- & thread-safety

- Borrow-checker is Rust's "killer app"
- Tradeoff:
  - Imposes additional requirements/constraints on you, the programmer
  - But! Compiler can now statically reason about ownership and lifetimes of memory objects
  - Can statically guarantee the absence of memory corruption bugs, race conditions
- Deterministic, predictable memory allocations/frees, only when necessary

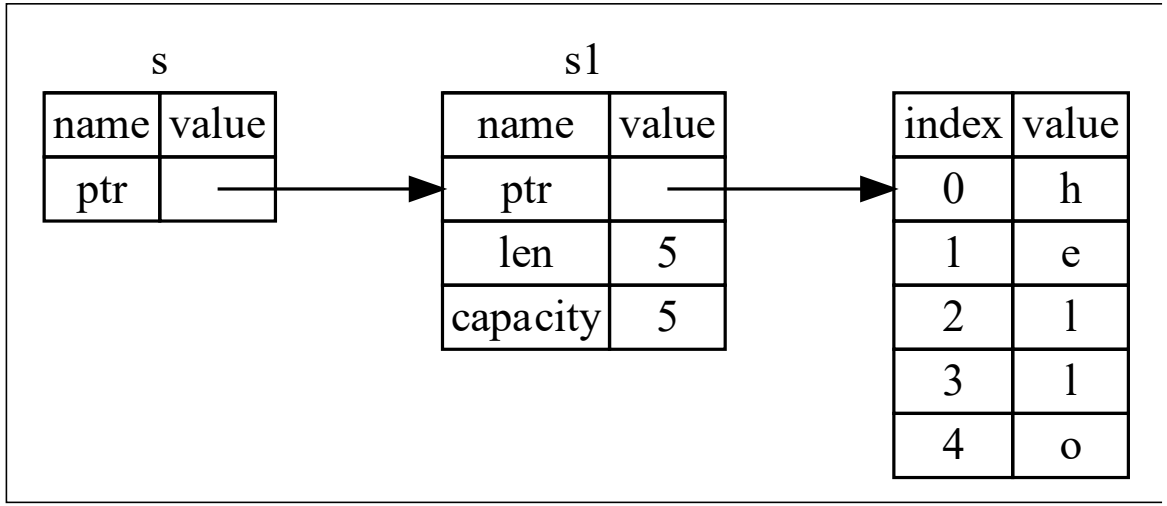


Diagram of a simple string-type object. Source: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Memory- & thread-safety

```
std::array<int, 4> = {-1, -2, -3, -4};

#pragma omp parallel shared(foo)

{

        int thread_id = omp_get_thread_num();

        foo[1] = thread_id;

}


// What is the value of “foo” here?
```

Memory- & thread-safety

```
std::array<int, 4> = {-1, -2, -3, -4};

#pragma omp parallel shared(foo)

{

        int thread_id = omp_get_thread_num();

        foo[1] = thread_id;

}


// What is the value of “foo” here?
```

Threads can execute in any order – race condition!!

```
[KAHLMAC-SL]Rust % ./a.out
-1  3  -3  -4
[KAHLMAC-SL]Rust % ./a.out
-1  0  -3  -4
[KAHLMAC-SL]Rust % ./a.out
-1  2  -3  -4
```

Memory- & thread-safety

- Rust references can be either shared *or* mutable, but not both
- References are immutable by default
- Forces you to be very deliberate about *when* objects are modified in memory
- Forces you to be deliberate about which sections of the program owns a resource
- Most of this is stuff you *should* be doing anyway (most of the time)
    - Guarantees freedom from race conditions, memory leaks, use-after-free errors
- Can disable with an *unsafe* block when really necessary (e.g. low-level device code)

```
>> {
let mut v = vec![-1, -2, -3, -4];
let v1 = &mut v;
let v2 = &mut v;
let z = v1[1] + v2[2];
println!("{z}");
}
[E0499] Error: cannot borrow `v` as mutable more than once at a time
    [ command:1:1 ]
 3  let v1 = &mut v;
                ──┬─
                  ╰─── first mutable borrow occurs here
 4  let v2 = &mut v;
                ──┬─
                  ╰─── second mutable borrow occurs here
 5  let z = v1[1] + v2[2];
              ──┬─
                ╰─── first borrow later used here
```

Example: Murchison Widefield Array data processing

- Radio telescope precursor to the square-kilometer array (SKA)
- "Digitally-steered" telescope – relies on complex signal processing to filter raw input from sensor array and "point" at things
- Needs to process huge (PB) amounts of data
- Large amounts of data processing pipeline in Rust
- I worked on a Pawsey Uplift Project to identify file I/O bottlenecks in hybrid Rust/C++ code
- Contra popular wisdom, bottleneck was in the legacy C++ code
- Rust was robust easy to debug, ready for prime-time

Community!

- Community is important!
- Good community => good technical foundation

1. Have a healthy and welcoming community
2. Attract skilled programmers from marginalized backgrounds (e.g. trans women, furries, people of colour)
3. You get their contributions *and* their friendship
4. Success!

- Rust did all of these things! Substantial reason for its technical success
  - Less so as it becomes corpo – opportunity?
  - Still way better than C++, Fortran, etc
- E.g. Best linear algebra crate in rust (`faer`) is maintained by a cool trans woman!

Closing thoughts

- Rust is really nice to work with
  - The programming language *helps* you write robust code, rather than fighting you
  - Strictly-enforced types and interface contracts are a huge win over existing scientific programming languages
  - Highly-composable
- Really good C FFI, mature ecosystem (it's in Linux!), package management
- Still some rough patches: MPI parallelism, GPU programming
- Probably don't want to rewrite EVERYTHING, but maybe consider using it for new projects
  - Especially good fit for parsing, data wrangling, stuff with complex edge-conditions
- It's fun!!!

Q&A

Useful links

- The Rust Programming Language: https://doc.rust-lang.org/book/
- Rust in a jupyter notebook: https://github.com/evcxr/evcxr
- Learn Rust the Dangerous Way: https://cliffle.com/p/dangerust/

Bonus slides