Rust for Programmers

MIT DCI Workshop — day 1

Interrupt!

And try to follow along on your laptop :)

Slides are intentionally verbose.

Why me?

Jon Gjengset

PhD Student at MIT's Parallel and Distributed Operating Systems group

Noria: 60k LOC Rust database

Several open-source Rust libraries

40+ hrs of Rust live-coding streams

jon@tsp.io

https://tsp.io

https://twitter.com/jonhoo

https://github.com/jonhoo

https://youtube.com/c/JonGjengset

Why Rust?

Fast, reliable, productive — pick three

By Mozilla, for “systems programming”

“Fearless concurrency”

Community-driven and open-source

Rapid industry adoption

Great tooling

https://www.rust-lang.org/

https://www.rust-lang.org/learn

https://doc.rust-lang.org/book/

https://newrustacean.com/

https://play.rust-lang.org/

Today: all the basics

The talk assumes some familiarity with systems programming (Go, C, etc.)

We’ll be moving quickly — stop me at any time.

Up and running

The Rust dev environment

Setting up a Rust dev environment

Rust syntax and basic concepts

Rust program lifecycle

Modules and crates

Ownership

Zero-cost abstractions

Case study: VecMap

Homework: build a hash table

Installing Rust

- Through your package manager of choice
  - If you're stuck and daring: `curl https://sh.rustup.rs -sSf | sh`
- You probably want `rustup`, not just Rust
  - Lets you keep multiple versions installed (later); and
  - Lets you easily install *tools* like rustfmt or clippy (later)
- You can also follow along using https://play.rust-lang.org/

Let’s see what we got

- `rustc --version`
    - The main Rust compiler — you will rarely call this directly
- `cargo --version`
    - The Rust build tool — manages dependencies and build stages for your
- `rustup install nightly && rustc +nightly --version`
    - The bleeding edge compiler — needed if you want unstable features

Time to start a project

- `cargo new --bin rusting`
  - Creates a minimal skeleton for a new application (--lib for libraries)
- `cd rusting && ls`
  - Cargo.toml — meta info about your application + dependencies
  - .gitignore — ignores various build artifacts
  - src/ — where all your source files will go
  - src/main.rs — entry point for our new application’s binary (src/lib.rs for libs)
- `cargo run`
  - “Hello, world!” — amazing!

What does cargo do?

- `cargo run`
  - Run the application’s binary (--bin <binname> if multiple)
- `cargo build`
  - Build the application (place in target/)
- `cargo build --release`
  - Build with optimizations! (also `cargo run --release`)
- … and **much** more (later)

Rust basics

Syntax & common concepts

What code did we just run?

```
// cat src/main.rs
fn main() {
    println!(“Hello, world!”);
}
```

What code did we just run?

```
// main just like in most languages
fn main() {
    // println lets you, well, print a line
    // strings in “” like we’re used to
    //     v--- we’ll get back to this !
    println!(“Hello, world!”);
    //        semicolons! ---^
}
```

Let’s quickly go over the basics

```
fn main() {
    // we can have variables
    let x = 42;
    let y = “Hello”;
    // format strings too:
    println!(“{}, {}!”, x, y);
    //        ^--- print using Display trait (later)
}
```

Variables have “mutability”

```
fn main() {
    // variables are immutable by default
    let x = 42;
    x += 1; // won’t compile: x is not mutable
    let mut x = 42;
    x += 1; // compiles just fine
    // (oh, yeah, and you can shadow!)
}
```

Variables are typed

```
fn main() {
    // variables are typed
    let x: u32 = 42;
    let y: bool = x; // compiler says no
    let y: u16 = x; // compiler says no
    // variable types are _usually_ inferred
}
```

No particularly surprising types

```
fn main() {
    // nums: {i,u}{8,16,32,64,128,size}, f{32,64}
    // tuples: (u8, bool)
    // fixed arrays: [u8; 8]
    // references: &Foo, &mut Foo (like pointers)
    // text: char, &str and String (later)
    // slices: &[T]
}
```

Standard library also has handy things

```rust
fn main() {
    // Vec<T>
    // HashMap<K, V>, HashSet<T>
    // BTreeMap<K, V>, BTreeSet<T>
    // BinaryHeap<T>,
    // etc.
}
```

There are functions

```
fn nonzero(x: u32) -> bool {
    // every block evaluates to its last statement
    // and everything is an expression (e.g., if)
    x > 0
}
fn main() {
    assert!(nonzero(1));
}
```

We have control flow

```
fn main() {
    loop {
        for i in 1..=100 /* inclusive range */ {
            if i % 3 == 0 && i % 5 == 0 { break; }
        }
    }
}
```

You can define types

```
struct Foo {
    is_bar: bool,
}

enum Either {
    Left,
    Right,
}
```

You can define methods on types

```rust
impl Foo {
    fn barify(&mut self) {
        // ...
    }
    fn is_barified(&self) -> bool {
        // ...
    }
}
```

Visibility modifiers on fields + methods

```
struct Foo {
    pub is_bar: bool,
}
impl Foo {
    pub fn barify(&mut self) {
        self.is_bar = true;
    }
}
```

There are closures too

```
(1..=100)
  .filter_map(|i| {
    if i % 15 == 0 { Some(“fizzbuzz”) }
    else if i % 5 == 0 { Some(“buzz”) }
    else if i % 3 == 0 { Some(“fizz”) }
    else { None }
  })
```

Some Rustisms — traits (behaviors)

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

for x in xs /* if xs implements Iterator */ {


}
```

Some Rustisms — abstract datatypes

```
enum Result<T, E> { // oh yeah, and generics
    Ok(T),
    Err(E),
}
enum Option<T> {
    Some(T),
    None
}
```

Some Rustisms — pattern matching

```
let s = match x {
    Some(s) if s == “fizz” => format!(“got fizz”),
    Some(s) => format!(“no fizz, just {}”, other),
    None => format!(“a dud”),
};
let u = if let Foo { x: Bar::Baz(_, val), .. } = z {
    val
};
```

Some Rustisms — error handling

```
fn write(w: …) -> Result<usize, io::Error> {
    let n = w.write(“hello ”)?; // roughly equal to:
    // match w.write(“hello”) {
    //     Ok(t) => t,
    //     Err(e) => return Err(e),
    // }
    Ok(n) // == Result::Ok(n)
}
```

Some Rustisms — macros

```
// anything that ends with ! is a macro
// macros are “hygienic” -- more on that later
// e.g.,
assert_eq!(x, y, “ohnoes”)
// expands to something like
if x != y {
    panic!(“assertion {} == {} failed”, x, y);
}
```

Some Rustisms — more macros

```
// we'll cover macros in depth later
// for now, some useful ones:
println!()
format!()
unreachable!()
unimplemented!()
vec![1, 2, 3, 4]
```

Some Rustisms — documentation comments

```rust
/// triple / indicates a “doc comment”
/// documents the following item
/// used for everything (fn, type, module, etc.)
/// markdown **is** supported
/// code blocks are tests!
pub fn haz_docs_wow() { /* ... */ }
```

Lifecycle

testing, documentation, releases, tools

Setting up a Rust dev environment

Rust syntax and basic concepts

Rust program lifecycle

Modules and crates

Ownership

Zero-cost abstractions

Case study: VecMap

Homework: build a hash table

The developer lifecycle:

- `cargo check`
  - Fast compiler pass to make sure everything builds
- `cargo test`
  - Run all unit, integration, and documentation tests
- `cargo doc --open`
  - Build (and open) documentation for your application
- `cargo publish`
  - Release new version of application to https://crates.io/

Built-in testing

```
#[cfg(test)] // only compile in test mode
#[test] // this function is a test
fn it_works() {
    assert!(true); // if function panics, test fails
}

// all code in doc comments is tested automatically!
// all files in tests/ considered integration tests
```

Built-in documentation generator

- Same as https://doc.rust-lang.org/.
- Uniform across all libraries you'll come across!
    - So much so that all released libraries have docs auto-generated.
    - https://docs.rs/
- Also generates documentation for your dependencies.

Built-in documentation generator

DOCS.RS | evmap-5.0.2 ▾ | Source | Platform ▾ | Find crate

Crate evmap

See all evmap's items

- Re-exports
- Modules
- Structs
- Enums
- Functions
- Type Definitions

Crates

evmap

Re-exports

```
pub use shallow_copy::ShallowCopy;
```

Modules

shallow_copy	Types that can be cheaply aliased.

Structs

Options	Options for how to initialize the map.
Predicate	Unary predicate used to retain elements
ReadHandle	A handle that may be used to read from the eventually consistent map.
ReadHandleFactory	A type that is both `Sync` and `Send` and lets you produce new `ReadHandle` instances.
WriteHandle	A handle that may be used to modify the eventually consistent map.

Enums

Operation	A pending map operation.

Functions

new	Create an empty eventually consistent map.
with_hasher	Create an empty eventually consistent map with meta information and custom hasher.
with_meta	Create an empty eventually consistent map with meta information.

Uniform library + application releases

- All available through cargo (incl. binaries)
- All uploaded to https://crates.io/
- All documentation on https://docs.rs/
- Semantic versioning
- Can also retrieve over git or by path, or override for debugging

Also many handy tools

- `rustfmt`
  - Opinionated code-formatter (like `go fmt`)
- `clippy`
  - Sophisticated linter (like `go vet`)
- `rls`
  - Compiler-assisted editor integration (autocompletion, type lookup, go-to-def, etc.)

Mostly from `rustup component add`; more from `cargo install`

Structure

Modules and crates

Setting up a Rust dev environment

Rust syntax and basic concepts

Rust program lifecycle

Modules and crates

Ownership

Zero-cost abstractions

Case study: VecMap

Homework: build a hash table

Every project is a “crate”

- `Cargo.toml` describes the contents of that crate
    - Dependencies (i.e., other crates)
    - Meta-information
    - Binaries and examples
- Every crate is a single compilation unit (sort of)
- Crates can be published for others to use
- src/lib.rs is main entrypoint to the library “part”

Using external crates

- Define dependency in `Cargo.toml`:

```
[dependencies]
regex = “1”
```

- The just use it in your code:

```
use regex::Regex;
fn main() {
  let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
  assert!(re.is_match("2014-01-01"));
}
```

Crates have modules

- `mod foo` says that there is a module called “foo”
  - either inline (`mod foo { .. }`)
  - or in ./foo.rs
  - or in ./foo/mod.rs
- Modules can then contain other modules
- Use from a module with `foo::bar::superfunction`
- Import into current scope with `use`
- Use `crate::` prefix to start from `crate`

Rust visibility rules

- By default: visible in module and its children
- `pub` makes item visible anywhere it can be named
    - note that it does *not* mean that it is available outside of module
    - for example, won't be accessible if parent is private
- `crate` makes item visible anywhere it can be named in this crate
- `pub(super)` makes item visible in parent module
- `pub(in pa::th)` makes item visible from `pa::th`
- `pub` in crate root makes item available in external interface

An aside on the ecosystem

- Already some very good crates out there!
  - `structopt` — convenient command-line tools
  - `serde` — convenient serialization/deserialization
  - `csv` — convenient access to tabularized data
  - `regex` — really fast regular expressions
  - `rayon` — parallel compute library (think OpenMP but safe and nice)

```rust
#[derive(Deserialize)]
struct Row {
    city: String,
    region: String,
    population: i32,
}

fn main() {
    let mut rdr = csv::Reader::from_reader(std::io::stdin());
    let mut region_max: HashMap<_, (String, i32)> = HashMap::new();
    for result in rdr.deserialize() {
        let Row { city, region, population } = result.unwrap();
        match region_max.entry(region) {
            Entry::Occupied(mut e) => {
                if e.get().1 < population {
                    *e.get_mut() = (city, population);
                }
            }
            Entry::Vacant(v) => { v.insert((city, population)); }
        }
    }
    println!("{:?}", region_max);
}
```

Ownership

Borrows & moves

In Rust, someone always owns a value

- First real departure from languages you may know
- **Key:** The owner of `T` is responsible for deallocating `T`
    - Follows from this that there can only be *one* owner
- What happens on destruction is dictated by `Drop`
    - If T is a file, owner closes the file
    - If T is heap-allocated, owner frees

Owners can then share values

- Owner can lend out access to T through *references*
- At any one time, there can be either:
  - Any number of shared (immutable) references; **or**
  - A single exclusive (mutable) reference
- Follows from this that you cannot have data races! †
- Owner can also *give away* T if there are no references

† Mostly — we'll get back to this later.

The borrow checker

- A `&T` (shared) or `&mut T` (exclusive) is a pointer to `T`
    - A reference must not outlive `T`, or we'd have a dangling pointer
- `T` by itself is an *owned value*
    - Always a value on the stack
    - Could own resources elsewhere (like on the heap: `Box<T>`)
- If we give away `T`, we say that `T` is *moved*
    - Any references would be invalidated!

The borrow checker

- The **borrow checker** checks these properties at compile time:

```rust
let xs = vec![1, 2, 3];
drop(xs);
let y = xs[0];
```

```
error[E0382]: borrow of moved value: `xs`
 --> src/main.rs:4:10
  |
2 | let xs = vec![1, 2, 3];
3 | drop(xs);
  |      -- value moved here
4 | let _y = xs[0];
  |          ^^ value borrowed here after move
```

More borrow examples

```rust
let mut x = vec![1, 2, 3];
let y = &mut x[1];
x.clear();
*y = 42;
```

```
error[E0499]: cannot borrow `x` as mutable more than
once at a time
 --> src/main.rs:4:1
  |
3 | let y = &mut x[1];
  |              - first mutable borrow occurs here
4 | x.clear();
  | ^ second mutable borrow occurs here
5 | *y = 5;
  | ------ first borrow later used here
```

More borrow examples

```
let mut xs = vec![1, 2, 3];
for x in &xs {
    if *x == 2 {
        xs.insert(0, 2);
    }
}
```

```
error[E0502]: cannot borrow `xs` as mutable because
it is also borrowed as immutable
 --> src/main.rs:5:9
  |
3 | for x in &xs {
  |          ---
  |          |
  |          immutable borrow occurs here
  |          immutable borrow later used here
4 |     if *x == 2 {
5 |         xs.insert(0, 2);
  |         ^^^^^^^^^^^^^^^ mutable borrow occurs
here
```

More borrow examples

```rust
let mut xs = vec![1, 2, 3];
for x in &mut xs {
    if *x == 2 {
        xs.insert(0, 2);
    }
}
```

```
error[E0499]: cannot borrow `xs` as mutable more
than once at a time
 --> src/main.rs:5:9
  |
3 | for x in &mut xs {
  |          --------
  |          |
  |          first mutable borrow occurs here
  |          first borrow later used here
4 |     if *x == 2 {
5 |         xs.insert(0, 2);
  |         ^^ second mutable borrow occurs here
```

More borrow examples

```rust
fn foo() -> &[u32] {
    let xs = vec![1, 2, 3];
    &xs[1..]
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:13
  |
3 | fn foo() -> &[u32] {
  |             ^ help: consider giving it a 'static
lifetime: `&'static`
  |
  = help: this function's return type contains a
borrowed value, but there is no value for it to be
borrowed from
```

More borrow examples

```
let mut xs = vec![1, 2, 3];
thread::spawn(|| {
    xs.push(42);
});
assert_eq!(xs.len(), 42);
```

```
error[E0373]: closure may outlive the current
function, but it borrows `xs`, which is owned by the
current function
 --> src/main.rs:3:20
  |
3 | std::thread::spawn(|| {
  |                    ^^ may outlive borrowed value
`xs`
4 |     xs.push(42);
  |     -- `xs` is borrowed here
  |
note: function requires argument type to outlive
`'static`
```

Lifetimes: how long to borrow for

```
note: function requires argument type to outlive `'static`
```

- What does that mean?
- Every reference has a *lifetime*
  - Exact meaning is subtle
  - *Basically* the span of time it has access to its referent
- Lifetime of mutable references cannot overlap
- Referent cannot disappear while lifetimes remain

Propagating lifetimes

```
fn f<’a>(xs: &’a [u32]) -> &’a [u32] {
    &xs[1..]
}
```

- The return value has the same lifetime as the argument
- The compiler is usually smart enough that you can omit `‘a`
- `‘static` (essentially) means “forever”

Copy semantics

- For most types, moving them invalidates the source
    - No references can remain across the move point
- This is different for types that implement the `Copy` trait
- If a type is `Copy`, it can be cheaply duplicated with `memcpy`
    - Moving a `Copy` type just copies it — the old value remains!

An aside on strings

- Rust has two (main) string types: `&str` and `String`
- Almost exactly analogous to `&[char]` and `Vec<char>`
    - Can’t change or append to `&str`
    - `String` is heap-allocated and can grow/shrink
    - `String -> &str` is trivial
    - `&str -> String` requires allocation + copy

Traits

Zero-cost abstractions

Traits = typeclasses ≈ interfaces

- A trait is a collection of behaviors

  and the types that implement that behavior
- We've already seen many:
  - `Display` — Types that can be printed
  - `Debug` — Types that can be printed verbosely
  - `Iterator` — Types that can be iterated over (e.g., with `for`; see docs!)
  - `Fn/FnMut/FnOnce` — Types that are callable (e.g., closures)
  - `Copy` — Types that have copy semantics
  - `Drop` — Types that need special treatment when dropped

Some more traits as examples

- For I/O
  - `Read` — Types that can be read from
  - `Write` — Types that can be written to
- For concurrency
  - `Send` — Types that can be safely moved to another thread
  - `Sync` — Types that can be safely accessed from another thread
- For datastructures
  - `Hash` — Types that can be hashed
  - `Eq` — Types that can be compared for equality
  - `Clone` — Types that you can clone new instances of

Creating a trait

```
trait Iterator {
    // implementors must define this type
    type Item;
    // implementors must define this method
    fn next(&mut self) -> Option<Self::Item>;
    // this method implementation is provided
    fn last(mut self) -> Option<Self::Item> {
        // .. call self.next() until None, return last Some(x) ..
    }
}
```

Implementing a trait

```
struct Range(usize, usize);
impl Iterator for Range {
    type Item = usize;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 == self.1 { return None; }
        self.0 += 1;
        Some(self.0 - 1)
    }
}
for i in Range(0, 50) { … } // now works!
```

Traits > interfaces

- You can implement traits when they are defined
  - e.g., implement your trait for type in `std`
- Traits can have “conditional” implementations:
  - e.g., `Vec<T>` is `Clone` if `T` is `Clone`
  - e.g., `Vec<T>sort` is only provided if `T` is `Ord`
  - e.g., `HashMap<K, V>` requires that `K` is `Hash`
- Traits “know” about the implementing class (i.e., can access `Self`)
  - Needed for traits like `Eq` where operand must also be `Self`

Generic data types

```
struct VecMap<K, V> {
    values: Vec<(K, V)>,
}

impl<K, V> VecMap<K, V>
  where K: Eq { // we can “bound” generic parameters by traits
    fn get(&self, key: &K) -> Option<&V> {
        self.values.iter().find(|(k, _)| k == key).map(|(_, v)| v)
    }
}
```

Generic functions

```
fn sort<T>(xs: &mut [T]) where T: Ord {
    // ...
}

fn map<I, F, T>(xs: I, f: F) -> impl Iterator<Item = T>
  where I: Iterator,
        F: FnMut(I::Item) -> T {
    // ...
}
```

Generics at runtime

- All generic code is monomorphized, so no runtime cost
  - e.g., `map<I, F, T>` will be optimized for the specific `I`, `F`, and `T` passed
  - This *can* increase compilation time + binary size though!
- You *can* opt in to dynamic dispatch:

```
fn map<I, T>(xs: I, f: &mut dyn FnMut(I::Item) -> T)
  -> impl Iterator<Item = T>
where I: Iterator {
    // ...
}
```

Case study

Building a map with a vector

Live-coding time!

If you're just following the slides, the following contains the important final code.

But doesn't really contain commentary on why things are what they are.

```rust
struct VecMap<K, V> {
    values: Vec<(K, V)>,
}

impl<K, V> Default for VecMap<K, V> {
    fn default() -> Self {
        VecMap {
            values: Vec::new(),
        }
    }
}
```

```rust
impl<K, V> VecMap<K, V> where K: Eq {
    pub fn get(&self, key: &K) -> Option<&V> {
        self.values.iter().find(|(k, _)| k == key).map(|(_, v)| v)
    }

    pub fn insert(&mut self, key: K, value: V) -> Option<V> {
        match self.values.iter().position(|(k, _)| k == &key) {
            Some(i) => {
                let (_, v) = self.values.swap_remove(i);
                self.values.push((key, value));
                Some(v)
            },
            None => {
                self.values.push((key, value));
                None
            }
        }
    }
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_get() {
        let mut m = VecMap::default();
        assert_eq!(m.get(&'x'), None);

        m.insert('x', 42);
        assert_eq!(m.get(&'x'), Some(&42));

        assert_eq!(m.get(&'y'), None);
    }
}
```

```rust
#[doc(hidden)]
pub struct VecMapIter<'a, K, V> {
    map: &'a VecMap<K, V>,
    at: usize,
}

impl<'a, K, V> Iterator for VecMapIter<'a, K, V> {
    type Item = (&'a K, &'a V);

    fn next(&mut self) -> Option<Self::Item> {
        if self.at < self.map.values.len() {
            let (ref k, ref v) = self.map.values[self.at];
            self.at += 1;
            Some((k, v))
        } else {
            None
        }
    }
}
```

```rust
impl<'a, K, V> IntoIterator for &'a VecMap<K, V> {
    type Item = (&'a K, &'a V);
    type IntoIter = VecMapIter<'a, K, V>;
    fn into_iter(self) -> Self::IntoIter {
        VecMapIter {
            map: self,
            at: 0
        }
    }
}

… test …
    for (k, v) in &m {
        assert_eq!(k, &'x');
        assert_eq!(v, &42);
    }
```

```rust
use std::iter::FromIterator;
impl<K, V> FromIterator<(K, V)> for VecMap<K, V> where K: Eq {
    fn from_iter<T>(iter: T) -> Self
    where
        T: IntoIterator<Item = (K, V)>,
    {
        let mut m = VecMap::default();
        for (k, v) in iter {
            m.insert(k, v);
        }
        m
    }
}

#[test]
fn from_iter() {
    let m: VecMap<_, _> = vec![('x', 42), ('y', 43)].into_iter().collect();
    assert_eq!(m.get(&'x'), Some(&42));
    assert_eq!(m.get(&'y'), Some(&43));
}
```

```rust
use serde_derive::{Serialize, Deserialize};

// pure magic -- all that’s needed for full serialization support
// including json, binary, messagepack, bson, ...
#[derive(Serialize, Deserialize)]
struct VecMap<K, V> {
    values: Vec<(K, V)>,
}
```

Try it out @
https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=7e64a2d925e6fd2139a81b5206b5d965

Homework

Build a hash table

Homework: Build a hash table

- Try to match the main example from `std::collections::HashMap`
  - Some parts will be harder than others; feel free to skip around.
  - To make life easier for yourself, remove the calls to `.to_string()`.
  - There's some trait magic in `get`'s `Q` parameter to accept more key types;
    ignore that in your own implementation, and change the example code instead.
- Don't worry so much about efficiency at first
- Check that the map works for your own types (see later examples)
- If you want a challenge, try to also implement the "Entry API"
- Send it to me at jon@tsp.io if you want feedback!

Rust in Depth

MIT DCI Workshop — day 2

Interrupt!

And try to follow along on your laptop :)

Slides are intentionally verbose.

Why me?

Jon Gjengset

PhD Student at MIT's Parallel and Distributed Operating Systems group

Noria: 60k LOC Rust database

Several open-source Rust libraries

40+ hrs of Rust live-coding streams

jon@tsp.io

https://tsp.io

https://twitter.com/jonhoo

https://github.com/jonhoo

https://youtube.com/c/JonGjengset

Today: the deep end

The talk assumes familiarity with Rust.

We’ll be moving quickly — stop me at any time.

We are covering a **lot** of advanced stuff — you won’t run into most of this.

unsafe

Programmer-enforced invariants

unsafe

Interior mutability

Send and Sync

Trait bounds

Hygienic macros

FFI

Asynchronous programming

Homework: ???

What do we mean by safe?

- Rust aims to provide both type and memory safety
  - Cannot make one type be another
  - Cannot have dangling pointers or buffer overflows
  - Cannot have data races
- It does this by restricting what your code can do
  - Enforced primarily by type checker and borrow checker
- *Most* of the time, this is what you want
  - *Very* occasionally, the compiler doesn't know that what you're doing is safe
  - That is when you need unsafe

unsafe, or “writing Rust like C”

- So what exactly *is* unsafe?
- In an unsafe block, you can:
  - Alias mutable pointers.
  - Transmutate same-sized types.
  - That’s it! All other restrictions still apply.
  - … but far-reaching implications.
- Essentially, it allows you to write code like in C.
  - **You** now have to check your invariants.
- **Primary use**: invariants the compiler cannot enforce.

When do we need unsafe?

- Zero-copy parsing:
    - We assert that that bytes matches in-memory layout
    - Or even simpler: `[u8; 8] -> u64`
- Race-free code with complex invariants
    - Mutex: “If this number is 0, we own this pointer”
    - Epochs: “Once this condition becomes true, there are no more readers”
    - Most lock-free code
- Calling C code: who knows what it’ll do
- Dereferencing pointers after pointer arithmetic

Unsafe is rare in practice

- Usually hidden behind safe interfaces (e.g., `std`)
- Usually safe interfaces are plenty fast enough
  - Premature optimization!
- Even when you **do** need it: encapsulate it safely
  - Still a hazard — now you know what code to review!
  - Still better than the wild west of C
  - Can also have unsafe functions (need unsafe block to call)
- If you’re going to write it, read https://doc.rust-lang.org/nomicon/

Unsafe trickiness: pointer aliasing

- In multi-threaded context, aliasing mutable pointers is bad
- But even in single-threaded context; consider:

```
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
}
```

Unsafe trickiness: transmuting types

- From the excellent “Nomicon” on transmutation:

> Even though this book is all about doing things that are unsafe, I really can't emphasize that you should deeply think about finding Another Way than the operations covered in this section. This is really, truly, the most horribly unsafe thing you can do in Rust. The railguards here are dental floss.

> The ways to cause Undefined Behavior with this are mind boggling.

Cells

Providing interior mutability

Mutating through shared references

- In some cases it’s okay!
  - If you’ve taken a lock.
  - Single-threaded code w/o pointers
  - Single-threaded code with reference counting
  - Atomic instructions
- These provide “interior mutability”
- Usually two primary uses:
  - Mutability is an implementation detail (e.g., new epoch on read)
  - Shared mutable state (e.g., `Arc<HashMap>`)

Single-threaded mutability

- Value in one thread, never touched by others
- **Rule**: can never have multiple mutable pointers!
- Two cases:
    - `Cell` — Never use pointers in the first place (only move)
    - `RefCell` — Check for aliasing at runtime (guard + reference counting)

Multi-threaded mutability

- Multiple threads with references to shared value
- Problem is if two threads race
  - May see intermediate state that they should not
- Two mechanisms
  - `std::sync::atomic` — Atomic operations have no intermediate state
  - `Mutex/RwLock` — Maintain invariants that give mutual exclusion

Unsafe interior mutability

- How do you *implement* one of these?
  - Rust **never** lets you turn `&` into `&mut` (even in unsafe).
- Example of something *unsafe* that *we* know is safe
  - If you have the lock, it's okay to modify!
  - Compiler can't check this invariant
- `UnsafeCell<T>` — holds a `T`, gives you `*mut T`.
  - You *probably* never need this type.

Thread-safety

The Send and Sync markers

Marking thread-safe code

- Types like `Cell` and `RefCell` aren't safe in concurrent code!
- Two *marker traits*: `Send` and `Sync`
  - `Send` — this type can be moved to other threads
  - `Sync` — this type can be accessed from multiple threads
    - Essentially: `T: Sync` if `&T: Send`
  - Implemented automatically if all members do
  - Can explicitly opt out if needed (e.g., `Rc`, `Cell`, `RefCell`, `*mut T`)

Thread-safety as bounds

- Send and Sync are useful because they can be used as trait bounds!

```
fn spawn<F>(f: F) -> JoinHandle
where F: FnOnce() + 'static + Send {
    // takes any closure that is safe to give to another thread!
}
```

```
impl<T> Send for Arc<T>
where T: Sync {
    // only share reference-counted pointer if value is thread-safe
}
```

Traits & Bounds

Taking advantage of types

unsafe

Interior mutability

Send and Sync

Trait bounds

Hygienic macros

FFI

Asynchronous programming

Homework: ???

Bounds let you specify more flexible APIs

- Going back to `HashMap::get`:

```
pub fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
where
    K: Borrow<Q>,
    Q: Hash + Eq,
```

  - `Q: ?Sized` — `Q` does not have to be a concrete type (e.g., `[u8]`)
  - `K: Borrow<Q>` — `K` can be borrowed as a `Q`
  - `Q: Hash + Eq` — `Q` can be hashed (as `K`!) and compared

Bounds let you specify more flexible APIs

- Or `&[]::get`:

```
pub fn get<I>(&self, index: I)
  -> Option<&<I as SliceIndex<[T]>>::Output>
where I: SliceIndex<[T]>,
```

  - `<I as SliceIndex<[T]>>::Output`
    - The type that `I` produces when used as an index in `[T]`
      - This is an associated type — think of them as “output types”
    - Compare `&xs[1]` and `&xs[1..]`

And complex requirements

- For example, chaining iterators:

```
fn chain<U>(self, other: U)
  -> Chain<Self, <U as IntoIterator>::IntoIter>
where U: IntoIterator<Item = Self::Item>,
```

  - `U: IntoIterator<Item = Self::Item>`
    - `U` must produce iterator that yields elements of the same type

… really complex requirements …

- What about being able to iterate a flattened iterator backwards?

```
impl<I, U> DoubleEndedIterator for Flatten<I>
where
    I: DoubleEndedIterator,
    U: DoubleEndedIterator,
    <I as Iterator>::Item: IntoIterator,
    <<I as Iterator>::Item as IntoIterator>::IntoIter == U,
    <<I as Iterator>::Item as IntoIterator>::Item
      == <U as Iterator>::Item,
```

Some other useful traits

- `Deref` and `DerefMut`
  - If `X` contains a `Y`, and `X` derefs to `Y`, you can access `Y` through `X` with `.`
- `AsRef` and `Borrow`
  - Trivial reference conversions (e.g., `&Vec<u8> -> &[u8]`)
- `Clone` and `Copy`
  - How to make more of your type (`Copy` **must** be derived!)
- `Into`, `From`, `TryInto`, and `TryFrom`
  - More elaborate type conversion (e.g., `usize: TryFrom<&str>`)

Macros

Transforming program syntax

unsafe

Interior mutability

Send and Sync

Trait bounds

Hygienic macros

FFI

Asynchronous programming

Homework: ???

Code that expands to other code

- If you’ve used them in C, Rust’s macros are *similar*
- Rust macros are *hygienic* — cannot (generally) affect surroundings
  - i.e.., cannot interact with items in caller’s scope implicitly
- Two ways to specify:
  - macro_rules! — declare macro inline
  - procedural macros — defined by program
- I recommend “The Little Book of Rust Macros” if you’re curious

A macro we’ve already used

```
// vec![1, 2, 3]
macro_rules! my_vec {
    ($($x:expr),*) => {{
        let mut xs = Vec::new();
        $(xs.push($x);)*
        xs
    }}
}
// it’s okay if there’s a variable called xs in caller!
// actual vec! is smarter -- avoids multiple allocs (see impl)
```

Macro arguments are typed

```
macro_rules! add_n_fn {
    ($fname:ident, $t:ty, $n:expr) => {
        fn $fname(x: $t) -> $t { x + $n }
    }
}
add_n_fn!(add_four, i32, 4);

fn main() {
    // can only call add_four here since we passed in the ident!
    assert_eq!(add_four(4), 8);
}
```

Programmatic macros

- “Procedural macros”
- Write a Rust function `TokenStream -> TokenStream`
  - Gets called for any item decorated with `#[macro_name]`
- *Really* powerful — we won’t go into too much detail
- Some common ones:
  - `format_args!()` (the core of `println!`)
  - `#[derive(Debug, Clone, PartialEq, Hash)]`
  - `#[derive(Serialize, Deserialize)]` — Serde (!!!)
  - `#[get(path = “/”)]` — Rocket

FFI

Interoperability

- unsafe
- Interior mutability
- Send and Sync
- Trait bounds
- Hygienic macros
- **FFI**
- Asynchronous programming
- Homework: ???

Not everyone uses Rust

- “Foreign Function Interface” — for playing with others
- Can *use* a foreign item, or *expose* an item to foreign code
- Essentially makes code conform to C ABI
  - Which is in turn supported by *lots* of languages!
- All FFI is unsafe!
  - We don’t know what foreign code does — might scribble all over memory
- Don’t need to write the bindings yourself!
  - bindgen and cbindgen generate everything for you

Calling out of Rust

```rust
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern { // other calling conventions are also possible
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
}
```

Encapsulating unsafe FFI bindings

- Wrap the unsafe library code in safe wrappers
- Many examples in the ecosystem:
  - `nix` — safe wrappers for most unix system calls
  - `ssh2` — safe bindings for libssh2
  - `openssl` — safe bindings for openssl
  - `libz` — safe bindings for zlib
  - `git2` — safe bindings for libgit2
- The nomicon is a good resource here too

Calling into Rust

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}

// in C:
extern char* hello_rust(void);
int main() {
    printf(“%s\n”, hello_rust());
}
```

Writing compatible Rust code

- `#[repr(C)]`
- Use raw pointers
- Null-terminated strings (`CStr` and `CString`)
- Use `libc` crate for C types
- Don’t panic across FFI boundary (`catch_unwind`)

async/await

Asynchronous programming

unsafe

Interior mutability

Send and Sync

Trait bounds

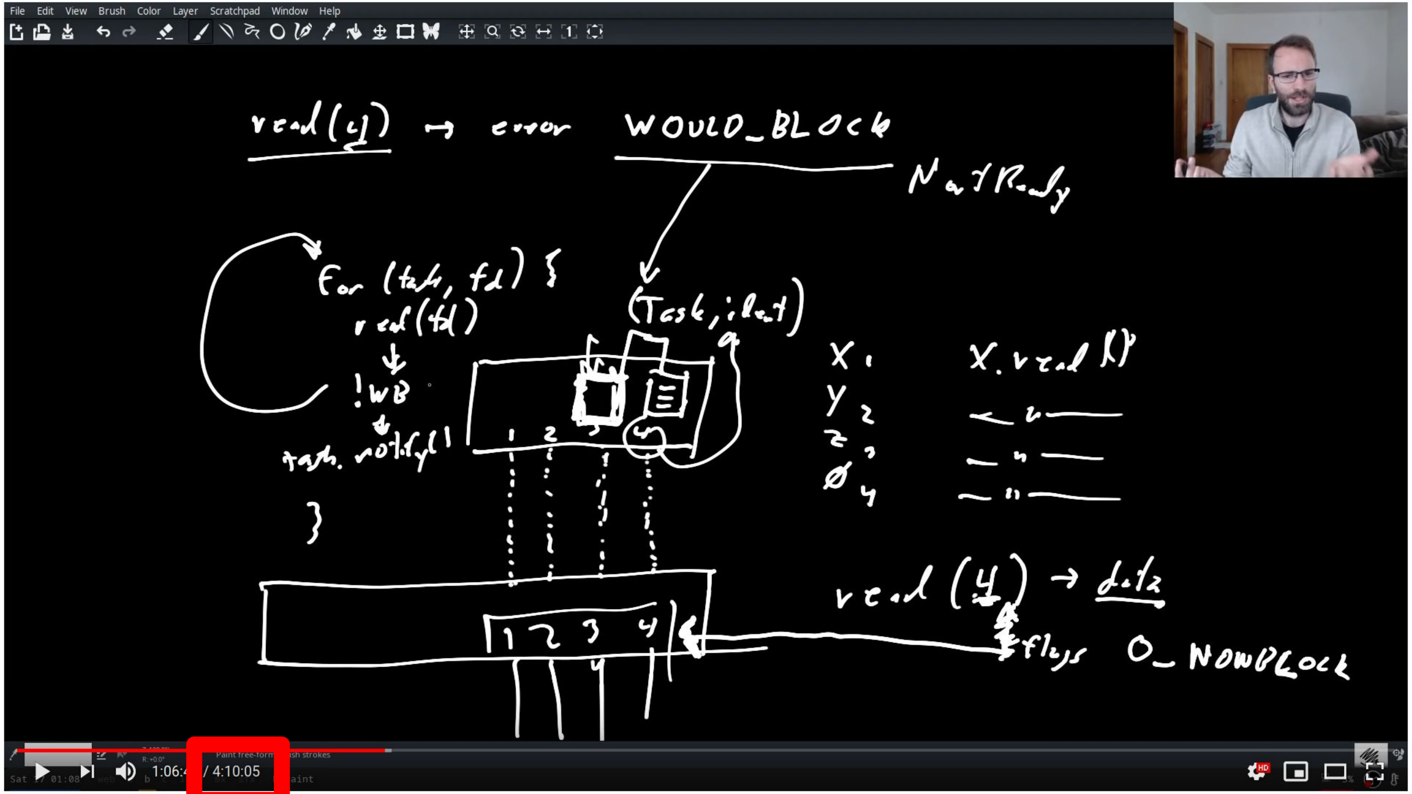
Hygienic macros

FFI

Asynchronous programming

Homework: ???


Here be dragons.



CAMBRIDGE

The What and How of Futures and async/await in Rust

16,788 views

463 1 SHARE SAVE

Futures — the heart of async Rust

```rust
pub trait Future {
    type Item;
    type Error;
    fn poll(&mut self) -> Poll<Self::Item, Self::Error>;
}
type Poll<T, E> = Result<Async<T>, E>;
pub enum Async<T> {
    Ready(T),
    NotReady,
}
```

This is the “old” version of futures -- we’ll get back to `std::Future`

Futures — the heart of async Rust

```
let result = loop {
    match fut.poll() {
        Ok(Async::Ready(t)) => break Ok(t),
        Ok(Async::NotReady) => {
            /* how do we avoid spinning? */
            /* can we avoid one thread per future? */
        },
        Err(e) => break Err(e),
    }
}
```

Creating a future does nothing!

The trait is only one part

- Asynchronous programming requires multiple components:
    - A *description* of the asynchronous computation (e.g., `Future`)
    - A way to *execute* the asynchronous computations (e.g., a thread pool)
    - A *notification mechanism* to know when progress can be made (e.g., `epoll`)
- In Rust, these are still evolving
    - `Future` is being added to `std`
    - Multiple executors (primarily `tokio`, also `juliex`)
    - Multiple “reactors” (primarily `tokio-reactor`, also `romio`; use `mio`)
    - Attempts to standardize: https://github.com/rustasync/runtime

Executing futures more concretely

```
fn main() {
    // assume that foo() returns a future
    let f = foo();
    // at this point, *no work has been done*
    // f just _describes_ the computation that will be performed
    // we need to _execute_ f (using, say, tokio) to get the value
    let v = tokio::run(f);
}
```

Making async compute convenient

- Most of current futures rely on *combinators*
  - Sort of like `Iterator` (think map, filter, and_then, flatten, etc.)
  - Leads to pretty ugly code and lots of closures (callback hell-ish)
- Next step: async/await
  - Language support for asynchronous operations
  - `async fn foo() -> Result<...>`
    - return type is *really* a `Future`
  - `let x = foo().await;`
    - similar to `return foo().and_then(|x| { /* rest of function */ }`
  - Allows you to write imperative code that is async!

Async/await is hard.

- Consider:

```
async fn foo() -> u32 {
    bar().await + baz().await
}
```

- We need to remember the first response somewhere!
    - Compiler needs to generate somewhere to store all intermediate results
- We need to resume at the right place!
    - If `bar().await` has completed, don’t run it again!
    - But `baz().await` may return `NotReady`!

Async/await is hard.

```
async fn foo() -> u32 {
    bar().await + baz().await
}
```

Is essentially the same as

```
fn foo() -> impl Future<Item = u32, Error = ()> {
    bar().and_then(|bar| {
        baz().map(|baz| bar + baz)
    })
}
```

Essentially: must encode the state machine

```
enum FooState {
        None,
        CalledBar(BarFuture),
        CalledBaz(BarReturnType, BazFuture),
        Done(u32)
}
impl Future for FooState {
  fn poll(&mut self) -> Poll<u32, ()> {
    match *self {
      FooState::None => {
        self = FooState::CalledBar(bar()),
        return self.poll();
      }
      FooState::CalledBar(ref mut fut) => {
        let bar_ret = try_ready!(fut.poll());
        self = FooState::CalledBaz(bar_ret, baz());
        return self.poll();
      }
      FooState::CalledBaz(bar_ret, ref mut fut) => {
        let baz_ret = try_ready!(fut.poll());
        self = FooState::Done(bar_ret + baz_ret);
        return self.poll();
      }
      FooState::Done(ret) => {
        return Ok(Async::Ready(bar_ret + baz_ret));
      }
    }
  }
}
```

With async/await, the *compiler* generates this for you!

Async/await is harder than that.

- Consider:

```
async fn foo(s: TcpStream) -> Result<u32, _> {
    let buf = [0u8; 1024];
    s.read(&buf[..]).await?;
    buf.parse()
}
```

- `buf` is on the stack, but must be valid across yield points
  - Compiler must generate “self-referential” code
  - **And** must guarantee that reference remains valid

Futures in the standard library

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

- Return type no longer required to be a `Result`
  - Some async operations are infallible, like pure compute
- Explicit `Context` used to “wake up” future later
  - Was previously done through implicit state in thread locals
- And then there’s `Pin`…

Pinning and self-referential values

- Recall that `await` means we need self-referential structs
    - e.g., a future holding a reference to a stack variable in an async fn
- Self-referential structs are okay as long as the target is *stable*
    - If a struct `T` lives on the heap, and isn't moved, then `T` can contain `&mut Self`
    - In fact, in general, we just require that `T` is not moved
    - And you cannot move a `T` without access to a `&mut T`
- Pin is a combination of a type and a contract (much like `Future`)
    - Cannot get `&mut T` from `Pin<T>` if `T` is self-referential
    - If you create `Pin<T>`, you are *promising* never to give out `&mut T`

A sketch of the argument for Pin

- `Future::poll` takes `Pin<&mut Self>`
- So executor must ensure `Future` won’t move to call poll
- So compiler can safely generate self-referential `Future` structs
  - It knows that poll will only be called if any `&mut Self` are still valid

- Types w/o self-references (`T: Unpin`) can ignore `Pin` contract
- Subtle stuff! See https://doc.rust-lang.org/std/pin/

Writing your own futures

- Mostly just writing an implementation of `Future::poll`
- Not actually that hard!
  - Can usually ignore `Pin` since your `T: Unpin`
- Main challenge: schedule wake-up after `NotReady`
  - Usually by “registering” on a reactor, but can call `notify()` manually
- Usually futures will mostly poll other futures
  - Contract upheld by induction (you return `NotReady` when they do)

Homework

Network-accessible HashMap

unsafe

Interior mutability

Send and Sync

Trait bounds

Hygienic macros

FFI

Asynchronous programming

Homework: ???

Homework: network-accessible HashMap

- Set up an efficient network frontend for your hash table
- Implement it with futures to avoid thread-per-connection
- Use `hyper` to get a futures-aware HTTP endpoint
- Maybe write a multi-threaded benchmark client too?