

CS 4414/5416: Systems Programming

Getting Started with Rust

David Bindel

2026-09-01

Warm-up exercise

```
enum Status { Assigned { to: String }, Done }

fn count_assignments(tickets: &[TicketStatus], name: &str) -> u32 {
    let mut count = 0;
    // TODO: Loop over tickets and count how many are assigned to name
    count
}

fn main() {
    let tickets = [ Status::Done,
                    Status::Assigned{ to: String::from("dbindel") },
                    Status::Assigned{ to: String::from("raja") },
                    Status::Assigned{ to: String::from("dbindel") }];
    println!("dbindel assigned {} tickets",
            count_assignments(&tickets, "dbindel"));
}
```

Recap

Last week (Ch 1-6):

- Most familiar bits (Ch 1-3): variables, simple types, arithmetic and logic ops, functions, conditionals, loops
- First take on ownership and borrowing
- **struct** and methods, **enum** and matchng

And some programming practice on Friday.

- I did teach pre-3410C a while back
- I know I went too fast at the end of Thu!
- We'll take more runs at ownership and borrowing
 - See also: spaced repetition
- And we'll see `enum` and `struct` in practice today

Where we're going (revised)

- Today
 - Traits and generics
 - Error handling
 - Modules and libraries (likely on your own)
- Thu
 - Memory layouts
 - Smart pointer types
 - Ownership, lifetimes, borrowing revisited
- Fri: Process registry lab, part 1

Where we're going (revised)

- Next Tue
 - Testing
 - Iterators and closures
 - Cargo and crates
- Next Thu
 - Dynamic traits and vtables
 - Advanced Rust, urther resources
- Fri: Process registry lab, part 2
- And then on to performance tuning.

There are lots of resources to learn Rust!

- The Rust book is great
- So is Rust by Example
- I also really like Comprehensive Rust
 - My slides are heavily inspired by Comprehensive Rust

Spend some time exploring! I have suggestions.

“If you want to go fast, go alone, if you want to go far, go together”

– Source unknown

Thus Thursday, Sep 3 from 4:30-5:40 in Gates 114.

- Maybe meet more folks from this class
- Or just meet more people generally!

Organized by Women in Computing at Cornell (WICC).

- Project list is posted!
 - First project repository is linked (or will be shortly)
 - Check-in submission due next Monday
 - Final submission due two Mondays hence
- We have Google cloud credits! See Ed for details
- Add deadline is Sep 8 – ticket if you need it!

Traits

What's a trait?

```
trait Player {  
    fn describe(&self) -> String;  
}
```

- Specifies method signatures (and associated types)
- Acts much like an interface in Java
- A foundation for generic behaviors

Implementing traits

```
trait Animal {  
    fn describe(&self) -> String;  
}  
  
struct Dog(String);  
  
impl Animal for Dog {  
    fn describe(&self) -> String {  
        format!("{}", self.0)  
    }  
}
```

- Uses an `impl` block like other methods
 - but with the `TraitName` for `StructName`

Super traits

```
struct Dog(String);

trait Animal {
    fn describe(&self) -> String;
}

impl Animal for Dog {
    fn describe(&self) -> String { format!("{}", self.0) }
}

trait Pet: Animal {
    fn greet(&self) -> String;
}

impl Pet for Dog {
    fn greet(&self) -> String { format!("Good boy, {}!", self.0) }
}
```

Super traits

```
struct Dog(String);

trait Animal {
    fn describe(&self) -> String;
}

impl Animal for Dog {
    fn describe(&self) -> String { format!("{}", self.0) }
}

trait Pet: Animal { // Must implement Animal to implement Pet
    fn greet(&self) -> String;
}

impl Pet for Dog {
    fn greet(&self) -> String { format!("Good boy, {}!", self.0) }
}
```

Associated types

```
trait LinearMap {  
    type Input; // Associated type  
    type Output; // Associated type  
    fn map(&self, x: &Self::Input) -> Self::Output;  
}  
  
struct RotateMap(f64);  
  
impl LinearMap for RotateMap {  
    type Input = [f64; 2];  
    type Output = [f64; 2];  
    fn map(&self, x: &Self::Input) -> Self::Output {  
        let c = self.0.cos();  
        let s = self.0.sin();  
        [c*x[0]-s*x[1], s*x[0]+c*x[1]]  
    }  
}
```

```
#[derive(Debug, Copy, Clone, PartialEq)]  
struct Point2d(f64, f64);
```

- A **derive** macro writes code for you
- Standard use case is implementing a trait
- Here, one line implements
 - **Debug**: Generate debug formatter
 - **Copy**, **Clone**: implement copying and **clone**
 - **PartialEq**: partial equality (why partial?)

Exercise

```
// From Rust standard library
// pub trait Add<Rhs = Self> {
//     type Output;
//     fn add(self, rhs: Rhs) -> Self::Output;
// }
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point2d(f64, f64);

// TODO: Implement the Add trait for Point2d
```

Generics

Generic functions

```
use std::fmt::Display;

fn show<T: Display>(x: T) {
    println!("Now showing: {x}");
}

fn main() {
    show(1);
    show("The Matrix");
}
```

- Include type parameters in angles
- Types usually constrained by *trait bounds*

Trait bounds

```
use std::ops::Add;
use std::ops::Mul;
use std::convert::From;

fn dot<T, const N: usize>(x: &[T; N], y: &[T; N]) -> T
where
    T: Add<Output=T> + Mul<Output=T> + From<u8> + Copy
{
    let mut result = T::from(0)();
    for i in 0..N {
        result = result + x[i]*y[i];
    }
    result
}
```

- Trait bounds can get complicated!
- **where** separates trait bounds from type declaration
- + allows trait bound intersection

Trait bounds

```
use std::ops::Add;
use std::ops::Mul;
use std::convert::From;

trait ScalarType: Add<Output=Self> + Mul<Output=Self> + From<u8> + Copy {}

impl ScalarType for f64 {}
impl ScalarType for f32 {}

fn dot<T: ScalarType, const N: usize>(x: &[T; N], y: &[T; N]) -> T {
    let mut result = T::from(0);
    for i in 0..N {
        result = result + x[i]*y[i];
    }
    result
}
```

Can also use supertraits to consolidate trait bounds.

Generic types

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point2d<T>(T,T);

impl<T: Add<Output=T>> Add for Point2d<T> {
    type Output=Self
    fn add(self, other: Self) -> Self {
        return Point2d<T>(self.0 + other.0, self.1 + other.1);
    }
}

fn main() {
    let p = Point2d(1,2);
    let q = Point2d(3,4);
    println!("{:?}", p+q);
}
```

We have already been using generic traits!

```
pub trait From<T>: Sized {  
    fn from(value: T) -> Self;  
}
```

- Examples are `String::from("foo")`, `T::from(0)`
- Will talk about `Sized` later

Exercise

```
use std::cmp::Ord;          // Method cmp(&self, &x) -> Ordering
use std::cmp::Ordering;    // enum Ordering { Less, Equal, Greater }

fn bubblesort<...>(v: ...) { // TODO: What should the types be?
    for _ in 0..v.len() {
        for i in 1..v.len() {
            // TODO: Call v.swap(i-1, i) if v[i-1] > v[i]
        }
    }
}

fn main() {
    let mut x = vec![8, 6, 7, 5, 3, 0, 9];
    bubblesort(&mut x);
    println!("{x:?}")
}
```

Error handling

Call `panic!()` for *unrecoverable* errors

- On **`assert!`** failure, out of bound access, OOM
- Causes stack to be unwound (cleans things up)
- Can be caught, but only used in rare cases
 - e.g. when Rust called from another language

For *recoverable* errors, use a **`Result`**

Result revisited

```
enum Result<T,E> {  
    Ok(T),  
    Err(E)  
}
```

Use `Result` for *recoverable* errors

- `unwrap()` or `expect(msg)` methods match `Ok` or panic
- Can check with `is_ok` or `is_err`
- Can handle with `match`, `if let`, `let else` constructs

- Various default handlers extract data
 - `or_else(f)` - Calls handler on `Err`
 - `unwrap_or(default)` - Returns `default` on `Err`
 - `unwrap_or_else(f)` - like `or_else`, but unwraps
- Can also transform data to `Option`
- Or apply mappings

More on these after we cover closures.

Error propagation

```
use std::fs::File;
use std::io::Read;

fn read_file(fname: &str) -> Result<String, std::io::Error> {
    let mut contents = String::new();
    File::open(fname)?.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() {
    let Ok(s) = read_file("input.txt") else {
        println!("Could not read file");
        return;
    };
    println!("{}", s);
}
```

Try operator (?) propagates errors (unwrap `Ok` or `return` error)

Actual behavior of try:

```
// Expansion of expression?  
match expression {  
    Ok(value) => value,  
    Err(err) => return Err(From::from(err))  
}
```

Useful for when there are multiple error types.

Error conversions

```
#[derive(Debug)]
enum MyError { IOError(std::io::Error), EmptyUserError }
impl Error for MyError {}
impl From<std::io::Error> for MyError {
    fn from(err: std::io::Error) -> Self { Self::IoError(err) }
}

fn read_username(path: &str) -> Result<String, MyError> {
    let mut username = String::new();
    std::io::fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(MyError::EmptyUserError);
    };
    Ok(username)
}
```

The `thiserror` crate provides an `Error` derive macro and various other utilities.

```
enum Option<T> {  
    Some(T),  
    None  
}
```

What is the difference between **Option** and **Result**?

- **None** carries no error information
- Because it usually does not mean an error!
 - “No result” is not the same as “there was a problem”

Exercise

```
use thiserror::Error;
struct User { id: String, name: String }

#[derive(Debug, Error)]
enum UserError {
    #[error("IO error: {0}")]
    IoError(std::io::Error),
    #[error("Missing username: {0}")]
    MissingError(String),
}

fn read_users(path: &str) -> ... { // TODO: Fill in the type
    // Implementation elided, but return Vec<User> or an I/O error
}

fn lookup_name(path: &str, id: &str) -> ... { // TODO: Type?
    // TODO: Call read_users to get to name (if id present)
}
```

Modules and libraries

```
mod foo {  
    pub fn do_thing() { println!("Doing the foo!"); }  
}  
  
mod bar {  
    pub fn do_thing() { println!("Going to the bar!"); }  
}  
  
fn main() {  
    foo::do_thing();  
    bar::do_thing();  
}
```

```
mod parent {  
    fn private() { println!("Very secret"); }  
    pub fn do_thing() { private(); }  
    pub mod child {  
        pub fn do_thing() { println("Inner: "); super::private(); }  
    }  
}  
  
fn main() {  
    parent::child::do_thing();  
}
```

- Module hierarchy tracks file hierarchy in a crate
 - Top level in `lib.rs` or `main.rs`
 - In crate, `boots::cats` says to look in `src/boots/cats.rs`
- `super` points to parent module
- `self` points to current module
- `use` brings a name into current space (`pub use` re-exports)

std structs and traits

In `std::vec::Vec`

```
pub struct Vec<T, A=Global> where A: Allocator {...}
```

Selected associated functions:

- `new()`: Create new empty `Vec`
- `with_capacity(c)`: Create empty `Vec` with capacity `c`

Selected methods:

- `push(v)` and `pop()`: Stack operations
- `insert(i,v)` and `remove(i)`: Indexed insert/remove
- `len()` and `capacity()`: Length and capacity

In `std::string::String`

```
pub struct String {...}
```

- `push` a character or `push_str` a string
- `len` and `capacity` are in *bytes*
- `nth(i)` for character `i`, index by bytes
- `chars` for iterator over characters

In `std::collections::HashMap`

```
pub struct HashMap<K, V, S = RandomState, A: Allocator = Global> { ... }
```

- `insert(k,v)` - Insert a key
- `get(k)` - Get value (returns `Option`)
- `entry(k)` - Get an `Entry`

In `std::cmp`

- `PartialEq<RHS>` for partial equivalence (`==` and `!=`)
- `Eq`: `PartialEq` if `PartialEq` actually an equivalence
- `PartialOrd`: `PartialEq` for partial ordering
- `Ord`: `Eq` + `PartialOrd` for complete ordering
- Floating point does *not* implement `Eq`

In `std::ops`

- `Add`, `Sub`, `Mul`, `Div` for implementing `+`, `-`, `*`, `/`
- `AddAssign`, etc for `+=` and the like
- `Index` and `IndexMut` to overload subscripting
- `Fn`, `FnMut`, and `FnOnce` to overload calling
 - We will see these more next week

In `std::convert`

- `From<T>` for `U` says how to convert a `T` to a `U`
- `Into<T>` for `U` says how to convert a `U` into a `T`
- `From<U>` for `T` implies `Into<T>` for `U`
- `TryFrom` and `TryInto` if failure is possible

In `std::io`

- `Read` trait implements `read()` to read bytes
- `Write` trait implements `write()` to write bytes

Wrapping up

Isn't this too fast?

Yes! Six lectures is only about 7.5 hours!

- *But* you have the slides (and pointers) and can read
- You have all learned a few languages already
- And real learning will mostly happen as you use Rust
- And you can ask questions in Ed and OH!

Anagrams!

You learn by doing, so we have a first project:

- *Anagrams* are words with the same letters rearranged
 - Example: **listen** and **silent**
 - This is an equivalence relation!
- Goal is to find anagram equivalence classes fast
 - By a representative word or by maximal size
- Harder goal is to include word pairs!

Check-in due next Monday, project due in two weeks.

Tell me either:

- One thing you're confused by, or
- One thing that surprised you, or
- One thing that I might not know (Rust or otherwise)