

# CS 4414/5416: Systems Programming

The End of the Tour

---

David Bindel

2026-09-10

## Warm-up exercise

```
/// Adapt an iterator to discard entries outside lo..=hi
/// This should work for iterators over any T implementing Ord
fn censor<T>(..., lo: ..., hi: ...) -> ...
where
    T: ...
{
    ...
}
```

- Who is your neighbor?
- What is (or was) their most fun *non-CS* undergrad class?

# Logistics

---

- Marked as graded the ones that (I think) are done
- New class sheets have combs!
- Feel free to print your own (with typed netids)

- You have a lab tomorrow
- These are *opportunities to practice Rust*
  - This is not work I need solved
  - AI producing this code misses the point
- This is a time to ask more questions!

- The `README.md` and files are updated
- You have new functionality to implement in `keys.rs`
- This includes tests for `anagrams.rs` and `keys.rs`
  - You should edit `tests/keys_test.rs`
- You should submit a `crate` file for P2
  - Just submitting `rs` files will not work!

- Exam from 7:30-9:30 (target 7:30-9:00) on 9/22
  - NetID A-L: Olin 155
  - NetID M-V: Olin 255
  - NetID W-Z: Olin 165
  - Early exams (5:30) in Statler 396
- There is a conflict form if main time does not work
- Covers the Rust material (first three weeks)

Performance of single-thread codes, requiring

- Modeling
- Measurement
- Understanding memory effects
- Using the compiler effectively

Two weeks:

- Next: Profiling, memory, and optimization
- Then: Compilation, linking, loading, and build systems

- Next week, we start using GCP
- Retrieval link for Google Cloud coupons is on Ed
  - One per student email
  - Please use your Cornell-based email!
- Recommended: GCE Linux VM walk-through
  - Use an inexpensive instance for setup
  - Clean it up when done

## Recap

---

# Why Rust?

- This is a class on *systems programming*
  - Low-level, performant, resource-constrained
  - Load-bearing code at the “bottom of the stack”
- Rust is a *safe* systems programming language
  - Safety improves security
    - See CISA/NSA joint guide (and 2024 report), “Safer with Google”
  - Safety improves reliability generally
  - Safety can *also* improve performance (more on this later)

- Started by Mozilla, currently about 13% of Firefox codebase
  - Currently more than C, less than C++
  - NB: about half of codebase is HTML, JS, or Python
- Used in Linux, Android, Microsoft (firmware and systems components), Amazon (across AWS), OpenAI (Codex CLI), Discord...
  - Comprehensive Rust on bare-metal, Android, Chromium
- Adoption is quite fast given legacy scale!
  - And Code Metal and Google and Microsoft working on large-scale translation of C/C++ into Rust

What's familiar from C/C++/Java?

- Lots of control flow and syntax, basic data types
- Traits a bit like Java interfaces
- Generics are reminiscent of C++ templates

What's familiar from functional languages?

- Broadly used type inference
- Enums, structure matching, `match`
- Some functional features

What's unfamiliar? Ownership and borrowing.

- Each entity has one owner
- One mutable or many shared refs (not both)
- All references must be valid

Also maybe error handling with `Result`? Macros?

Stated module goals:

- Understanding fundamentals of Rust
- Understanding how some of this is implemented

But:

- Three weeks is just enough for a survey
- Learn more by doing the projects this semester!

Okay, on to some remaining topics...

## Modules (try 2)

---

```
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)

## Dynamic traits

---

## Different uses of traits

- *Static* polymorphism (generics)
  - Everything resolved at compile time
  - May produce multiple versions of code
  - No (direct) run-time cost, inlining possible
- *Dynamic* polymorphism (dyn traits)
  - Resolve version to dispatch at run time
  - Single code version works for multiple types
  - Requires dispatch via a *vtable*

# Usage

```
trait Hello { fn speak(&self); }  
struct Dog { name: &'static str }  
struct Cat { name: &'static str }  
  
impl Hello for Dog {  
    fn speak(&self) { println!("{}", self.name); }  
}  
  
impl Hello for Cat {  
    fn speak(&self) { println!("{}", self.name); }  
}  
  
fn speak_to(who: &dyn Hello) { who.speak(); }  
  
fn main() {  
    speak_to(&Dog {name: "Fido"});  
    speak_to(&Cat {name: "Pete"});  
}
```

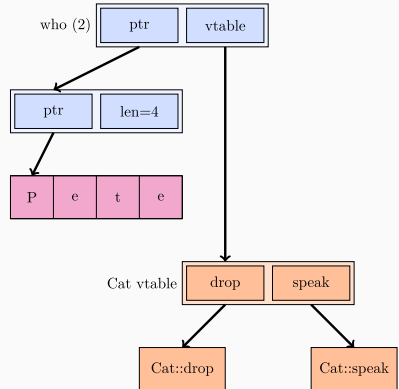
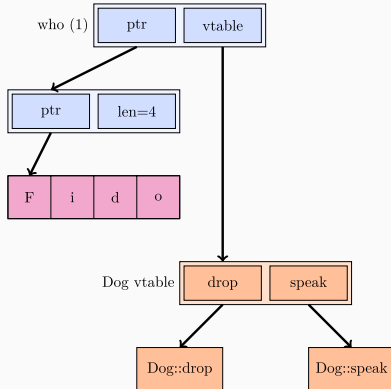
```
// Hello, Dog, Cat, speak_to as before

fn schrodinger(name: &'static str) -> Box<dyn Hello> {
    Box::new(&Cat {name})
}

fn main() {
    speak_to(&*schrodinger("??"));
}
```

- Don't know size of a dyn trait object
  - Can only get to it via some pointer type
  - Implementation is a fat pointer: `(ptr, vtable ptr)`
- Methods are stored in a virtual function table (vtable)
  - Assembled by the compiler, one slot per method
  - Must also contain information on how to **drop**
- **dyn Trait** interfaces must be type generic
  - Can't use associated types
  - Can't return **Self**

# Implementation



## Dynamically sized types

---

## What's a DST?

Dynamically sized type examples:

- Trait objects
- Slices
- `str`

Size is *not known* at compile time.

Two special trait bounds:

- **Sized** when size is known at compile time
  - Can call `std::mem::size_of::<T>()` for **Sized** types
- **?Sized** when unknown size is allowed

Type parameters and associated types are **Sized** unless explicitly declared **?Sized**

## New Type idiom

---

An *idiom* is a common code pattern

- Every language has its own idioms
  - Often libraries, frameworks, etc do as well
- Code following standard patterns is *idiomatic*
- One standard Rust idiom is *NewTypes*

## NewType idea

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

fn main() {
    let height = Meters(2.0); // Says it on the label
    let h2 = 180;             // Probably centimeters?
    if height > Meters(1.0) {
        // You can go on the rides
    }
    if height > h2 {           // ERROR: Types don't match
        // You are confused
    }
}
```

- Just a single-field struct!
- Type carries semantic information (e.g. units)
- Can add functionality via methods, traits

## NewType example

```
#[derive(Debug, Clone, PartialEq)]
pub struct GithubId(String);

#[derive(Debug, Clone, PartialEq)]
pub struct Password(String);

#[derive(Debug, Clone, PartialEq)]
pub struct UserRecord(usize);
```

- The structs are public, the fields are not!
- GithubId, Password, and String are now different types

## NewType example

```
pub fn lookup_user(user: &GithubId) -> Result<UserRecord, GithubId> {  
    //...  
}  
  
pub fn validate_password(user_rec: UserRecord, password: &Password)  
    -> Result<(), PasswordError> {  
    //...  
}  
  
pub fn login(user: &GithubId, password: Password)  
    -> Result<Session, LoginError> {  
    let user_info = lookup_user(user);  
    validate_password(user_info, &password)?;  
    //...  
}
```

Student's choice

---

Five additional actions:

- Dereference a raw pointer
- Call unsafe functions or methods (includes FFI)
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of **union**

# What unsafe Rust is not

Not a free-for-all!

- **unsafe** code is scoped (use in small doses)
- References are still checked in unsafe code
- Ownership guarantees must still be respected
  - OK C code may be erroneous in analogous unsafe Rust
  - e.g. it's Not Good to mutate something when a shared reference has been borrowed

For more see the Rustonomicon

- CS colloquium today - Jonathan Aldrich on BorrowSanitizer and Rust/C FFI (video link will go up presently)
- We will indeed see unsafe Rust again when dealing with FFI

## Basic rules

- Each entity has one owner
- One mutable ref or many immutable refs (not both)
- References must always be valid

... with addenda for smart pointers (esp `RefCell`)

Good examples in Rust By Example!

# Ownership

```
struct Password(String);

fn check_password(password: Password) {
    // Consumes the password
}

fn main() {
    let password = get_password();
    login(password);
    // ...
    login(password); // Error, ownership transferred on first call!
}
```

- Each entity has one owner
- Ownership can be transferred (e.g. functions that *consume* arguments)

## Reference exclusivity rules

```
struct KeyedDict<'a> {  
    words: &'a [String],  
    index: Vec<(usize, usize)>  
}  
  
fn main() {  
    let mut dict: Vec<String> = load_dict();  
    add_word(&mut dict, String::from("vibecode")); // OK  
    let index = keyed_index(&dict); // OK (mut ref is done)  
    add_word(&mut dict, String::from("truthiness")); // ERROR!  
}
```

- One mutable ref or many immutable

```
fn bad_idea() -> &usize {  
    let x: usize = 1;  
    &x  
}
```

- No way to try to create uninitialized references in Rust
- Checked: references cannot outlive what they point to

# Generics and trait bounds

```
use num_traits::{Num, NumOps};

fn dot<T: Num + NumOps, const N: usize>(x: &[T;N], y: &[T;N]) -> T {
    let mut result = T::zero();
    for (xi,yi) in x.iter().zip(y.iter()) {
        result = result + xi*yi;
    }
}
```

Rust generics parameterize over types, constants

- But not every type will work!
- *Trait bounds* specify required functionality

## Generics and trait bounds

```
fn mymax<T,Titer>(mut t: Titer) -> Option<T>
where
    T: Ord,
    Titer: Iterator<Item=T>
{
    let Some(item) = t.next() else { return None; };
    let mut max_item = item;
    for item in t {
        if item > max_item {
            max_item = item;
        }
    }
    Some(max_item)
}

fn main() {
    let v = vec![1, 2, 2, 1];
    println!("{}", mymax(v.iter()));
}
```

Basic idea:  $T : S + R$  for traits  $S$  and  $R$  means

- Type  $T$  must implement  $S$  and  $R$
- Implementing trait  $T$  requires  $S$  and  $R$ 
  - We say  $S$  and  $R$  are *supertraits* of  $T$

Again, Rust by Example has great examples!

## Wrapping up

---

- We are done talking *just* about Rust
- But there will be lots more Rust ahead
  - ... in the interest of actual systems programming
- Start reading from CS:APP3e!

What are your thoughts on course logistics?

- Slides + exercise format?
- Class sheets?
- Labs?

Constructive feedback would be great!