

CS 4414/5416: Systems Programming

Getting Started with Rust

David Bindel

2026-08-27

```
fn mystery(n: u32) -> u32 {  
    let mut result = 0;  
    for i in 1..=n {  
        result += if i%2 == 0 { i } else { 0 };  
    }  
    result  
}  
  
fn main() {  
    let answer = mystery(10);  
    println!("The answer is {answer}");  
}
```

Ask someone near you (different from Tue):

- Name? Salty snacks or sweet?
- What is the output of the code above?

First lab tomorrow!

- Bring your computer (or find a friend with one)
- Install the Rust toolchain in advance
- We will assume you are able to
 - Work in a favored text editor
 - Open a terminal
 - Use `git`

Recap



- See <https://rust-lang.org/> (particularly the playground)
- You will want to install locally using Rustup
- Make sure you are using the most recent version (1.98.0)

Tue: the most-familiar bits (Ch 1-3)

- Variable bindings with `let` (and `let mut`)
- Scalar types, tuples, arrays, strings
- Standard arithmetic and logical ops
- Functions
- Conditionals and loops

Mostly unsurprising for a C programmer.

Today: the more Rust-specific bits (Ch 4-6)

- First take on ownership and borrowing
- **struct** and methods
- **enum** and matching

Week 2: heap objects, memory layouts, smart pointers Week 3: error handling, traits and generics (And then we move on to systems programming)

A silly demo

```
fn main() {  
    let mut v = vec![1, 2, 3];  
    v.push(4);  
    println!("{:?}", v);  
}
```

What do you think this outputs?

A silly demo

```
fn main() {  
    let mut v = vec![1, 2, 3];  
    v.push(4);  
    println!("{:?}", v);  
}
```

- Macro **vec!** creates and initializes a **Vec**
- This is a growable *heap*-based array type
 - Implemented as a **struct** (more soon)

A silly demo

```
fn main() {  
    let mut v = vec![1, 2, 3];  
    v.push(4);  
    println!("{:?}", v);  
}
```

- Vec type implements the *method* push
 - Can attach methods to a **struct**

```
fn main() {  
    let mut v = vec![1, 2, 3];  
    v.push(4);  
    println!("{:?}", v);  
}
```

- Then print using **Debug** formatting (`:?`)
 - Works because **Vec** implements **Debug** *trait*
 - `println!` macro quietly *borrow*s a *reference* to `v`

A silly demo

```
fn main() {  
    let mut v = vec![1, 2, 3];  
    v.push(4);  
    println!("{:?}", v);  
}
```

- Creating **Vec** allocated dynamic memory
- Automatically deallocated (**Dropped**) at end

Let's dig into this a little deeper.

Ownership and borrowing, take 1

- C manages dynamic memory *explicitly* (`malloc/free`)
 - Prone to use-after-free, uninitialized use, etc
- Java is *garbage collected*
 - Gets rid of use-after-free errors
 - But GC adds unpredictable overhead
- C++ automates this some with RAI pattern
 - Calls destructor when object goes out of scope
 - Better than C, but still error prone

- No garbage collector
 - Explicit management is a benefit for systems coding
 - Avoid GC overheads (particularly for embedded)
- Supports clean-up similar to C++ destructors
 - Allow user-defined cleanup through **Drop**
 - Invoke when *owner* goes out of scope
- Type system *enforces* safe resource management
 - C++ RAII is a programmer pattern, not type checked

A starter example

```
fn printvec(v: Vec<i32>) {  
    println!("{:?}", v);  
}  
  
fn main() {  
    let mut v1 = vec![1, 2, 3];  
    printvec(v1);  
}
```

A starter example

```
fn printvec(v: Vec<i32>) {  
    println!("{:?}", v);  
}  
  
fn main() {  
    let mut v1 = vec![1, 2, 3];  
    printvec(v1);  
    v1.push(4); // Error: borrow of moved value  
    printvec(v1);  
}
```

- Problem: Ownership moved in `printvec` call!
- After move, `v1` cannot be used

A starter example

```
fn printvec(v: Vec<i32>) -> Vec<i32> {  
    println!("{:?}", v);  
    return v;  
}  
  
fn main() {  
    let mut v1 = vec![1, 2, 3];  
    let mut v2 = printvec(v1);  
    v2.push(4);  
    printvec(v2);  
}
```

- First `Vec` is owned by `v1`
- Ownership moved to argument of `printvec`
- Ownership moved to `v2` via return
- When `v2` goes out of scope, `drop` value

- Every Rust value has a single *owner* at any time
- Ownership can *move* by **let**, function parameter, or return
- Role of owner is to clean up resources
 - Happens automatically when owner goes *out of scope*
- Simple Rust values implement **Copy** (just make a copy)
 - Some types might implement **Clone** (`v.clone()`)
 - Excessive **clone** calls are inefficient

```
fn main() {  
    let a = String::from("a");  
    let b = String::from("b");  
    {  
        let c = String::from("c");  
    } // drop(c) called implicitly here  
    drop(b);  
} // drop(a) called implicitly here
```

- Rust is *lexically scoped* by code blocks
- Drop values when owner goes out of scope
 - “Please take out your trash when you leave the building.”

Reference example

```
fn printvec(v: &Vec<i32>) {  
    println!("{:?}", v);  
}  
  
fn main() {  
    let mut v1 = vec![1, 2, 3];  
    printvec(&v1);  
    v1.push(4);  
    printvec(&v1);  
}
```

- A *reference* is a safe pointer (non-owning)
 - Unsafe Rust also allows raw pointers
- In Rust we *borrow* a reference
 - Hence the *borrow checker* enforces reference safety

Reference operations

```
fn main() {  
    let x = 1;  
    let rx = &x;  
  
    let mut y = 1;  
    let ry = &mut y;  
    *ry = 2;  
  
    println!("{}", *rx);  
}
```

- Borrow a reference with `&` (or a mutable ref with `&mut`)
- The `*` operator dereferences
- The `.` operator dereferences implicitly as needed (no `->`)
- Comparison operators also implicitly dereference

What could go wrong?

```
fn main() {  
    let mut v1 = vec![1, 2, 3];  
    let r1 = &v1[1];  
    v1.push(4);  
    println!("{}", *r1);  
}
```

Rust will not compile this. Why?

1. References must always be valid
2. **One** mutable reference *xor* many immutable references

Reader/writer safety enables “fearless concurrency”

- Rule 2 makes it easier to maintain data invariants
- *Particularly* in concurrent programs (later in semester)

```
fn printvec2(v: &[i32]) {  
    println!("{:?}", v);  
}  
  
fn main() {  
    let v = vec![1, 2, 3, 4];  
    printvec2(&v);           // Prints [1, 2, 3, 4]  
    printvec2(&v[1..3]);    // Prints [2, 3]  
}
```

- A *slice* is a (**ref**, **len**) pair (a “fat pointer”)
- Also have *string slice* (**&str**) for UTF-8 strings
- Same shared/mutable rules as references
- Rust supports implicit conversions to slices

Exercise

A 2D rotation of the point $(x^{\text{in}}, y^{\text{in}})$ by θ looks like

$$x^{\text{out}} = \cos(\theta)x^{\text{in}} - \sin(\theta)y^{\text{in}}$$

$$y^{\text{out}} = \sin(\theta)x^{\text{in}} + \cos(\theta)y^{\text{in}}$$

Suppose we represent 2D points as `[f64; 2]`. How would you complete the following function?

```
fn rotate(theta: f64, xy: ...) { // Possible types for xy?  
    let c = theta.cos();  
    let s = theta.sin();  
    // Overwrite xy with rotated xy  
}
```

What does a call to `rotate` look like?

Structs

Structs and tuple structs

```
// Named struct
struct Point2dNamed {
    x: f64,
    y: f64,
}

// Tuple struct
struct Point2d(f64, f64);

fn main() {
    let pn = Point2dNamed {x: 1.0, y: 2.0, };
    let pt = Point2d(3.0, 4.0);
    println!("Named: {}, {}", pn.x, pn.y);
    println!("Tuple: {}, {}", pt.0, pt.1);
}
```

- Named **struct** looks a lot like C
- Tuple **struct** fields indexed, not named

Initialization

```
struct Point2dNamed {  
    x: f64,  
    y: f64,  
}  
  
fn main() {  
    let x = 1.0;  
    let y = 1.0;  
    let pn = Point2dNamed {x, y};  
    println!("Named: {}, {}", pn.x, pn.y);  
}
```

- Some syntactic sugar for field initialization

Destructuring

```
struct Point2dNamed {  
    x: f64,  
    y: f64,  
}  
  
struct Point2d(f64, f64);  
  
fn main() {  
    let pn = Point2dNamed {x: 1.0, y: 2.0, };  
    let pt = Point2d(3.0, 4.0);  
    let Point2dNamed{x: pnx, y: pny} = pn;  
    let Point2d(ptx,pty) = pt;  
    println!("Named: {}, {}", pnx, pny);  
    println!("Tuple: {}, {}", ptx, pty);  
}
```

- Support for *destructuring* is surprisingly handy

Methods and associated functions

```
struct Point2d(f64, f64);

impl Point2d {

    // Associated function
    fn origin() -> Self { Self(0.0,0.0) }

    // Method
    fn len(&self) -> f64 {
        let Point2d(x,y) = self;
        (x*x+y*y).sqrt()
    }
}

fn main() {
    println!("{}", Point2d(3.0,4.0).len());
    println!("{}", Point2d::origin().len());
}
```

Macro-generated methods

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

fn main() {
    let p = Point2d(1.0, 2.0);
    println!("{p:?}");
}
```

- **derive** macros auto-implement methods and traits
- This eliminates a lot of boilerplate!
- We will discuss *using* macros
- Leave *implementing* macros to the Rust book (not on exams)

A 2D rotation of the point $(x^{\text{in}}, y^{\text{in}})$ by θ looks like

$$x^{\text{out}} = \cos(\theta)x^{\text{in}} - \sin(\theta)y^{\text{in}}$$

$$y^{\text{out}} = \sin(\theta)x^{\text{in}} + \cos(\theta)y^{\text{in}}$$

- Suppose we use `struct Point2d(f64, f64)`.
- Write the code for a method `p.rotate(theta)` to produce a rotated version of the point `p`

Enums

Enum basics

```
enum Direction {  
    North,  
    South,  
    East,  
    West,  
}  
  
enum PlayerMove {  
    Skip,  
    Run(Direction),  
    Jump {x: u32, y: u32},  
}
```

- As in C, `enum` introduces an enumerated type
- Unlike C, can attach a payload to options

Option

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

- This is an enum with *type parameters* (<T>)
- **Option** type used for optional results (of type T)
- Often used where C might have a sentinel value (or **NULL**)

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

- `Result` type used for error handling
- We will have much more to say about this soon

Matching

```
fn option_used<T>(x: &Option<T>) -> bool { // Generic function!
    match x {
        Some(_) => true,
        None => false,
    }
}

fn main() {
    let x = Option::Some(1);
    println!("{}", option_used(&x));
}
```

- `match` is a little like C `switch` (or OCaml `match`)
- Can have non-enum matches as well
- All cases covered by some “arm” (`pattern => result`)
- Can add default arm of `_ => result`

Control flow (if-let)

```
fn print_some<T: std::fmt::Display>(x: &Option<T>) {  
    if let Some(xx) = x {  
        println!("{xx}");  
    }  
}  
  
fn main() {  
    let x: Option<i32> = Some(1);  
    print_some(&x);  
}
```

- `if-let` is often more ergonomic than `match`
- Note: `T` must implement `Display` *trait* (more to come)

Control flow (let-else)

```
fn print_some<T: std::fmt::Display>(x: &Option<T>) {  
    let Some(xx) = x else { return; };  
    println!("{xx}");  
}  
  
fn main() {  
    let x: Option<i32> = Some(1);  
    print_some(&x);  
}
```

- Can use `let-else` for defaults or fast return
- Rust book: this is “staying on the happy path”

Exercise (if time)

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

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

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

Getting Rusty

But why?

A language that doesn't affect the way you think about programming, is not worth knowing. – Alan Perlis

Learn Rust because:

- It's fun!
- You'll need it for the rest of this semester
- You might write Rust code at a future job
- I hope it will *change how you think*

(Asking AI to generate Rust will not change *your* thinking.)

How to get to Carnegie Hall

- Practice, practice, practice!
 - Do the exercises in Ch 1-6 of the Rust Book
 - Or Day 1 of Comprehensive Rust
 - Or through 3.2 in 100 Exercise to Learn Rust
- Come to lab tomorrow and practice with friends!
- Ask questions (and share insights) on Ed!
- You learned C (and likely Java, Python, OCaml)... you can learn this

For your sheet: what was most confusing today?