

CS 4414/5416: Systems Programming

Getting Started with Rust

David Bindel

2026-08-25

Why Rust?

Why Rust?

```
int main(int argc, char** argv) {  
    int a[1];  
    a[1] = -1;  
    return 0;  
}
```

Systems programming is *resource-constrained* programming

- This often means *low-level* code
- Traditional languages are C and C++
- But these have lots of *undefined behavior*
 - Out-of-bound access, use-after-free, race conditions, ...

Is this the inevitable cost of low-level control? No!

Why Rust?

- A (mostly) *safe* systems programming language
 - Designed for performance and low-level control
 - Also type safety, concurrency, and memory safety
- Early public versions sponsored by Mozilla
 - Version 0.1 was in 2012
 - Taken over by Rust Foundation in 2021
- And it has taken off!
 - Used at Google, Microsoft, Meta, in the Linux kernel
 - Has a sizeable developer community

For any reasonable language, we either have a choice

- Allow some incorrect programs (with UB)
- Only allow correct programs (but reject some as well)

Safe Rust: restrict to programs it can prove correct.

(There is also *unsafe* Rust, which we will mostly avoid.)

- If it type checks, it is guaranteed free some problems!
- If it doesn't type check, the messages are helpful.

For heap objects (and the like):

- There is exactly one owner at any time
- Pointer access (created by “borrowing a reference”) involves
 - Any number of readers *xor*
 - One writer
- The object has a *lifetime* that no reference can exceed

This is all checked by a *borrow checker*.

Challenge: A performant doubly-linked list in Rust is hard!

Getting Started



- 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 (1.97.1)

Key reference: <https://doc.rust-lang.org/book/>

Rough plan is

- Today: the mostly-familiar bits (Rust book Ch 1-3)
- Thurs: overview of 4-6 in Rust book
- Next week: pointers, memory, ownership, borrow
- Two weeks: error handling, traits and generics, iterators

Do read and practice outside lecture! And join us on Ed!

Hello world

At the shell, run `cargo new hello` for new Rust project

```
hello/  
  Cargo.toml  
  src/  
    main.rs
```

Within `hello`, run `cargo run`

```
Compiling hello v0.1.0 (/Users/dbindel/work/class/cs4414-  
f26/hello)  
Finished `dev` profile [unoptimized + debuginfo] target(s)  
Running `target/debug/hello`  
Hello, world!
```

We can also try this out in the playground.

What is in `src/main.rs`?

```
fn main() {  
    println!("Hello, world!");  
}
```

- `fn` introduces a keyword or function
- Curly braces for code blocks (as in C)
 - Semicolons *separate* statements
 - Last statement gives a value for the block
- `println!` is a macro (vs a function)

Variables

```
fn main() {  
    let answer: u32 = 42;  
    println!("The answer is {answer}");  
}
```

- `let name: type = value` defines an immutable variable
- The type can frequently be inferred (and omitted)
- Value of `answer` is interpolated into the format string

Scalar Types and Values

	Types	Literals
Signed	i8, i16, i32, i64, i128, isize	-1, 0, 1_000, 123_i64
Unsigned	u8, u16, u32, u64, u128, usize	0, 123, 10_u16
Floating point	f32, f64	3.14, -6.02e23, 2_f32
Unicode	char	'a', '🐶'
Boolean		true, false

Compound Types and Values

```
let tup: (i32,i32) = (1,2);           // Tuple
let arr: [f64; 2] = [3.14, 2.72, 6.7]; // Array
let s: &[f64] = &arr[0..2];           // Array slice

let tfirst = tup.0; // Tuple access
let afirst = arr[0]; // Array access
```

- The `&[T]` type is a *slice*
 - Can also have mutable slices `&mut [T]`
 - Uses a “fat pointer” with `(ptr, len)`
- `Vec` type (a `struct`) for variable len arrays (on heap)
- We’ll discuss `structs` and `enums` later

Strings

```
fn main() {  
    let ferris: &str = "🦀";  
    println!("{ferris} says: Hello, world!")  
}
```

- Strings are UTF-8 encoded (variable length)
 - So no array-style indexing (`s[0]`)
 - Also no C-style null terminators
 - Represented as a type of slice
- Literals use quotes and backslash-escapes like C
- Write `b"foo"` for `u8` literal arrays
- There is also a heap-based `String` type (a struct)

Rust has *similar* arithmetic (and logic) operations to C.

- Integer and floating point arithmetic
- Bitwise operations (shift, and, or not)
- Logical operations and short-circuited `&&` and `||`

But consider:

```
fn main() {  
    let a: i8 = 100;  
    let answer = a*a;  
    println!("The answer is {answer}")  
}
```

Error turns into a runtime *panic* if we move the multiplication into a function.

```
fn square8(x: i8) -> i8 { x*x }

fn main() {
    let a: i8 = 100;
    let answer = square8(a);
    println!("The answer is {answer}")
}
```

- Different behavior for
 - `strict_mul`: Panic on overflow
 - `saturating_mul`: Largest integer on overflow
 - `wrapping_mul`: Two's complement wrap on overflow
- The behavior of `a * b` varies with compile mode
 - `strict_mul` in `debug` mode
 - `wrapping_mul` in `release` mode
- Similar variants for `add` and other operations
- But behavior is *defined* in every case

What is the output in release mode?

```
fn main() {  
    let a: i8 = 100;  
    let answer = square8(a);  
    println!("The answer is {answer}")  
}
```

Hint: $10000 = 8192 + 1024 + 512 + 256 + 16$

We saw something like this already:

```
fn square8(x: i8) -> i8 { x*x }
```

- Syntax is `fn name(args) -> return_type`
- Arguments are comma-separated `name: type` pairs
- Return via `return` or through value of block

What happens with this code? Let's try!

```
fn main() {  
    let answer = 41;  
    answer = 42; // Changed my mind!  
    println!("The answer is {answer}");  
}
```

```
fn main() {  
    let mut answer = 41;  
    answer = 42; // Changed my mind!  
    println!("The answer is {answer}");  
}
```

- If we want mutability, we have to ask for it!
- We still get a warning that 41 was never used

```
fn fib1(n: u64) -> u64 {  
    if n <= 1 { n } else { fib(n-1) + fib(n-2) }  
}
```

- `if` syntax like C, but without parens on condition
- Can produce a value (like the C ternary `cond ? a : b`)
- Condition must be a `bool` (no automatic conversion)

```
fn my_sqrt(x: f32) -> f32 {  
    let mut y = x/2.0;  
    loop {  
        let yprev = y;  
        y = (y+x/y)/2.0;  
        if (yprev/y-f32).abs() < 1e-7 {  
            break (y+x/y)/2.0;  
        }  
    }  
}
```

- **loop** introduces an otherwise-infinite loop
- **break** and **continue** similar to C
- **break** can produce a value for the loop
- NB: **abs** is a *method call* (more on this later)

```
fn gcd(mut a: u64, mut b: u64) -> u64 {  
    while b != 0 {  
        let temp = b;  
        b = a % b;  
        a = temp;  
    }  
    a  
}
```

- `while` syntax like C, but without parens on condition

```
fn fib2(n: u64) {  
    if n <= 1 {  
        n  
    } else {  
        let mut fm1 = 1;  
        let mut fm2 = 1;  
        for _ in 2..=n {  
            let fi = fm1+fm2;  
            fm2 = fm1;  
            fm1 = fi;  
        }  
        fm1  
    }  
}
```

- `for` iterates over an iterator
- Write `lo..hi` for range `lo <= i < hi`
- Write `lo..=hi` for range `lo <= i <= hi`

Many things about Rust “rhyme with” C/C++

- Part of the “curly braces” language family
- Many similar basic types (not all)
- Similar control flow syntax

We start seeing the big differences on Thursday

- **Structs** and (especially) **Enums**
- Memory safety features (ownership and borrowing)
- Error handling, the trait system

- Wrap up the paper
 - What are you most excited about for the course?
 - What are you most worried about?
- Read Ch 1-3 in <https://doc.rust-lang.org/book/>
- Skim Ch 4-6 when you have a moment
- Install the Rust toolchain (before lab on Friday)