Systems Programming
The Rust Programming Language

Instructors:

Amber Hu Justin Hsia

Teaching Assistants:

Ally Tribble	Blake Diaz	Connor Olson
Grace Zhou	Jackson Kent	Janani Raghavan
Jen Xu	Jessie Sun	Jonathan Nister
Mendel Carroll	Rose Maresh	Violet Monserate

Relevant Course Information

- We are finished with testable course content! 
  - Today's content and Friday's wrap-up lecture will not be tested
- HW 4 is due tomorrow @ 11:59 PM
- Course evals
  - Separate lecture and section evals
  - Lecture has only 17 responses out of 203 so far!
- Final Exam is Wednesday, 3/11 @ 12:30-2:20 PM
  - Kane Hall 110 and 120
  - See Ed post #630
  - Final review session on Friday @ 4:30-6:20 PM, CSE2 G10 + Zoom

Lecture Overview

- **Why is C/C++ difficult?**
- Rust programming language
- C/C++ vs. Rust
  - Type safety
  - Defaults
  - Memory safety
  - Race conditions
  - Tools

Why is Writing (Good) C/C++ Difficult?

- Tools
  - Compilation/linking is hard, Makefiles are tedious
  - Error messages are inscrutable, esp. linking and template errors
- Weird defaults
  - Functions in global namespace, mutable by default, `char` = 1 byte
- Type (un)safety
  - Reinterpreting bits, `void*`
- Memory errors
  - No bounds checking, invalid pointer dereferencing
- Data races
  - Hard to keep track of which memory is shared b/w threads
  - Not exclusive to C/C++

Why Write New C/C++?

- Despite all this, new C/C++ code remains common
  - Why?
- Backwards compatibility
  - Legacy C code is everywhere
  - C++ allows us to directly build on top of legacy systems
- Close to the hardware
  - Embedded systems (IoT, robotics, vehicles) often require granular control of hardware only exposed in C
  - High performance applications are often written in C with a high-level wrapper (*e.g.*, `numpy` in Python)

Enter: The Rust Programming Language

- Rust is a language designed to both:
  - Fill the **niche** that C/C++ currently occupy, and…
  - **Fix** what makes writing good C/C++ difficult
- Overview
  - No runtime or garbage collection
    - Compiles to a variety of assembly/OS targets
  - Type system provides **strong** compile-time guarantees
    - Safe Rust eliminates all data races, memory leaks, and invalid reads
    - "Fearless concurrency" —The Rust Book
  - Tooling designed for productivity
    - Built-in package manager and build/test/docs system
    - Human-readable error messages and linter/formatter

Rust in the Real World

- Created in 2006, sponsored by Mozilla in 2009
  - Didn't gain much adoption until version 1.0 in 2015
  - Named the "most admired programming language" from 2016—2025 in the StackOverflow developer survey

- Who is using Rust?
  - Linux Kernel
  - AWS Lambda (trillions of requests/mo)
  - Cloudflare web proxy (3 trillion requests/mo)
  - Firefox (200M active users/mo)
  - Discord, Dropbox, Meta, Google, Microsoft, and more

Rust: Hello World!

See hello.rs

```
fn main() {
    let name : String = std::env::args().nth(1)
        .expect("Did not provide name");
    println!("Hello, {name}!");
}
```

```
amberhu@rocky-wsl:examples$ rustc hello.rs
amberhu@rocky-wsl:examples$ ./hello Amber
Hello, Amber!
amberhu@rocky-wsl:examples$ ./hello

thread 'main' (86215) panicked at hello.rs:5:10:
Did not provide name
note: run with `RUST_BACKTRACE=1` environment variable to
display a backtrace
```

A panic is an error that crashes the program

Lecture Overview

- Why is C/C++ difficult?
- Rust programming language
- **C/C++ vs. Rust**
  - **Tools**
  - **Defaults**
  - **Memory safety**
  - **Type safety**
  - **Race conditions**

C/C++: Tools

- Importing libraries is difficult
  - No centralized package ecosystem (*e.g.*, `pip` for Python)
  - Must download the source files and compile it yourself, or…
    - Hope that there's a binary compatible with your platform
  - No standard tool to generate documentation
- No standard tools
  - Many compilers, testing frameworks, and documentation tools
  - Usually not portable! Need different tools for different platforms
- Error messages are hard to interpret
  - Linking errors
  - C++ template errors are notoriously obtuse

Rust: Tools

- One toolset, almost any platform
  - `rustup` (Installs `rustc` and `cargo` for your platform)
  - `rustc` ≈ `gcc`
  - `cargo` (Manages build pipeline and packages)
    - `cargo build` ≈ `make`
    - `cargo test` ≈ `./test_suite`
    - `cargo doc` ≈ `doxygen, javadoc`
    - `cargo add package` ≈ `pip install package`
    - `cargo run` ≈ `make bin && ./bin`
      - `cargo run --example foo` = Runs `examples/foo.rs`

- Readable error messages
  - Error message often suggests a possible fix!

Using Packages: `cargo`

Type inferred as `DateTime<Local>`
Easy to read and write, like Python, but statically typed like Java/C++

See `time.rs`

```
use chrono::Local;

fn main() {
    let time = Local::now();
    println!("It is now {}", time.format("%H:%M"));
}
```

```
amberhu@rocky-wsl:examples$ cargo run --example time
   Compiling cse333-rust-demo v0.1.0 (~/)
error[E0432]: unresolved import `chrono`
 --> examples/time.rs:1:5
  |
1 | use chrono::Local;
  |     ^^^^^^ use of unresolved module or unlinked crate `chrono`
  |
  = help: if you wanted to use a crate named `chrono`, use `cargo add
chrono` to add it to your `Cargo.toml`

For more information about this error, try `rustc --explain E0432`.
```

C/C++: Weird Defaults

- `char` type is 1 byte, always
  - Designed to support Latin character sets (*e.g.*, ASCII)
  - Websites in 2026: 99% Unicode (UTF-8), 1% ASCII-based
  - UTF-8 is variable length: each character is 1-4 bytes
    - The n^{th} character in a UTF-8 string is not necessarily the $(n-1)^{th}$ byte!

- All functions visible in global namespace
  - Marking "`static`" makes it invisible outside current file
  - Helper functions need a `_priv.h` to use across multiple files

- Mutable (non-const)
  - Easy to accidentally mutate a variable without noticing
  - `string greeting = hello.append(name);`

Rust: Immutable by Default

See `greet.rs`

```
fn main() {
    let greeting = "你好,";
    let name : String = std::env::args().nth(1)
        .expect("Did not provide name");
    println!("{greeting} {name}!");
    greeting = "再見,";
    println!("{greeting} {name}!");
}
```

```
error[E0384]: cannot assign twice to immutable variable `greeting`
  --> examples/greet.rs:10:5
   |
 4 |     let greeting = "你好,";  // hmmm
   |         -------- first assignment to `greeting`
...
10 |     greeting = "再見,";
   |     ^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable
   |
help: consider making this binding mutable
   |
 4 |     let mut greeting = "你好,";  // hmmm
   |         +++

For more information about this error, try `rustc --explain E0384`.
```

Rust: Better Defaults

- `char` type is Unicode (superset of ASCII)
  - Strings are encoded in UTF-8
- Variables are immutable by default
  - Must declare variables as `mut` to be mutable
- Namespaces are restricted by file
  - Functions are private by default, must declare as `pub` to be public
  - `use` declarations are similar to Python imports

Rust: References, Not Pointers

- In "safe" Rust, you don't directly manipulate pointers
  - Rust references are like C++ references, but you can rebind them
  - Written as `&Type` or `&mut Type`
  - Rust references have the following guarantees
    - They cannot be null
    - They refer to a living variable (*i.e.*, it hasn't been deallocated)

- However, there is also "unsafe" Rust
  - The compiler allows you to do things it would normally ban
  - You must explicitly use the `unsafe` keyword
    - Makes you really think twice!
  - Pointers do exist in Rust, but they are treated as unsafe

- See `borrow.rs`

C/C++: Type (Un)safety

- Easy to accidentally reinterpret bits
  - `char c = 'A'; cout << (c+2) << endl;` ⇒ 67
  - `int32_t x = -1; printf("%u\n", x);` ⇒ 4294967295
- Arithmetic overflow
  - `uint8_t x = 255; x++; printf("%d\n", x);` ⇒ 0
- No guarantee what a `void*` points to
  - Could be an `int32_t`, stack-allocated `vector`, `char[]` …
  - Might not even be a pointer!
    - *e.g.*, storing integer values in `LinkedList` in HW2/3
  - C++ templates tried to fix this, but system call code is written in C

Rust: Type Safety

- All pointer manipulation and bit manipulation is unsafe
  - Requires use of `unsafe` keyword
- Arithmetic overflow will cause a panic
  - Useful for testing, disabled in production for speed
- Safer, readable error handling
  - `Result<T, E>` is either `Ok` with type `T` or `Err` with type `E`
  - If the current fn also returns a `Result`, use the `?` operator
  - `Ok(val) => val, Err(msg) => return Err(msg)`
  - Unhandled errors will result in a compile-time warning
  - See `result.rs`

C/C++: Memory Errors

- Sometimes causes a crash
  - Dereferencing NULL
- Sometimes silently causes problems
  - Read from uninitialized memory
  - Writing one past the end of an array
- Sometimes causes security vulnerabilities!
  - Buffer overflow
  - Use after free
- Memory leaks
  - C++11 tried to fix this with smart pointers, but adoption is slow

Rust: Memory Safety

- Buffer overflows are impossible
  - Out of bounds indexing causes a panic at runtime
- Read from uninitialized memory is impossible
  - All data must be initialized before reading
- Memory leaks are almost impossible
  - Data gets destructed (Dropped) when it goes out of scope
  - Still have to watch out for cycles (think `weak_ptr`)
- See `memory.rs`

C/C++: Data Races

- Data races occur when multiple threads share memory that *could* be modified
  - 2 threads with const pointers?
    - OK
  - 2 thread with mutable pointers?
    - Compiles, with data race
  - 1 thread with a mutable pointer, 1 thread with a const pointer?
    - Compiles, with data race

- We had to remember to use mutex locks to synchronize
  - Easy to accidentally forget to lock or unlock!

Rust: No Data Races

- Safe Rust eliminates all data races
  - Borrow checker prevents two threads from sharing memory if it could be mutated
    - 1 mutable reference: OK
    - 1 mutable reference and 1+ other references: Does not compile!
  - Borrow checker rules apply in single threaded code as well

- Synchronization primitives still available (*e.g.*, `Mutex<T>`)
  - Cannot mutate the data without calling `.lock()` on Mutex
  - Lock automatically released when it goes out of scope!

- See `threads.rs`

Want to Learn More?

- Rust by Example
- Rust Book
  - Available offline with `rustup doc --book`
  - Brown University's version has interactive quizzes
  - Final project: build a multi-threaded web server (sound familiar?)
    - In some ways harder than HW4
      - Rust's corrow checker rules take time to get used to
    - In some ways easier: compiler **guarantees** memory and thread safety
- CSE 334: Modern Concurrency
  - New course, taught for the first time in 26wi as 493L
  - Intended to be the canonical "Rust course"