

Rust

Lukáš Hozda

NPRG082 · winter 2026/27

Course organization

- Alternating weeks lectures/labs
- Bring your notebook to the labs
- Visitors welcome & feel free to share the materials
- All materials will be published and lectures will be recorded

Contacts

- *Course Discord*
- *Rust CZ/SK Discord*
- You can email me also:
 - `lukas@lambda-symbolics.com`



`discord.gg/E4XpeWbJpx`

Course topics

- Ownership, borrowing, and data structures
- Enums, pattern matching, and error handling
- Modules, traits, generics, and iterators
- Threads, synchronization, and async
- Dependency management, library design, testing, documentation etc.

Rust history

- 2006: Graydon Hoare starts the project
- 2009: Mozilla begins sponsoring development
- 2015: Rust 1.0
- 2018: non-lexical lifetimes
- 2019: async/await
- 2021: Rust Foundation
- 2022: Rust in Linux kernel
- 2025: Rust 2024 edition (lol)

Influences

- OCaml/ML-family: basic syntax, algebraic data types and matching
- C++: resource management and zero-overhead abstractions
 - And the rest of syntax
- Haskell: traits (cf. Haskell type classes)
- Cyclone: memory safety
- Pre 2015 Rust also tried Erlang-style concurrency, classes, garbage collectors...

Design goals

- Memory safety without a CG
- Control over allocation, layout, and resource lifetime
- “Zero-cost abstractions” (-> compile to direct and efficient assembly)
- No data-races in safe code
- Interop with existing libs
- Explicit > implicit
- Clearly delimited unsafe

Resource management

- Lifetimes, borrowing, and ownership
 - Ownership: a value has exactly 1 owner
 - Borrowing: temporary access to a resource
 - Lifetimes: abstract information about how long can a value exist
- RAII (Resource Acquisition Is Initialization)
 - Resources (memory, handles...) are initialized when a value is created
 - When a value is discarded (=dropped), resources are cleaned up

Static analysis

Rust tries to provide most of its guarantees at compile time already:

- This generally means type safety, conversions
- Memory safety
- With the clippy linter and root attributes you can enforce more rules at compile time.

Installing Rust

- Install through *rustup.rs*
 - System packages for Rust should generally be avoided
- Linux, macOS, *BSDs
 - You also need a native linker/toolchain
- Windows:
 - You also need MSVC build tools (or use WSL)

Editor setup

- VS Code or VSCodium:
 - rust-analyzer plugin
 - TOML plugin
- Neovim or Emacs: rust-analyzer through an LSP client
 - I use lsp-mode in Emacs
- RustRover is another option

Standard tools

```
rustup component add rustfmt clippy rust-src  
cargo fmt  
cargo clippy  
cargo test  
cargo doc --open
```

Cargo project

```
cargo new mfdemo  
cd mfdemo  
cargo check  
# quick debug build  
cargo run  
# release build  
cargo run --release
```

- `Cargo.toml`: package metadata and dependencies
- `src/main.rs`: executable entry point

Crates and packages

crate == package == project - A crate may be comprised of a library and more th

Dependencies

```
cargo add regex
```

- Rust std is intentionally minimal and does not have things you might expect
 - e.g. no RNG, regex, DateTime, serialization, whatever nonsense is Python stdlib...
 - “The standard library is where code goes to die...”
- crates.io: package registry
- docs.rs: published API documentation
- lib.rs: an alternative crate directory

Hello-world (no cargo)

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

- `println!`: macro with compile-time fmt validation
- Save to e.g. `hello.rs`
- Compile with `rustc hello.rs`
- Run `./hello` on Unix or `.\hello.exe` on Windows

Basic values

```
fn main() {  
    let count: u32 = 42;  
    let ratio: f64 = 0.25;  
    let ready: bool = true;  
    let letter: char = 'λ';  
    println!("{count} {ratio} {ready} {letter}");  
}
```

- Rust is statically typed, but many local type annotations are/can be inferred
- Bindings are immutable unless declared `mut`

Primitive types

- Signed integers:
 - i8 through i128
- Unsigned integers:
 - u8 through u128
- isize, usize: ptr-sized integers
- f32, f64: floats
- bool
- char: Unicode scalar value, 4b (not 1 byte!)
- (): unit, a value with no stored data
 - more precisely a 0-tuple

Some more types

- `[T; N]`: fixed-size array
 - On the stack by default
- `Vec<T>`
 - pointer, length, capacity
- `String`
 - an **owned** UTF-8 buffer
- `&str`: a **borrowed** UTF-8 view into a string (does not have to `String!`)

Assignment of an owned value

```
fn main() {  
    let text = String::from("hello");  
    let other = text;  
    println!("{text}");  
    println!("{other}");  
}
```

Moving values

- `text` initially owns the string's allocation
- Assignment transfers ownership to `other`
- Old binding is invalid
- The string's contents are not implicitly cloned
- Two independent owners must not free the same allocation

Explicit copying

```
fn main() {  
    let text = String::from("hello");  
    let other = text.clone();  
    println!("{text} / {other}");  
}
```

- `clone` makes a copy of the allocation and contents
- Rust generally does not do implicit copies of heap allocations

Borrowed results

```
fn first_word(input: &str) -> &str {  
    input  
        .split_whitespace()  
        .next()  
        .unwrap_or("")  
}  
  
fn main() {  
    let line = String::from("systems programming");  
    println!("{}", first_word(&line));  
}
```

- The result may refer into the input's allocation
- Its validity is tied to the input borrow's lifetime

References and mutation

```
fn main() {  
    let mut values = vec![10, 20, 30];  
    let first = &values[0];  
    values.push(40);  
    println!("{first}");  
}
```

Vector growth

- push may allocate a larger buffer -> allocation may move
- A reference into the old buffer would be a dangling ref
- Uh oh
- => shared and exclusive access cant overlap

Reserved capacity

```
fn main() {  
    let mut values = Vec::with_capacity(64);  
    values.push(10);  
    let first = &values[0];  
    values.push(20);  
    println!("{first}");  
}
```

- This snippet is still rejected The compiler isn't psychic and there will always be valid programs rejected by the borrow checker

Unsafe code

```
#[allow(deref_nullptr)]  
unsafe {  
    // i'm crashing this plane, with no survivors  
    *(0 as *const u8)  
}
```

- `unsafe` blocks don't disable any compiler checks
- But let you do things that are potentially unsafe
- You need them for low-level/native things, when the borrowck is not smart enough, and when interfacing with foreign libraries
- Safe APIs can encapsulate an unsafe implementation

Resource cleanup

- Ownership applies to more than memory
- `File`: close an owned file handle
- `MutexGuard`: release a lock when the guard is dropped
- Cleanup normally follows scope exit, including early returns
- Leaks are possible; memory safety does not require reclamation

Absence as a value

```
fn main() {  
    let values = [10, 20];  
    match values.get(5) {  
        Some(value) => println!("{value}"),  
        None => println!("no element"),  
    }  
}
```

- `Option<T>` distinguishes a value from its absence
- No invalid reference represents the missing element

Errors as values

```
fn main() {  
    let result = "70000".parse::<u16>();  
    match result {  
        Ok(port) => println!("port {port}"),  
        Err(error) => println!("{error}"),  
    }  
}
```

- `Result<T, E>` includes the error type in the contract
- Callers can handle errors or propagate them with ?

Exhaustive matching

```
enum State { Stopped, Running }  
fn main() {  
    let state = State::Running;  
    match state {  
        State::Running => println!("running"),  
    }  
}
```

Next time

- Install Rust and configure your editor
- Build and run a Cargo project
- Bring a laptop