



ACCU
2022

A FRIENDLY INTRODUCTION TO RUST FOR C++ DEVELOPERS

HENDRIK NIEMEYER



A Friendly Introduction to Rust For C++ Developers

Hendrik Niemeyer (he/him)





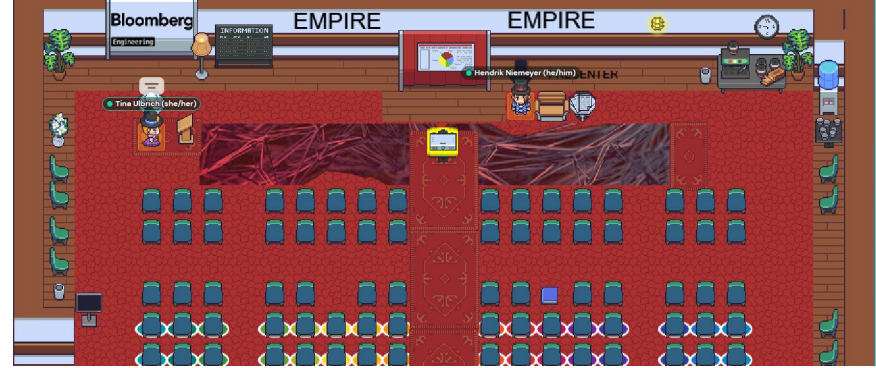
Link to Slides:

<https://tinyurl.com/y6exty57>



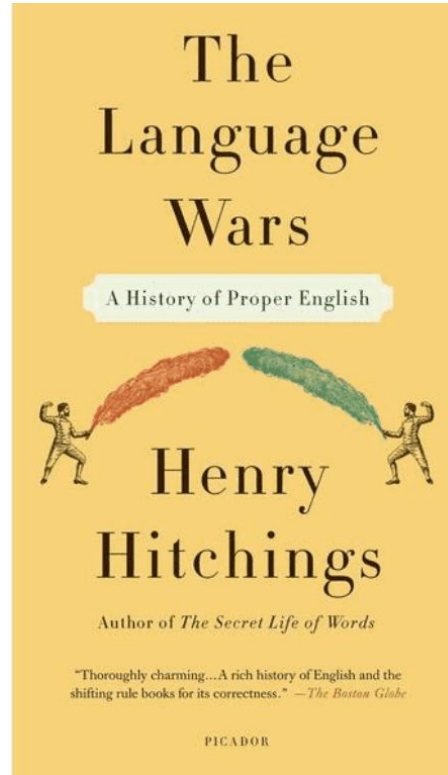
Feedback and Questions

- Twitter: [@hniemeye](https://twitter.com/hniemeye)
- LinkedIn: [hniemeyer87](https://www.linkedin.com/in/hniemeyer87)
- GitHub: [hniemeyer](https://github.com/hniemeyer)



Ask questions in Q&A section of zoom now or later in the gathertown room.

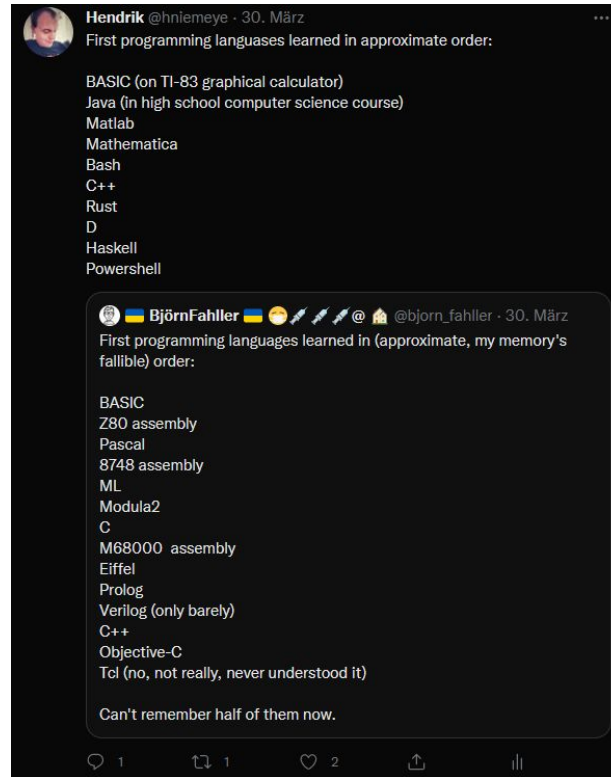
Disclaimer



If You Want to Try Out Code

- [Compiler Explorer](#) supports Rust
- [Rust Playground](#)
- [Install Rust](#)

Motivation



Motivation: Tooling

- Setting up a new project should be easy
- (de facto) standard build system, package manager, testing framework and static analyzer makes life easier
- C++ has many, very good tools for these tasks
- Rust has cargo and crates.io

Motivation: Safety

```
int main()
{
    std::vector<int> v({-7, 1, 2, 3});
    int& x = v[0];
    v.push_back(12);
    std::cout << "The number is: " << x;
}
```

The number is: 0

<https://gcc.godbolt.org/z/8Mao4T>

Motivation: Safety

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
--> <source>:4:5
  |
3 |   let x = &v[0];
  |           - immutable borrow occurs here
4 |   v.push(12);
  |   ^^^^^^^^^ mutable borrow occurs here
5 |   println!("The number is: {}", x);
  |                                   - immutable borrow later used here
```

Motivation



RustConf 2020 - Opening Keynote

What Is Rust?

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: **with no runtime or garbage collector**, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust's rich type system and ownership model guarantee **memory-safety** and thread-safety — enabling you to **eliminate many classes of bugs at compile-time**.

Productivity

Rust has great documentation, a friendly compiler with **useful error messages**, and top-notch tooling — an integrated **package manager and build tool**, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Rust at ACCU 2022

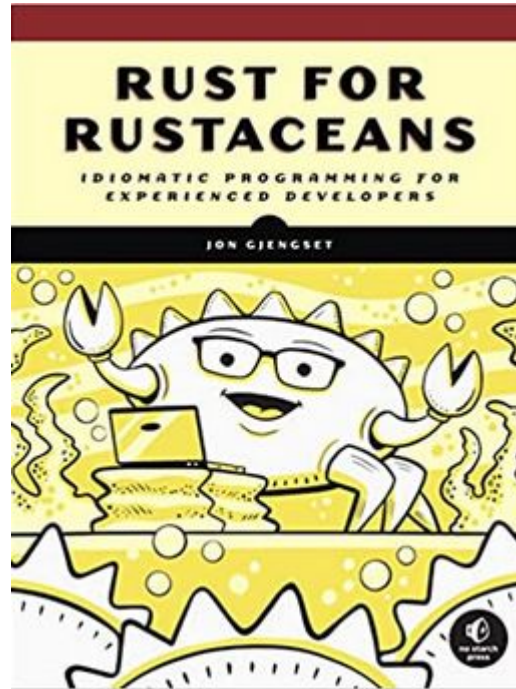
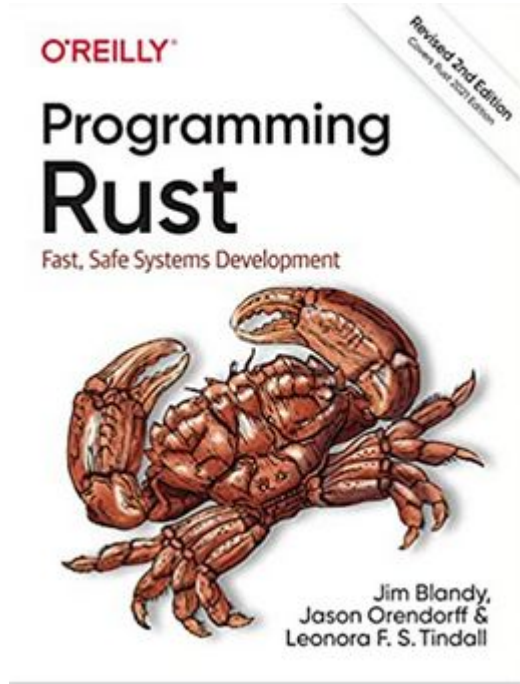
Coroutines: C++ vs Rust: 11:00 - 12:30 Friday 8th April 2022 BST

Neutron – writing a single-tasking ‘DOS’ for Arm microcontrollers, in Rust:
14:00 - 15:30 Friday 8th April 2022 BST

Free Resources to Learn Rust

- [Rust Book](#)
- [Rust Cookbook](#)
- [Rust by Example](#)
- [ferrous systems - teaching material](#)
- <https://jrvidal.github.io/explaine.rs/>

Paid Resources to Learn Rust



Syntax and Mutability

```
fn add(x: i32, y: i32) -> i32
{ x+y }
```

```
fn main() {
    let a = 5;
    let b = 10;
    let mut res = add(a,b);
    res += 5;
    println!("The result is
{} ", res);
}
```

```
int add(int x, int y)
{ return x + y; }
```

```
int main() {
    const auto a = 5;
    const auto b = 10;
    auto res = add(a, b);
    res += 5;
    std::cout << "The result is
" << res;
}
```


const in Rust

```
const fn fib(n: i32) -> i32 {  
    if n < 2 { 1 }  
    else { fib(n-1) + fib(n-2) }  
}
```

```
fn main() {  
    let x = fib(5);  
    const S: i32 = fib(8);  
    println!("{}", x);  
    println!("{}", S);  
}
```

```
constexpr int fib(int n) {  
    if (n < 2)  
        return 1;  
    else  
        return fib(n - 1) + fib(n - 2);  
}
```

```
int main() {  
    const auto x = fib(5);  
    constexpr auto S = fib(8);  
    std::cout << x << '\n';  
    std::cout << S << '\n';  
}
```

Variables and References

- Each value in Rust has a variable that's called its owner
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped
- At any given time, you can have either one mutable reference or any number of immutable references but not both
- References must always be valid

References

```
let mut s = String::from("hello");
```

```
let r1 = &s; // no problem
```

```
let r2 = &s; // no problem
```

```
let r3 = &mut s; // BIG PROBLEM
```

```
println!("{}", {}, and {}", r1, r2, r3);
```

References

```
let mut s = String::from("hello");
```

```
let r1 = &s; // no problem
```

```
let r2 = &s; // no problem
```

```
println!("{}", r1, r2);
```

```
// r1 and r2 are no longer used after this point
```

```
let r3 = &mut s; // no problem
```

```
println!("{}", r3);
```

Dangling References

```
fn main() {  
    let reference_to_nothing = dangle();  
}  
  
fn dangle() -> &String {  
    let s = String::from("hello");  
  
    &s  
}
```

this function's return type contains a borrowed value, but there is no value for it to be borrowed from.

Lifetimes

- Lifetimes describe the duration a value remains in memory (compared to other values)
- They are descriptive types which the compiler usually can infer itself
- used by compiler to reason about memory safety of code

[Rust Playground](#)

<https://gcc.godbolt.org/z/vdTMzzWaP>



Are there any questions?



Copy Semantics

```
fn main() {  
    let a = 5;  
    let b = a;  
    println!("The result is {}", a);  
}
```


Move Semantics

```
struct WrappedNumber {  
    value: i32  
}
```

```
fn main() {  
    let a = WrappedNumber {value: 5};  
    let b = a;  
    println!("The result is {}", a.value);  
}
```

Move Semantics

```
error[E0382]: borrow of moved value: `a`  
--> src/main.rs:8:34
```

```
6 |     let a = WrappedNumber {value: 5};  
    - move occurs because `a` has type `WrappedNumber`, which does not implement the `Copy` trait  
7 |     let b = a;  
    - value moved here  
8 |     println!("The result is {}", a.value);  
                                   ^^^^^^^^ value borrowed here after move
```

Expressions

```
fn main() {  
    let x = 12;  
    let number = if x < 11 { 5 } else { 6 };  
    let mut counter = 0;  
    let result = loop {  
        counter += number;  
        if counter > 3*x {  
            break counter * x;  
        }  
    };  
    println!("The value of number is: {}", number);  
    println!("The value of result is {}", result);  
}
```



Are there any questions?



Structs

```
struct Square {  
    length: f32  
}
```

```
fn main() {  
    let a = Square {length: 5.2};  
    println!("My square is {} long!", a.length);  
}
```

Methods

```
impl Square {  
  
    fn area(&self) -> f32 {  
        self.length * self.length  
    }  
  
    fn grow(&mut self, factor: f32) {  
        self.length *= factor;  
    }  
}
```

Methods

```
impl Square {  
    fn new(l: i32) -> Self {  
        Self {length: l}  
    }  
}  
  
fn main() {  
    let my_square = Square::new(3);  
    println!("{}", my_square.length)  
}
```

Vector

```
fn main() {  
    let mut my_vec = Vec::new();  
    my_vec.push(8);  
  
    let mut my_other_vec = vec![1,2,3];  
  
    for i in &mut my_other_vec {  
        *i += 2;  
    }  
  
    let sum: i32 = my_other_vec.iter().sum();  
    println!("{}", sum);  
}
```


Slices

- Slices in Rust are similar to `std::span` and `std::string_view` in C++
- (possibly mutable) view to data owned somewhere else

<https://gcc.godbolt.org/z/P7njMxfda>

Traits

```
trait Shape {  
    fn area(&self) -> f32;  
}  
  
impl Shape for Square {  
    fn area(&self) -> f32 {  
        self.length * self.length  
    }  
}  
  
fn print_area(shape: &impl Shape) {  
    println!("The area is {}", shape.area());  
}
```

Generics and Trait Bounds

```
fn print_area<T: Shape>(shape: &T) {  
    println!("The area is {}", shape.area());  
}
```

```
fn print_area<T: Shape + Density>(shape: &T) {  
}
```

```
fn some_function<T, U>(t: &T, u: &U) -> i32  
    where T: Display + Clone,  
           U: Clone + Debug  
{ }
```

Derivable Traits

```
#[derive(Clone, Copy)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let a = Rectangle {width: 5, height: 3};
    let b = a;
    println!("{}", a.width);
    println!("{}", b.width);
}
```

Trait Objects and Smart Pointers

- a trait object can represent any struct which implements the mentioned trait
- can be used to create vectors of different types which share a trait
- trait objects perform dynamic dispatch (similar to virtual function calls in C++)
- Box is smart pointer whose contents are stored on the heap

<https://gcc.gnu.org/z/EoT3ofzd3>

Traits and Functions

<https://gcc.godbolt.org/z/naPzb3j3T>

Tuple-like Structs

```
struct Color(i32, i32, i32);  
struct Point(i32, i32, i32);  
  
fn main() {  
    let black = Color(0, 0, 0);  
    let origin = Point(0, 0, 0);  
    let my_tuple = (1, 2, 3);  
    let (x, _, _) = my_tuple;  
    let Color (r,g,b) = black;  
    println!("{}", r, g, b);  
    println!("{}", origin.1, x);  
}
```



Are there any questions?



Enums

```
enum Action {  
    Stay,  
    Move {x: i32, y: i32},  
    Fight(i32),  
    Say(String),  
}  
  
fn main() {  
    let my_action = Action::Move {x: 12, y: 15};  
}
```

Pattern Matching

```
fn main() {  
    let my_action = Action::Move {x: 12, y: 15};  
    let result = match my_action {  
        Action::Stay => 5,  
        Action::Move {x, y} => {println!("Moving");  
                                x+y},  
        _ => 20  
    };  
    println!("{}", result);  
}
```

if let

```
fn main() {  
    let my_action = Action::Move {x: 12, y: 15};  
    let result = if let Action::Move {x,y} = my_action { x+y }  
    else {  
        5  
    };  
    println!("{}", result);  
}
```

Option

```
enum Option<T> {  
    Some(T),  
    None,  
} //defined by standard lib
```

```
let some_number = Some(5);  
let absent_number: Option<i32> = None;
```

Error Handling

```
use std::fs::File;

enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Error Handling

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
    let g = File::open("world.txt").expect("Failed to open
world.txt");
}
```

Error Handling: The ? operator

```
fn read_file() -> Result<File, io::Error> {  
    let f = File::open("hello.txt");  
  
    let f = match f {  
        Ok(file) => file,  
        Err(e) => return Err(e),  
    };  
    Ok(f)  
}  
  
fn main() {  
    let my_file = read_file().unwrap();  
}
```

Error Handling: The ? operator

```
use std::fs::File;
use std::io;

fn read_file() -> Result<File, io::Error> {
    let f = File::open("hello.txt");
    Ok(f)
}

fn main() {
    let my_file = read_file().unwrap();
}
```




Are there any questions?



Iterators

```
struct EvenNumbers {  
    value: u32,  
}  
  
impl EvenNumbers {  
    fn new() -> Self {  
        Self { value: 2 }  
    }  
}  
  
impl Iterator for EvenNumbers {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        let old_value = Some(self.value);  
        self.value += 2;  
        old_value  
    }  
}
```

Iterators

```
fn main() {  
    let my_series = EvenNumbers::new();  
    let result: u32 = my_series.skip(5).map(|x| x*x).take(10).sum();  
    println!("{}", result);  
}
```

Tooling: Cargo

https://github.com/hniemeyer/rust_tooling_demo



Are there any questions?



Interop with C++

- Rust ist cool but shall we rewrite our 1.5 million LOC C++ codebase in Rust?
- of course not
- Maybe new projects in Rust.
- what options do we have for interop are there?
- [rust-bindgen](#) (unsafe bindings, ffi)
- [cxx crate](#) (safe bindings, ffi)
- [grpc](#) (rpc) or [Thrift](#) (rpc)

Interesting Topics for Further Exploration

- multithreading
- async
- WASM



Are there any questions?



My Other Talk

What Has C++20 Ever Done For Templates?: 11:00 - 12:30 Friday 8th April 2022
BST

