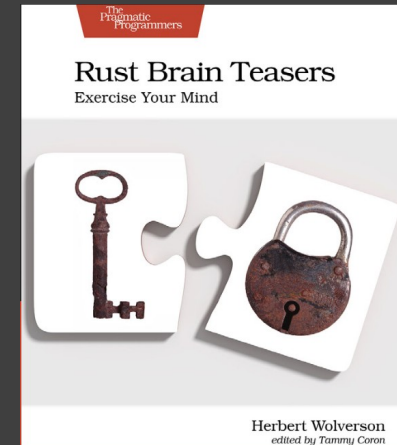


Rust Debugging & Optimization

Presented by Herbert Wolverson

Who am I?

- Rust Trainer at Ardan Labs
- Author of Hands-on Rust, Rust Brain Teasers
- Maintainer of bracket-lib
- Contributor to LibreQoS
- IT Consultant and Trainer



What's in this Class?

- Debugging Rust
 - Formatting Data
 - Logging to the Console
 - The “log” and “envlog” crates
 - Logging to Syslog and Beyond
 - Debugging with Visual Studio Code
 - Avoiding Making Bugs
- Optimizing Rust
 - Cargo Optimization Profiles
 - Link Time Optimization
 - Building Benchmarks
 - Optimizing for Size

This is a shortened (1 hour) version of the 5-day class.

PART 1: DEBUG LOGS

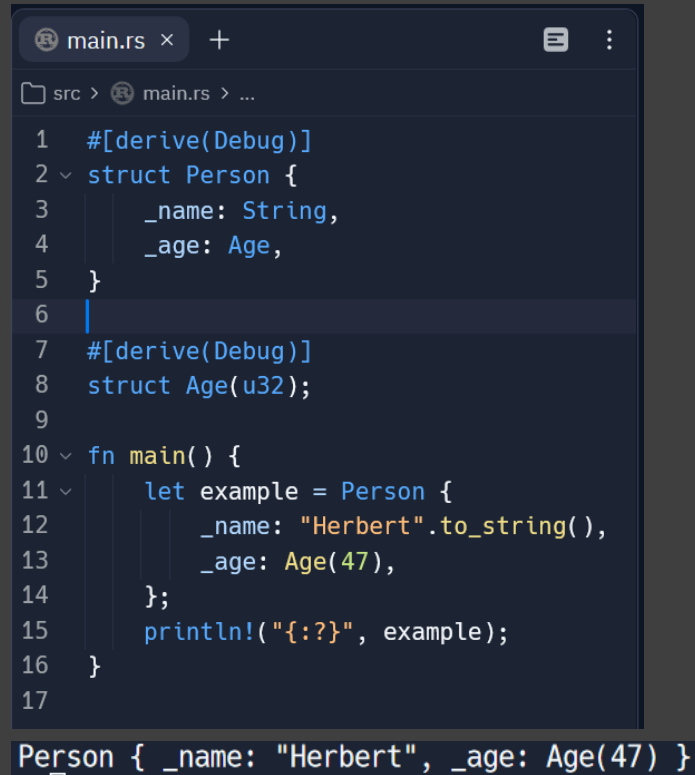
Practical techniques for debugging Rust with the console and log files.

Why debug to the console?

- Not every platform *has* a debugger.
- You can't always attach a debugger and pause the world.
- The console (in some form) is always available.
- Everything you learn about formatting to the console *also* works for logging.

Formatting Structures

- The easy way:
`#[derive(debug)]`
- `println!("{}", my_structure);`
- Downsides:
 - Everything in the structure must also support Debug
 - Limited control over the appearance of the output.



```
main.rs x +
src > main.rs > ...
1  #[derive(Debug)]
2  struct Person {
3      _name: String,
4      _age: Age,
5  }
6
7  #[derive(Debug)]
8  struct Age(u32);
9
10 fn main() {
11     let example = Person {
12         _name: "Herbert".to_string(),
13         _age: Age(47),
14     };
15     println!("{}", example);
16 }
17
Person { _name: "Herbert", _age: Age(47) }
```

Try it online: <https://replit.com/@HerbertWolverso/PrintDebug#src/main.rs>

Pretty Printing with Debug

- **You still**
`#[derive(Debug)]`
- `println!("{}", my_structure);`
- **Downsides:**
 - The output can be HUGE.
 - You still have limited control over what prints.

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

```
Person {  
  _name: "Herbert",  
  _age: Age(  
    47,  
  ),  
}
```

Try it online: <https://replit.com/@HerbertWolverso/PrettyPrintDebug#src/main.rs>

Implementing Display

- Implementing Display for a structure gives you control.
- You can now: `println!("{{my_struct}}");`

```
use std::fmt;

struct Person {
    name: String,
    age: Age,
}

struct Age(u32);

impl fmt::Display for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{{}} ({{}})", self.name, self.age.0)
    }
}

fn main() {
    let example = Person {
        name: "Herbert".to_string(),
        age: Age(47),
    };
    println!("{}", example);
}
```

Herbert (47)

Try it online: <https://replit.com/@HerbertWolverso/DisplayDebug#src/main.rs>

Nested Display – Total Control

```
use std::fmt;

struct Person {
    name: String,
    age: Age,
}

struct Age(u32);

impl fmt::Display for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{} ({})", self.name, self.age)
    }
}

impl fmt::Display for Age {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        if self.0 > 45 {
            write!(f, "Too Old!")
        } else {
            write!(f, "Young and spry")
        }
    }
}

fn main() {
    let example = Person {
        name: "Herbert".to_string(),
        age: Age(47),
    };
    println!("{}", example);
}
```

Herbert (Too Old!)

Try it online: <https://replit.com/@HerbertWolverso/NestedDisplay#src/main.rs>

The “log” and “env_logger” crates

- Add dependencies to Cargo.toml

```
[dependencies]
log = "0"
env_logger = "0"
```

- Replace println! with log::warn!

```
use log::warn;

fn main() {
    env_logger::init();
    warn!("Hello, world!");
}
```

- Run with an environment variable:

```
> RUST_LOG=info cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.14s
    Running `target/debug/my-project`
[2023-01-19T19:30:44Z WARN my_project] Hello, world!
> □
```

Try it online: <https://replit.com/@HerbertWolverso/LogCrate#src/main.rs>

Formatting still works with log

```
use std::fmt;

struct Person {
    name: String,
    age: Age,
}

struct Age(u32);

impl fmt::Display for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{} ({})", self.name, self.age)
    }
}

impl fmt::Display for Age {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        if self.0 > 45 {
            write!(f, "Too Old!")
        } else {
            write!(f, "Young and spry")
        }
    }
}

fn main() {
    env_logger::init();
    let example = Person {
        name: "Herbert".to_string(),
        age: Age(47),
    };
    log::warn!("{example}");
}
```

```
❖ RUST_LOG=info cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.20s
  Running `target/debug/my-project`
[2023-01-19T20:32:27Z WARN my_project] Herbert (Too Old!)
❖ □
```

Try it online: <https://replit.com/@HerbertWolverso/LogDisplay#src/main.rs>

Sending logs elsewhere

- Replace `log` with `log4rs`
- Completely configurable logging

- Start `main()` with:

```
log4rs::init_file("log4rs.yml", Default::default()).unwrap();
```

- Configure with YAML:

```
appenders:  
  syslog:  
    kind: libc-syslog  
    openlog:  
      ident: log4rs-syslog-example  
      option: LOG_PID | LOG_NDELAY | LOG_CONS  
      facility: Daemon  
    encoder:  
      pattern: "{M} - {m}"  
root:  
  level: trace  
  appenders:  
    - syslog
```

Implementing display functions

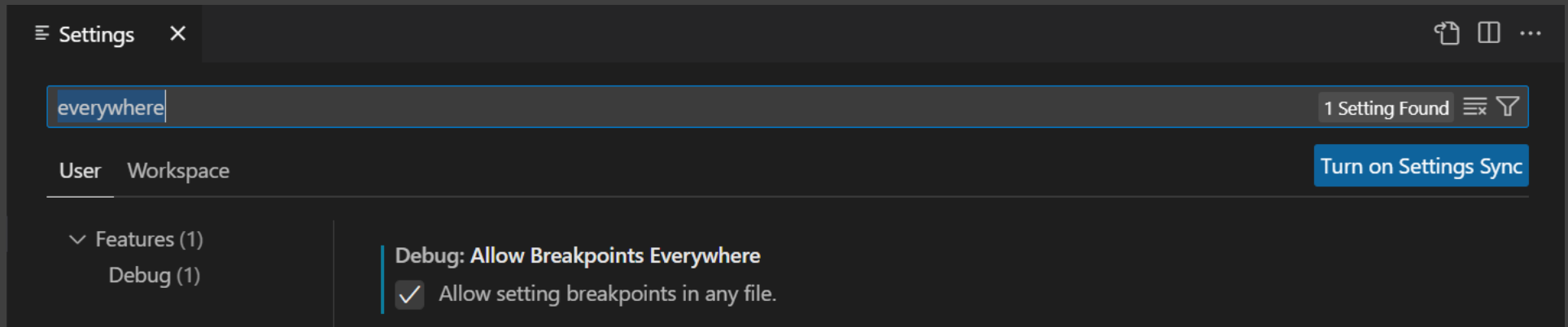
- Sometimes, you want to adjust the display for a specific event.
- Create a function that returns a String – don't print directly.
- You can log strings however you want – capturing stdout is trickier.

PART 2: DEBUGGING

Debugging in Visual Studio Code

Setup Visual Studio Code

- **Install Rust Analyzer**
 - Not needed for debugging, but you want it!
- **Install either:**
 - CodeLLDB (preferred)
 - Microsoft C++
- **Open Settings (ctrl + ,)**
 - Search for “everywhere”
 - Ensure “allow breakpoints everywhere” is checked.



Let's Debug a Program

```
fn main() {
    println!("Hello, what's your name?");
    let mut buffer: String = String::new();
    let stdin: Stdin = std::io::stdin();
    stdin.read_line(buf: &mut buffer).expect(
        msg: "Unable to read standard input"
    );

    if buffer == "Herbert" {
        println!("Hello Herbert");
    } else {
        println!("You aren't an authorized user.")
    }
}
```

```
Hello, what's your name?
Herbert
You aren't an authorized user.
```

- Can you spot the bug?
- Even though we typed “Herbert”, the program rejects it.
- Let's look in a debugger...

Set a Breakpoint

- Mouse over to the left of the line on which to break.
- Click, and a red circle appears.

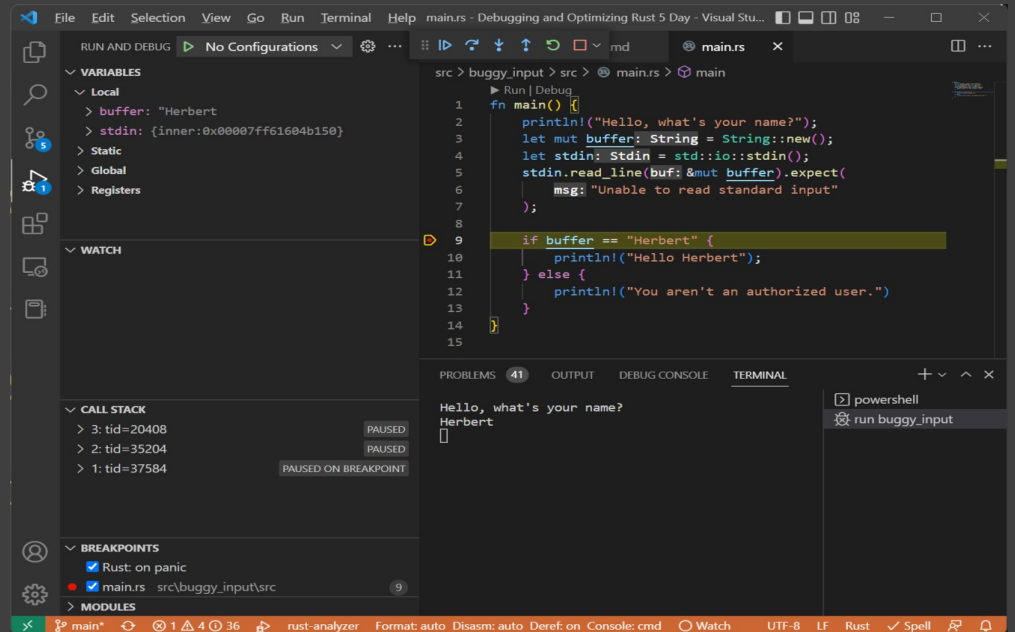
```
src > buggy_input > src > main.rs > main
  ▶ Run | Debug
1  ∨ fn main() {
2      println!("Hello, what's your name?");
3      let mut buffer: String = String::new();
4      let stdin: Stdin = std::io::stdin();
5  ∨  stdin.read_line(buf: &mut buffer).expect(
6      |   msg: "Unable to read standard input"
7      );
8
9  Breakpoint if buffer == "Herbert" {
10     |   println!("Hello Herbert");
11  ∨   } else {
12     |   println!("You aren't an authorized user.");
13     |   }
14     |   }
15 }
```

Start the Debugger

- Open the Command Palette
 - Ctrl+Shift+P
 - **OR** View → Command Palette
- Choose “Rust-Analyzer: Debug”
- Select the project to debug
- Where’s the bug?

```
> |  
rust-analyzer: Debug
```

```
|  
run buggy_input  
restart
```



There's the bug

```
src > buggy_input.rs
  ▶ Run | De
1  fn main() {
2      println!();
3      let mut buffer = String::new();
4      let stdin = stdin.lock();
5      std::io::Read::read_line(&mut buffer, &mut stdin, &mut io::stdout()).unwrap();
6      // println!("{}", buffer);
7  }
8
9  if buffer == "Herbert" {
10     println!("Hello Herbert");
11 } else {
12     println!("You aren't an authorized user.");
13 }
14 }
15
```

"Herbert"

- [0]: 'H'
- [1]: 'e'
- [2]: 'r'
- [3]: 'b'
- [4]: 'e'
- [5]: 'r'
- [6]: 't'
- [7]: '\r'
- [8]: '\n'

> [raw]: alloc::string::String

Hold Alt key to switch to editor language hover

- Hovering over “buffer” shows that the input string contains extra characters: line-feed and line-break.
- You can fix the problem by adding `.trim()` to the string.

When (not) to use a Debugger

- When you don't have single-user access to a development environment.
- In a distributed or micro-services environment, it's not always clear which program to debug!
- Don't breakpoint on a live system. Nobody will thank you for pausing the world.

PART 3: DON'T WRITE BUGS!

If only it were that simple?

Rust can help you not make bugs to begin with

Use Error Handling

- Use Results
 - Any function can wrap a result in a `Result<>` type.
 - Don't ignore the result – check it.
 - Combine with defensive programming
- Anyhow to make it easier

```
1 use anyhow::{Error, Result};
2
3 fn do_some_math(n: i32) -> Result<i32> {
4     if n == 0 {
5         Err(Error::msg("n must be greater than 0"))
6     } else {
7         120 / n
8     }
9 }
10
11 fn main() {
12     match do_some_math(12) {
13         Ok(answer) => println!("The answer is {answer}"),
14         Err(e) => println!("{:?}", e),
15     }
16 }
```

Try it online: <https://replit.com/@HerbertWolverso/ErrorHandling#src/main.rs>

Require Error Acknowledgment

- Decorate functions that return a result with `#[must_use]`
- Not checking the result is now a compiler warning.

```
use anyhow::Result;

#[must_use]
fn do_something() -> Result<> {
    // Do something important that may fail
    Ok(())
}

fn main() {
    do_something();
}
```

```
warning: unused `Result` that must be used
--> src/main.rs:10:5
10 |     do_something();
    |
    = note: `#[warn(unused_must_use)]` on by default
    = note: this `Result` may be an `Err` variant, which should be handled
```

Try it online: <https://replit.com/@HerbertWolverso/ErrorMustUse#src/main.rs>

Avoid Bugs with Unit Tests

- Unit testing is built into Rust & Cargo
- Run your tests with cargo test

```
pub fn square(n: u32) -> u32 {
    n * n
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn test_right_answer() {
        assert_eq!(square(2), 4)
    }
}
```

```
❯ cargo test
   Compiling my-project v0.1.0 (/home/runner/UnitTestExample)
   Finished test [unoptimized + debuginfo] target(s) in 1.06s
   Running unittests src/lib.rs (target/debug/deps/my_project-b3e56510ae34b9f5)

running 1 test
test test::test_right_answer ... ok
```

Try it online: <https://replit.com/@HerbertWolverso/UnitTestExample#src/lib.rs>

PART 4: OPTIMIZATION

Cargo Optimization Profiles

Quick tool-driven optimization

Debug Mode

- Minimal optimizations
- Full debug information
- Numeric overflow is checked
- Can be slow
- It's the default –
`cargo build` and
`cargo run` use
debug mode by
default.

Tip: Optimized Debug Mode

- Still has debug information
- Disables overflow checks
- Allows some compiler optimizations

- Add to Cargo.toml:

```
[profile.dev]
opt-level = 1           # Use slightly better optimizations.
overflow-checks = false # Disable integer overflow checks.
```

- Perfect for when debug isn't fast enough, but you still need a debugger

Release Builds

- Removes debug information
- Removes assertion and overflow checks
- Runs full compiler optimizations
- Execute with
 - `cargo run --release`
 - `cargo build --release`

Link Time Optimization

- LTO permits cross-crate inlining.
 - **false**: none is performed
 - **thin**: Some is performed – relatively fast compile time.
 - **fat**: optimize all calls. Compilation can be very slow.

Cargo.toml:

```
[profile.release]  
lto = (false/thin/fat)
```

Benchmarking

Measure twice, cut once.

Only spend time optimizing things that are actually
slow...

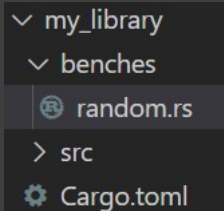
... and prove that your optimization made a
difference!

Criterion Boilerplate

- In Cargo.toml:
 - Add “criterion” has a dev dependency.
 - Add benchmark to Cargo.toml
- Add “benches” folder.
- Add empty “random.rs” file.

```
[dev-dependencies]
criterion = { version = "0.3", features = [ "html_reports" ] }
```

```
[[bench]]
name = "random"
harness = false
```



A file explorer view showing the project structure. The root directory is 'my_library', which contains a 'benches' folder. Inside 'benches', there is a file named 'random.rs'. Below the 'benches' folder, there is a 'src' folder and a 'Cargo.toml' file.

Benchmarking Random Numbers

```
random.rs x +
benches > random.rs > ...
1 use criterion::{black_box, criterion_group, criterion_main, Criterion};
2 use my_library::RandomNumberGenerator;
3
4 pub fn criterion_benchmark(c: &mut Criterion) {
5     c.bench_function("random", |b| {
6         let mut rng = RandomNumberGenerator::new();
7         b.iter(|| {
8             let n: u64 = rng.next();
9             black_box(n);
10        })
11    });
12 }
13
14 criterion_group!(benches, criterion_benchmark);
15 criterion_main!(benches);
16
```

cargo bench

random time: [13.644 ns **14.669 ns** 15.669 ns]

Try it online: <https://replit.com/@HerbertWolverso/Benchmark#benches/random.rs>

Faster Random Number Algorithm

- Add `rand_xoshiro` to `Cargo.toml`
- Replace “Rng” with “Xoshiro256Plus”
- Rerun benchmark
- Random number generation is down to nanoseconds.
- Why not always use Xoshiro?

```
random          time: [2.6827 ns 2.9343 ns 3.2147 ns]
```

Try it online: <https://replit.com/@HerbertWolverso/BenchmarkFast#src/lib.rs>

Optimizing for Size

Rust in Embedded Development

Optimizing for Size on Embedded Platforms

- The never ending quest for a tiny “hello world”
- Take the standard “hello world” program
- Building in Debug:
 - 155136 bytes executable
 - 1380352 bytes debug info!
- Building in Release:
 - 151552 bytes executable
- With opt-level “z”
 - 151552 bytes executable

151,552 bytes is huge for embedded!

No Standard Library

- Here's "hello world" without the standard library.
- It compiles to 14k – better.

```
#![no_std]
#![no_main]

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}

#[link(name = "c")]
extern "C" {
    fn write(fd: i32, buf: *const i8, count: usize) -> isize;
}

#[no_mangle]
pub extern "C" fn main() -> isize {
    unsafe { write(1, b"Hello, World!\n" as *const u8 as *const i8, 14) };
    0
}
```

And finally...

- <https://github.com/kmcallister/tiny-rust-demo>
- “Hello World” in 151 bytes.
- This illustrates the final point: optimize as much as you need to. You *can* jump through hoops to make tiny and/or really fast code: but you’ll spend a lot of developer time doing it.
- Optimize where it’s needed.

Wrap-Up

- Any Questions?
- @herberticus on Twitter
-