Ultimate Rust 1: The Rust Ecosystem

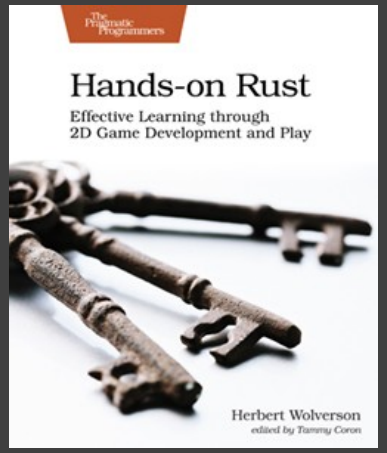
Presented by Herbert Wolverson

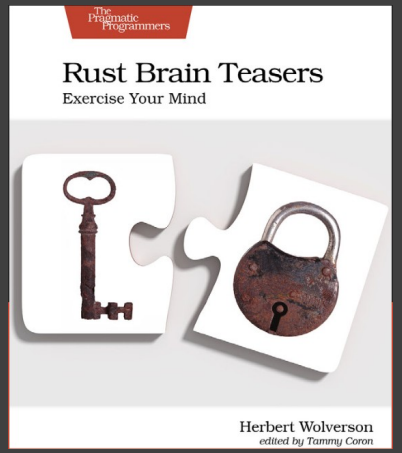
Source code:

https://github.com/thebracket/UltimateRust1-EcoSystem

Who am I?

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





What’s in this Module?

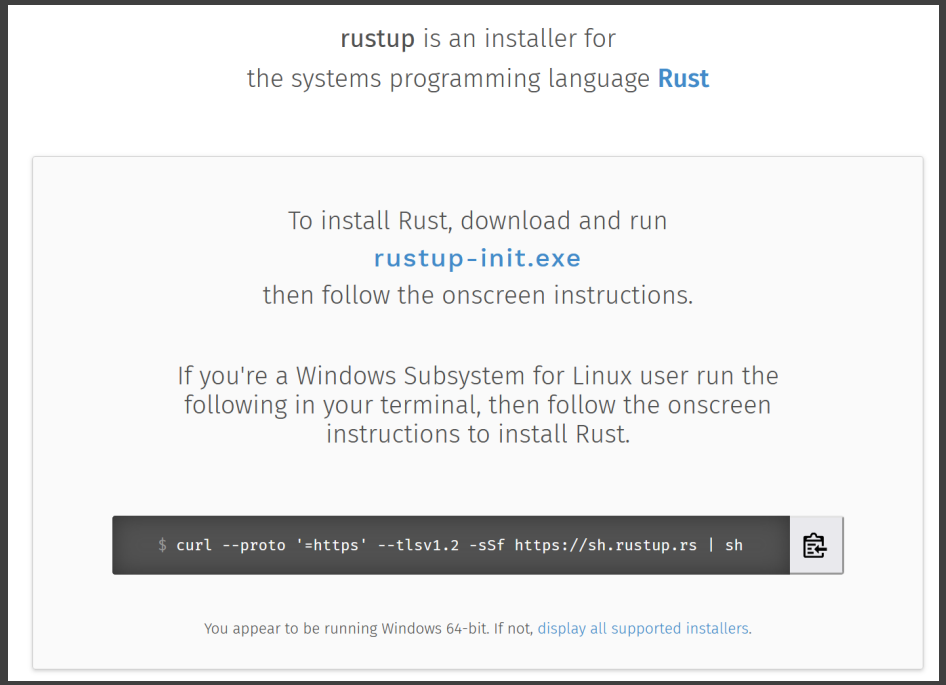
- *This is the shortened version for demonstration purposes.*
- Installing Rust & Working with Rust
- The Rust Toolchain
- Introducing Cargo – “Hello World”
- Modules & Namespaces
- Dependencies
- Workspaces
- Your First Library
- Unit Testing
- Rust’s Safety Guarantees

Two Popular Ways to Install Rust

- RustUp for single-user deployment
  - `https://rustup.rs/`
  - Follow the instructions on the web page.
- OS Package Managers
  - e.g. `apt install rust-all`

Let’s install Rust with RustUp.rs

RustUp.rs in Action



- Go to rustup.rs
- Either:
  - Download the Windows installer
  - Copy the terminal/shell install command.

Install Rust

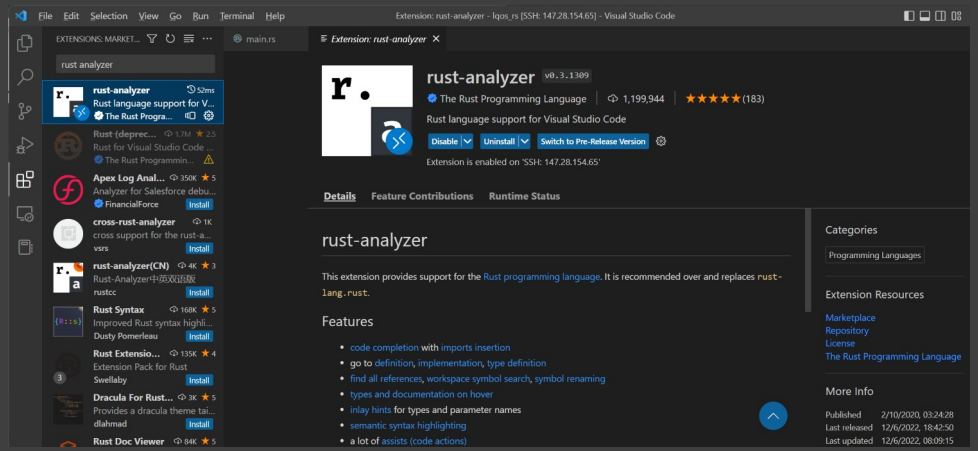
```
               profile: default
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>1

info: profile set to 'default'
info: default host triple is x86_64-unknown-linux-gnu
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
info: latest update on 2022-11-03, rust version 1.65.0 (897e37553 2022-11-02)
info: downloading component 'cargo'
info: downloading component 'clippy'
info: downloading component 'rust-docs'
info: downloading component 'rust-std'
info: downloading component 'rustc'
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
 18.8 MiB /  18.8 MiB (100 %)  15.7 MiB/s in  1s ETA:  0s
info: installing component 'rust-std'
 30.0 MiB /  30.0 MiB (100 %)  20.0 MiB/s in  1s ETA:  0s
info: installing component 'rustc'
 56.2 MiB /  56.2 MiB (100 %)  23.9 MiB/s in  2s ETA:  0s
info: installing component 'rustfmt'
info: default toolchain set to 'stable-x86_64-unknown-linux-gnu'

  stable-x86_64-unknown-linux-gnu installed - rustc 1.65.0 (897e37553 2022-11-02)


Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, run:
source "$HOME/.cargo/env"
herbert@trial2:~$
```

- The install script installs Rust and tooling for you.
- If you need any additional packages, they will be listed here.

IDE/Text Editor Selection

- Rust Analyzer
- VIM and EMACS
- JetBrains *IntelliJ* and C-Lion
- Microsoft *Visual Studio Code*
- We’ll use *Visual Studio Code* for this class

Configuring Rust Analyzer



- Click the “Extensions” button
- Search for “rust-analyzer”
- Click Install

Rust Analyzer Features

```rust
async fn get_data() -> Result<DataResult> {
    let mut result: DataResult = DataResult {
        totals: (0, 0, 0, 0),
        top: Vec::new(),
    };
    let mut stream: TcpStream = TcpStream::connect(addr: BUS_BIND_ADDRESS).await?;
    let test: BusSession = BusSession {
        auth_cookie: 1234,
        requests: vec![
            BusRequest::GetCurrentThroughpu
            BusRequest::GetTopNDownloaders(
        ],
    };
    let msg: Vec<u8> = encode_request(&test
    stream.write(src: &msg).await?;
    let mut buf: Vec<u8> = Vec::new();
    let _ = stream.read_to_end(&mut buf).aw
    let reply: BusReply = decode_response(b

    for r: &BusResponse in reply.responses.iter() {
        match r {
```

```rust
tokio::net::tcp::stream::TcpStream
pub async fn connect<A>(addr: A) -> io::Result<TcpStream>
where
    A: ToSocketAddrs,
```

Opens a TCP connection to a remote host.

`addr` is an address of the remote host. Anything which implements the [`ToSocketAddrs`] trait can be supplied as the address. If `addr` yields multiple addresses, connect will be attempted with each of the addresses until a connection is successful. If none of the addresses result in a successful connection, the error returned from the last connection attempt (the last address) is returned.

To configure the socket before connecting, you can use the [`TcpSocket`] type.

```rust
let test: BusSession = BusSession {
```

Extract...
- Extract into variable
- Extract into function

Rewrite...
- Replace let with if let
- Insert explicit type `BusSession`

```rust
tThroughput,
wnloaders(10),
uest(&test)?;
?;
let mut buf: Vec<u8> = Vec::new();
```

\[EXPANSION].rs

```rust
// Recursive expansion of vec! macro
// ==================================

(<[_]>::into_vec(
    #[rustc_box]
    $crate::boxed::Box::new([command]),
))
```

- Real-time code warning detection
- Syntax highlighting
- Documentation tool-tips
- Refactoring

Quick Tool Overview

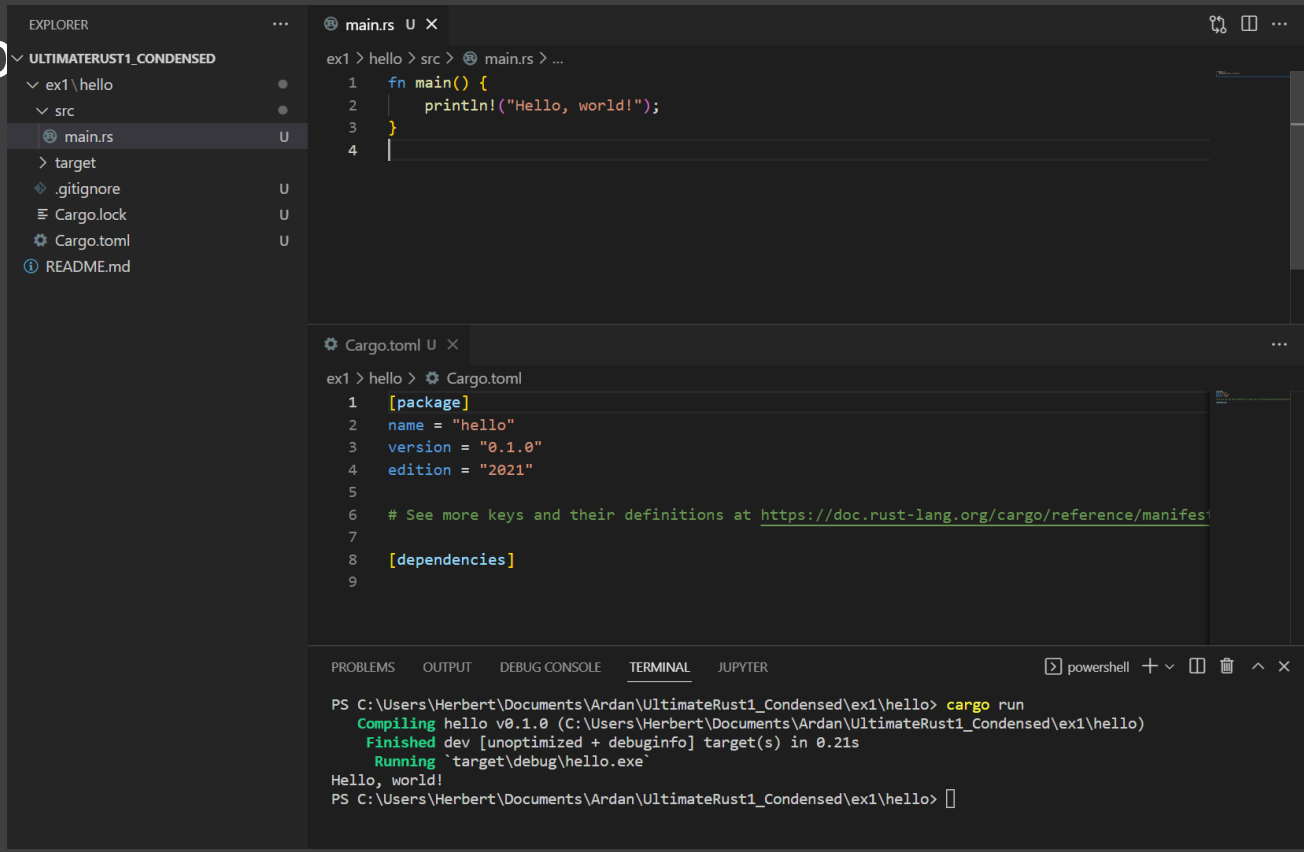
- Rustup
- Cargo
- Rustc
- Clippy
- Cargo’s sub tools

Stay up-to-date with RustUp

- `rustup self update`
  - Update rustup itself to the latest release
- `rustup update`
  - Update all installed Rust tools to the latest release
- `rustup toolchain install wasm32-unknown-unknown`
  - Install a specific “toolchain” – you can cross-compile to other targets.

Create “Hello World” with Cargo

- `cargo init hello`
- Git repo created
- src/main.rs
- `cargo run`
- Everything you need to get started



Code in ex1/hello

Generated Cargo.toml

- Sections in `[square brackets]`
- `Name`: the “crate” name
- `Version`: The version of your Rust program.
- `Edition`: The “edition” of Rust to use – 2021 is current.

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at

[dependencies]

```

Generated main.rs

- `fn` denotes a *function*
- Most programs require a `main` function – the entry point.
- `println!` Prints text to the screen
- Having a `main.rs` file indicates that this project is an executable. If it were `lib.rs`, the output would be a library.

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

```

Ask the User’s Name

```rust
use std::io;                                        // Import the io namespace from the standard lib

fn greet_user() -> String {                         // Declare a function
    println!("Hello, what is your name?");
    let mut buffer: String = String::new();         // Create a MUTABLE string
    let stdin: Stdin = io::stdin();                 // Acquire Standard Input from OS
    stdin.read_line(buf: &mut buffer).unwrap();     // Read a line into the buffer
    buffer                                          // Return the buffer
}

▶ Run | Debug
fn main() {
    let user_name: String = greet_user();           // Call our function, store result in user_name
    println!("Hello, {user_name}");                 // Use the println! macro to say hello
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER powershell

```
PS C:\Users\Herbert\Rust\hello> cargo run
   Compiling hello v0.1.0 (C:\Users\Herbert\Rust\hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target\debug\hello.exe`
Hello, what is your name?
Herbert
Hello, Herbert
```

- Import functionality from namespaces
- Create a function
- Mutable vs Immutable.
- `println!` Is a *macro*.
- Error handling and `unwrap()`

Code in ex2/hello

Dividing Your Code with Modules

main.rs U × greeter.rs U

src > main.rs > ...

```rust
mod greeter;              // Include "greeter.rs" in the build
use greeter::greet_user;  // Import "greet_user" from the "greeter" module

▶ Run | Debug
fn main() {
    let user_name: String = greet_user();
    println!("Hello, {user_name}");
}
```

main.rs U greeter.rs U ×

src > greeter.rs > ...

```rust
use std::io;

pub fn greet_user() -> String {             // The function is PUBLIC
    println!("Hello, what is your name?");
    let mut buffer: String = String::new();
    let stdin: Stdin = io::stdin();
    stdin.read_line(buf: &mut buffer).unwrap();
    buffer
}
```

- Create `greeter.rs`
- Public vs Private
- Include it with `mod greeter`
- Use the `greet_user` function

Code in ex3/hello

Dependencies & Errors

Cargo.toml U

Cargo.toml

```toml
name = "hello"
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1" # Import the "anyhow" crate from Cargo/Crates.io
```

greeter.rs U

src > greeter.rs > greet_user

```rust
use std::io;
use anyhow::{Result, Error}; // Use Result and Error from anyhow.

pub fn greet_user() -> Result<String> { // Return a Result, wrapping your actual result
    println!("Hello, what is your name?");
    let mut buffer: String = String::new();
    let stdin: Stdin = io::stdin();
    stdin.read_line(buf: &mut buffer)?; // ? means "if an error occurs, exit the fn with an error"
    if buffer.trim().to_lowercase() != "herbert" { // Trim any special characters
        Err(Error::msg(message: "Access denied!")) // Return an error
    } else {
        Ok(buffer) // Return the buffer, wrapped in Ok to indicate success
    }
}
```

main.rs U

src > main.rs > main

```rust
mod greeter;
use greeter::greet_user;

▶ Run | Debug
fn main() {
    let result: Result<String, Error> = greet_user(); // result is a Result type
    if let Ok(user_name: String) = result { // If its ok, if let will unwrap the result
        println!("Hello, {user_name}"); // and let you print the name.
    } else {
        println!("{:?}", result); // Otherwise, use {:?} to "debug print" the result.
    }
}
```

- `cargo search` to find available packages.
- Include in `[dependencies]` to use.
- *anyhow* makes error-handling easier.
- Results are not exceptions!

Code in ex4/hello

Create a Library

lib.rs U

greeter > src > lib.rs > greet_user

```rust
use std::io;
use anyhow::{Result, Error};

pub fn greet_user() -> Result<String> {
    println!("Hello, what is your name?");
    let mut buffer: String = String::new();
    let stdin: Stdin = io::stdin();
    stdin.read_line(buf: &mut buffer)?;
    if buffer.trim().to_lowercase() != "herbert" {
        Err(Error::msg(message: "Access denied!"))
    } else {
        Ok(buffer)
    }
}
```

lib.rs U Cargo.toml U

greeter > Cargo.toml

```toml
[package]
name = "greeter"
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1"
```

- `cargo init –lib greeter`
- Copy `greeter.rs` into the new `greeter/src/lib.rs` file
- Move anyhow dependency into `greeter/Cargo.toml`

Code in ex5/

Use your Library

main.rs U

src > main.rs > main

```rust
use greeter::greet_user;

▶ Run | Debug
fn main() {
    let result: Result<String, Error> = greet_user();
    if let Ok(user_name: String) = result {
        println!("Hello, {user_name}");
    } else {
        println!("{:?}", result);
    }
}
```

Cargo.toml U

Cargo.toml

```toml
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

[dependencies]
greeter = { path = "greeter/" }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
PS C:\Users\Herbert\Rust\hello> cargo tree
hello v0.1.0 (C:\Users\Herbert\Rust\hello)
└── greeter v0.1.0 (C:\Users\Herbert\Rust\hello\greeter)
    └── anyhow v1.0.66
PS C:\Users\Herbert\Rust\hello>
```

- Remove “`mod greeter`” from main.rs
- Remove *anyhow* from `Cargo.toml`
- Add greeter to `Cargo.toml`
- Use `cargo tree` to understand your dependencies.

Cargo Workspaces

Cargo.toml

```toml
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

[dependencies]
greeter = { path = "greeter/" }

[workspace]
members = [ "greeter" ]
```

- Combine projects with *workspaces*.
- Add `[workspace]` and `members = [ … ]` to `Cargo.toml`
- Why use workspaces?
  - Reclaim disk space
  - Shared compilation: faster

Code in ex6/

Add an `is_valid()` function

greeter > src > lib.rs > ...

```rust
use std::io;
use anyhow::{Result, Error};

pub fn greet_user() -> Result<String> {
    println!("Hello, what is your name?");
    let mut buffer: String = String::new();
    let stdin: Stdin = io::stdin();
    stdin.read_line(buf: &mut buffer)?;
    if is_valid_user(&buffer) {                    // Call "is_valid_user"
        Err(Error::msg(message: "Access denied!"))
    } else {
        Ok(buffer)
    }
}

fn is_valid_user(user: &str) -> bool {             // Function is private
    user.trim().to_lowercase() == "herbert"        // Function returns last expression
}
```

- Move the name checking logic into a new function.
- The function is private: you can't access it from outside this module.

Code in ex7/

Unit Testing is_valid()

greeter > src > lib.rs > ...

```rust
#[cfg(test)]
▶ Run Tests | Debug
mod test {
    use super::*;

    #[test]
    ▶ Run Test | Debug
    fn test_valid() {
        assert!(is_valid_user("herbert"));
    }

    #[test]
    ▶ Run Test | Debug
    fn test_valid_case() {
        assert!(is_valid_user("HeRbErT\r\n"));
    }

    #[test]
    ▶ Run Test | Debug
    fn test_invalid() {
        assert!(!is_valid_user("Bob"));
    }
}
```

- `#[cfg(test)]` ensures the tests only compile when you are running tests.
- `mod test` makes a module inside your module
- `use super::*` imports everything from the parent module.
- `#[test]` indicates a test function
- `cargo test` runs all tests
  `cargo test --all` runs all tests in the workspace.

Other Cargo Tools

- Cargo is extensible. It can also:
  - Search for packages with cargo search
  - Install new features with cargo install
  - “Lint” your code with cargo clippy
  - Format your code with cargo fmt
  - Compile C and C++ code as part of your build
  - Install new Cargo features and do even more...

Life without Cargo

- Some projects require that you integrate into an existing Make, CMake or other build system.
- Cargo calls rustc for actual compilation.
- You can use rustc in Makefiles, and CMake projects.

Life without the Standard Library

- Sometimes you need *small* binaries.
- “Hello World” is 1 Mb – because it links the entire “standard library”.
- You can add #[no_std] to your project to not include the standard library – and make binaries as small as 700 bytes.
- Downside: You don’t have the standard library.

Rust’s Safety Guarantees

- Memory Safety
  - You can’t have mutable access to a section of memory more than once.
  - All types are bounded (and bounds-checked in debug mode) – no more buffer overruns.
  - “Use after free” is a compiler error.
- No Data Races
- The “unsafe” keyword

Wrap-Up

- Any Questions?
- (Contact information)