

Rust Networking Tutorial

Khalid Hourani

University of Houston

April 14, 2021



Table of Contents

1 The Rust Language

2 Networking



Table of Contents

1 The Rust Language

2 Networking



What is Rust?



What is Rust?

Rust is a multi-paradigm programming language with



What is Rust?

Rust is a multi-paradigm programming language with

- no runtime (e.g. garbage collector)



What is Rust?

Rust is a multi-paradigm programming language with

- no runtime (e.g. garbage collector)
- while still guaranteeing memory safety



What is Rust?

Rust is a multi-paradigm programming language with

- no runtime (e.g. garbage collector)
- while still guaranteeing memory safety
 - **at compile time**



Performance



Performance

Rust has performance comparable to C/C++ with no runtime or garbage collector, and can



Performance

Rust has performance comparable to C/C++ with no runtime or garbage collector, and can

- handle performance-critical services



Performance

Rust has performance comparable to C/C++ with no runtime or garbage collector, and can

- handle performance-critical services
- run on embedded devices



Safety



Safety

Rust guarantees memory and thread safety **at compile time** with



Safety

- Rust guarantees memory and thread safety **at compile time** with
- rich, algebraic type system



Safety

Rust guarantees memory and thread safety **at compile time** with

- rich, algebraic type system
 - e.g. can utilize return value error handling without typical boilerplate overhead



Safety

Rust guarantees memory and thread safety **at compile time** with

- rich, algebraic type system
 - e.g. can utilize return value error handling without typical boilerplate overhead
- a borrow checker



Safety

Rust guarantees memory and thread safety **at compile time** with

- rich, algebraic type system
 - e.g. can utilize return value error handling without typical boilerplate overhead
- a borrow checker
 - very strict rules set by the compiler



Safety

Rust guarantees memory and thread safety **at compile time** with

- rich, algebraic type system
 - e.g. can utilize return value error handling without typical boilerplate overhead
- a borrow checker
 - very strict rules set by the compiler
 - allows compiler to verify memory and thread safety



Safety

Rust guarantees memory and thread safety **at compile time** with

- rich, algebraic type system
 - e.g. can utilize return value error handling without typical boilerplate overhead
- a borrow checker
 - very strict rules set by the compiler
 - allows compiler to verify memory and thread safety
- lifetimes



Safety

Rust guarantees memory and thread safety **at compile time** with

- rich, algebraic type system
 - e.g. can utilize return value error handling without typical boilerplate overhead
- a borrow checker
 - very strict rules set by the compiler
 - allows compiler to verify memory and thread safety
- lifetimes
 - specify the scope during which objects will survive in memory



Ownership



Ownership

Rust has an **ownership** system



Ownership

Rust has an **ownership** system

- every value has a **unique** *owner* (variable)



Ownership

Rust has an **ownership** system

- every value has a **unique** *owner* (variable)
- when an owner's scope ends, the value is destroyed



Ownership

Rust has an **ownership** system

- every value has a **unique** *owner* (variable)
- when an owner's scope ends, the value is destroyed
 - common known as RAII



Ownership

Rust has an **ownership** system

- every value has a **unique** *owner* (variable)
- when an owner's scope ends, the value is destroyed
 - common known as RAII
 - Resource Acquisition is Initialization



Ownership

Rust has an **ownership** system

- every value has a **unique** *owner* (variable)
- when an owner's scope ends, the value is destroyed
 - common known as RAII
 - Resource Acquisition is Initialization



Ownership

Rust has an **ownership** system

- every value has a **unique** *owner* (variable)
- when an owner's scope ends, the value is destroyed
 - common known as RAII
 - Resource Acquisition is Initialization

The **borrow checker** enforces this system



The Borrow Checker



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`

The borrow checker essentially enforces the following:



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`

The borrow checker essentially enforces the following:

- at any time, there can be



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`

The borrow checker essentially enforces the following:

- at any time, there can be
 - any number of immutable references to an object, or



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`

The borrow checker essentially enforces the following:

- at any time, there can be
 - any number of immutable references to an object, or
 - **exactly one** mutable reference to an object



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`

The borrow checker essentially enforces the following:

- at any time, there can be
 - any number of immutable references to an object, or
 - **exactly one** mutable reference to an object
- but not both



The Borrow Checker

To avoid copying everything by value, Rust allows passing by reference. Every reference can be

- immutable
 - `&T`
- mutable
 - `&mut T`

The borrow checker essentially enforces the following:

- at any time, there can be
 - any number of immutable references to an object, or
 - **exactly one** mutable reference to an object
- but not both
- implicit readers-writers lock



Rust Syntax



Rust Syntax

Syntactically, Rust is somewhat similar to C++. For example, the Hello World in Rust looks like



Rust Syntax

Syntactically, Rust is somewhat similar to C++. For example, the Hello World in Rust looks like

Hello World

```
1 fn main() {  
2     println!("Hello, World!");  
3 }
```



Rust Syntax

Syntactically, Rust is somewhat similar to C++. For example, the Hello World in Rust looks like

Hello World

```
1 fn main() {  
2     println!("Hello, World!");  
3 }
```

Like C and C++, Rust has a **main** entry point



Syntax (Continued)

- Blocks of code are delimited by curly braces



Syntax (Continued)

- Blocks of code are delimited by curly braces
- **Statements** are ended with semi-colons



Syntax (Continued)

- Blocks of code are delimited by curly braces
- **Statements** are ended with semi-colons
- **if**, **else**, **while**, **for** are control-flow keywords



Syntax (Continued)

- Blocks of code are delimited by curly braces
- **Statements** are ended with semi-colons
- **if**, **else**, **while**, **for** are control-flow keywords
- **match** keyword works similar to C **switch**



Syntax (Continued)

- Blocks of code are delimited by curly braces
- **Statements** are ended with semi-colons
- **if**, **else**, **while**, **for** are control-flow keywords
- **match** keyword works similar to C **switch**
- **let** keyword is used for assignment



Syntax (Continued)

- Blocks of code are delimited by curly braces
- **Statements** are ended with semi-colons
- **if**, **else**, **while**, **for** are control-flow keywords
- **match** keyword works similar to C **switch**
- **let** keyword is used for assignment
 - Compiler will attempt to infer type if not specified



Syntax (Continued)

- Blocks of code are delimited by curly braces
- **Statements** are ended with semi-colons
- **if**, **else**, **while**, **for** are control-flow keywords
- **match** keyword works similar to C **switch**
- **let** keyword is used for assignment
 - Compiler will attempt to infer type if not specified
 - e.g. **let** `x = 3;` *// x is u32*



Syntax (Continued)



Syntax (Continued)

- Expression-based



Syntax (Continued)

- Expression-based
 - Blocks are expressions and can be assigned to variables



Syntax (Continued)

- Expression-based
 - Blocks are expressions and can be assigned to variables



Syntax (Continued)

- Expression-based
 - Blocks are expressions and can be assigned to variables

```
1 let x = 3;
2 let y = {
3     let x_squared = x * x;
4     let x_cubed = x_squared * x;
5
6     // This expression will be assigned to `y`
7     x_cubed + x_squared + x
8 };
```



Syntax (Continued)

- Expression-based
 - Blocks are expressions and can be assigned to variables

```
1 let x = 3;
2 let y = {
3     let x_squared = x * x;
4     let x_cubed = x_squared * x;
5
6     // This expression will be assigned to `y`
7     x_cubed + x_squared + x
8 };
```



Syntax (Continued)



Syntax (Continued)

- Implicit function returns



Syntax (Continued)

- Implicit function returns
 - when semi-colon is **omitted**, last expression is returned



Syntax (Continued)

- Implicit function returns
 - when semi-colon is **omitted**, last expression is returned
 - Rust also allows explicit returns



Syntax (Continued)

- Implicit function returns
 - when semi-colon is **omitted**, last expression is returned
 - Rust also allows explicit returns
 - style is to only use explicit returns for early termination



Syntax (Continued)

- Implicit function returns
 - when semi-colon is **omitted**, last expression is returned
 - Rust also allows explicit returns
 - style is to only use explicit returns for early termination



Syntax (Continued)

- Implicit function returns
 - when semi-colon is **omitted**, last expression is returned
 - Rust also allows explicit returns
 - style is to only use explicit returns for early termination

```
1 fn is_prime(n: u32) -> bool {  
2     for i in (2..n) {  
3         if n % i == 0 {  
4             return false;  
5         }  
6     }  
7     true  
8 }
```



Syntax (Continued)



Syntax (Continued)

Rust supports **structs**



Syntax (Continued)

Rust supports **structs**

- similar to classes



Syntax (Continued)

Rust supports **structs**

- similar to classes
- groups data



Syntax (Continued)

Rust supports **structs**

- similar to classes
- groups data
- supports **methods** unlike C



Syntax (Continued)

Rust supports **structs**

- similar to classes
- groups data
- supports **methods** unlike C
 - implicit functions



Syntax (Continued)

Rust supports **structs**

- similar to classes
- groups data
- supports **methods** unlike C
 - implicit functions
 - writing using **impl** keyword



Syntax (Continued)

Rust supports **structs**

- similar to classes
- groups data
- supports **methods** unlike C
 - implicit functions
 - writing using **impl** keyword
- e.g. can create struct Rectangle and call rectangle.area()



Table of Contents

1 The Rust Language

2 Networking



Networking in Rust



Networking in Rust

Networking can be done using the `std::net` standard library



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication
- `IpAddr` — IPv4 and IPV6 addresses



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication
- `IpAddr` — IPv4 and IPV6 addresses
 - `Ipv4Addr`



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication
- `IpAddr` — IPv4 and IPV6 addresses
 - `Ipv4Addr`
 - `Ipv6Addr`



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication
- `IpAddr` — IPv4 and IPV6 addresses
 - `Ipv4Addr`
 - `Ipv6Addr`
- `SocketAddr` — socket address of IPv4 and IPv6



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication
- `IpAddr` — IPv4 and IPV6 addresses
 - `Ipv4Addr`
 - `Ipv6Addr`
- `SocketAddr` — socket address of IPv4 and IPv6
 - `SocketAddrV4`



Networking in Rust

Networking can be done using the `std::net` standard library

- `TcpListener` — TCP Socket Server
- `TcpStream` — Stream between local and remote socket
- `UdpSocket` — Functionality for UDP communication
- `IpAddr` — IPv4 and IPV6 addresses
 - `Ipv4Addr`
 - `Ipv6Addr`
- `SocketAddr` — socket address of IPv4 and IPv6
 - `SocketAddrV4`
 - `SocketAddrV6`



TcpListener

```
1 fn handle_client(stream: TcpStream) {
2     // ...
3 }
4
5 fn main() -> Result<()> {
6     let address = "127.0.0.1:80";
7     let listener = TcpListener::bind(address)?;
8
9     // accept connections and process serially
10    for stream in listener.incoming() {
11        handle_client(stream?);
12    }
13    Ok(())
14 }
```



TcpStream

```
1 fn main() -> Result<()> {
2     let addr = "127.0.0.1:34254"
3     let mut stream = TcpStream::connect(addr)?;
4
5     stream.write(&[1])?;
6     stream.read(&mut [0; 128])?;
7     Ok(())
8 } // the stream is closed here
```



UdpSocket

```
1 fn main() -> Result<()> {
2     {
3         let addr = "127.0.0.1:34254"
4         let mut socket = UdpSocket::bind(addr)?;
5
6         let mut buf = [0; 10];
7         let (amt, src) = socket.recv_from(&mut buf)?;
8
9         let buf = &mut buf[..amt];
10        buf.reverse();
11        socket.send_to(buf, &src)?;
12    } // the socket is closed here
13    Ok(())
14 }
```



Comparison with C

```
1 let args: Vec<String> = env::args().collect();
2 match args.len() {
3     1 => panic!("Please pass port number to command line"),
4     _ => (),
5 }
6 let port = &args[1];
7 let address = format!("localhost:{}", port);
8 let listener = TcpListener::bind(address)?;
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31 //
```

```
1 if (argc < 2) {
2     perror("Please pass port number to command line");
3     exit(EXIT_FAILURE);
4 }
5 int port_number = atoi(argv[1]);
6 int socket_fd = socket(AF_INET6, SOCK_STREAM, 0);
7 if (socket_fd < 0) {
8     perror("Error creating socket");
9     exit(EXIT_FAILURE);
10 }
11 int set = 1;
12 int ret = setsockopt(socket_fd, SOL_SOCKET, SO_REUSEADDR, &set,
13 if (ret < 0) {
14     perror("setsockopt failed");
15     return -1;
16 }
17 struct sockaddr_in6 server_address;
18 memset(&server_address, 0, sizeof(server_address));
19 server_address.sin6_family = AF_INET6;
20 server_address.sin6_port = htons(port_number);
21 server_address.sin6_addr = in6addr_any;
22 ret = bind(socket_fd, (struct sockaddr*)&server_address, sizeof
23 if (ret < 0) {
24     perror("Bind failed");
25     exit(EXIT_FAILURE);
26 }
27 ret = listen(socket_fd, 1);
28 if (ret < 0) {
29     perror("Listen failed");
30     exit(EXIT_FAILURE);
31 }
```

