Brief Introduction to Rust-lang for HPC

GöHPCoffee
2023-10-25

Artur Wachtel (GWDG)

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

What is Rust

* Programming language

  which addresses the problem of _shared mutable state_

  reliability   - memory safety, thread safety, backwards compatibility
  performance   - compiled to machine code, no garbage collector, fearless concurrency
  productivity - convenient tooling, helpful compiler errors

  https://rust-lang.org

* Release of version 1.0 in 2015

* Most loved/admired language (StackOverflow) for multiple years in a row

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

What is Rust used for

* Low level → High level
  * Embedded / Systems programming
  * Command line tools
  * Web development

* Growing adoption in the industry
  * Firefox
  * Android
  * Linux Kernel
  * Windows Kernel

What do we learn today?

* How to read Rust code
* What makes Rust different
* Built-in features for fearless concurrency

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

Rust tooling

* `rustup` the Rust toolchain manager
* `rust-analyzer` the Rust language server for IDE integration

* `rustc` the Rust compiler, built on top of LLVM
  rustc version (1.73.0) and Rust Language Edition (2015, 2018, 2021)
* `clippy` the Rust linter to catch common mistakes and enforce code style
* `cargo` the Rust build system and package manager

* minimal standard library
* extensive community crate registry on crates.io

Rust on the SCC

```
$ module load rust/1.65.0
$ cargo new hello
$ cd hello
$ cargo run
[...]
Hello, world!
```

Cargo

```
$ cargo new hello
Create new project

Results in

  hello/
    .git/
    src/
      main.rs
    Cargo.toml

$ cargo check
Check source code without compiling

$ cargo build [--release]
Pull dependencies and compile [optimized]

$ cargo run [--release]
Run the [optimized] binary
```

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

```toml
#  Cargo.toml  created by $ cargo new


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

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

```
//  src/main.rs  created by $ cargo new

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

- Rust.md
- What is Rust.md
- What is Rust used for.md
- What do we learn today.md
- Tooling.md
- Cargo new.md
- Cargo.toml
- main.rs
- println.rs
- Flow.rs
- Variables.rs
- Types.rs
- Functions.rs
- References.rs
- Ownership.md
- Structs.rs
- Enums.rs
- Generics.rs
- Null.rs
- Heap.rs
- Escape hatches.md
- Iterators.rs
- Rayon.rs
- Threads.rs
- Mutable State.rs
- Mutex.rs
- Channels.rs

```rust
//  The println! macro

// Hello, world1!
fn main() {
    let a = "Hello";
    let b = "world";
    let x = 1;
    println!("{}, {}{}!", a, b, x);  // variable number of arguments
}

// The exclamation point signifies that println! is a macro

// The three variables we created (a, b, x) have a known type at compile type.
// But for ease of use, the compiler will infer many types so you don't have to manually specify them.
// In this case a and b are simple strings and x is a 32-bit signed integer.
```

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

```
//  Control flow

fn main() { // Rust provides the common control flow constructs:
            // loops via `for` or `while`, `if` and `else` for branching

    for i in 1..42 { // for loops use iterators

        if i % 3 == 0 {
            if i % 5 == 0 {
                println!("Fizz Buzz");
            }
            else {
                println!("Fizz");
            }
        }
        else if i % 5 == 0 {
            println!("Buzz");
        }
        else {
            println!("{}", i);
        }
    }
}
```

Rust.md
What is Rust.md
What is Rust used for.md
What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs

```rust
//  Variables

fn main() {

    // declaration of variables
    let x;

    // initialization
    x = 42;

    // both in a single line, with (optional) type annotation after the variable name
    let y: i32 = 42;

    // ⚡ Compile time error!
    // assignment of value to immutable variable is forbidden by compiler
    y = 5;

    // mutability is opt-in!!   variables are IMMUTABLE BY DEFAULT
    let mut y = 1.0; // f64
    y = 2.0 * (x as f64);  // primitive type cast is never implicit

}
```

- What do we learn today.md
- Tooling.md
- Cargo new.md
- Cargo.toml
- main.rs
- println.rs
- Flow.rs
- Variables.rs
- Types.rs
- Functions.rs
- References.rs
- Ownership.md
- Structs.rs
- Enums.rs
- Generics.rs
- Null.rs
- Heap.rs
- Escape hatches.md
- Iterators.rs
- Rayon.rs
- Threads.rs
- Mutable State.rs
- Mutex.rs
- Channels.rs
- Summary.md
- What we did not talk about.md
- References.md

```rust
//  Basic Types

fn main() {

    // Primitive types
    let t: bool = true; // boolean - true, false
    let m: i32 = -5; // signed integers - i8, i16, i32, i64, i128
    let n: u64 = 42; // unsigned integers - u8, u16, u32, u64, u128
    let i: isize = 1; // integers with length depending on CPU arch - isize, usize
    let x: f64 = 2_000_000.0; // floating point numbers - f32, f64

    let crab: char = '🦀'; // four bytes Unicode scalar value
    let s: &str = "utf8 string literal 🦀"; // static string

    // Compound types
    let array: [i32; 4] = [1, 3, 7, 8]; // fixed-size arrays of uniform type
    let tuple: (i32, f64, char) = (0, 1., '🦀'); // fixed-size tuple of mixed types

    array[0]; // array indexing starts at 0
    (tuple.0, tuple.1); // tuple member access via dot

    // Unit type
    let unit: () = (); // tuple with no elements
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
//  Functions

// functions with no arguments and no return value
fn hello() -> () {
    println!("Hello, world!");
}

// arguments and return values *must* have type annotations
fn square(x: i32) -> i32 {
    // Last expression in a function is the return value.
    // Take note of the missing semicolon!!
    x*x
}

fn main() -> () { // In fact, the `-> ()` can be omitted as we have seen previously
    // calling a fuction
    hello();

    // functions are first class citizens
    let f = square;
    let cube = |x| {x*x*x};

    println!("4*4 = {}", f(4));
    println!("4*4*4 = {}", cube(4u8));
}
```

```rust
//  References and Borrowing

fn multiply(a: &f64, b: &f64) -> f64 {  // `&T` is an immutable reference
    a*b
}

fn multiply_to(a: &mut f64, b: &f64) {  // `&mut T` is a mutable reference
    *a = (*a) * b;   // (*a) is the dereference operation
}

fn main() {
    let mut x: f64 = 3.0;
    let mut y: f64 = 5.0;

    // You can pass a mutable reference where an immutable reference is expected.
    let product1 = multiply(&mut y, &x);

    // You can create as many immutable references as you want.
    let product2 = multiply(&x, &x);

    // XOR at most one mutable reference.
    multiply_to(&mut y, &x);

    // ⚡ Compile time error!
    // You *cannot* have a mutable and immutable reference
    multiply_to(&mut y, &y);
    // You *cannot* have two mutable references
    multiply_to(&mut y, &mut y);

}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

Ownership rules

* Each value in Rust is owned by a variable.

* There can only be one owner at a time.

* When the owner goes out of scope, the value will be dropped.

Borrowing rules

* At any given time, you can have _either_ one mutable reference XOR any number of immutable references.

* References must always be valid.

Safety guarantees

* No null pointers, dangling pointers, or data races.

- What do we learn today.md
- Tooling.md
- Cargo new.md
- Cargo.toml
- main.rs
- println.rs
- Flow.rs
- Variables.rs
- Types.rs
- Functions.rs
- References.rs
- Ownership.md
- Structs.rs
- Enums.rs
- Generics.rs
- Null.rs
- Heap.rs
- Escape hatches.md
- Iterators.rs
- Rayon.rs
- Threads.rs
- Mutable State.rs
- Mutex.rs
- Channels.rs
- Summary.md
- What we did not talk about.md
- References.md


```rust
//  Structured data with shared behavior

struct Rect { width: f64, height: f64 } // structs store related data

impl Rect {
    fn new() -> Self {
        Rect { height: 1.0, width: 1.0 }
    }
}

trait Area { fn area(&self) -> f64; } // traits allow shared behavior

impl Area for Rect {
    fn area(&self) -> f64 { self.width * self.height }
}

#[derive(PartialEq, PartialOrd)]  // We derive traits from the std library for operator overloading
struct Circ(f64); // tuple struct

impl Area for Circ {
    fn area(&self) -> f64 { 2.0 * 3.14 * self.0 * self.0}
}

fn main() {
    let r = Rect::new(); // associated function as "constructor"
    let c1 = Circ(1.0);
    let c2 = Circ(42.0);
    if c2 > c1 {  // we can compare two circles because we derived the PartialOrd trait
        println!("{}, {}", r.area(), c2.area()); // method syntax equivalent to Area::area(&r)
    }
}
```

```rust
//  Enums and the Match Expression

enum Color{
    Black,
    White,
    RGB(u8, u8, u8),  // Enum variants can hold arbitrary data, in this case a tuple
}

fn what_color(c: &Color) {

    let g: u8 = match c { // match expression is exhaustive
        Color::RGB(0, 0, 0) => {
            println!("Deepest black."); 0
        }
        Color::Black => {
            println!("Deepest black."); 0
        }
        Color::White | Color::RGB(255, 255, 255) => { // expression matching can be very elegant
            println!("Purest white."); 255
        }
        Color::RGB(_, g,_) => { *g } // expression matching can also grant access to internal data
    };
    println!("Color with green value {}.", g);
}

fn main() {
    let c1 = Color::RGB(255, 255, 255);
    let c2 = Color::RGB(128, 128, 255);
    what_color(&c1);
    what_color(&c2);
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
// Generic Types

struct Rect<T> { width: T, height: T, }  // Generic types are annotated via <T>

impl<T> Rect<T> { // Generics are monomorphized during compile time
    fn new(a: T, b: T) -> Self { // nomomorphized functions are statically dispatched
        Rect { height: a, width: b }
    }
}

struct Circ<T>(T);

trait Area { fn area(&self) -> f64; }

impl<T> Area for Rect<T>
where T: Copy, f64: From<T>  // You can condition Generics with trait bounds
{                            // where Type1: Trait1 + Trait2, Type2: Trait3 + Trait4
    fn area(&self) -> f64 {
        f64::from(self.width) * f64::from(self.height)
    }
}

impl<T> Area for Circ<T>
where T: Copy, f64: From<T>
{
    fn area(&self) -> f64 {
        2.0 * 3.14 * f64::from(self.0) * f64::from(self.0)
    }
}

fn main() {
    let r = Rect::new(2, 3);  // Note that we do not have to specify the generic type explicitly!
    let c = Circ(42u8);
    println!("{}, {}", r.area(), c.area());
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
// There is no Null

fn main() {

    // Absense of value is represented with the enum Option<T> from the std library.
    // Its variants are Some(T) and None. Often used as return type from functions.

    let maybe_one: Option<f64> = Some(1.0);
    let maybe_two: Option<f64> = None;

    match (maybe_one, maybe_two) {  // match is always exhaustive and throws compile time errors
        (Some(a), Some(b)) => {     // if you forget to cover any case
            println!("{}", a*b);
        },
        _ => {  // catchall pattern for other cases: (None, Some(_)), (Some(_), None), (None, None)
            println!("Absense of value detected!");
        },
    }

    if let Some(a) = maybe_one { // `if let` is a concise way to cover exactly one case
        println!("variable carries the value {}", a);
    }

}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
// Heap allocation

fn call_by_value(s: String) {
    println!("{}", s);
}

fn main() {

    let mut s = String::from("Hello,"); // Dynamically sized strings.
    s.push_str("world!");

    let mut v: Vec<i32> = vec![1, 2, 3, 4]; // Heap allocation for dynamically sized arrays.
    v.push(5);

    let one: Box<i32> = Box::new(1); // Heap allocation + smart pointer to that value.
    let two = 1 + *one; // You can dereference a Box as if it was a reference.

    let v2 = v; // Transfer of ownership invalidates previous variable.
    call_by_value(s); // Call-by-value will move ownership too.

    let v3 = v2.clone(); // Deep copy of heap data via clone method.

    // ⚡ Compile time error!
    // Trying to access invalidated variables
    println!("{}", v[0]);
    println!("{}", s);

    // All heap allocations are automatically cleaned up at the end of scope
    // No manual freeing of memory, no double free, no use after free
}
```

(Partial) Escape hatches

* `Copy` trait to avoid move of ownership - but only for primitive types and stack data
* `Rc<T>` multiple ownership via reference counting - at run time
* `RefCell<T>` interior mutability with borrowing rules - at run time
* Extreme case: `unsafe` keyword to work with raw pointers directly

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
//  Iterators

//  Σ_i {i²}

fn sum_of_squares(n: u64) -> u64 {
    n * (n + 1) * (2 * n + 1) / 6
}

fn main() { // Iterator manipulation compiles down to equivalent for loops.
            // Advantage: you don't need to handle the internal mutable variables.
    let n: u64 = 1_000_000;
    let s: u64 = (1..=n)
        .map(|x| x * x)
        .sum();

    println!("{} - {}", s, sum_of_squares(n));

    let v1 = vec![1.0, 2.0, 3.0];
    let v2 = vec![5.0, 6.0, 7.0];

    let scalar_product: f64 = v1.iter() // You can iterate over many collection types
        .zip(v2)
        .map(|(x, y)| {x*y})
        .sum();
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```
//  Easy Data Parallelism with Rayon


use rayon::prelude::*;

fn sum_of_squares(n: u64) -> u64 {
    n * (n + 1) * (2 * n + 1) / 6
}

fn main() {  // Here we use parallel iterators from the `rayon` crate
             // Our source code looks the same, but rayon handles the task with a thread pool
    let n: u64 = 1_000_000;
    let s: u64 = (1..=n).into_par_iter()
        .map(|x| x * x)
        .sum();

    println!("{} - {}", s, sum_of_squares(n));

    let v1 = vec![1.0, 2.0, 3.0];
    let v2 = vec![5.0, 6.0, 7.0];

    let scalar_product: f64 = v1.par_iter()
        .zip(v2)
        .map(|(x, y)| {x*y})
        .sum();
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
// Concurrency with Operating System Threads

use std::thread; // Threads are part of the standard library

fn main() {

    let handle = thread::spawn(|| { // Threads take an anonymous function (Closure)
        println!("Hello from the thread!");
    });

    handle.join().unwrap();
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
// Shared Mutable State

use std::thread;

// ⚡ Compile time error!
// This code will not compile!
fn main() {

    let mut state: u64 = 0;

    let state_ref = &mut state;

    let handle1 = thread::spawn(move || { // The move keyword tries to explicitly capture variables
        *state_ref += 1;                  // from the environment, creating a true Closure
    });

    let handle2 = thread::spawn(move || {
        *state_ref += 1;
    });  // But you cannot get this code to compile when trying to grant
         // shared mutable access to `state`. Even trying to wrap it into Rc<T> or RefCell<T>
         // will fail. Those are not thread safe.
    handle1.join().unwrap();
    handle2.join().unwrap();

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

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```rust
// Shared-State Concurrency

use std::thread;

// Instead we need mutually exclusive access (Mutex) and atomic reference counting (Arc)
fn main() {

    let mut state = Arc::new(Mutex::new(0u64));
    let mut handles = vec![];

    for _ in 0..2 {
        let state = Arc::clone(&state);
        let handle = thread::spawn(move || {
            let mut n = state.lock().unwrap();  // via `n` we have exclusive access
            *n += 1; // the type of `n` is a smart poniter to the inner value
        }); // When n goes out of scope, the lock is automatically lifted

        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("{}", *state.lock().unwrap());
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```
//  Channels for message passing between threads


use std::sync::mpsc;
use std::thread;

fn main() {

    // Multiple Producer Single Consumer channel with transmittter tx and receiver rx
    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone(); // we can clone the transmitter to create multiple producers
    thread::spawn(move || { // here we move ownership of the transmitter tx1 into the thread
        let vals = vec![ 1, 3, 5, 7, 9 ];

        for val in vals {
            tx1.send(val).unwrap(); // tx.send(val) is a call-by-value and moves ownership!
        }
    });
    thread::spawn(move || { // here we move ownership of the original transmitter tx
        let vals = vec![ 2, 4, 6, 8, 10 ];

        for val in vals {
            tx.send(val).unwrap();
        }
    });

    for received in rx { // main thread holds onto the receiver and loops over incoming values
        println!("Got: {}", received);
    }
}
```

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```
#  Summary

* Rust is a strongly typed programming language that compiles to machine code.

* Rust manages memory automatically but without garbage collector.

* Rust enforces memory safety at _compile time_
  via Ownership and Borrowing rules
  thus preventing common error types:

  * Dereferencing of Null pointers
  * Dangling pointers
  * Data races

* Rust provides high-level abstractions at zero cost.

* Rust helps to write fast and correct concurrent code.
```

What we did not talk about

* Managing projects with packages, crates, modules
* Error handling with the `Result<T, E>` type instead of `.unwrap()`
* Lifetimes of `&T` when returned from functions
* Trait objects and dynamic dispatch
* Metaprogramming with procedural macros
* Unsafe Rust
* Interoperability with other languages

Interesting crates from crates.io

* `num` numeric types and traits
* `rand` random number generation
* `serde` serialization/deserialization
* `itertools` more powerful iterators
* `ndarray` multi-dimensional arrays (à la NumPy)
* `crossbeam` tools for concurrent programming
* `rayon` easy concurrency for data parallelism
* `mpi` Message Passing Interface

What do we learn today.md
Tooling.md
Cargo new.md
Cargo.toml
main.rs
println.rs
Flow.rs
Variables.rs
Types.rs
Functions.rs
References.rs
Ownership.md
Structs.rs
Enums.rs
Generics.rs
Null.rs
Heap.rs
Escape hatches.md
Iterators.rs
Rayon.rs
Threads.rs
Mutable State.rs
Mutex.rs
Channels.rs
Summary.md
What we did not talk about.md
References.md

```
#   References

    Rust Homepage
        https://www.rust-lang.org/

    The Rust Programming Language, 2nd Ed
        No Starch Press
        ISBN-13: 9781718503106
        https://doc.rust-lang.org/book/


#   Further Reading

    The Rust Performance Book
        https://nnethercote.github.io/perf-book

    The Rustonomicon
        Unsafe Rust
        https://doc.rust-lang.org/nomicon/
```