Basics of the Rust language

For FOSS developers

Lars Wirzenius Consulting Ltd

Boot

🟦 While you're waiting for the class to start

Relax.

Breathe.

Don't worry.

It will be OK.

In memory of a programmer. v0.3.

In a corner of the cemetery,
on grass, beneath an old tree,
lies a tombstone, fallen,
covered by moss and leaf,
a name, two dates, five words,
a summary of a life of grief:
"how hard can it be?"

Boot

Goals of this training

- You can make sense of Rust code you read.
- You can learn more Rust on your own.

Method

We will be using mix of teaching / learning methods.

- Lecture.
- Group discussion.
- Hands-on practice.
- Active participation expected.

Discussion

Why are you interested in learning Rust?

Overview of Rust

🟦 A Rust "hello, world" program

To verify that you have a working Rust installation:

```
$ cargo init hello
$ cd hello
$ cargo check
$ cargo build
$ cargo run
$ cargo clippy
$ cargo doc --open
```

Overview of Rust

A Rust "hello, world" program

This the output of `cargo init hello`

`Cargo.toml`

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

[dependencies]
```

`src/main.rs`

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

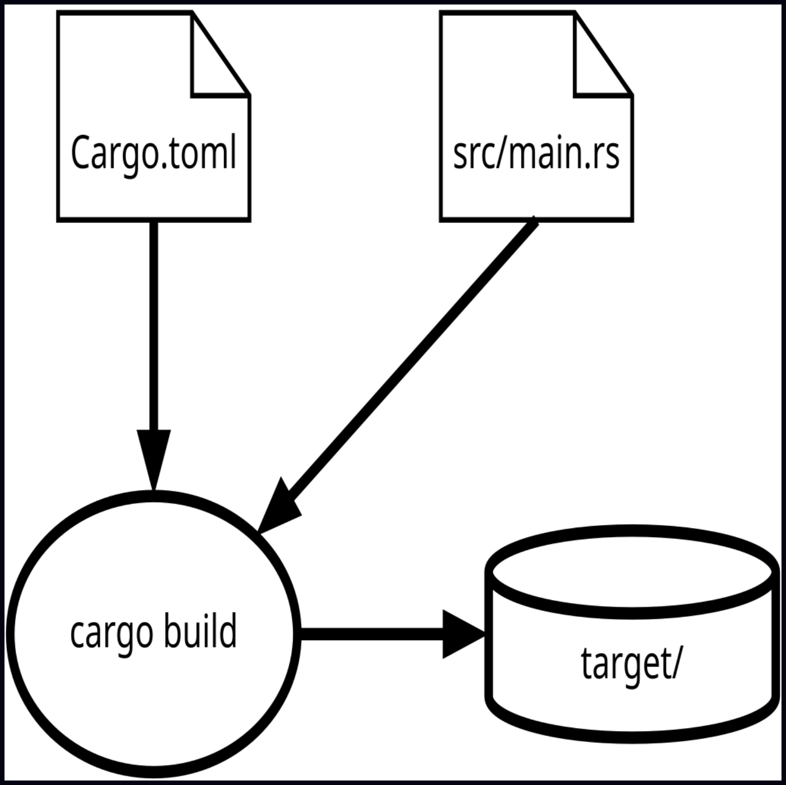
Overview of Rust

🟦 Cargo

- workflow tool
  - think `make` or `cmake` that is also `pip` or `apt-get`
- downloads dependencies
- builds code
- runs the built program
- runs tests
- runs benchmarks

Overview of Rust

- Cargo build process



Overview of Rust

Rust strengths

- memory safety
  - no memory leaks, no use-after-free, no dangling pointers, no NULL pointers
  - no data races, almost painless concurrency
- strong type system with inference
- `Result` and `Option` types
  - no runtime exceptions
  - no NULL values
- performance
  - zero cost abstractions, iterators
  - fast execution speed
  - control of how memory is used
- pretty good tooling
  - friendly compiler
  - strong support for IDEs, programmer's editors
  - fearless refactoring
- evolves carefully, rarely breaks working code

Overview of Rust

Rust weaknesses

- builds can be slow
- statically linked by default
    - binaries are large
- not very good for rapid prototyping: requires careful thought
- doesn't support as many target architectures as C does
- still young, keeps changing

Overview of Rust

Rust concepts

- automatic memory management, borrow checker, lifetimes
- enumerated types where variants can contain data
- traits, generics
- match on values, structure, unpacking
- crate
- edition

Note that Rust is not an object oriented language. It does not have classes or inheritance. Traits serve similar needs.

Overview of Rust

- Rust ecosystem
  - Rust Foundation
  - various teams: compiler, libraries, toolchain, ...
  - default central, public repository of crates: https://crates.io
    - as of 2025-09-20...
    - about 197 thousand crates
    - about 175 billion downloads
  - cultural bias against very small libraries
  - semantic versioning, heavily relied-on by cargo
  - heavy emphasis on being careful, not breaking things
  - heavy emphasis on being welcoming and constructive

Overview of Rust

Installing Rust

- Common and preferred: **rustup**
  - https://www.rust-lang.org/tools/install
  - bad: downloads code from the Internet, runs it
  - good: does its best to be safe and secure
  - good: easy to get latest version of Rust toolchain and tooling
- Packaged in Debian, other Linux distributions.
  - good: uses system package manager
  - bad: tends to lag behind Rust development
  - perfectly fine for learning
- Other implementations are starting to appear but are not ready for production use yet.

Overview of Rust

- Important sites

```
https://www.rust-lang.org/
https://crates.io/
https://docs.rs/
https://doc.rust-lang.org/std/

https://doc.rust-lang.org/book/
https://stevedonovan.github.io/rust-gentle-intro/
https://www.chiark.greenend.org.uk/~ianmdlvl/rust-polyglot/

https://blessed.rs/
https://serde.rs/
```

Enterprise hello

🟦 Plan

We will develop an enterprise grade version of the "hello, world" program that `cargo init` produces. We will do this in several steps:

1. ☑ the first, simplistic, version from `cargo init`
2. □ allow the user to specify who gets greeted on the command line
   - add command line parsing using the `clap` library
   - with default value
3. □ read who is greeted from a file
   - reading a file while handling errors

Enterprise hello

🟦 Whom should we greet? The `Cargo.toml` file

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

[dependencies]
clap = { version = "4.0.2", features = ["derive"] }
```

To add a dependency using a tool:

```
$ cargo add clap --features derive
```

Enterprise hello

- Whom should we greet? The `src/main.rs` file

```rust
use clap::Parser;

#[derive(Parser)]
struct Args {
    #[clap(default_value = "world")]
    whom: String,
}

fn main() {
    let args = Args::parse();
    println!("hello, {}", args.whom);
}
```

Enterprise hello

- Whom should we greet? The demo

```
$ cargo run -q
Hello, world!
$ cargo run -q -- there
Hello, there!
$
```

Enterprise hello

- Read name from file: the `Cargo.toml` file

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

[dependencies]
clap = { version = "4.0.2", features = ["derive"] }
thiserror = "1.0.37"
```

Enterprise hello

- Read name from file: command line arguments

```rust
use clap::Parser;
use std::fs::read;
use std::path::{Path, PathBuf};

#[derive(Parser)]
struct Args {
    #[clap(default_value = "world")]
    whom: String,

    #[clap(short, long)]
    filename: Option<PathBuf>,
}
```

Enterprise hello

- Read name from file: error codes

```
#[derive(Debug, thiserror::Error)]
enum HelloError {
    #[error("failed to read file {0}")]
    Read(PathBuf, #[source] std::io::Error),

    #[error("failed to parse file {0} as UTF-8")]
    Utf8(PathBuf, #[source] std::string::FromUtf8Error),
}
```

Enterprise hello

- Read name from file: read name from file

```rust
impl Args {
  fn whom(&self) -> Result<String, HelloError> {
    if let Some(filename) = &self.filename {
      let whom = Self::read(filename)?;
      Ok(whom)
    } else {
      Ok(self.whom.clone())
    }
  }
  fn read(filename: &Path) -> Result<String, HelloError> {
    let data = read(filename)
      .map_err(|e| HelloError::Read(filename.into(), e))?;
    let whom = String::from_utf8(data)
      .map_err(|e| HelloError::Utf8(filename.into(), e))?;
    Ok(whom.trim().to_string())
  }
}
```

Enterprise hello

🟦 Aside: self in Rust

Method arguments:

- Reference to the value that owns the method
  - `fn foo(&self) { ... }`
- Mutable reference
  - `fn foo(&mut self) { ... }`
- Transfer ownership of value to the method
  - `fn foo(self) { ... }`

Alias for the type being implemented:

```
impl Foo {
    fn new() -> Self {
        Self { ... }
    }
}
```

Enterprise hello

- Read name from file: main program

```
use std::error::Error;

fn main() {
    if let Err(e) = fallible_main() {
        eprintln!("ERROR: {}", e);
        let mut err = e.source();
        while let Some(underlying) = err {
            eprintln!("caused by: {}", underlying);
            err = underlying.source();
        }
    }
}

fn fallible_main() -> Result<(), HelloError> {
    let args = Args::parse();
    println!("hello, {}", args.whom()?);
    Ok(())
}
```

Enterprise hello

- Read name from file: the demo

```
$ cargo run -q
Hello, world!
$ cargo run -q -- there
Hello, there!
$ echo Earth > name.txt
$ cargo run -q -- -f name.txt
Hello, Earth!
$ cargo run -q -- -f who-me.txt
ERROR: failed to read file who-me.txt
caused by: No such file or directory (os error 2)
$
```

Enterprise hello

🟦 Hands-on

Install a Rust program from crates.io and try it.

```
$ cargo install ripgrep
```

Common command line tools you may enjoy:

- ripgrep---a fast, versatile "grep"
- bat---a "less" with colors
- starship---a fancy shell prompt

Or you can find something else.

Strings

There can't be only one string type

- arbitrary binary data: `Vec<u8>`
  - sub-vector or slice: `&[u8]`
  - binary string literal `b"hello"`
- human-oriented text as UTF8: `String`
  - `String::from("hello, world")`
  - string literal: "hello, world", type `&str`
  - slice: `&str`, `&s`, `&s[10..20]` or `s.as_str()`
- file names: `std::path::PathBuf`
  - `PathBuf::from("file.txt")`
  - slice: `std::path::Path`
- native text for operating system: `std::ffi::OsString`
  - slice: `std::ffi::OsStr`
  - command line arguments, environment variables, ...

Strings

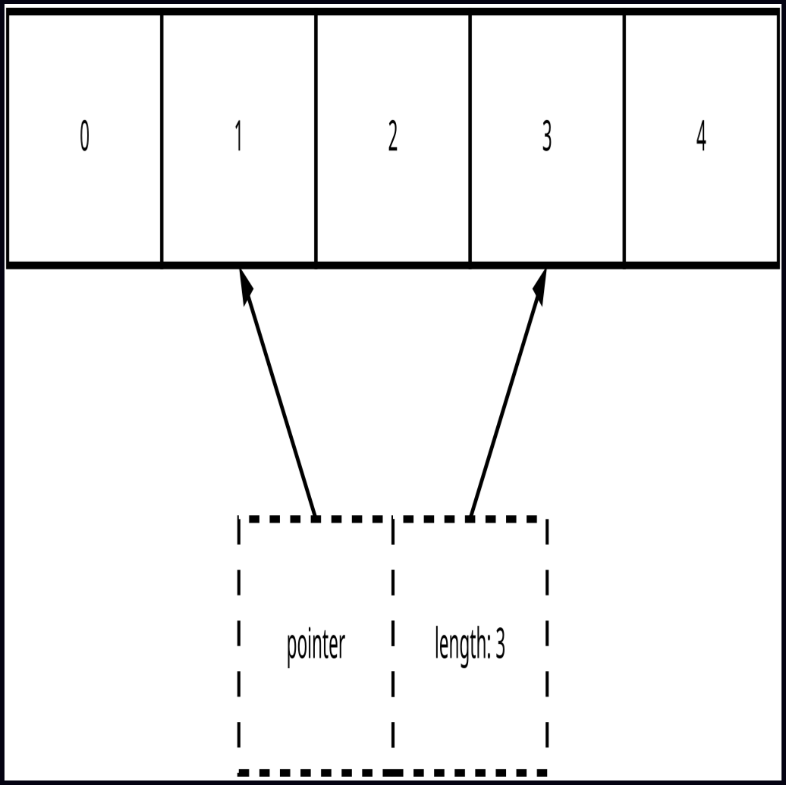
- Vectors, slices

```rust
let mut v = vec![]; // v is Vec<i32>
v.push(97);
v.push(98);
v.push(99);
println!("v1: {:?}", v);
// v1: [97, 98, 99]

let v2 = vec![97, 98, 99]; // v2 is Vec<i32>
println!("v2: {:?}", v2);
// v2: [97, 98, 99]

let v3 = &v2[1..]; // v3 is &[i32]
println!("v3: {:?}", v3);
// v3: [98, 99]
```

Strings

Vectors, slices (2)

- A vector `Vec` allocates memory for the values.
- A slices references values stored elsewhere.



Strings

- Strings, filenames

```rust
let s = String::from("hello, world");
println!("s: {:?}", s);
println!("s: {}", s);
// s: "hello, world"
// s: hello, world

let bytes: Vec<u8> = vec![97, 98, 99];
let s2 = String::from_utf8_lossy(&bytes);
println!("s2: {:?}", s2);
// s2: "abc"

let filename = PathBuf::from("README.md");
println!("filename: {:?}", filename);
println!("filename: {}", filename.display());
// filename: "README.md"
// filename: README.md
```

Strings

Native strings for the operating system

- command line arguments
- environment variable names and values
- raw filenames
- ...

As Rust types:

- `std::ffi::OsString`
- `std::ffi::OsStr`

Strings

🟦 Hands-on: byte counting

Using the enterprise version of "hello, world" as an example, write this program:

- user gives filenames on the command line
- iterate over all the filenames
- read each file
- count number of bytes in each file
- for each file, output the filename and number of bytes
- at the end, output the total number of bytes in all files

For extra credit, if you spare time, count number of lines instead of bytes.

`https://codeberg.org/liw-rust-training/enterprise-hello.git`

Generics

Generic types

- This is advanced, but it's used very commonly in Rust, so you need to understand it.
- Types that contain values of some type, or functions that act on values of a type, but don't mind what the actual type is.
  - The contained type is expressed using a *type variable*.
- There can be some constraints on the contained type.
  - it might need to have a size known at the compile time
- A vector is a container of values of some type T: `Vec<T>`
  - `Vec` needs to know how large values of type T are
  - `Vec` doesn't do anything with the values, just stores them
  - `Vec` is **generic for type T**

Generics

■ The `Option` type

- An `Option` either contains a value of a specific type, or doesn't
  - implemented using an `enum`
- Always use `Option` if a value might be there or not be there. There is no "NULL pointer" or "nil reference" or "zero value".
  - the compiler understands the `Option` type and can help you get your code correct; it doesn't understand that, say, an empty string is special
  - you **can't** get the contained type without checking that the value exists

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Generics

- Unpacking an `Option` value: pattern matching

```rust
fn flaunt(it: Option<i32>) {
    if let Some(value) = it {
        println!("{}", value);
    }
}

fn flaunt2(it: Option<i32>) {
    match it {
        Some(value) => println!("{}", value);
        _ => (),
    }
}
```

Generics

The `Result` type

- A *fallible* operation returns a result—the operation either succeeded or failed
- If successful, return a useful value of some type T,   otherwise return an error value of some type E
- Not a special magic value of the return type to indicate an error—is -1 a valid integer or does it indicate an error?
- The compiler warns if results are not used
  - this is not an error by default, but you can make it be one—the compiler is relentless and forces you to use a result
  - you can ignore the result if you can't be bothered to do something about errors, but you have to be explicit about it

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

Generics

- Container: for any type T (using Vec)

```rust
#[derive(Debug, Default)]
struct Container<T> {
    values: Vec<T>,
}

impl<T> Container<T> {
    fn len(&self) -> usize {
        self.values.len()
    }

    fn is_empty(&self) -> bool {
        self.values.is_empty()
    }
}
```

Generics

- Container: constrained by a trait

```rust
impl<T: Eq> Container<T> {
    fn find(&self, v: &T) -> Option<usize> {
        for (i, x) in self.values.iter().enumerate() {
            if x == v {
                return Some(i);
            }
        }
        None
    }
}
```

Generics

Hands-on: generic stack

Implement a simple stack of value of any type. The following code must work with your stack.

```
let mut stack = Stack::new();
stack.push(3);
stack.push(2);
stack.push(1);
while !stack.is_empty() {
    println!("{}", stack.pop().unwrap());
}
```

Hint: Look up the `Vec` type methods `push` and `pop` methods in the standard library documentation: `https://doc.rust-lang.org/std`

Generics

Homework (for later)

Skim the documentation and code for the `Option` type and the `Iterator` trait in the standard library.

What's the most interesting method for you?

Iterators

You can implement your own iterator

- `for` loops and similar constructs want iterators
  - anything that implements the `Iterator` trait – OR the `IntoIterator` trait
- You can implement those traits for your own types.

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Iterators

Items returned by iterators

- Some iterators return a reference to a value
  - `type Item = &Bar;`
- Others return the actual items
  - `type Item = Bar;`
  - this moves ownership if an implicit copy can't be made
- `for bar in foo`
  - `foo` must be implement `Iterator` or `IntoIterator`
  - sometimes this ends up being an iterator that returns items
  - this can lead to problems of ownership
- it can be clearer to always create an iterator explicitly: there is often a method `iter` for collection types, such as vectors
  - `for i in vec.iter()`

Iterators

Sequence of integers: mission statement

Produce a sequence of increasing integers from a starting value until a goal. Don't include the goal.

```rust
struct Seq {
    goal: i32,
    next: i32,
}
```

Iterators

- Sequence of integers: using sequence

```rust
fn main() {
    // 0, 1, 2, etc, through to 9, but not including 10
    for i in Seq::new(10) {
        print!("{} ", i);
    }
    println!();

    // -10, -9, etc, through to 9, but not including 10
    for i in Seq::range(-10, 10) {
        print!("{} ", i);
    }
    println!();
}
```

Iterators

- Sequence of integers: constructors

```
impl Seq {
    fn new(goal: i32) -> Self {
        Self {
            goal,
            next: 0,
        }
    }

    fn range(start: i32, goal: i32) -> Self {
        Self {
            goal,
            next: start,
        }
    }
}
```

Iterators

▓▓▓ Sequence of integers: iterator

```rust
impl Iterator for Seq {
    type Item = i32;
    fn next(&mut self) -> Option<Self::Item> {
        if self.next < self.goal {
            let item = Some(self.next);
            self.next += 1;
            item
        } else {
            None
        }
    }
}
```

Modules

🟦 Any Rust source file may contain a module

```rust
fn main() {
    println!("random number is {}", foo::random());
}

mod foo {
    pub fn random() -> usize {
        42
    }
}
```

- `pub` is necessary for any symbol exported from a module   even for "local" modules
- Often used for unit tests.
- Also useful for name space control.

Modules

- A Rust source file is a module

File `src/foo.rs`

```rust
pub fn random() -> usize {
    42
}
```

File `src/main.rs`

```rust
mod foo;

fn main() {
    println!("random number is {}", foo::random());
}
```

Modules

- The `lib.rs` module is special

File `src/lib.rs`

```
pub fn random() -> usize {
    42
}
```

File `src/main.rs`

```
use foocrate::random;

fn main() {
    println!("random number is {}", random());
}
```

Modules

Dark mysterious secrets of the ancient world

There's more to modules in Rust, but this will get you started

Memory

Why?

Computer	year	RAM (KiB)
PDP-7	1965	9.2 KiB
Commodore 64	1982	64 KiB
Cray X-MP	1982	128 MiB
Linus' first PC	1991	4 MiB
Nokia X10	2021	6 GiB

- Static allocation: at compile time; wasteful.
- Dynamic memory allocation.
    - fit more into less
    - get more bang for your buck
    - waste not, want not
    - simple idea, but hard to get right

Memory

Manual memory management

Example: C

Promise:

> I'll give you the simplest possible tools to manage memory dynamically. You will make mistakes and they'll be hard to debug. They'll also be security problems.

Motto:

> "Suffering builds character"

Memory

Garbage collection

Examples: LISP, Python, Ruby, Go, Java, ...

Promise:

> I'll free memory you're not using anymore. You don't need to do anything special, but your programs will sometimes stall briefly at run time.

Motto:

> "Things will usually... wait for it... work."

Memory

🟦 Automatic based on ownership

Example: Rust

Promise:

> I will give you simple rules to follow that I can check at compile
> time. I will know at compile time when memory needs to be allocated
> and when it can be freed. I will tell you if you make a mistake,
> and I will try to suggest how to fix it.

Motto:

> "Prove to me you manage memory correctly."

Memory

- Allocating memory

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };
let farfaraway = Point { x: 32000, y: -32000 };
```

Memory

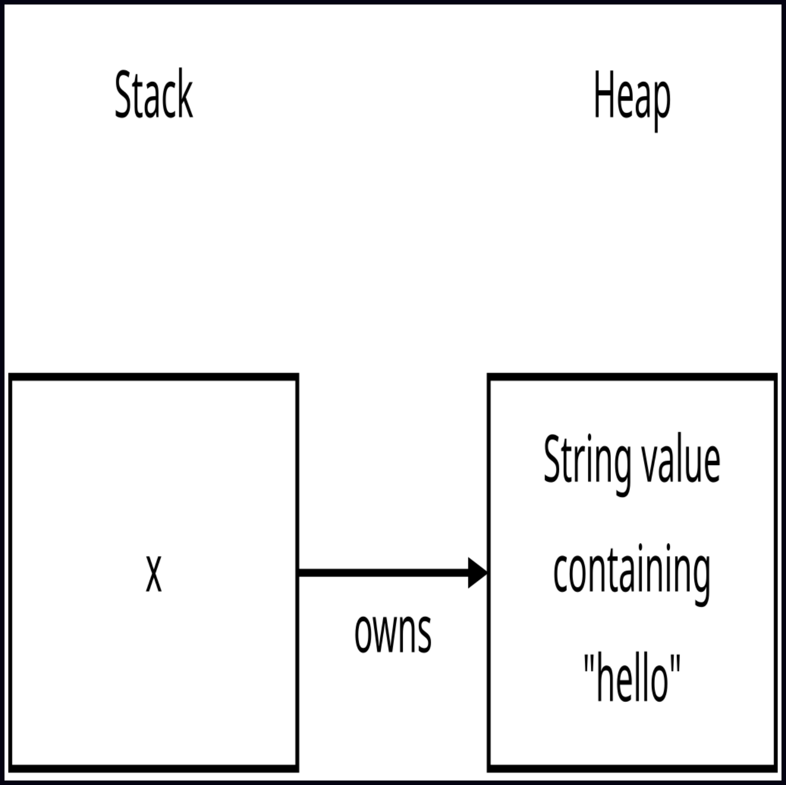
Ownership, freeing memory

- Every value is stored in memory
  - local variables on the stack, dynamic memory on the heap
- Each value has exactly one **owner**
  - There can only be one owner at a time
- When the owner goes **out of scope**, the value will be freed
  - "dropped"

```
{
    let x = String::from("hello"); // allocate on heap
    // value on heap exists
}
// value on heap no longer exists
```

Memory

- Values on stack may own values on the heap



Memory

Keeping track of ownership

- Easy: value is allocated on stack.
  - compiler knows what it removes things from the stack
- Hard: value is allocated on heap.
  - owner may be on stack, owns value on heap
  - owner may be another value on the heap
  - compiler does this for you
- Ownership can be **moved** → same value, new owner.
  - compiler keeps track, no code is generated
  - e.g. value is returned from function
- Values can be **copied** or **cloned** → new value, new owner.
  - executed at run time
  - `Copy` trait → copy the bits of the value, e.g., integers
  - `Clone` trait → construct new value that is semantically equal

Memory

Borrowing

Borrow = get a reference to a value.

1. At any given time, you can have **either** one mutable reference **or** any number of immutable references.
2. References must always be valid.

This prevents:

- Race conditions when data is changed.
- Using memory before it's been allocated or after it's been freed.
- NULL pointers.

Doesn't prevent:

- Other race conditions.
- Deadlocks.
- Live locks.

Memory

- Mutability and borrowing

```rust
let a = String::new(); // immutable a and b
let b = &a;

let mut x = String::new(); // mutable x
x.push_str("hello");
let y = &mut x;
y.push_str(", world"); // modify contents of x
println!("y={y:?}"); // OK: we don't use y after this!

let mut z = &x; // immutable reference to x
println!("first z={z:?}");
z = &a; // change what z refers to

println!("a={a:?} b={b:?} z={z:?}");
```

Memory

- Mutability and borrowing, output

```
$ cargo run -q
y="hello, world"
first z="hello, world"
a="" b="" z=""
```

Memory

- Lifetime example

```rust
fn main() {
    let mut refs: Vec<&String> = vec![];
    {
        let x = String::from("hello");
        refs.push(&x);
    }
    for s in refs {
        println!("{}", s);
    }
}
```

Memory

- Borrow checker error message

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:19
  |
5 |   refs.push(&x);
  |             ^^ borrowed value does not live long enough
6 | }
  | - `x` dropped here while still borrowed
7 | for s in refs {
  |          ---- borrow later used here

For more information about this error, try
`rustc --explain E0597`.
```

Memory

🟦 Hands-on: generic key/value container

- Create a generic key/value container type.
  - any key and value type, as long as keys can be compared
- Method to insert a key and value.
  - if key already in container, replace previous value with new
- Method to retrieve value.

Memory

- Hands-on: key/value container interface

```rust
struct Container<K: Eq, V> { values: Vec<(K, V)> }
impl<K: Eq, V> Container<K, V> {
    fn new() -> Self { /* FIXME */ }
    fn insert(&mut self, k: K, v: V) { /* FIXME */ }
    fn get(&self, k: &K) -> Option<&V> { /* FIXME */ }
}
fn main() {
    let alice = "alice".to_string();
    let bob = "bob".to_string();
    let robert = "Robert".to_string();
    let mut cont = Container::new();
    cont.insert(bob.clone(), bob.clone());
    cont.insert(bob.clone(), robert);
    println!("{} -> {:?}", &alice, cont.get(&alice));
    println!("{} -> {:?}", &bob, cont.get(&bob));
    // Output should be Robert
}
```

Memory

🟦 Homework (for later)

Read the documentation for the container types provided by the standard library:

https://doc.rust-lang.org/std/collections/index.html

- Can you find use for them in your own programs?
- What else would you like to have? Can you find that on crates.io?

Concurrency

Why?

Computer	year	price	cores
Cray X-MP	1982	$15 million	4
Rasp Pi 3B	2016	$50	4
Nokia 6.1	2018	$200	8

- CPU cores aren't getting significantly faster anymore
- Even cheap CPUs now have more than one core or hyperthread
- To get results faster, compute more things at the same time
- Traditionally **really hard** to get right

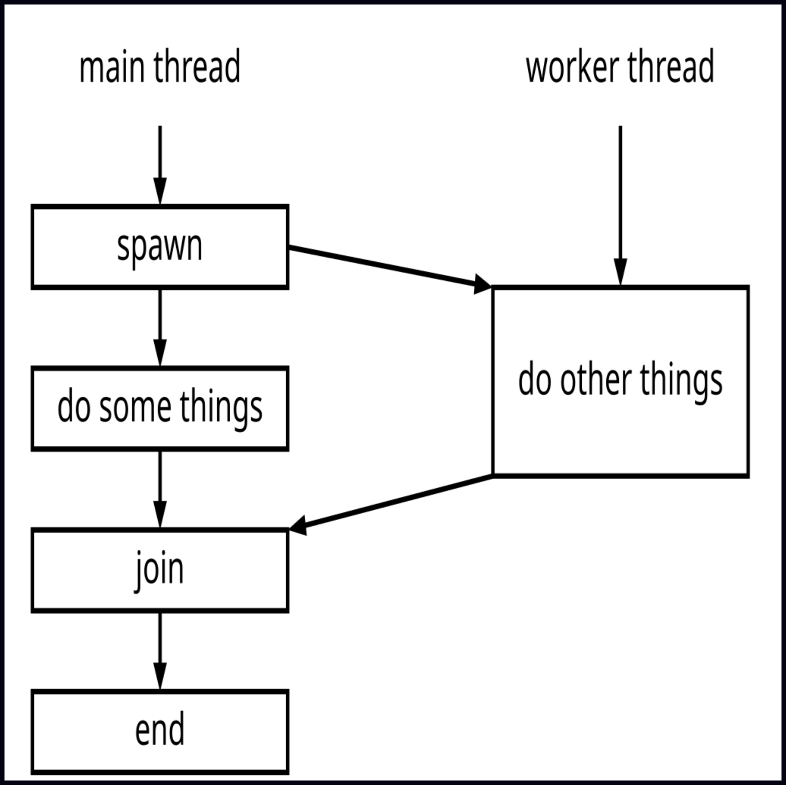
Concurrency

Overview

- Fearless concurrency.
  - safety rules apply: no data races → you *must* use locking if anything mutates
- Threads.
  - pre-emptive
  - map well into operating system threads
  - mature, well supported, part of `std`
  - good choice for CPU intensive applications
- `async` / `await`
  - collaborative
  - fairly new, maturing fast
  - needs additional crates, e.g., `tokio`
  - good choice for I/O intensive applications

Concurrency

- Threads, conceptually



Concurrency

- Threads, as code

```rust
use std::thread::spawn;

let mut handles = vec![];
for filename in args.filenames {
    let handle = spawn(move || checksum(&filename));
    handles.push(handle);
}

for handle in handles {
    let sumresult = handle.join().expect("thread join");
    let sum = sumresult?;
    sum.print();
}
```

Concurrency

Hands on: Concurrent file checksums

- `https://codeberg.org/liw-rust-training`
    - repository `/checksums-hands-on.git`
- Open that page, clone the repository, read the README.
- You may ask questions.
- This slide will not self-destruct in five seconds.
- Complete your mission.

Concurrency

Async: conceptually

- Operating system threads tend to be "heavy"
    - RAM, context switches
    - thread runs until it blocks, or its time slot ends
    - careful management of inter-thread communication
- Co-operative multi-tasking can be light-weight
    - task runs alone in its thread until it blocks
    - almost like writing sequential code
    - little RAM, no extra task switches
    - enormous numbers of tasks is feasible
- `async` is provided by many languages: JS, Python, Rust, ...
    - `async fn` → return *promise* of a value existing in the future
    - `await` on a promise returns when value is computed
    - a *runtime* executes futures to compute actual values

Concurrency

Async: the Rust story

- `rustc` implements the `async` and `await` syntax and related semantics.
- `std` implements futures, and other necessary types for using async.
- Crates provide run-times (executors):
    - `tokio`
    - `async-std`
    - `smol`
    - ...
    - vary by maturity, functionality, size, intended use, etc
    - you can write your own

Concurrency

- Async: example (1/2)

```
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let args = Args::parse();
    let mut tasks = vec![];
    let client = reqwest::Client::builder()
        .danger_accept_invalid_certs(true)
        .build()?;
    for _ in 0..args.n {
        let url = args.url.clone();
        let client = client.clone();
        let x = tokio::spawn(
            async move { client.get(&url).send().await });
        tasks.push(x);
    }
    println!("Created {} tasks", args.n);
```

Concurrency

- Async: example (2/2)

```
    for task in tasks {
        let result = task.await?;
        let response = result?;
        if !response.status().is_success() {
            println!("{:?} {}", args.url,
                     response.status());
        }
    }
    println!("All went OK");
    Ok(())
}
```

Concurrency

Hands-on: Concurrent HTTP requests

- `https://codeberg.org/liw-rust-training/get.git`
- Make sure you can get that code to work.
    - be kind: don't hit on a public site hard, at most 100 repetitions
- Then change the code so it's given only URLs on the command line, and fetches each concurrently, and prints the status code for each URL at the end.

End

Advice for writing Rust, at first

- Use `clone` liberally, if the borrow checker gets in the way.
  - it's wasteful, but OK when learning
- Use `cargo fmt` and `cargo clippy` frequently.
- `anyhow` is easy, but use `thiserror` for better error messages.
- Learn to use and implement traits.
- Take small steps. No, much smaller than that.

End

🟦 Advice for writing Rust, at first

▓▓▓ Now what?

- Write code.
- Read std docs.
- Read docs for crates.
- Read code.
- Join community fora.
- Start or join an internal group.

End

FIN

No, really.

It's over.

I hope you enjoyed it.

If you want to, I would appreciate a public review of this training, on your blog or social media.