Rust programming

Module B: Application programming

Last time...

- Generic code
- Traits
- Trait generics & associated types
- Lifetime annotations

Any questions?

In this module

Learn how to use Rust for writing high quality applications

Learning objectives

- Work with `crate` dependencies
- Create your own crate with a nice API
- Test and benchmark your code

During tutorial:

- Divide your code into logical parts with modules
- Use common crates
- Set up your own Rust application and library

Module B

Application programming

Content overview

- Working with `crate`s
- API guidelines
- Testing and benchmarking

Creating Rust projects

Cargo

Most daily usage of Rust will involve using cargo in one way or another.

Some of the more common tasks are:

- Creating new projects
- Managing dependencies
- Building projects
- Executing the resulting binaries
- Running tests and benchmarks
- Generating and viewing local documentation

Cargo configuration

Cargo is managed through the `Cargo.toml` configuration file. Toml is an easy to read configuration file fairly similar to ini files.

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

[dependencies]
serde = "1.0"
```

Adding dependencies

You can add dependencies to `Cargo.toml` in multiple ways

Add a line in cargo.toml

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

[dependencies]
serde = "1.0"
itertools = "0.10"
```

Use `cargo add`

```
cargo add itertools
```

Dependencies? Crates!

The crate is the compilation unit for Rust

- Binary crates:
  - Result in a compiled binary program that you can execute.
  - Binaries have a `main` function as entrypoint of the program
- Library crates:
  - define functionality that can be used by other crates.
  - No specific `main` function

Each crate in Rust has a root file. For binary crates this typically is `main.rs`, but for libraries this typically is `lib.rs`.

Using a `crate`

Crates included in `Cargo.toml` can be:

- imported with a `use`
- qualified directly using path separator `::`

```
// Import an item from this crate, called `my_first_app`
use my_first_app::add;
// Import an item from the `tracing` dependency
use tracing::info;

fn main() {
    // Use qualified path
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::DEBUG)
        .init();

    let x = 4;
    let y = 6;

    // Use imported items
    let z = add(x, y);
    info!("Let me just add {x} and {y}: {z}")
}
```

Other dependency sources

- Local
- Git

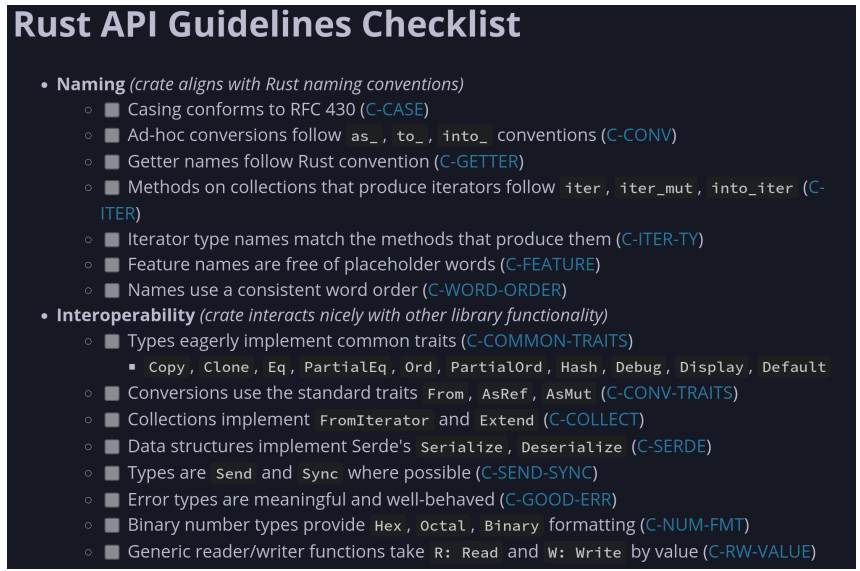
```
$ cat Cargo.toml
# -snip-
[dependencies]
my_local_dependency = { path = "../path/to/my_local_dependency" }
my_git_dependency = { git = "<Git SSH or HTTPS url>", rev="<commit hash or tag>", branch = "<branch>" }
```

- Private crate registries are WIP

Creating a nice API

Rust API guidelines

- Defined by Rust project
- Checklist available (Link in exercises)



Rust API Guidelines Checklist

- **Naming** *(crate aligns with Rust naming conventions)*
  - ☐ Casing conforms to RFC 430 (C-CASE)
  - ☐ Ad-hoc conversions follow `as_`, `to_`, `into_` conventions (C-CONV)
  - ☐ Getter names follow Rust convention (C-GETTER)
  - ☐ Methods on collections that produce iterators follow `iter`, `iter_mut`, `into_iter` (C-ITER)
  - ☐ Iterator type names match the methods that produce them (C-ITER-TY)
  - ☐ Feature names are free of placeholder words (C-FEATURE)
  - ☐ Names use a consistent word order (C-WORD-ORDER)
- **Interoperability** *(crate interacts nicely with other library functionality)*
  - ☐ Types eagerly implement common traits (C-COMMON-TRAITS)
    - `Copy`, `Clone`, `Eq`, `PartialEq`, `Ord`, `PartialOrd`, `Hash`, `Debug`, `Display`, `Default`
  - ☐ Conversions use the standard traits `From`, `AsRef`, `AsMut` (C-CONV-TRAITS)
  - ☐ Collections implement `FromIterator` and `Extend` (C-COLLECT)
  - ☐ Data structures implement Serde's `Serialize`, `Deserialize` (C-SERDE)
  - ☐ Types are `Send` and `Sync` where possible (C-SEND-SYNC)
  - ☐ Error types are meaningful and well-behaved (C-GOOD-ERR)
  - ☐ Binary number types provide `Hex`, `Octal`, `Binary` formatting (C-NUM-FMT)
  - ☐ Generic reader/writer functions take `R: Read` and `W: Write` by value (C-RW-VALUE)

Read the checklist, use it!

General recommendations

Make your API

- Unsurprising
- Flexible
- Obvious

Next up: Some low-hanging fruits

Make your API

Unsurprising

Naming your methods

```
pub struct S {
    first: First,
    second: Second,
}

impl S {
    // Not get_first.
    pub fn first(&self) -> &First {
        &self.first
    }

    // Not get_first_mut, get_mut_first, or mut_first.
    pub fn first_mut(&mut self) -> &mut First {
        &mut self.first
    }
}
```

Other example: conversion methods `as_`, `to_`, `into_`, name depends on:

- Runtime cost
- Owned ↔ borrowed

Implement/derive common traits

As long as it makes sense public types should implement:

- `Copy`
- `Clone`
- `Eq`
- `PartialEq`
- `Ord`
- `PartialOrd`
- `Hash`
- `Debug`
- `Display`
- `Default`
- `serde::Serialize`
- `serde::Deserialize`

Make your API

Flexible

Use generics

```
pub fn add(x: u32, y: u32) -> u32 {
    x + y
}

/// Adds two values that implement the `Add` trait,
/// returning the specified output
pub fn add_generic<O, T: std::ops::Add<Output = O>>(x: T, y: T) -> O {
    x + y
}
```

Accept borrowed data if possible

- User decides whether calling function should own the data
- Avoids unnecessary moves
- Exception: non-big array `Copy` types

```rust
/// Some very large struct
pub struct LargeStruct {
    data: [u8; 4096],
}

/// Takes owned [LargeStruct] and returns it when done
pub fn manipulate_large_struct(mut large: LargeStruct) -> LargeStruct {
    todo!()
}

/// Just borrows [LargeStruct]
pub fn manipulate_large_struct_borrowed(large: &mut LargeStruct) {
    todo!()
}
```

Make your API

Obvious

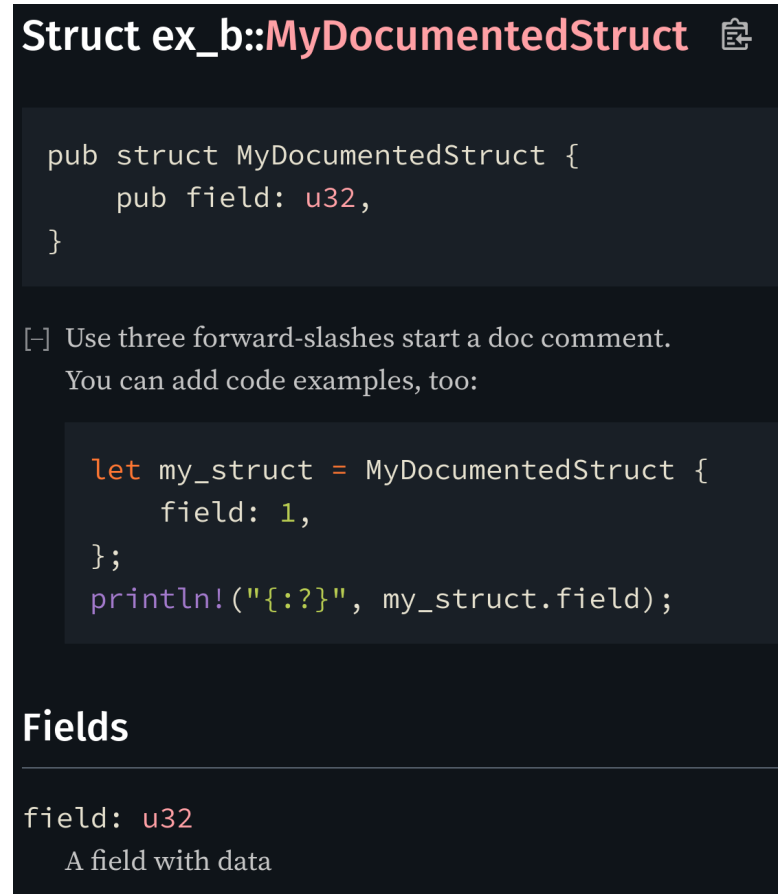
Write Rustdoc

- Use 3 forward-slashes to start a doc comment
- You can add code examples, too

```rust
/// A well-documented struct.
/// ```rust
/// # // lines starting with a `#` are hidden
/// # use ex_b::MyDocumentedStruct;
/// let my_struct = MyDocumentedStruct {
///     field: 1,
/// };
/// println!("{:?}", my_struct.field);
/// ```
pub struct MyDocumentedStruct {
    /// A field with data
    pub field: u32,
}
```

To open docs in your browser:

```
$ cargo doc --open
```



Include examples

Create examples to show users how to use your library

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── examples
│   └── say_hello.rs
└── src
    └── lib.rs
$ cargo run --example say_hello
   Compiling my_app v0.1.0 (/home/henkdieter/tg/edu/my_app)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/examples/say_hello`
Hello, henkdieter!
```

Use semantic typing (1)

Make the type system work for you!

```rust
/// Fetch a page from passed URL
fn load_page(url: &str) -> String {
    todo!("Fetch");
}

fn main() {
  let page = load_page("https://101-rs.tweede.golf");
  let crab = load_page("🦀"); // Ouch!
}
```

`&str` is not restrictive enough: not all `&str` represent correct URLs

Use semantic typing (2)

```rust
struct Url<'u> {
  url: &'u str,
}

impl<'u> Url<'u> {
  fn new(url: &'u str) -> Self {
    if !valid(url) {
      panic!("URL invalid: {}", url);
    }
    Self { url }
  }
}

fn load_page(remote: Url) -> String {
    todo!("load it");
}

fn main() {
    let content = load_page(Url::new("□ ")); // Not good
}

fn valid(url: &str) -> bool {
    url != "□ " // Far from complete
}
```

```
    Compiling playground v0.0.1 (/playground)
     Finished dev [unoptimized + debuginfo] target(s) in 2.90s
      Running `target/debug/playground`
thread 'main' panicked at 'URL invalid: □ ', src/main.rs:11:7
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- Clear intent
- Input validation: security!

Use the `url` crate

Use Clippy and Rustfmt for all your projects!

```
$ cargo clippy
$ cargo fmt
```

Testing your crate

Testing methods

- Testing for correctness
  - Unit tests
  - Integration tests
- Testing for performance
  - Benchmarks

Unit tests

- Tests a single function or method
- Live in child module
- Can test private code

To run:

```
$ cargo test
[...]
running 2 tests
test tests::test_swap_items ... ok
test tests::test_swap_oob - should panic ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
[..]
```

*Rust compiles your test code into binary using a **test harness** that itself has a CLI*:

```
# Don't capture stdout while running tests
$ cargo test -- --nocapture
```

```rust
/// Swaps two values at the `first` and `second` indices of the slice
fn slice_swap_items(slice: &mut [u32], first: usize, second: usize) {
    let tmp = slice[second];
    slice[second] = slice[first];
    slice[first] = tmp;
}

/// This module is only compiled in `test` configuration
#[cfg(test)]
mod tests {
    use crate::slice_swap_items;

    // Mark function as test
    #[test]
    fn test_swap_items() {
        let mut array = [0, 1, 2, 3, 4, 5];
        slice_swap_items(&mut array[..], 1, 4);
        assert_eq!(array, [0, 4, 2, 3, 1, 5]);
    }

    #[test]
    // This should panic
    #[should_panic]
    fn test_swap_oob() {
        let mut array = [0, 1, 2, 3, 4, 5];
        slice_swap_items(&mut array[..], 1, 6);
    }
}
```

Integration tests

- Tests crate public API
- Run with `cargo test`
- Defined in `tests` folder:

```
$ tree
.
├── Cargo.toml
├── examples
│   └── my_example.rs
├── src
│   ├── another_mod
│   │   └── mod.rs
│   ├── bin
│   │   └── my_app.rs
│   ├── lib.rs
│   ├── main.rs
│   └── some_mod.rs
└── tests
    └── integration_test.rs
```

Tests in your documentation

You can even use examples in your documentation as tests

```
/// Calculates fibonacci number n
///
/// # Examples
///
/// ```
/// # use example::fib;
/// assert_eq!(fib(2), 1);
/// assert_eq!(fib(5), 5);
/// assert_eq!(fib(55), 55);
/// ```
pub fn fib(n: u64) -> u64 {
    if n <= 1 {
        n
    } else {
        fib(n - 1) + fib(n - 2)
    }
}
```

```
cargo test --doc
```

Benchmarks

- Test *performance* of code (vs. correctness)
- Runs a tests many times, yield average execution time

*Good benchmarking is **Hard***

- Beware of optimizations
- Beware of initialization overhead
- Be sure your benchmark is representative

More in exercises

Summary

- Set up your own Rust application and library
  - Using `cargo new`
- Divide your code into logical parts with modules
  - Modules
  - Workspaces
- Create a nice API
  - Unsurprising, Flexible, Obvious
  - API guidelines
- Test and benchmark your code
  - Unit tests, integration tests, benchmarks

Tutorial time!

- Exercises A3 recap
- Exercises B in 101-rs.tweede.golf
- Live code on ex B1 and first part of B2

Don't forget to `` `git pull` ``!

END