Rust programming

Module 3: Crate Engineering

Unit 1

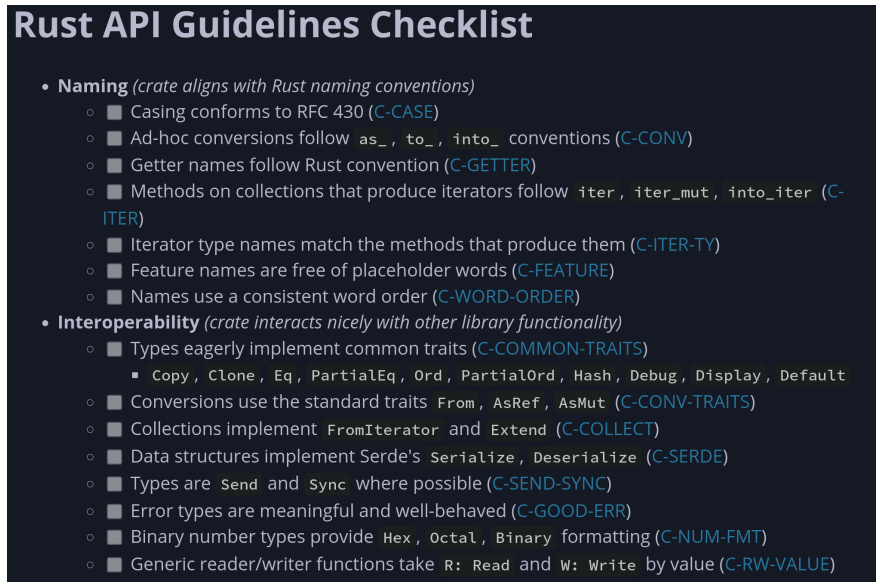
Crate Engineering

Learning objectives

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!

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

fn main() {
  let page = load_page("https://teach-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 go
}

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

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s)
     Running `target/debug/playground`
thread 'main' panicked at 'URL invalid: 🦀', src/mai
note: run with `RUST_BACKTRACE=1` environment variab
```

- Clear intent
- Input validation: security!

Use the `url` crate

Use Clippy and Rustfmt for all your projects!

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

Design patterns in Rust

Why learn design patterns?

- Common problems call for common, tried an tested solutions
- Make crate architecture more clear
- Speed up development
- Rust does some patterns ever-so-slightly differently

Learning common Rust patterns makes understanding new code easier

What we'll do

```
const PATTERNS: &[Pattern] = &[
    Pattern::new("Newtype"),
    Pattern::new("RAII with guards"),
    Pattern::new("Typestate"),
    Pattern::new("Strategy"),
];
fn main() {
    for pattern in PATTERNS {
        pattern.introduce();
        pattern.show_example();
        pattern.when_to_use();
    }
}
```

1. The Newtype pattern

a small but useful pattern

Newtype: introduction

Wrap an external type in a new local type

```
1   pub struct Imei(String)
```

That's it!

Newtype: example

```rust
pub enum ValidateImeiError { /* - snip - */}

pub struct Imei(String);

impl Imei {
    fn validate(imei: &str) -> Result<(), ValidateImeiError> {
        todo!();
    }
}

impl TryFrom<String> for Imei {
    type Error = ValidateImeiError;

    fn try_from(imei: String) -> Result<Self, Self::Error> {
        Self::validate(&imei)?;
        Ok(Self(imei))
    }
}

fn register_phone(imei: Imei, label: String) {
    // We can certain `imei` is valid here
}
```

Newtype: when to use

Newtype solves some problems:

- Orphan rule: no `impl`s for external `trait`s on external types
- Allow for semantic typing (`url` example from mod B)
- Enforce input validation

2. The RAII guard pattern

More robust resource handling

RAII Guards: introduction

- Resource Acquisition Is Initialization (?)
- Link acquiring/releasing a resource to the lifetime of a variable
- Guard constructor initializes resource, destructor frees it
- Access resource through the guard

Do you know of an example?

RAII Guards: example

```rust
pub struct Transaction<'c> {
    connection: &'c mut Connection,
    did_commit: bool,
    id: usize,
}

impl<'c> Transaction<'c> {
    pub fn begin(connection: &'c mut Connection)
     -> Self {
        let id =
            connection.start_transaction();
        Self {
            did_commit: false,
            id,
            connection,
        }
    }

    pub fn query(&self sql: &str) { /* - snip - */}

    pub fn commit(self) {
        self.did_commit = true;
    }
}
```

```rust
impl Drop for Transaction<'_> {
    fn drop(&mut self) {
        if self.did_commit {
            self
                .connection
                .commit_transaction(self.id);

        } else {
            self
                .connection
                .rollback_transaction(self.id);
        }
    }
}
```

RAII Guards: when to use

- Ensure a resource is freed at some point
- Ensure invariants hold while guard lives

3. The Typestate pattern

Encode state in the type

Typestate: introduction

- Define uninitializable types for each state of your object

```
pub enum Ready {} // No variants, cannot be initialized
```

- Make your type generic over its state using `` `std::marker::PhantomData` ``
- Implement methods only for relevant states
- Methods that update state take owned `` `self` `` and return instance with new state

👻 *`` `PhantomData<T>` `` makes types act like they own a `` `T` ``, and takes no space*

Typestate: example

```rust
pub enum Idle {} // Nothing to do
pub enum ItemSelected {} // Item was selected
pub enum MoneyInserted {} // Money was inserted

pub struct CoffeeMachine<S> {
    _state: PhantomData<S>,
}

impl<CS> CoffeeMachine<CS> {
    /// Just update the state
    fn into_state<NS>(self) -> CoffeeMachine<NS> {
        CoffeeMachine {
            _state: PhantomData,
        }
    }
}

impl CoffeeMachine<Idle> {
    pub fn new() -> Self {
        Self {
            _state: PhantomData,
        }
    }
}
```

```rust
impl CoffeeMachine<Idle> {
    fn select_item(self, item: usize) -> CoffeeMachi
        println!("Selected item {item}");
        self.into_state()
    }
}

impl CoffeeMachine<ItemSelected> {
    fn insert_money(self) -> CoffeeMachine<MoneyInse
        println!("Money inserted!");
        self.into_state()
    }
}

impl CoffeeMachine<MoneyInserted> {
    fn make_beverage(self) -> CoffeeMachine<Idle> {
        println!("There you go!");
        self.into_state()
    }
}
```

Typestate: when to use

- If your problem is like a state machine
- Ensure *at compile time* that no invalid operation is done

4. The Strategy pattern

Select behavior dynamically

Strategy: introduction

- Turn set of behaviors into objects
- Make them interchangeble inside context
- Execute strategy depending on input

Trait objects work well here!

Strategy: example

```
trait PaymentStrategy {
    fn pay(&self);
}

struct CashPayment;
impl PaymentStrategy for CashPayment {
    fn pay(&self) {
        println!("🪙💸");
    }
}

struct CardPayment;
impl PaymentStrategy for CardPayment {
    fn pay(&self) {
        println!("💳");
    }
}
```

```
fn main() {
    let method: &str
        = todo!("Read input");
    let strategy: &dyn PaymentStrategy
        = match method {
        "card" => &CardPayment,
        "cash" => &CashPayment,
        _ => panic!("Oh no!"),
    };
    strategy.pay();
}
```

Strategy: when to use

- Switch algorithms based on some run-time parameter (input, config, ...)

Anti-patterns

What *not* to do

The Deref polymorphism anti-pattern

A common pitfall you'll want to avoid

Deref polymorphism: Example

```
use std::ops::Deref;

struct Animal {
    name: String,
}

impl Animal {
    fn walk(&self) {
        println!("Tippy tap")
    }
    fn eat(&self) {
        println!("Om nom")
    }
    fn say_name(&self) {
        // Animals generally can't speak
        println!("...")
    }
}
```

```
struct Dog {
    animal: Animal
}
impl Dog {
    fn eat(&self) {
        println!("Munch munch");
    }
    fn bark(&self) {
        println!("Woof woof!");
    }
}

impl Deref for Dog {
    type Target = Animal;

    fn deref(&self) -> &Self::Target {
        &self.animal
    }
}

fn main (){
    let dog: Dog = todo!("Instantiate Dog");
    dog.bark();
    dog.walk();
    dog.eat();
    dog.say_name();
}
```

The output

```
Woof woof!
Tippy tap
Munch munch
...
```

Even overloading works!

Why is it bad?

- This is no 'real' inheritance: `Dog` is no subtype of `Animal`
- Traits implemented on `Animal` are not implemented on `Dog` automatically
- `Deref` and `DerefMut` are intended 'pointer-to-`T`' to `T` conversions
- Deref coercion by `.` 'converts' `self` from `Dog` to `Animal`
- Rust favours explicit conversions for easier reasoning about code

It will only add confusion: for OOP programmers it's incomplete, for Rust programmers it is unidiomatic

⚠️ Don't do try to inherit in Rust!

What to do instead?

- *Move away from inheritance constructs*
- Compose your structs
- Use facade methods
- Use `` `AsRef` `` and `` `AsMut` `` for explicit conversion

More anti-patterns

- Forcing dynamic dispatch in libraries
- `clone()` *to satisfy the borrow checker*
- `unwrap()` or `expect()` *to handle conditions that are recoverable or not impossible*

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