

# 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

```
1  pub struct S {  
2      first: First,  
3      second: Second,  
4  }  
5  
6  impl S {  
7      // Not get_first.  
8      pub fn first(&self) -> &First {  
9          &self.first  
10     }  
11  
12     // Not get_first_mut, get_mut_first, or mut_first.  
13     pub fn first_mut(&mut self) -> &mut First {  
14         &mut self.first  
15     }  
16 }
```

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

```
1  pub fn add(x: u32, y: u32) -> u32 {
2      x + y
3  }
4
5  /// Adds two values that implement the `Add` trait,
6  /// returning the specified output
7  pub fn add_generic<O, T: std::ops::Add<Output = O>>(x: T, y: T) -> O {
8      x + y
9  }
```

# Accept borrowed data if possible

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

```
1  /// Some very large struct
2  pub struct LargeStruct {
3      data: [u8; 4096],
4  }
5
6  /// Takes owned [LargeStruct] and returns it when done
7  pub fn manipulate_large_struct(mut large: LargeStruct) -> LargeStruct {
8      todo!()
9  }
10
11  /// Just borrows [LargeStruct]
12  pub fn manipulate_large_struct_borrowed(large: &mut LargeStruct) {
13      todo!()
14  }
```

Make your API

# Obvious

# Write Rustdoc

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

```
1  /// A well-documented struct.
2  /// ```rust
3  /// # // lines starting with a `#` are hidden
4  /// # use ex_b::MyDocumentedStruct;
5  /// let my_struct = MyDocumentedStruct {
6  ///     field: 1,
7  /// };
8  /// println!("{:?}", my_struct.field);
9  /// ```
10 pub struct MyDocumentedStruct {
11     /// A field with data
12     pub field: u32,
13 }
```

*To open docs in your browser:*

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

## Struct ex\_b::MyDocumentedStruct

```
pub struct MyDocumentedStruct {
    pub field: u32,
}
```

[–] Use three forward-slashes start a doc comment.  
You can add code examples, too:

```
let my_struct = MyDocumentedStruct {
    field: 1,
};
println!("{:?}", my_struct.field);
```

## Fields

field: u32

A field with data

# Include examples

Create examples to show users how to use your library

```
1  $ tree
2  .
3  ├── Cargo.lock
4  ├── Cargo.toml
5  ├── examples
6  │   └── say_hello.rs
7  └── src
      └── lib.rs
9  $ cargo run --example say_hello
10     Compiling my_app v0.1.0 (/home/henkdieter/tg/edu/my_app)
11     Finished dev [unoptimized + debuginfo] target(s) in 0.30s
12     Running `target/debug/examples/say_hello`
13 Hello, henkdieter!
```

# Use semantic typing (1)

Make the type system work for you!

```
1  /// Fetch a page from passed URL
2  fn load_page(url: &str) -> String {
3      todo!("Fetch");
4  }
5
6  fn main() {
7      let page = load_page("https://teach-rs.tweede.golf");
8      let crab = load_page("🦀"); // Ouch!
9  }
```

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

# Use semantic typing (2)

```
1 struct Url<'u> {
2     url: &'u str,
3 }
4
5 impl<'u> Url<'u> {
6     fn new(url: &'u str) -> Self {
7         if !valid(url) {
8             panic!("URL invalid: {}", url);
9         }
10        Self { url }
11    }
12 }
13
14 fn load_page(remote: Url) -> String {
15     todo!("load it");
16 }
17
18 fn main() {
19     let content = load_page(Url::new("🐞")); // Not gc
20 }
21
22 fn valid(url: &str) -> bool {
23     url != "🐞" // Far from complete
24 }
```

```
1 Compiling playground v0.0.1 (/playground)
2 Finished dev [unoptimized + debuginfo] target(s)
3 Running `target/debug/playground`
4 thread 'main' panicked at 'URL invalid: 🐞', src/main.rs:8:13
5 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!

```
1 $ cargo clippy  
2 $ cargo fmt
```

# Design patterns in Rust

# Why learn design patterns?

- Common problems call for common, tried and 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

```
1  const PATTERNS: &[amp;Pattern] = &[
2      Pattern::new("Newtype"),
3      Pattern::new("RAII with guards"),
4      Pattern::new("Typestate"),
5      Pattern::new("Strategy"),
6  ];
7  fn main() {
8      for pattern in PATTERNS {
9          pattern.introduce();
10         pattern.show_example();
11         pattern.when_to_use();
12     }
13 }
```

# 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

```
1  pub enum ValidateImeiError { /* - snip - */}
2
3  pub struct Imei(String);
4
5  impl Imei {
6      fn validate(imei: &str) -> Result<(), ValidateImeiError> {
7          todo!();
8      }
9  }
10
11 impl TryFrom<String> for Imei {
12     type Error = ValidateImeiError;
13
14     fn try_from(imei: String) -> Result<Self, Self::Error> {
15         Self::validate(&imei)?;
16         Ok(Self(imei))
17     }
18 }
19
20 fn register_phone(imei: Imei, label: String) {
21     // We can certain `imei` is valid here
22 }
```

# 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

```
1 pub struct Transaction<'c> {
2     connection: &'c mut Connection,
3     did_commit: bool,
4     id: usize,
5 }
6
7 impl<'c> Transaction<'c> {
8     pub fn begin(connection: &'c mut Connection)
9         -> Self {
10         let id =
11             connection.start_transaction();
12         Self {
13             did_commit: false,
14             id,
15             connection,
16         }
17     }
18
19     pub fn query(&self sql: &str) { /* - snip - */}
20
21     pub fn commit(self) {
22         self.did_commit = true;
23     }
24 }
```

```
1 impl Drop for Transaction<'_> {
2     fn drop(&mut self) {
3         if self.did_commit {
4             self
5                 .connection
6                 .commit_transaction(self.id);
7         } else {
8             self
9                 .connection
10                .rollback_transaction(self.id);
11        }
12    }
13 }
14 }
```

# 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

```
1 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

```
1  pub enum Idle {} // Nothing to do
2  pub enum ItemSelected {} // Item was selected
3  pub enum MoneyInserted {} // Money was inserted
4
5  pub struct CoffeeMachine<S> {
6      _state: PhantomData<S>,
7  }
8
9  impl<CS> CoffeeMachine<CS> {
10     /// Just update the state
11     fn into_state<NS>(self) -> CoffeeMachine<NS> {
12         CoffeeMachine {
13             _state: PhantomData,
14         }
15     }
16 }
17
18 impl CoffeeMachine<Idle> {
19     pub fn new() -> Self {
20         Self {
21             _state: PhantomData,
22         }
23     }
24 }
```

```
1  impl CoffeeMachine<Idle> {
2      fn select_item(self, item: usize) -> CoffeeMachine<ItemSelected> {
3          println!("Selected item {item}");
4          self.into_state()
5      }
6  }
7
8  impl CoffeeMachine<ItemSelected> {
9      fn insert_money(self) -> CoffeeMachine<MoneyInserted> {
10         println!("Money inserted!");
11         self.into_state()
12     }
13 }
14
15 impl CoffeeMachine<MoneyInserted> {
16     fn make_beverage(self) -> CoffeeMachine<Idle> {
17         println!("There you go!");
18         self.into_state()
19     }
20 }
```

# 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 interchangeable inside context
- Execute strategy depending on input

*Trait objects work well here!*

# Strategy: example

```
1
2 trait PaymentStrategy {
3     fn pay(&self);
4 }
5
6 struct CashPayment;
7 impl PaymentStrategy for CashPayment {
8     fn pay(&self) {
9         println!("💰💳");
10    }
11 }
12
13 struct CardPayment;
14 impl PaymentStrategy for CardPayment {
15     fn pay(&self) {
16         println!("💳");
17     }
18 }
```

```
1
2 fn main() {
3     let method: &str
4         = todo!("Read input");
5     let strategy: &dyn PaymentStrategy
6         = match method {
7         "card" => &CardPayment,
8         "cash" => &CashPayment,
9         _ => panic!("Oh no!"),
10    };
11    strategy.pay();
12 }
```

# 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

```
1 use std::ops::Deref;
2
3 struct Animal {
4     name: String,
5 }
6
7 impl Animal {
8     fn walk(&self) {
9         println!("Tippy tap")
10    }
11    fn eat(&self) {
12        println!("Om nom")
13    }
14    fn say_name(&self) {
15        // Animals generally can't speak
16        println!("...")
17    }
18 }
```

```
1 struct Dog {
2     animal: Animal
3 }
4 impl Dog {
5     fn eat(&self) {
6         println!("Munch munch");
7     }
8     fn bark(&self) {
9         println!("Woof woof!");
10    }
11 }
12
13 impl Deref for Dog {
14     type Target = Animal;
15
16     fn deref(&self) -> &Self::Target {
17         &self.animal
18     }
19 }
20
21 fn main () {
22     let dog: Dog = todo!("Instantiate Dog");
23     dog.bark();
24     dog.walk();
25     dog.eat();
26     dog.say_name();
27 }
```

# The output

```
1  Woof woof!  
2  Tippy tap  
3  Munch munch  
4  ...
```

*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 OOP in Rust!

# What to do instead?

- *Move away from OOP 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:

```
1  $ cargo test
2  [...]
3  running 2 tests
4  test tests::test_swap_items ... ok
5  test tests::test_swap_oob - should panic ... ok
6
7  test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
8  [..]
```

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

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

```
1  /// Swaps two values at the `first` and `second` indices of the slice
2  fn slice_swap_items(slice: &mut [u32], first: usize, second: usize) {
3      let tmp = slice[second];
4      slice[second] = slice[first];
5      slice[first] = tmp;
6  }
7
8  /// This module is only compiled in `test` configuration
9  #[cfg(test)]
10 mod tests {
11     use crate::slice_swap_items;
12
13     // Mark function as test
14     #[test]
15     fn test_swap_items() {
16         let mut array = [0, 1, 2, 3, 4, 5];
17         slice_swap_items(&mut array[..], 1, 4);
18         assert_eq!(array, [0, 4, 2, 3, 1, 5]);
19     }
20
21     #[test]
22     // This should panic
23     #[should_panic]
24     fn test_swap_oob() {
25         let mut array = [0, 1, 2, 3, 4, 5];
26         slice_swap_items(&mut array[..], 1, 6);
27     }
28 }
```

# Integration tests

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

```
1  $ tree
2  .
3  ├── Cargo.toml
4  ├── examples
5  │   └── my_example.rs
6  ├── src
7  │   ├── another_mod
8  │   │   └── mod.rs
9  │   ├── bin
10 │   │   └── my_app.rs
11 │   ├── lib.rs
12 │   ├── main.rs
13 │   └── some_mod.rs
14 └── tests
15     └── integration_test.rs
```



# Tests in your documentation

You can even use examples in your documentation as tests

```
1  /// Calculates fibonacci number n
2  ///
3  /// # Examples
4  ///
5  /// ```
6  /// # use example::fib;
7  /// assert_eq!(fib(2), 1);
8  /// assert_eq!(fib(5), 5);
9  /// assert_eq!(fib(55), 55);
10 /// ```
11 pub fn fib(n: u64) -> u64 {
12     if n <= 1 {
13         n
14     } else {
15         fib(n - 1) + fib(n - 2)
16     }
17 }
```

```
1  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*

# To do

Issue: tweedegolf/teach-rs#70

# To do

Issue: tweedegolf/teach-rs#69

# Summary