Best Practices and Idiomatic Rust

Performant Software Systems with Rust — Lecture 14

Baochun Li, Professor
Department of Electrical and Computer Engineering
University of Toronto

Best Practices and Idiomatic Rust

- Using `clippy` for static checking
- Writing tests and running them using `cargo test`
- Cargo, crates, and modules: building a larger project
- Design patterns in Rust
- Idiomatic Rust

Using `clippy` for Static Checking

Just run `cargo clippy`!

Live Demo

Writing Tests

- We use a Rust attribute `#[cfg(test)]` to tell the Rust compiler that a piece of code should only be compiled when the `test` config is active
  - `#[cfg(...)]` is one of the **built-in** attributes in Rust

Writing Tests: Example

```rust
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test] // indicates a test function
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

Live Demo with `cargo new hello --lib`

Checking Results with the `assert!` Macro

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Checking Results with the `assert!` Macro

Testing Equality with the `assert_eq!` and `assert_ne!` Macros

- `assert_eq!` is equivalent to `assert!` with the `==` operator
- `assert_ne!` is most useful when we know what a value definitely shouldn't be
- More convenient than `assert!` — they also print the values on the left and right if the assertion failed

Adding Custom Failure Messages

```rust
pub fn greeting(name: &str) -> String {
    format!("Hello {name}!")
}

#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{result}`"
    );
}
```

Checking for Panics with `should_panic`

Using `Result<T, E>` in Tests

- This allows the use of the `?` operator in the body of tests
- You can't use the `#[should_panic]` annotation on tests that use `Result<T, E>`
- Use `assert!(value.is_err())` to assert that an operation returns an `Err` variant

Using `Result<T, E>` in Tests

```rust
#[test]
fn it_works() -> Result<(), String> {
    let result = add(2, 2);

    if result == 4 {
        Ok(()) // returns `Ok(())` when the test passes
    } else {
        Err(String::from("two plus two does not equal four"))
    }
}
```

`cargo test` **command-line options**

- `cargo test --help` displays the options you can use with `cargo test`
- `cargo test -- --help` displays the options you can use after the separator that go to the resultant test binary

`cargo test` command-line options

- `cargo test -- --test-threads=1` runs tests consecutively rather than in parallel
- `cargo test -- --show-output` shows what's printed to standard output
- `cargo test add_two_and_two` runs only the named test `add_two_and_two`
- `cargo test add` runs all tests with `add` in the name

Ignoring Tests

```rust
#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

- `cargo test -- --ignored` to run only the ignored tests
- `cargo test -- --include-ignored` to run all tests

Unit Tests vs. Integration Tests

- To add **unit tests**, create a module named `tests` in each file to contain the test functions, annotated with `cfg(test)`
- **Integration tests** are external to your library, and are in the `test` directory

Managing Large Projects

- Our assignments have always been in one module and one file, `main.rs`
- As a project grows, you should organize code by splitting it into multiple modules and then multiple files

Features in Rust that Helps Manage Large Projects

- **Crates**: A tree of modules that produces a library or executable
- **Packages**: A Cargo feature that lets you build, test, and share crates
- **Modules** and **use**: Let you control the organization, scope, and privacy of paths
- **Paths**: A way of naming an item, such as a struct, function, or module

Crates and Packages

- The smallest amount of code that the Rust compiler considers at a time
- Contains **modules**
- Comes in a **library** or a **binary** form
- A **package** is a bundle of one or more crates that provides a set of functionality

Defining Modules to Control Scope and Privacy

- Grouping related code in modules
- Private (default) vs. public visibility
- Use the `use` keyword and a relative (with `super::`) or absolute path to create shortcuts
- Separate modules into different files

Live Demo

Design Patterns

- **Design patterns** are “general reusable solutions to a commonly occurring problem within a given context in software design”
- They are very language-specific and sometimes controversial

Design Patterns

- If overused, design patterns can add unnecessary complexity to programs
- Features in Rust allow us to throw out many conventional design patterns, which were invented in the prime times of object orientation
  - The Strategy pattern is no longer useful as we can just use traits

Prefer Small Crates

- Prefer small crates and do not over-engineer the design
- The `url` crate only provides tools for working with URLs
- The `num_cpus` crate only provides a function to query the number of CPUs on a machine

The **Singleton** Design Pattern

- The **singleton** pattern restricts the instantiation of a type to a **singular** instance
- The **best practice** is to avoid using this pattern completely

Implementing the Singleton Pattern

Example: Implementing a `ReportLogger` for my discrete-event network simulator

```rust
use std::sync::{Arc, LazyLock, Mutex, RwLock};

// `LazyLock` is a thread-safe value which is initialized on the first
// access. It is available since Rust 1.80 and can be used in statics
// `RwLock` is a reader-writer lock that allows multiple readers or
// one writer at any time
pub static REPORT_INTERVAL: LazyLock<RwLock<f64>> =
    LazyLock::new(|| RwLock::new(f64::MAX));

pub struct ReportLogger {
    report_logger: CsvLogger,
    report_interval: f64,
}
```

Implementing the Singleton Pattern

```rust
impl ReportLogger {
    pub fn new() -> ReportLogger {
        ReportLogger {
            report_logger: CsvLogger {},
            report_interval: *REPORT_INTERVAL.read().unwrap(),
        }
    }

    pub fn get_instance() -> Arc<ReportLogger> {
        static INSTANCE: LazyLock<Mutex<Option<Arc<ReportLogger>>>> =
            LazyLock::new(|| Mutex::new(None));

        let mut instance = INSTANCE.lock().unwrap();
        if instance.is_none() {
            *instance = Some(Arc::new(ReportLogger::new()));
        }
        Arc::clone(instance.as_ref().unwrap())
    }
```

Idiomatic Rust

Make Illegal States Unrepresentable

Make Illegal States Unrepresentable

Example: Managing a list of users

```rust
struct User {
    username: String,
    birthdate: chrono::NaiveDate,
}
```

Make Illegal States Unrepresentable

But what happens if we create a user with an **empty** username?

```rust
let user = User {
    username: String::new(),
    birthdate: chrono::NaiveDate::from_ymd(1990, 1, 1),
};
```

Not what we want — the type system is our friend!

Define a Type that Represents a Username

```rust
struct Username(String);

impl Username {
    // 'static: reference lives for the remaining lifetime of
    // the running program; a string literal here is a `&'static str`
    fn new(username: String) -> Result<Self, &'static str> {
        if username.is_empty() {
            return Err("Username cannot be empty");
        }
        Ok(Self(username))
    }
}
```

Define a Type that Represents a Username

```rust
struct User {
    username: Username,
    birthdate: chrono::NaiveDate,
}

let username = Username::new("johndoe".to_string())?;
let birthdate = NaiveDate::from_ymd(1990, 1, 1);
let user = User { username, birthdate };
```

What About the Birthdate?

A new user that is **1000 years old** is perhaps not what we want either!

```rust
struct Birthdate(chrono::NaiveDate);

impl Birthdate {
    fn new(birthdate: chrono::NaiveDate) -> Result<Self, &'static str> {
        let today = chrono::Utc::today().naive_utc();
        if birthdate > today {
            return Err("Birthdate cannot be in the future")
        }

        let age = today.year() - birthdate.year();
        if age < 12 {
            return Err("Not old enough to register")
        }
        if age >= 122 {
            return Err("The longest living person was 122 years old")
        }

        Ok(Self(birthdate))
```

It's Now Time to Write Some Tests!

```rust
[cfg(test)]
mod tests {
    use super::*;
    use chrono::Duration;

    #[test]
    fn test_birthdate() {
        let today = chrono::Utc::today().naive_utc();
        // Birthdate cannot be in the future
        assert!(Birthdate::new(today + Duration::days(1)).is_err());
        // Excuse me, how old are you?
        assert!(Birthdate::new(today - Duration::days(365 * 122)).is_err())
        // Not old enough
        assert!(Birthdate::new(today - Duration::days(365 * 11)).is_err());
        // Ok
        assert!(Birthdate::new(today - Duration::days(365 * 15)).is_ok());
    }
}
```

Using Enums to Represent State

Using Enums to Represent State

Do not use `bool` to represent **state**!

```rust
struct User {
    // ...
    active: bool,
}
```

What does `active = false` mean anyway?

Using Enums to Represent State

Can we use an unsigned integer to represent state?

```rust
struct User {
    // ...
    active: bool,
}

const ACTIVE: u8 = 0;
const INACTIVE: u8 = 1;
const SUSPENDED: u8 = 2;
const DELETED: u8 = 3;

let user = User {
    // ...
    status: ACTIVE,
};
```

It’s still not ideal!

Using Enums to Represent State

Enums are a great way to model state!

```rust
#[derive(Debug)]
pub enum UserStatus {
    /// The user is active and has full access
    /// to their account and any associated features.
    Active,

    /// The user's account is inactive.
    /// This state can be reverted to active by
    /// the user or an administrator.
    Inactive,

    /// The user's account has been temporarily suspended,
    /// possibly due to suspicious activity or policy violations.
    /// During this state, the user cannot access their account,
    /// and an administrator's intervention might
    /// be required to restore the account.
    Suspended,

```

Aim for Immutability

Aim for Immutability

- Variables in Rust are **immutable** by default
- The `mut` keyword should be used sparingly, preferably only in tight scopes
- Move instead of `mut`
- Don’t be afraid of copying data!

Aim for Immutability

```rust
ub struct Mailbox {
    /// The emails in the mailbox
    // Obviously, don't represent emails as strings in real code!
    // Use higher-level abstractions instead.
    emails: Vec<String>,
    /// The total number of words in all emails
    total_word_count: usize,
}

impl Mailbox {
    pub fn new() -> Self {
        Mailbox {
            emails: Vec::new(),
            total_word_count: 0,
        }
    }

    pub fn add_email(&mut self, email: &str) {
```

Aim for Immutability

```rust
pub struct Mailbox {
    emails: Vec<String>,
}

impl Mailbox {
    pub fn new() -> Self {
        Mailbox {
            emails: Vec::new(),
        }
    }

    pub fn add_email(&mut self, email: &str) {
        self.emails.push(email.to_string());
    }

    pub fn get_word_count(&self) -> usize {
        self.emails
             iter()
```

That’s It for the Course!

- First of its kind in **Canada**
- In U.S. universities, only offered at **Northwestern** and **UPenn**
- I hope you learned a lot from this course, and good luck with your project!

Required Additional Reading

The Rust Programming Language, Chapter 7 and 11

Rust Design Patterns

Idioms in Rust

A Blog on Idiomatic Rust