**Rust Refactoring for Beginners**

[[Michał Fita](https://michalfita.medium.com/?source=post_page-----15a3270ce45d--------------------------------)](https://michalfita.medium.com/?source=post_page-----15a3270ce45d--------------------------------)

[[Better Programming](https://betterprogramming.pub/?source=post_page-----15a3270ce45d--------------------------------)](https://betterprogramming.pub/?source=post_page-----15a3270ce45d--------------------------------)

[Michał Fita](https://michalfita.medium.com/?source=post_page-----15a3270ce45d--------------------------------)

·

Follow

Published in

[Better Programming](https://betterprogramming.pub/?source=post_page-----15a3270ce45d--------------------------------)

·

10 min read

·

Jul 7

47

1



Recently [Neeraj Avinash](https://www.linkedin.com/in/neeraj-avinash-33407b9) posted his code on Rust Programming Language Group on LinkedIn. His goal is to learn some Rust basics, but I found his example being a good foundation for my article. The intention is to show how to improve Rust’s code in stages and demonstrate what mistakes beginners can avoid starting with their code. For the sake of simplicity, please ignore the obvious deficiencies of this simple program.

**The Baseline**

I’m not going to post the whole code here, even if this is short, as it would discourage readers willing only to skim through the text. You can see the whole code at commit being my starting point [here](https://github.com/neerajbg/rust-student-mini-project/tree/11c7526a839c3223a531429c1f79c44bfbe7bb36). I use only code snippets with explanations in the text that follows. This should help understand the process better, not seeing just the final result. Each section below forms the commit of [this PR](https://github.com/neerajbg/rust-student-mini-project/pull/1).

**Lack of Rust Idioms**

For experienced Rustacean it stings eyes that the following function doesn’t return Result<>, but a tuple:

fn add\_student() -> (Student, bool)

This approach is not only not idiomatic, but misleading for the code reader — “What the bool value means?” someone might ask. Then to react to the outcome of this function something as complicated as the below has to be written:

// Add student to course  
let (st, err) = add\_student();  
  
// Check for error. If error, continue the loop  
if !err {  
 continue;  
}

Five lines with comments for the reader to understand the code. Short variable names are another bad practice.

**Refactor**

Let’s refactor these bits first. From:

fn add\_student() -> (Student, bool) {  
 // ...  
 let mut st = Student {  
 name: "".to\_string(),  
 age: 0,  
 };  
 // ...  
 if student\_name.len() < 3 {  
 // ...  
 return (st, false);  
 }  
 // ...  
 (st, true)  
}

To be more idiomatic and readable:

fn add\_student() -> Result<Student, &'static str> {  
 // ...  
 if student\_name.len() < 3 {  
 // ...  
 return Err("Student's name too short");  
 }  
 // ...  
 let age = age.parse.map\_err(|\_| "Cannot parse student's age")?;  
  
 Ok(Student {  
 name: student\_name,  
 age  
 })  
}

I’m aware returning static strings as errors is a non-sustainable practice, but it’s good enough for this example. If the second part of this article is ever created about improving code with the use of external crates then I demonstrate recognized good practices.

The .map\_err() method used in the example allows conversion of the type instance held by Err(e) enum's value into one that's supported by our function.

Our declared type, in this case, is &'static str (Rust's equivalent of const char\* type idiom in C), so our texts within quotes match. The ? operator is in fact one of the best features Rust has - it checks the instance of Result<> before it, if the value is Err(e) it returns that result; continues otherwise. In the past, there was try!() macro you may spot in older code.

As a result our check if we have expected output from the function evolves into the following:

let student = if let Ok(student) = add\_student() {  
 student  
} else {  
 continue;  
}  
  
student\_db.push(student.clone());

This condition isn’t ideal, as it effectively discards any error. We do that here on assumptions we’re allowed to but consider handling the Err(e) enum on an individual basis.

**Beginner’s Mistake: Infinite**loop

The original code has two problems with loop:

1. There’s still an incremental counter in the context, that is unnecessarily mutable
2. The exit\_var condition display "outcome" before leaving

Both above practices are common beginner’s mistakes. Let’s improve the code and turn:

let mut i: i8 = 1;  
loop {

into

for i in 1usize.. {

based on the assumption the typical 64-bit size of usize these days is good enough for ages to come.

Then we remove:

i += 1;

that way we simplified the code and made it more readable.

Then we move the invocation of showing the list of collected students out of the loop:

if exit\_var == "q" {  
 println!("Exiting...");  
 display\_students\_in\_course(&student\_db);  
 break;  
}

turns into:

/// ...  
 if exit\_var == "q" {  
 break;  
 }  
}  
println!("Exiting...");  
display\_students\_in\_course(&student\_db);

**Encapsulation**

This program reminds me of myself in the ’80s when writing such programs in BASIC was pretty common. But BASIC was a basic imperative language lacking many features that are common today, for example, functions. But as a result, straightforward thinking about the problem led to straightforward code. The code we work on here is precisely that, an example of straightforward thought about the recipe of how to achieve the expected result.

This works for the start, but usually quite quickly becomes unmaintainable. As a remedy, universities teach students object-oriented programming. Despite them teaching it wrong, without going into details we’re going to use some of its principles to improve the code for the future.

In simplest words **encapsulation** is confining basic elements to prevent unwanted access and *hide* implementation details from the user.

Rust by nature is not object-oriented as its type and trait model is closer to functional languages than to *proper* object-oriented languages[^oo-lang]. It’s *good enough* to encapsulate things in our simple program.

In this example, I will demonstrate how to use modules for refactoring, even though it may not be necessary for a program of this small size.

**Refactoring**

Let’s start with this line of code:

let mut student\_db: Vec<Student> = Vec::new();

that creates an empty mutable vector. But this type doesn’t say much about what the user of this primitive database is allowed to do. In our case, it’s not much.

Let’s create src/db.rs module and include the following code in it:

use super::Student;  
  
pub struct StudentDatabase {  
 db: Vec<Student>  
}  
impl StudentDatabase {  
 pub fn new() -> Self {  
 Self {  
 db: vec![]  
 }  
 }   
}

then at the start of src/main.rs we have to add the:

mod db;

for this module to be taken into account during compilation.

But this simple code only initializes the inner vector. But this simple example is good enough to quickly get through how types can wield the power of methods. Rust’s idiom about new() method, contrary to other languages, is about it being the simplest constructor of instance on the stack.

Huh? If you feel confused here you have to do your learning about stack usage in programs written in languages not using garbage collection. Without going into too much detail (this topic is worth a whole article) other languages tend to use new() about memory allocation on the heap (I'm thinking about C++ and Java).

Moving forward we need the ability to add students — let’s simply do this freeing the user from instantiating the Student type by the API user (this is one way, not always ideal, but not a subject of this lesson). New code to add to impl StudentDatabase:

pub fn add(&mut self, name: String, age: u8) {  
 self.db.push(Student {  
 name,  
 age  
 })  
}

assuming it cannot fail gracefully, what’s in alignment with .push() method of std::Vec - it can panic. Observe that Rust can match the names of the function's arguments with the field names of the type we're using.

This is **field init shorthand** that simplifies code and improves readability and doesn't require writing name: name.

Another aspect worth mentioning is the fact the name argument is **consuming**, which in Rust's semantics using it we move the instance of that string into the scope of the function's body. Hence, the need for .clone() in the original code. That's not the most efficient way of working with strings, but it's beyond the scope of this article to discuss other options.

To fully finish refactoring in src/main.rs we need to add one more method. Normally you'd start refactoring straight away letting your editor show errors in the code, but to avoid such confusion at this stage we prepare all components we need upfront. Here we go:

pub fn display(&self) {  
 for student in self.db.as\_slice() {  
 println!("Name: {}, Age: {}", student.name, student.age);  
 }  
}

At this stage to fulfill all requirements of the existing code, we need to know the length of the database. In the next stage, we’ll improve encapsulation to remove the need for that, however, it’s a useful function in the public API of the database.

To check length:

pub fn len(&self) -> usize {  
 self.db.len()  
}

Now is the time to apply our new code to the src/main.rs. First, replace:

let mut student\_db: Vec<Student> = Vec::new();

into

let mut student\_db = db::StudentDatabase::new();

then remove:

display\_students\_in\_course(&student\_db);

from the maximum length condition body. And the definition of that function has to be removed as well to avoid warnings about dead code.

Then replace the same line at the bottom of the main()'s body with:

student\_db.display();

Afterward, the original addition into the vector:

student\_db.push(student.clone());

replace with:

student\_db.add(student.name.clone, student.age);

This shows a deficiency of our earlier decision about arguments to add(). In languages supporting overloading, we easily could do this both ways, but in Rust, you need explicit method names, so we'll leave this for now. Let's focus on add\_student() function, which doesn't add any students, so have an incorrect name. So, we start with renaming:

// Function to add a new student to DB  
fn add\_student() -> Result<Student, &'static str> {

into

fn input\_student() -> Result<Student, &'static str> {

and invocation as well.

The code of that function tries to do some things redundantly:

let student\_name = &input[..input.len() - 1].trim();  
// ...  
let age = input.trim();  
age.to\_string().pop(); // Remove newline character  
let age = age.parse().map\_err(|\_| "Cannot parse student's age")?;

so this requires fixing into much simpler idiomatic Rust:

let student\_name = input.trim();  
// ...  
let age = input.trim().parse().map\_err(|\_| "Cannot parse student's age")?;

What becomes clearly visible in the result is the repeated pattern of:

let mut input = String::new();  
let \_ = stdin().read\_line(&mut input);

where the result of read\_line() is ignored. Such ignorance is considered a bad practice.

A quick fix would be adding the following function and replacing repeated code with it:

fn prompt\_input<T: FromStr>(prompt: &str) -> Result<T, &'static str> {  
 println!("{}: ", prompt);  
 let mut input = String::new();  
 let \_ = stdin().read\_line(&mut input);  
 input.trim().parse().map\_err(|\_| "Cannot parse input")  
}

So, we end up with the function input\_student() as compact as:

fn input\_student() -> Result<Student, &'static str> {  
 print!("#### Adding New Student ####\n");  
 let student\_name: String = prompt\_input("Enter Student Name")?;  
 // Check for minimum 3 character length for Student name  
 if student\_name.len() < 3 {  
 println!(  
 "Student name cannot be less than 3 characters. Record not added.\n Please try again"  
 );  
 return Err("Student's name too short");  
 }  
 let age = prompt\_input("Age of the Student")?;  
 Ok(Student {  
 name: student\_name.to\_string(),  
 age,  
 })  
}

It’s far from great, but a significant improvement, don’t you think?

In this section, we covered fundamentals of encapsulation, which as well helps in keeping code **DRY** (**D**on’t **R**epeat **Y**ourself). We do our best to isolate the using function from dealing with implementation details. The final code is far from perfection (if ever can be one), but as it acts as an illustration of steps taken certain parts have to be left for later.

**Unit Tests**

One of the very good practices missing in the original examples is unit tests. Unit tests in our code act as the first quality gate capable to catch many errors and mistakes that can be costly to catch and fix if escape into the next gate, and even more costly if escape further. The latter happens quite frequently in younger projects as they at the start can afford only manual testing.

As we separated important parts of core logic into separate modules, we have a good starting point to introduce unit tests. Some people may moan at this point good code starts with unit tests. But the reality is that it’s a utopian vision as most code in real life starts with some early draft of a prototype. So, let’s put that discussion aside.

Rust is a very friendly language for writers of unit tests. Basic mechanisms are built-in. They’re not ideal in every aspect, but good enough to start testing without much fuzz.

Let’s add to src/db.rs at the bottom:

#[cfg(test)]  
mod tests {  
 use super::\*;  
  
 #[test]  
 fn add\_to\_database() {  
 let mut db\_ut = StudentDatabase::new();  
 db\_ut.add("Test Student".to\_string(), 34);  
 assert\_eq!(db\_ut.len(), 1);  
 }  
}

This is a very primitive test, but a good start. Running cargo test would produce:

Compiling student v0.1.0 (/home/teach/rust-student-mini-project)  
 Finished test [unoptimized + debuginfo] target(s) in 0.16s  
 Running unittests src/main.rs (target/debug/deps/student-f5f1fdf375ff16cf)  
  
running 1 test  
test db::tests::add\_to\_database ... ok  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

so our test passes.

But the other part of our API is displaying the content of the database. So we bump into the issue of the output from that function. Capturing the output in the test is theoretically possible, but it’s complex and way beyond the scope of this article.

Let’s focus on a practice called **reversed dependency injection** that in simplest terms is about providing interfaces for our unit under test for everything this may depend on. In practice, this is limited to aspects improving testability and — as a result — reusability of our code.

To achieve this we need to change .display() method of StudentDatabase into:

pub fn display\_on(&self, output: &mut impl std::io::Write) -> io::Result<()> {  
 for student in self.db.as\_slice() {  
 write!(output, "Name: {}, Age: {}\n", student.name, student.age)?;   
 }  
 Ok(())  
}

then invocation in src/main.rs into:

student\_db.display\_on(&mut stdout()).expect("Unexpected output failure");

So we in effect are able to create the following test below:

#[test]  
fn print\_database() {  
 let mut db\_ut = StudentDatabase::new();  
 db\_ut.add("Test Student".to\_string(), 34);  
 db\_ut.add("Foo Bar".to\_string(), 43);  
  
 let mut output: Vec<u8> = Vec::new();  
 db\_ut.display\_on(&mut output).expect("Unexpected output failure");  
 let result = String::from\_utf8\_lossy(&output);  
 assert\_eq!(result, "Name: Test Student, Age: 34\nName: Foo Bar, Age: 43\n");  
}

Then we see both tests pass:

running 2 tests  
test db::tests::add\_to\_database ... ok  
test db::tests::print\_database ... ok

**Conclusion**

In a couple of stages, this article demonstrated how to get from the state where code looks like written by a teenage noob at the early beginning of a programming apprenticeship, to the point where code is close to the level expected from a junior engineer working in the first job for couple months. And I don’t write this to offend anyone, just to underscore the role of experience in designing a basic code structure. Writing *spaghetti* code is easy, planning well how code should be structured with the right amount of separation of concerns is hard and in my 25 years career, I still learn how to do this better. This learning never ends for good programmers.

Even the refactored version has flaws, I’m aware of that. It’s just polishing it now to the level of full satisfaction would blur the sense of this article and would make changes show much more complex to explain. I leave it to the readers’ exercise to find what I missed and how more unit tests may help prevent embarrassment.

I did some of the major changes the code required to be improved and committed changes done in this stage for reference. None of demonstrated *solutions* are ultimate, but most should be close to what any very experienced programmer would change in this code in the first place. You have the right, however, to have your opinion on that matter. I invite you to challenge my approach by writing your article about proposed changes to the baseline.

Other than that I hope you enjoyed it.

¹ The proper Object Oriented Language is Smalltalk, however, essential concepts of OO in Simula predate it. Many claim programming languages today are an abomination of OO.