Extracted from:

Hands-on Rust

Effective Learning through 2D Game Development and Play

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Hands-on Rust

Effective Learning through 2D Game Development and Play

Herbert Wolverson

edited by Tammy Coron

Hands-on Rust

Effective Learning through 2D Game Development and Play

Herbert Wolverson

The Pragmatic Bookshelf
Raleigh, North Carolina

Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: Vanya Wong
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-816-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2021

Capturing User Input

Most computer programs operate in a cycle of accepting input from the user and transforming that into some form of—hopefully useful—output. A calculator without buttons is useless, and a computer program without input is equally limited to always doing the same thing. You used println! in "Hello, World" to output text; you can use read_line() to accept data from the terminal.

In this section, you'll use the terminal to ask the visitor to type their name and receive the result. Finally, you'll make use of Rust's formatting system to print a personalized greeting to the terminal.

Prompting for the Visitor's Name

When a visitor arrives at your swanky new treehouse, you need to ask them for their name. In Printing Text, on page ?, you used println! to print text to the screen. You'll do the same thing here, too.

Replace println!("Hello, World") with:

FirstStepsWithRust/hello_yourname/src/main.rs

```
println!("Hello, what's your name?");
```

Why Did the Project Name Change?

Don't worry, you're still working on the treehouse project. The source code examples in this book are provided in chunks, representing each stage of development within the chapter. When you see the source file name change, it means that the code is referring to the next example along the way—you don't need to change anything.

You replaced the output string asking for the visitor's name. Now you're ready to receive and store the answer.

Storing the Name in a Variable

You'll store the visitor's name in a variable. Rust variables default to being *immutable*. Once an immutable variable is assigned, you cannot change the value stored in the variable. You can make more variables, reference, or copy a previously assigned variable, but you can't change the immutable variable once it is assigned. You can explicitly mark a variable as *mutable* with the mut keyword. Once marked as mutable, a variable may be changed as needed.

Add a second line of code to your program:

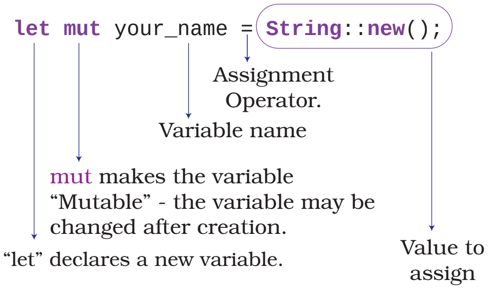
FirstStepsWithRust/hello_yourname/src/main.rs

```
let mut your_name = String::new();
```

Be Wary of Mutation

Mutants are scary and mutable variables can be too. It’s tempting to mark everything as mutable so you don’t need to remember to add mut when you need it. Rust/Clippy will warn you when a variable doesn’t need to be marked as mut. It’s a good idea to heed the warnings, because it’s much easier to think about what a program does if you can be sure that a variable still means what you think it means.

This code creates a mutable variable named your_name, and sets it to be an empty text string. The syntax for the variable declaration looks like this:[1]



```
let mut your_name = String::new();
```

Assignment Operator.
Variable name
mut makes the variable “Mutable” - the variable may be changed after creation.
“let” declares a new variable.
Value to assign

String is a *type*, built into Rust.[2] Types can have functions associated with them; you’ll learn how to do this for your types in Grouping Data with Structs, on page ?.

Use snake_case for Variable Names

Rust encourages you to use snake_case for variable names. Use lowercase and replace spaces with _. Clippy will remind you if you forget.

Let’s obtain the user’s name from the keyboard and store it in a string.

Receiving Keyboard Input

Rust’s standard input system provides an easy way to receive keyboard input. Rust provides terminal input functions in std::io::stdin.[3] You can find read_line as std::io::stdin::read_line. That’s a lot of typing just to read a line of text from the keyboard. Let’s import the name with Rust’s use keyword so that you don’t have to type out the full version every time.

[1] https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html.

[2] https://doc.rust-lang.org/1.7.0/book/strings.html.

[3] https://doc.rust-lang.org/std/io/struct.Stdin.html.

Add the following line to the top of main.rs:

FirstStepsWithRust/hello_yourname/src/main.rs

```
use std::io::stdin;
```

This line imports std::io::stdin into your project. Now you can just type stdin instead of remembering all of the namespace prefix.

Reading User Input

Now that you have access to stdin, and a variable in which to store the user's name, you're ready to read that name from the console input. Add the following code to your main function, immediately after the variable declaration:

FirstStepsWithRust/hello_yourname/src/main.rs

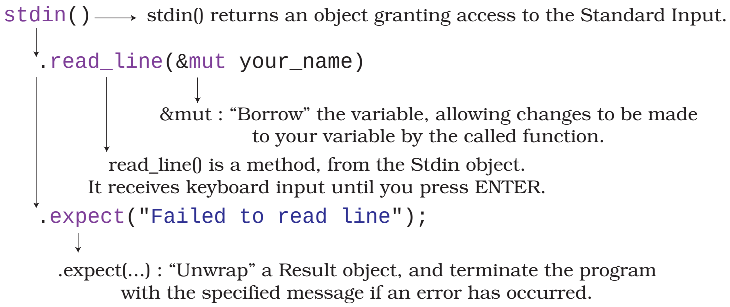
```
stdin()
    .read_line(&mut your_name)
    .expect("Failed to read line");
```

Combining functions like this is called *function chaining*. Starting from the top, each function passes its results to the next function. It's common to format a function chain with each step on its line, indented to indicate that the block belongs together. The cargo fmt command (see Formatting Your Code, on page ?) will automatically apply this formatting standard for you.

Why Create the Variable First?

read_line() wants to write its results into an existing string, rather than returning to a new one. You have to create the empty String first so that it has somewhere to store the function's results.

Here are the sections of the read_line call explained:



```
stdin()
 .read_line(&mut your_name)
 .expect("Failed to read line");
```

stdin() returns an object granting access to the Standard Input.

&mut : "Borrow" the variable, allowing changes to be made to your variable by the called function.

read_line() is a method, from the Stdin object. It receives keyboard input until you press ENTER.

.expect(...) : "Unwrap" a Result object, and terminate the program with the specified message if an error has occurred.

You can learn two important concepts from this code:

- Prefixing a variable with an ampersand (&) creates a *reference* to the variable. A reference passes access to the variable itself, not a copy of the variable. This is also called *borrowing*—you're *lending* the variable to the function you are calling. Lending with &mut permits the borrowing function to mutate your variable. Any changes it makes to the variable are written directly into the variable you lent. Passing &mut your_name to read_line allows the read_line function to write directly into your_name.

- You expect the read_line function to work correctly. If it doesn't, your program will crash. Rust is returning a Result object, and you are checking that the function worked by calling expect. Don't worry about the details of this yet. You'll learn about error handling in Handling Errors in the Main Function, on page ?.

Printing with Placeholders

Now that the your_name variable contains the visitor's name, you can greet them properly. Greeting the user requires another println! call:

FirstStepsWithRust/hello_yourname/src/main.rs

```rust
println!("Hello, {}", your_name)
```

The println macro is almost the same as before, but it has gained a *placeholder*. Including {} in your println! string indicates that a variable's value goes here. You then provide the variable as a second parameter to the macro call. Rust includes a very powerful formatting system and can take care of most of your string formatting needs out of the box.[4]

The Completed Greeter Program

Your treehouse admission program now looks like this:

FirstStepsWithRust/hello_yourname/src/main.rs

```rust
use std::io::stdin;

fn main() {
    println!("Hello, what's your name?");
    let mut your_name = String::new();
    stdin()
        .read_line(&mut your_name)
        .expect("Failed to read line");
    println!("Hello, {}", your_name)
}
```

Run the program with cargo run (see Run Hello, World, on page ? if you need a refresher on running programs), and you'll see the following:

```
⇒ cargo run
❮ Hello, what's your name?
```

[4] https://doc.rust-lang.org/std/fmt/

```
⇛ Herbert
❮ Hello, Herbert
```

Congratulations, you now have working input and output. Let's learn about *functions* by moving your input code into a reusable block of code.

Moving Input to a Function

You're frequently going to be asking the user for their name in this chapter. Whenever you have commonly used code, it's a good idea to move it into a function. This has two advantages: you don't keep typing the same code, and a single call to `what_is_your_name()` is less disruptive of the overall flow of your function, which lets you concentrate on the important parts. This is a form of *abstraction*: you replace detailed code with a function call and move the detail into a function.

When Should I Use a Function?

Try to use a function when you are typing the same code repeatedly. This is called the DRY principle: *Do not Repeat Yourself*. *Code Complete [McC04]* provides an excellent overview of the DRY Principle and its practical application.

You should also consider breaking code up into functions if it becomes very large. It's much easier to read a shorter function that calls other functions, especially when you come back to a piece of code after a break.

In Hello, World, on page ?, you declared the `main` function; making your own functions is similar:

FirstStepsWithRust/hello_yourname_function/src/main.rs

```
use std::io::stdin;

① fn what_is_your_name() -> String {
②     let mut your_name = String::new();
      stdin()
          .read_line(&mut your_name)
          .expect("Failed to read line");
③     your_name
  }

  fn main() {
      println!("Hello, what's your name?");
④     let name = what_is_your_name();
      println!("Hello, {}", name);
  }
```

❶ The function signature is very similar to the main function. The function name is different, and -> String denotes that it *returns* a String.

❷ The read_line code is the same, moved into your function.

❸ This line doesn't end with a semicolon. This is Rust shorthand for return. Any expression may return this way. It's the same as typing return your_name;. Clippy will complain if you type return when you don't need it.

❹ Instead of calling read_line directly, call your function and store the result in name.

Now that you have your input function, you're ready to move on.

Trimming Input

The program's output looks good on screen, but it contains a subtle bug. The string contains some extra characters representing the ENTER key. You can see this by replacing your last println! call with the following:

FirstStepsWithRust/treehouse_guestlist_problem/src/main.rs

```
println!("{:?}", name);
```

Replacing the {} placeholder with {:?} uses the *debug* placeholder. Any type that supports debug printing will print a detailed debugging dump of its contents, rather than just the value. If you run the program now, you can see the problem:

```
❮ Hello, what's your name?
⇒ Herbert
❮ "Herbert\r\n" (or "Herbert\n" on UNIX-based systems)
```

\r is a special character that means carriage return. On old printers, it returned the printhead to the left of the page. \n means a new line. Windows generates these two characters for an ENTER keypress. UNIX-derived systems just append \n.

Rust's strings include a trim() function to remove these extra characters. If you don't remove these characters, you'll be surprised when you type "Bert" but Bert in your code doesn't match, because the string contains Bert\r\n.

It's also a good idea to convert the input to lowercase. This allows "Bert," "bert," and even "bErt" to correctly match a name. Rust's strings provide the to_lowercase() function to do this for you.

Amend your function to use both trim and to_lowercase:

FirstStepsWithRust/treehouse_guestlist_trim/src/main.rs

```rust
fn what_is_your_name() -> String {
    let mut your_name = String::new();
    stdin()
        .read_line(&mut your_name)
        .expect("Failed to read line");

    your_name
        .trim()
        .to_lowercase()
}
```

That’s much better. Your input is now always lowercase, and it doesn’t include non-printing characters. A treehouse with only one visitor isn’t much of a party. Let’s add support for more of your friends.