The Rust Programming Language

Amir Khayatzadeh

Department of Computer Engineering
Islamic Azad University, Mashhad Branch

Spring 2024

Table of Contents I

Table of Contents II

By default, all variables in Rust are immutable. When a variable is immutable, once a value is bound to a name, you can't change that value.[3]

```rust
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

Figure: Filename: src/main.rs[3]

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

Figure: Error Message On Compilation[3]

Put "mut" before the identifier to make the variable mutable.

```rust
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

Figure: Filename: src/main.rs[3]

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

Figure: Output of The Code[3]

```rust
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Figure: An example of a constant declaration.[3]

Differences between immutable variables[3]:

1. The constant's type must be annotated.
2. Constants are compile-time. Immutable variables are runtime.
3. Constants are set only to a constant expression, not the result of a value that could only be computed at runtime.
4. Immutable variables cannot be global.

```
error: expected item, found keyword `let`
 --> src/main.rs:1:1
  |
1 | let x = 5;
  | ^^^ consider using `const` or `static` instead of `let` for global variables

error: could not compile `variables` (bin "variables") due to 1 previous error
```

Figure: Compilation Error For Declaring An Immutable Variable Globally.

Table of Contents I

Table of Contents II

Rust, A Statically Typed Language

Compiler usually can infer the type based on the value, but in cases where multiple types are possible, we need to annotate the type:

```rust
let guess: u32 = "42".parse().expect("Not a number!");
```

Figure: Convert a string to a numeric type.[3]

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0284]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^        ----- type must be known at this point
  |
  = note: cannot satisfy `<_ as FromStr>::Err == _`
help: consider giving `guess` an explicit type
  |
2 |     let guess: /* Type */ = "42".parse().expect("Not a number!");
  |              ++++++++++++

For more information about this error, try `rustc --explain E0284`.
error: could not compile `no_type_annotations` (bin "no_type_annotations") due
```

Figure: If the type is not specified, we get compiler error.[3]

Integer Types

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Figure: Integer Types in Rust[3]

the isize and usize types depend on the architecture of the computer your program is running on, which is denoted in the table as “arch”: 64 bits if you’re on a 64-bit architecture and 32 bits if you’re on a 32-bit architecture.[3]

Floating-Point Types

```rust
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Figure: An example of floating-point variables in rust.[3]

The default type is f64 because on modern CPUs, it's roughly the same speed as f32 but is capable of more precision. All floating-point types are signed.[3]

The Boolean and Character Types

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

Figure: An example of declaring boolean values.[3]

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Figure: An example declaring char values.[3]

- Declared using single quotations.
- 4 bytes in size.
- Supports Unicode values.

The Tuple Type

- Groups of values with different types.
- Tuples have a fixed length.

```rust
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Figure: An example with optional type annotations.[3]

```rust
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

Figure: An example of destructuring a tuple.[3]

The Array Type

- Groups of values with the same type.
- Arrays have a fixed length.
- Allocated on the stack rather than the heap.

```rust
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Figure: Declaring an array with type annotation and length.[3]

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

Figure: Accessing array elements.[3]

An Example of Rust's Memory Safety

```rust
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

Figure: An example of accessing array elements from input.[3]

Panicking

```
thread 'main' panicked at src/main.rs:19:19:
index out of bounds: the len is 5 but the index is 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Figure: Runtime Error (Panicking) if 10 is given as the index.[3]

This is an example of Rust's memory safety principles in action. In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed.[3]

Table of Contents I

Table of Contents II

Definition And Parameters Declaration

```rust
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Figure: An example of defining functions with parameters.[3]

Rust: An Expression-Based Language

Statements

- Statements are instructions that perform some action and do not return a value.

Creating a variable and assigning a value to it with the let keyword is a statement. Statements do not return values. Therefore, you can't assign a let statement to another variable.

```
fn main() {
    let x = (let y = 6);
}
```

Figure: Code Using The let Statement In The Wrong Way.[3]

- Different from C and other languages, where the assignment returns the value of the assignment.
- In other languages, you can write $x = y = 6$; not in Rust.

Rust: An Expression-Based Language

Expressions

- Expressions evaluate to a resultant value.[3]

Consider a math operation, such as $5 + 6$, which is an expression that evaluates to the value 11.[3]

- The 6 in the statement let $y = 6$; is an expression that evaluates to the value 6.
- Calling a function/macro is an expression.
- Scope blocks are expressions.

```rust
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

Figure: An example showcasing expressions.[3]

- The $x + 1$ line doesn't have a semicolon at the end.
- Expressions do not include ending semicolons.
- Adding semicolons turn it to a statement, which do not return a value.

Examples of Functions that Return Values

```rust
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

Figure: A function that returns a value.[3]

```rust
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Figure: A function that add 1 and returns the result.[3]

A Common Mistake When Returning Values

```rust
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

Figure: Adding Semicolon to The Function's Last Expression.[3]

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` (bin "functions") due to 1 previous error
```

Table of Contents I

Table of Contents II

How to Write Comments?

```
// hello, world
```

Figure: A simple comment[3]

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
```

Figure: Multi-Line Comment[3]

Table of Contents I

Table of Contents II

If Expression

```rust
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

Figure: An example using the if expression[3]

```rust
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

Figure: An erroneous example using the if expression[3]

Using if in a let Statement

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

Figure: Assigning the result of an if expression to a variable[3]

```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

Figure: An erroneous example using the if let statement[3]

Infinite Loop

```rust
fn main() {
    loop {
        println!("again!");
    }
}
```

Figure: An example using the loop keyword[3]

Returning Values from Loops

```rust
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Figure: An example that returns a value in a loop[3]

Loop Labels

```rust
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Figure: An example using loop labels[3]

Conditional Loops with while

```rust
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

Figure: Using a while loop to run code while a condition holds true[3]

Looping Through a Collection with for

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

Figure: Looping through each element of a collection using a for loop[3]

Table of Contents I

Table of Contents II

Understanding

Why?

- Making memory safety guarantees without needing a garbage collector.[3]
- None of the features of ownership will slow down your program while it's running.[3]

Topics to talk about:

- Borrowing.
- Slices.
- How Rust lays data out in memory.

What?

- A set of rules governing how a Rust program manages memory.[3]
- If any of the rules are violated, the program won't compile.[3]
- The main purpose of ownership is to manage heap data.[3]

Ownership rules:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

The memory is automatically returned once the variable that owns it goes out of scope.[3]

```
    {
        let s = String::from("hello"); // s is valid from this point forward

        // do stuff with s
    }                                  // this scope is now over, and s is no
                                       // longer valid
```

Figure: An example of declaring a string in heap.[3]

After going out of scope, Rust calls the *drop* function.

Assignment of Variables

```rust
let x = 5;
let y = x;
```

Figure: Assigning the integer value of variable x to y.[3]

Because integers are simple values with a known, fixed size, and these two 5 values are pushed onto the stack.[3]

```rust
let s1 = String::from("hello");
let s2 = s1;
```

Figure: Doing the same with the String type.[3]

- These two are not the same.
- The second line does not make a copy of the value in s1 and does not bind it to s2.

What Happens to String?

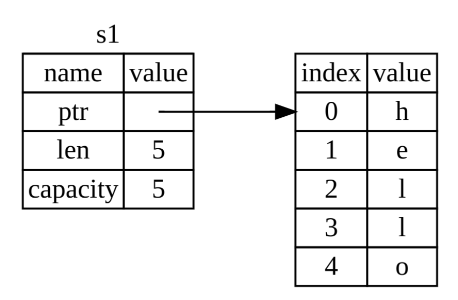

Figure: Representation in memory of a String holding the value "hello" bound to s1[3]

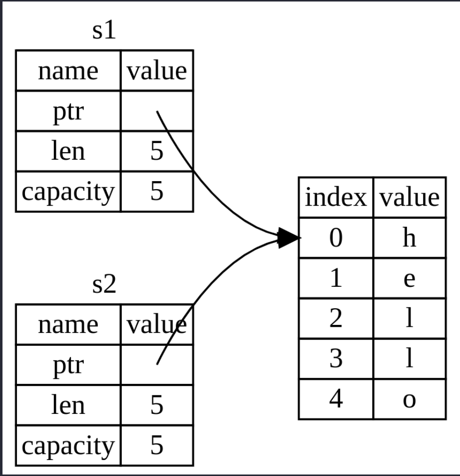

Figure: Representation in memory of the variable s2 that has a copy of the pointer, length, and capacity of s1[3]

Double Free Error

Both s1 and s2 point to the same memory location, leading to two times calling the drop function when they go out of scope.
To solve this problem and ensure memory safety:

- After the line let s2 = s1;, Rust considers s1 as no longer valid.
- Therefore, Rust doesn't need to free anything when s1 goes out of scope.

```rust
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Figure: Erroneous code that tries to copy s1 to s2[3]

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not impl
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which co
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

Figure: Rust prevents using an invalidated reference[3]

Shallow Copy vs. Move

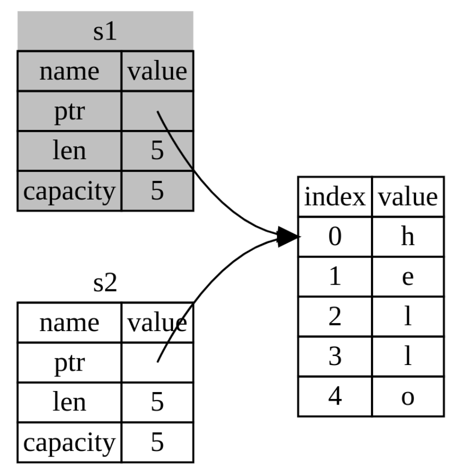

Figure: Representation in memory after s1 has been invalidated[3]

- Shallow Copy and Deep Copy
- In Rust it is a *move* because the first variable gets invalidated.[3]
- We say that s1 was *moved* into s2.[3]
- Rust will never automatically create "deep" copies of your data.[3]
- Therefore, any automatic copying can be assumed to be inexpensive in terms of runtime performance.[3]

Using Clone to Copy Deeply

If copying the heap data is desired, we can use a method called clone.

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Figure: Using The Clone Method[3]

- When you see a call to clone, you know that some arbitrary code is being executed and that code may be expensive.[3]

Types with The Copy Trait

Why in this code, x is still valid and wasn't moved into y?

```rust
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

Figure: An example of copying variables with simple types[3]

- They are stored entirely on the stack.
- Copying of these variables is quick and inexpensive.

For variables that are stored in the stack, Rust has a special annotation called the Copy trait. If a type implements this trait, variables that use it do not move.

- All the integer types, such as u32.
- The Boolean type, bool, with values true and false.
- Tuples, if they only contain types that also implement Copy. For example, (i32, i32) implements Copy, but (i32, String) does not.

Passing is Like Assigning

Passing a variable to a function will move or copy, just as assignment does.[3]

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Figure: Functions with ownership and scope annotated[3]

Return Values and Scope

Returning values can also transfer ownership.[3]

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                              // some_string is returned and
                                             // moves out to the calling
                                             // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope

    a_string  // a_string is returned and moves out to the calling function
}
```

Figure: Transferring ownership of return values[3]

It's quite annoying that anything we pass in also needs to be passed back if we want to use it again.[3]
Rust allows returning multiple values as a tuple:

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Figure: Returning ownership of parameters

But this is too much work, so a feature is needed to address this problem.

Table of Contents I

Table of Contents II

References

- Like pointers.
- That data is owned by some other variable.
- Unlike a pointer, a reference is guaranteed to point to a valid value.

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Figure: Using a reference to pass the value without loosing ownership[3]

References

In The Memory

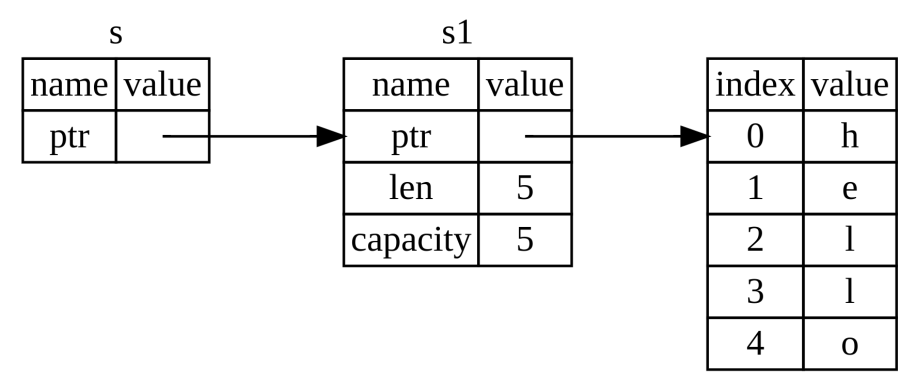

Figure: A diagram of &String s pointing at String s1[3]

This action of creating a reference is called borrowing. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back. You don't own it.[3]

Table of Contents I

Table of Contents II

String Slices: A Reference to A Part of A String

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

Figure: Using Slices[3]

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

Figure: Slices without the first number[3]

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

Figure: Slices without the second number[3]

Slices In Memory

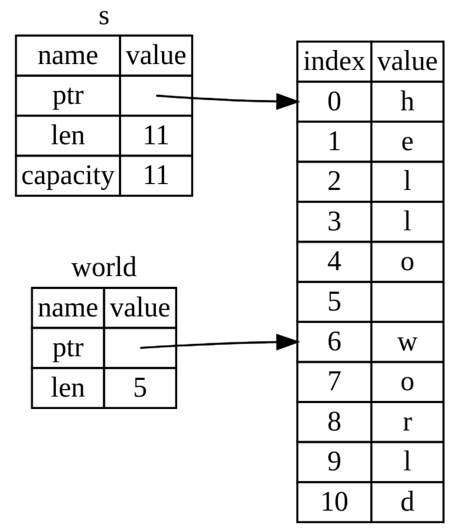

Figure: String slice referring to part of a String[3]

Table of Contents I

Table of Contents II

```rust
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Figure: A User struct definition[3]

```rust
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

Figure: Creating an instance of the User struct[3]

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

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Figure: Defining an area method on the Rectangle struct[3]

Table of Contents I

Table of Contents II

```rust
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

Figure: an enum and declaring two variables with it[3]

The match Control Flow Construct

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Figure: An enum and a match expression that has the variants of the enum as its patterns[3]

Table of Contents I

Table of Contents II

How to Read A File

```rust
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

Figure: Reading the contents of the file specified by the second argument[3]

Collecting Command-Line Arguments Into A Vector

```rust
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

Figure: Creating variables to hold the query argument and file path argument[3]

Table of Contents I

Table of Contents II

```rust
let v: Vec<i32> = Vec::new();
```

Figure: Creating a new, empty vector to hold values of type i32[3]

```rust
let v = vec![1, 2, 3];
```

Figure: Creating a new vector containing values[3]

```rust
let v = vec![100, 32, 57];
for i in &v {
    println!("{i}");
}
```

Figure: Printing each element in a vector by iterating over the elements using a for loop[3]

PNGWING, "Rust icon." https://www.pngwing.com.

Rust, "Rust website." https://www.rust-lang.org.

S. Klabnik and C. Nichols, *The Rust Programming Language*. International series of monographs on physics, William Pollock, 2022.