The Rust Programming Language

Jaideep Ganguly

September 29, 2020

Reference Book

Moving to the Rust programming language by Jaideep Ganguly
free download from https://jganguly.github.io

Agenda

1. Fundamental Principles behind Rust
2. A Quick Comparison with Go
3. Cargo Package Manager
4. Ownership, Borrowing, Referencing
5. Struct & Trait & Trait Bound
6. Enum, Pattern Matching & Error Handling
7. Closure
8. Smart Pointer
9. Concurrency

Fundamental Principles behind Rust

1. Ownership and safe borrowing of data
2. Functions, methods and closures to operate on data
3. Tuples, structs and enums to aggregate data
4. Matching pattern to select and destructure data
5. Traits to define behavior on data

A comparison with Go

1. Go is incredibly easy to learn. Go is small. Finding and using libraries from the ecosystem is very easy. Rust has a steeper learning curve.
2. Go takes lots of bits from other languages and improves them, there is little new. Rust has unique concepts.
3. Go compiles faster than Rust. Rust runs much faster than Go. Rust runs as well or faster than C. Rust performance
4. Go is Garbage collected, Rust is not. Go is not a systems language, Rust is. Go has no Macros, no Generics. Go code, including error handling, becomes repetitive quickly. Go Interfaces are not sophisticated. Go has no first class enums. Go switch may be non-exhaustive. Go routines and channels have lightweight syntax for spawning Go routines.

Why Rust?

1. **Rust is safe by default**. All memory accesses are checked and it is not possible to corrupt memory by accident.

2. With direct access to hardware and memory, Rust is an ideal language for embedded and bare-metal development. One can write extremely low-level code such as OS kernels. It is also a very pleasant language to write application code as well.

3. Rust's strong type system and emphasis on memory safety, all enforced at compile time, mean that it is **extremely common to get errors when compiling your code**.

4. If a program has been written so that no possible execution can exhibit undefined behavior, we say that program is well defined. If a language's safety checks ensure that every program is well defined, we say that language is type safe. **Rust is Type Safe. Rust guarantees that concurrent code is free of data races.**

Why Rust?

5. Rust strives to have as many **zero-cost abstractions** as possible, abstractions that are as equally performant as corresponding hand-written code. Zero-cost abstraction means that there's no extra runtime overhead for certain powerful abstractions or safety features that you do have to pay a runtime cost for other languages.

6. Rust strives to have a very fast run time. It does this in part by compiling to an executable and **injecting only a very minimal language runtime and does not provide a memory manager**, i.e., garbage collector that operates during the executable's runtime.

7. Rust gives you the choice of storing data on the stack or on the heap and determines at compile time when memory is no longer needed and can be cleaned up. **This allows efficient usage of memory as as well more performant memory access**.

1. Ownership begins with assignment and ends with scope. When a variable goes out of scope, its associated value, if any, is **dropped**. A dropped value can never be used again because the resources it uses are immediately freed. However, a value can be dropped before the end of a scope if the compiler determines that the owner is no longer used within the scope.

```rust
pub fn scope() {
    {
        let x = 1;
        println!("x: {}", x);
    }

    // println!("x: {}", x); // ERROR
}
```

Reassignment is a Move

2. Reassignment of ownership (as in **let b = a)** is known as a **move**. A move causes the former assignee to become uninitialized and therefore not usable in the future.

```rust
pub fn reassignfail() {
  let a = vec![1, 2, 3];  // a growable array literal
  let b = a;              // a can no longer be used beyond this line
  println!("b: {:?}", b);
  // println!("a: {:?}", a); // ERROR
}
```

3. Another form of reassignment occurs while returning a value from a function. But this will work as functions no longer have ownership of the returned values once its scope ends.

```rust
pub fn inc_vec(x: i32) -> Vec<i32> {
  let result = vec![x, x+1, x+2, x+3, x+4]; // allocated on heap
  result
}
```

Reassignment of Stack Variables

❹ **During reassignment, for variables in the stack, instead of moving the values owned by the variables, their values are copied.** The following code will work correctly.

```
pub fn copy_trait_example() {
  let a = 42;
    let b = 94;
    let c = a + b;
    println!("The sum of {} and {} is {}", a, b, c); // NO ERROR
}
```

❺ To do deep copy the heap data of the String, use **clone** .

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

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

Struct & Copy

6. Structs do not implement Copy by default. Reassignment of a struct variable leads to a move, not a copy. However, it is possible to automatically derive the Copy and Clone trait as follows.

```
pub fn struct_copy_example() {

    #[derive(Debug,Clone,Copy)]
    struct Person {
        age: i8
    }

    let alice = Person { age: 42 };
  let bob = alice;

  println!("alice: {:?}\nbob: {:?}", alice, bob);
}
```

7. Many resources are too expensive in terms of time or memory be copied for every reassignment. Rust offers the option to borrow using **&**.

```
pub fn ref_example() {
    let s = String::from("hello");
  let len = calculate_length(&s);
  println!("The length of '{}' is {}.", s, len);  // no error
}
```

8. To mutate a reference, annotate the type with **mut** in the caller function and with **&mut** in the function arguments.

Listing 1: Mutable reference

```
pub fn mut_ref_example() {
  let mut s = String::from("Hello");
    change(&mut s);
    println!("{}",s);
}
```

Mutable Reference Restrictons

9. But mutable references have one big restriction. **You can have only one mutable reference to a particular piece of data in a particular scope.**

```
pub fn mut_ref_restrict() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    // ERROR: will not compile
    // cannot borrow 's' as mutable more than once at a time
    println!("{}, {}", r1, r2);
}
```

Mutable Reference Restrictons

10. **We also cannot have a mutable reference while we have an immutable one.**

Listing 3: Mutable reference restriction

```
pub fn mut_ref_restrict2() {
    let mut s = String::from("hello");

    // ERROR: will not compile
    // cannot borrow 's' as mutable because it is also borrowed as
        immutable.
    let r1 = &s;     // no problem
    let r2 = &mut s; // problem

    println!("{}, {}", r1, r2);
}
```

Mutable References

⓫ However, the following code will work because the last usage of the immutable references occurs before the mutable reference is introduced.

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

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{} and {}", r1, r2);
// r1 and r2 are no longer used after this point

let r3 = &mut s; // no problem
println!("{}", r3);
```

These restrictions prevent data races at compile time which can happen if (a) two or more pointers access the same data at the same time, (b) at least one of the pointers is being used to write to the data, (c) there is no mechanism being used to synchronize access to the data.

Dangling Reference

12. The Rust compiler guarantees that references will never be dangling references.

```
fn dangle() -> &String { // ERROR: will not compile
  let s = String::from("hello");
  &s
}
```

13. The solution here is to simply return the String directly.

```
fn no_dangle() -> String {
  let s = String::from("hello");
  s
}
```

Slices

14 Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. **Slices do not have ownership**.

Listing 4: Slice

```
    let s = String::from("hello world");
    let hello = &s[0..5];
    let world = &s[6..11];
```

Lifetime

15. Sometimes we will want a function to return a borrowed value.

```
pub fn lifetime_example(x: &str, y: &str) -> &str { // Error
    if x.bytes().len() > y.bytes().len() {
        x
    } else {
        y
    }
}
```

16. With **lifetime**, the compiler is able to determine that the valid scope of the value whose borrowed reference it returns, matches the lifetime of the parameters x and y.

```
pub fn lifetime_example<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.bytes().len() > y.bytes().len() {
        x
    } else {
        y
    }
}
```

Struct & Method

1. A struct allows us to group related code together and model our application after entities in the real world.

```rust
// 'derive'  auto creates implementation to print the struct
#[derive(Debug)]
struct Rect {
  width: u32,
  height: u32,
}
impl Rect {
  fn area(&self) -> u32 {
    self.width * self.height
  }
}

fn main() {
  let rect = Rect {
    width: 30,
    height: 50,
  };
  println!("The area of the rectangle is {}", rect.area());
}
```

Trait

1. A **trait** is an equivalent of a Java interface.

```
pub trait Animal {
    fn eat(&self) {
        println!("I eat grass");
    }
}

pub struct Herbivore;

impl Animal for Herbivore{
    fn eat(&self) {
        println!("I eat plants");
    }
}

pub struct Carnivore;

impl Animal for Carnivore {
    fn eat(&self) {
        println!("I eat meat");
    }
}
```

Trait ... Contd.

2. Usage:

```
use tra::Animal;

let h = tra::Herbivore;
h.eat();

let c = tra::Carnivore;
c.eat();
```

Trait Bound

3. Consider the following.

```
pub trait Activity {
    fn fly(&self);
}

#[derive(Debug)]
pub struct Eagle;

impl Activity for Eagle {
    fn fly(&self) {
         println!("{:?} is flying",&self);
    }
}

pub fn activity<T: Activity + std::fmt::Debug>(bird: T) {
    println!("I fly as an {:?}",bird);
}
```

Trait Bound ... Contd.

4. But adding the following line will result in a compile error. This is because the **struct Hen** does not implement the **trait** **Activity**.

```
        let hen = tra::Hen;
        tra::activity(hen);   // COMPILE  ERROR
```

5. The function **activity** takes a generic **T** as an argument, the generic **T** must implement **trait Activity**. **Trait bounds** allow a a function to only **accept types** that **implement** a certain **trait**.

6. **Any invocation of the function with an instance of a struct that does not implement the trait will result in a compile error**. Such a function is said to be **trait bound**.

Trait Object

7. **Trait objects** behave more like traditional objects, they contain both **data** and **behavior**. **In trait objects, the data is referenced through a pointer to the data that is actually stored in the heap.**

8. The size of a trait is not known at compile-time. Therefore, traits have to be wrapped inside a **Box** when creating a vector trait object. A trait object is an object that can contain objects of different types at the same time (e.g., a vector). The **dyn** keyword is used when declaring a trait object. So,
   1. **Box<Trait>** becomes **Box<dyn Trait>**
   2. **&Trait** and **&mut** Trait become **&dyn Trait** and **&mut dyn Trait**

9. Various struct

```
pub struct Hen;

#[derive(Debug)]
pub struct Horse;

#[derive(Debug)]
pub struct Deer;

#[derive(Debug)]
pub struct Tiger;

#[derive(Debug)]
pub struct Duck;
```

```
pub trait Sound {
    fn sound(&self);
}
```

Trait Object ... Contd.

10 Implementations

```
impl Sound for Horse {
    fn sound(&self) {
        println!("{:?} neighs",&self)
    }
}

impl Sound for Deer {
    fn sound(&self) {
        println!("{:?} barks",&self)
    }
}

impl Sound for Tiger {
    fn sound(&self) {
        println!("{:?} roars",&self)
    }
}
```

Trait Object ... Contd.

11 Implementations

```
impl Sound for Duck {
    fn sound(&self) {
        println!("{:?} quacks",&self)
    }
}

pub struct SoundBook {
    pub sounds: Vec<Box<dyn Sound>>
}

impl SoundBook {

    pub fn run(&self) {
        for s in self.sounds.iter() {
            s.sound();
        }
    }
}
```

Enum

1. Rust enums can contain context, it can be a different for each variant of the enum. We can put data directly into each **enum** variant.

```
#[derive(Debug)]
pub struct MyBlack {
    pub name: String,
    pub rgb: (u8,u8,u8)
}

#[derive(Debug)]
pub enum Color {
    Black(MyBlack),
    White(u8,u8,u8)
}

impl Color {
    pub fn printColor(&self) {
        println!("Hi!");
    }
}
```

Enum & Matching

2. Invoking:

```
let my_black = enu::MyBlack {
  name: String::from("my black"),
  rgb: (10,10,10)
};
let black = enu::Color::Black(my_black);
let white = enu::Color::White(255,255,255);
println!("{:?}",black);
println!("{:?}",white);
```

3. **match** can be used to compare values stored in an **enum**.

```
match black {
    fn_07_enu::Color::White(x,y,z) => println!("{} {} {}",x,y,
        z),
    fn_07_enu::Color::Black(x) => println!("{:?}",x.rgb),
}
```

Option Enum

4 **Option** is a predefined **enum** in the Rust standard library.

```
enum Option<T> {
  Some(T), // used to return a value
  None     // used to indicate null, Rust does not support
      null
}
```

Rust does not support the null keyword.

```
let x: Option<u32> = Some(2);
assert_eq!(x.is_some(), true);

let x: Option<u32> = None;
assert_eq!(x.is_some(), false);

let y = x.unwrap();  // unwraps and gets the value
```

Matches are Exhaustive

5. Matches in Rust are exhaustive. We use the special pattern `_` instead to handle the rest. The `()` is just the unit value.

```
    let some_u8_value = 4u8;
    match some_u8_value {
        1 => println!("One"),
        3 => println!("Three"),
        5 => println!("Five"),
        7 => println!("Seven"),
        9 => println!("Nine"),
        _ => (),
    }
```

6. **if let** .

```
fn main() {
  let some_u8_value = Some(0u8);
  if let Some(3) = some_u8_value {
    println!("three");
  }
}
```

Error Handling with Result Enum

7. The enum **Result <T,E>** is used to handle recoverable errors.

```
enum Result<T,E> {
  OK(T),
  Err(E)
}
```

```
use std::fs::File;
let f = File::open("mypicture.jpg");   // file does not
    exist
match f {
  Ok(f)=> {
    println!("file found {:?}",f);
  },
  Err(e)=> {
    println!("file not found \n{:?}",e);   //handled error
  }
}
println!("I will print this");
```

Macro

1. An awesome and powerful feature of Rust is its ability to use and create macros. Macros are created using **macro_rules!**

```
macro_rules! hi {
  ($name : expr) => {
    println!("Hi {:?}", $name);
  };
}
```

2. Macros simply allows you to invent your own syntax and write code that writes more code. This is called metaprogramming, which allows for syntactic sugars that make your code shorter and make it easier to use your libraries. You could even create your own **DSL** (Domain-Specific Language) within rust.

Macro ... Contd.

3. Many of the macros can take multiple inputs.

```
macro_rules! map {
  ( $($key : expr => $value : expr), * ) => {{
      let mut hm = HashMap::new();
      $( hm.insert($key,$value); )*
      hm
  }};
}
```

```
use std::collections::HashMap;
let person = map! (
  "name" => "Tim",
  "gender" => "male"
);
println!("{:?}", person);
```

```
{"name": "Tim", "gender": "male"}
```

4. In Rust, **println!** and **vec!** are **macros**.

Closure

5. Rust's closures are anonymous functions that can be saved in a variable or can be passed as arguments to other functions.

6. Closures do not require annotating the types of the parameters or the return values.

7. Closures are not exposed through interfaces.

```
    let some_closure = |number: u32| -> u32 {
      println!("calculating ...");
      thread::sleep(Duration::from_secs(3));
      number + 1
    };
```

8. To define a closure, we start with a pair of vertical pipes | , inside which we specify the parameters to the closure. This syntax is similar to the closure definitions in **Smalltalk** and **Ruby** languages.

Function Receiving a Closure

9. Example with traits **FnOnce (self)**, **FnMut (&mut self)**, **Fn (&self)**

```
pub fn closure_example3(x:i32) -> i32 {

  let y = 3;
  let add = |x| {
    x + y
  };
  let result = receive_closure(add, x);
  result
}

// function receives a closure and returns an i32
fn receive_closure<F>(f: F, x: i32) -> i32
where
F: Fn(i32) -> i32
{
  f(x) as i32
}
```

```
let result = fn_11_clo::closure_example3(5);
println!("Result from closure is {}",result);
```

struct Cacher

10 struct

```rust
struct Cacher<T>
where
    T: Fn(u32) -> u32,  // trait bound
{
    calc: T,       // calc stores the closure that is trait bound
    value: Option<u32>, // Result of calling the function calc
}
```

11 Memoization or Lazy Evaluation Pattern

```
impl<T> Cacher<T>
where
  T: Fn(u32) -> u32 , // trait bound
{
  fn new(calc: T) -> Cacher<T> {
    Cacher {    // expression returning the function
      calc,
      value: None
    }
  }

  fn func(&mut self, arg: u32) -> u32 {
    match self.value {
      Some(v) => v, // value exists, return v
      None => {   // value does not exit
        let v = (self.calc)(arg); // invoke calc with arg
        self.value = Some(v);   // wrap value in Option
        v              // return v
      }
    }
  }
```

Cacher Example

12 Example usage of **Cacher**

```
use std::thread;
use std::time::Duration;
use core::fmt;

pub fn generate_force(hp: u32, random_number: u32) {

    let mut my_closure = Cacher::new(|number| {
        println!("calculating HP ...");
        thread::sleep(Duration::from_secs(1));
        number
    });
```

13 Example usage of **Cacher**

```
    if hp < 25 {
        println!("Low HP drive slow {}", my_closure.func(hp));
        println!("Low HP drive steady {}", my_closure.func(hp));
    } else {

        if random_number == 3 {
            println!("No HP generated");
        } else {
            println!(
                "Sufficient HP {}", my_closure.func(hp)
            );
        }
    }
}
```

Cacher Example

14 Example usage of **Cacher**

```
    if hp < 25 {
        println!("Low HP drive slow {}", my_closure.func(hp));
        println!("Low HP drive steady {}", my_closure.func(hp));
    } else {

        if random_number == 3 {
            println!("No HP generated");
        } else {
            println!(
                "Sufficient HP {}", my_closure.func(hp)
            );
        }
    }
}
```

Smart Pointers

1. References are pointers that only borrow data.
2. **Smart pointers, in many cases, own the data they point to.** Smart pointers are mostly implemented using structs. These structs implement the **Deref** and **Drop** traits.
3. The **Deref** trait allows an instance of the smart pointer struct to behave like a reference so that the code works with either references or smart pointers.
4. The **Drop** trait allows us to customize the code that is run when an instance of the smart pointer goes out of scope.

Smart Pointers

5. **Boxes allow you to store data on the heap rather than the stack.** What remains on the stack is the pointer to the heap data. Boxes do not have any performance overhead, other than storing their data on the heap instead of on the stack. **Box** is useful under these circumstances.
6. A type whose size is unknown at compile time and we want to use a value of that type in a context that requires an exact size.
7. A large amount of data, we want to transfer ownership but do not want the data to be copied.
8. Own a value and its type must implement a certain trait rather being of a particular type.

```
    let x = Box::new(100);
    println!("x = {}", x);
```

Deref trait

9. Implementing the **Deref** trait allows you to customize the behavior of the dereference operator.

```
#[derive(Debug)]
struct MyBox<T> { // same as: struct MyBox<T>(T);
  a: T
}
```

```
use std::ops::Deref;
impl<T> Deref for MyBox<T> {
  type Target = T;

  fn deref(&self) -> &T {
    &self.a
  }
}
```

```
let x = MyBox{a:100};
println!("{}",*(x.deref()));
```

Drop trait

10. In languages such as C/C++, the programmer must call code to free memory or resources every time they finish using an instance of a smart pointer. If they forget, the system might become overloaded and crash.

11. In Rust, you specify a particular bit of code be run whenever a value goes out of scope and the compiler will insert this code automatically when you implement the **Drop** trait.

12. While implementing the **Drop** trait on a type, you can specify what needs to happen which can include activities such as releasing resources such as files, network connections, DB connections, etc.

Drop trait

13. Implement Drop

```
struct mysmaptr {
  data: String
}
```

```
impl Drop for mysmaptr {
  fn drop(&mut self) {
    println!("Dropping struct mysmaptr with data {}", self.data);
  }
}
```

```
let x = mysmaptr{ data : String::from("Hello") };
println!("struct mysmaptr with data {}", x.data);
```

Concurrency

1. Developers of Rust discovered that the ownership and type systems are the keys to help manage memory safety and address concurrency problems.
2. By leveraging Rust's unique concept of ownership and type checking, many concurrency errors are reduced to compile-time errors in Rust rather than runtime errors.
3. Rust developers have nicknamed this aspect of Rust as fearless concurrency. Fearless concurrency allows you to write code that is free of subtle bugs and is easy to refactor without introducing new bugs.

4. No risk of forgetting to unlock the mutex

```
pub fn mutex_example() {
    let counter = Arc::new(Mutex::new(0)); // atomic ref count
    let mut handles = vec![];   // stores references to the
        threads

    for _ in 0..10 {
        let counter = Arc::clone(&counter); // clone the arc

        // use the move closure and spawn 10 threads
        let handle = thread::spawn( move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
    // join the threads
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Mutex

5. **Mutex<T>** is a smart pointer.

6. The call to lock returns a smart pointer called **MutexGuard**, wrapped in a **LockResult** that is handled with the call to **unwrap**.

7. The **MutexGuard** can be dereferenced to point to the data.

8. The **MutexGuard** has a drop implementation that releases the lock once **MutexGuard** goes out of scope. **With this, we do not risk forgetting to unlock the mutex because this is done automatically in Rust.**

9. The smart pointer **Arc<T>**, an **Atomically Referenced Counted** type. It is needed for thread safety in multi-threaded programs.

Channel

10. An increasingly popular approach to ensuring safe concurrency is message passing, where **threads** or **actors** communicate by sending each other messages containing data.

11. **"Do not communicate by sharing memory; instead, share memory by communicating."**

12. Rust has implementation of a **channel** to send and receive messages between concurrent sections of the code. A **channel** has two halves, a transmitter and a receiver. Let's look at the following code that has multiple producers of messages and a single receiver.

13 Channel

```rust
pub fn concur_example2() {
    // multiple producer, single consumer
    let (tx, rx) = mpsc::channel();

    // clone a second producer
    let tx2 = mpsc::Sender::clone(&tx);

    // spawn a thread and move the transmitter into the closure
    // spawned thread will now own the transmitter
    thread::spawn( move || {
        let vals = vec![
            String::from("Hello"),
            String::from("from"),
            String::from("thread-1"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
```

14 Channel

```
    // same comments of the previous code block apply here.
    thread::spawn( move || {
        let vals = vec![
            String::from("Hi"),
            String::from("your"),
            String::from("thread-2"),
        ];

        for val in vals {
            tx2.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    // receive the result, timeout beyond 1 sec
    let result =
        rx.recv_timeout(Duration::from_millis(1000));
```

15 Channel

```
            rx.recv_timeout(Duration::from_millis(1000));

    match result {
        Err(e) => {
            println!("{:?}",e);
            process::exit(0);
        },
        Ok(x) => {
            for received in rx {
                println!("Got: {}", received);
            }
        }
    }
}
```

Async

16 Async/await are special pieces of Rust syntax that make it possible to yield control of the current thread rather than blocking it allowing other code to make progress while waiting on an operation to complete.

17 The async/await syntax lets you write code that feels syn-chronous but is actually asynchronous. In Rust, deferred computations due to "long" running programs are called futures.

18 While most of the concepts are fairly similar with other programming languages, in Rust you need to pick a runtime to actually run your asynchronous code.

19 The de facto standard library providing a runtime system for green threads and asynchronous I/O is **tokio** which we will use. It has zero-cost abstractions and delivers bare-metal performance.

Async

20. Async/Await

Listing 5: Async/Await example

```
use std::error::Error;
use std::time::{Duration, Instant};
use std::thread;
use futures::future;
use futures::join;
use tokio::macros::support::Future;
```

21 Async/Await

Listing 6: Async/Await example

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {

    // Sequential execution
    let t1 = Instant::now();
    let mut x1 = 100;
    let r1 = long_running_fn_1(&mut x1).await;
    let r2 = long_running_fn_2().await;
    let t2 = Instant::now(); println!("{} {} {:?}",r1,r2,t2-t1);

    // Concurrent execution
    let tasks = vec![
        tokio::spawn(async move { long_running_fn_1(&mut x1).await
            }),
        tokio::spawn(async move { long_running_fn_2().await }),
    ];
```

Async

22. Async/Await

Listing 7: Async/Await example

```
    // join the tasks
    let t1 = Instant::now();
    let r = futures::future::join_all(tasks).await; let t2 =
        Instant::now();
    println!("{:?} {:?}",r,t2-t1);
    Ok(())
}
```

Async

23 Async/Await

Listing 8: Async/Await example

```
fn long_running_fn_1(x: &mut i32) -> impl Future<Output = i32> {
    thread::sleep(Duration::from_secs(1));
    *x = *x + 1;
    future::ready(*x)
}

async fn long_running_fn_2() -> i32 {
    thread::sleep(Duration::from_secs(3));
    42
}
```

Questions?

Thank you!