Rust Crash Course

Part3

Agenda

- Error handling

- Ownership
- Borrowing
- Life times

- Smart pointers
- Global variables

Most content & examples from https://tourofrust.com

Error handling

- Most languages handle errors via one of two ways:
  - Try/catch exceptions (Python, Java, JavaScript, etc.)
  - Returning a separate error value (Go)
- Many times, errors are not handled properly or are tedious to handle.
- Rust tries to make error handling easier. It's not always smooth sailing though.

Error handling

- The idiomatic way to handle errors in Rust is via `Result<T, E>`:

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

- Functions that may not complete successfully should return a `Result`.
  - If the result is `Ok`, the caller can access the returned value (of generic type `T`).
  - If the result is `Err`, additional information (of generic type `E`) about the error is returned.

- If a program cannot recover from an error,
  you can `panic!` instead of returning a `Result`.

Error handling

- Example:

```rust
use std::fs::File;

fn main() {
   let f = File::open("hello.txt");
   let f = match f {
      Ok(file) => file,
      Err(error) => panic!("Problem opening the file: {:?}", error),
   };
}
```

- Opening a file can fail, so `File::open` returns a `Result`.
- We check if the `open` was successful, if not, we panic

Error handling

- Matching on `Results` all the time can be tedious.
- If we know we are going to panic on an `Err`, we can use `unwrap()` instead:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

- `unwrap()` panics on error; otherwise, it returns the data contained in the `Ok` variant.

Error handling

- Another shortcut is the `?` operator:

```
use std::fs::File; use std::io; use std::io::Read;

fn read_file() -> Result<String, io::Error> {
   let mut f = File::open("hello.txt")?;
   let mut s = String::new();
   f.read_to_string(&mut s)?;
   Ok(s)
}
```

- If a result is an `Err`, `?` causes the function to return that `Err`.
- Otherwise, `?` unwraps the `OK` variant.

Owner

- Instantiating a type and **binding** it to a variable name creates a memory resource that the Rust compiler will validate through its whole **lifetime**.
- The bound variable is called the resource's **owner**.
- Example:
  - Variable `foo` is the owner of a instantiated type `struct Foo`

```
struct Foo {
    x: i32,
}

fn main() {
    // We instantiate structs and bind to variables
    // to create memory resources
    let foo = Foo { x: 42 };
    // 'foo' is the owner
}
```

Scope-Based Resource Management

- Rust uses the end of scope as the place to deallocate a resource.
- The term for this deallocation is called a **drop**.

```
struct Foo {
    x: i32,
}

fn main() {
    // We instantiate structs and bind to variables
    // to create memory resources
    let foo = Foo { x: 42 };
    // 'foo' is the owner

    println!("{}", foo.x);
    // 'foo' is dropped here
}
```

Dropping is Hierarchical

- When a struct is dropped, the struct itself is dropped first, then its children are dropped individually, and so on.

```rust
struct Bar {
    x: i32,
}

struct Foo {
    bar: Bar,
}

fn main() {
    let foo = Foo { bar: Bar { x: 42 } };
    println!("{}", foo.bar.x);
    // 'foo' is dropped first
    // then 'foo.bar' is dropped
}
```

Moving Ownership

- When an owner is passed as an argument to a function, ownership is moved to the function parameter.
- After a **move** the variable in the original function can no longer be used.
- During a move the stack memory of the owners value is copied to the function call's parameter stack memory.

```rust
struct Foo {
    x: i32,
}

fn do_something(f: Foo) {
    println!("{}", f.x);
    // 'f' is dropped here
}

fn main() {
    let foo = Foo { x: 42 };
    // 'foo' is moved to 'do_something'
    do_something(foo);
    // 'foo' can no longer be used
}
```

Returning Ownership

- Ownership can also be returned from a function.

```rust
struct Foo {
    x: i32,
}

fn do_something() -> Foo {
    Foo { x: 42 }
    // ownership of 'Foo' is moved out
}

fn main() {
    let foo = do_something();
    // 'foo' becomes the owner
    // 'foo' is dropped because of end of function scope
}
```

Borrowing ownership with references

- References allow us borrow read-only access to a resource with the & operator.
- References are also dropped like other resources.
- Example:
  - We can access the instantiated struct through `foo` and `f`

```rust
#[derive(Debug)]
struct Foo {
    x: i32,
}

fn main() {
    let foo = Foo { x: 42 };
    let f = &foo;
    println!("{}", f.x);
    // 'f' is dropped here
    println!("{:?}", foo);
    // 'foo' is dropped here
}
```

Borrowing ownership with mutable references

- We can also borrow mutable access to a resource with the &mut operator.
- A resource owner cannot be moved or modified while mutably borrowed.
- Memory detail:
  - Rust prevents having two ways to mutate an owned value because it introduces the possibility of a data race.

```rust
struct Foo {
    x: i32,
}

fn do_something(f: Foo) {
    println!("{}", f.x);
    // 'f' is dropped here
}
```

```rust
fn main() {
    let mut foo = Foo { x: 42 };
    let f = &mut foo;

    do_something(foo);

    // FAILURE: 'do_something(foo)' would fail
    // because 'foo' cannot be moved here
    // while mutably borrowed to 'f'
```

Borrowing ownership with mutable references

Heinrich Heine Universität Düsseldorf

- We can also borrow mutable access to a resource with the &mut operator.
- A resource owner cannot be moved or modified while mutably borrowed.
- Memory detail:
  - Rust prevents having two ways to mutate an owned value because this could cause data races.

```
struct Foo {
    x: i32,
}

fn do_something(f: Foo) {
    println!("{}", f.x);
    // f is dropped here
}
```

```
fn main() {
    let mut foo = Foo { x: 42 };
    let f = &mut foo;

    f.x = 13;
    // 'f' is dropped here because it's no
    // longer used after this point

    // move ownership of 'foo' to a function is OK
    do_something(foo);
}
```

Dereferencing for copyable types

- Dereferencing is done using the * operator
  - You can read/write the owner's value
  - Or you get a copy of an owned value, if the value has copyable type

test.rs

```
fn main() {
    let mut foo = 42;
    let f = &mut foo;
    let bar = *f; // get a copy of the owner's value
    *f = 13; // set the reference's owner's value
    println!("{}", bar);
    println!("{}", foo);
}
```

```
MBP22:tmp mschoett1$ ./test
42
13
```

Dereferencing for non-copyable types

- Dereferencing is done using the * operator
  - Or the value is moved, if the value has a non-copyable type, e.g. struct types

```
#[derive(Debug)]
struct Foo {
    x: i32,
}
```

```
fn main() {
    let mut foo = Foo { x: 42 };
    let f = &mut foo;
    let bar = *f; // move occurs
    (*f).x = 13;  // set the reference's owner's value
    println!("{:?}", bar);
    println!("{:?}", foo);
}
```

Dereferencing for non-copyable types

- Dereferencing is done using the * operator
  - Or the value is moved, if the value has a non-copyable type, e.g. struct types
  - You can make struct types copyable by implementing the trait `Copy` or derive it

```
#[derive(Debug)]
#[derive(Clone)]
#[derive(Copy)]
struct Foo {
    x: i32,
}
```

```
fn main() {
    let mut foo = Foo { x: 42 };
    let f = &mut foo;
    let bar = *f; // copy occurs
    (*f).x = 13; // set the reference's owner's value
    println!("{:?}", bar);
    println!("{:?}", foo);
}
```

Dereferencing for non-copyable types

- `Copy` is implicit, inexpensive, and cannot be re-implemented.
- `Clone` is explicit, may be expensive, and may be re-implemented.
- See also Part1

Example: references that outlive referents

```rust
fn main() {
    let r;
    {
        let x = 1;
        r = &x;
    }
    println!("{}", r)
}
```

x is dropped here

Borrow no longer valid ⚡

```rust
fn main() {
    let r;
    let x;
    {
        x = 1;
        r = &x;
    }
    println!("{}", r)
}
```

Explicit Lifetimes

- Even though Rust doesn't always show it in code, the compiler understands the lifetime of every variable and will attempt to validate that a reference never exists longer than its owner.

- Functions can be explicit by parameterizing the function signature with symbols that help identify which parameters and return values share the same lifetime.

- Lifetime specifiers always start with a ' (e.g. 'a, 'b, 'c)

Explicit lifetimes

- Example:

```rust
fn smallest_number<'a>(n: &'a [i32]) -> &'a i32 {
    ...
}
```

- Again, we're basically saying:
  - For any lifetime `'a`, `smallest_number` takes a slice `&[i32]` and returns a reference to the smallest element `&i32` in that slice that has the same lifetime.
- This ensures that we can't borrow the returned reference from `smallest_number` if it doesn't life at least as long as the variable we've assigned it to.

- Thus this will not compile:
  - Rust will tell us that numbers doesn't live long enough

```rust
let s;
{
    let numbers = [2, 4, 1, 0, 9];
    s = smallest_number(&numbers);
}
println!("{}", s)
```

Lifetimes in data types

- Similarly to functions, data types can be parameterized with lifetime specifiers of its members.

- Rust validates that the containing data structure of the references never lasts longer than the owners its references point to.

- We can't have structs running around with references pointing to nothing!

Lifetimes in data types

- Example:

```rust
struct Config {
    ...
}

struct App {
    config: &Config
}
```

```rust
struct Config {
    ...
}

struct App<'a> {
    config: &'a Config
}
```

- We need to tell the compiler, that config, which is of type &Config, has the same lifetime as App.

Multiple lifetimes in data types

- Example:

```
struct Point<'a> {
    x: &'a i32,
    y: &'a i32
}
```

```
fn main() {
    let x = 3;
    let r;
    {
        let y = 4;
        let point = Point { x: &x, y: &y };
        r = point.x
    }
    println!("{}", r);
}
```

- After this update the code works:

```
struct Point<'a,'b> {
    x: &'a i32,
    y: &'b i32
}
```

Static lifetime

- The 'static lifetime is a special lifetime that represents the entire duration of the program.
- Any reference with a 'static lifetime can be used anywhere without worrying about its scope.
- Should only be used if necessary
- 'static resources will never drop.

```
struct ListNode {
    size: usize,
    next: Option<&'static mut ListNode>,
}
```

- Examples

```
// string literals have a 'static lifetime
const MSG: &'static str = "Hello World!";

fn main() {
    println!("msg = {}", MSG);
}
```

Smart Pointers

- Smart pointers in rust are similar to the ones in C++
- Basically you can overwrite the deref operators `*` and `.`
- You write code implementing traits: `Deref`, `DerefMut`, and `Drop` to specify the logic of what should happen when during dereferencing

Example: Smart Pointers

```rust
use std::ops::Deref;

struct Person {
   name: String,
   age: u32,
}

// Implement the Deref trait for the Person struct
impl Deref for Person {
  type Target = String; // Specify the type that the deref operation will produce

   fn deref(&self) -> &Self::Target {
      &self.name
   }
}

fn main() {
   let person = Person { name: String::from("Alice"), age: 30, };

   // Use the dereference operator (*) to access the name field of person
   println!("Name: {}", *person);
}
```

Example: Smart Unsafe Code


```rust
fn main() {
  let a: [u8; 4] = [86, 14, 73, 64];

  // This is a raw pointer. Getting the memory address of something as a number is safe
  let pointer_a = &a as *const u8 as usize;
  println!("Data memory location: {}", pointer_a);

  // Turning our number into a raw pointer to a f32 is also safe to do.
  let pointer_b = pointer_a as *const f32;

  // This is unsafe because we are telling the compiler to assume our pointer is
  // a valid f32 and dereference it's value into the variable b.
  // Rust has no way to verify this assumption is true.
  let b = *pointer_b;

  println!("I swear this is a pie! {}", b);
}
```

Example: Smart Unsafe Code

```rust
fn main() {
  let a: [u8; 4] = [86, 14, 73, 64];

  // This is a raw pointer. Getting the memory address of something as a number is safe
  let pointer_a = &a as *const u8 as usize;
  println!("Data memory location: {}", pointer_a);

  // Turning our number into a raw pointer to a f32 is also safe to do.
  let pointer_b = pointer_a as *const f32;

  let b = unsafe {
    // This is unsafe because we are telling the compiler to assume our pointer is
    // a valid f32 and dereference it's value into the variable b.
    // Rust has no way to verify this assumption is true.
    *pointer_b
  };
  println!("I swear this is a pie! {}", b);
}
```

Global variables

- Easy as long as they are read only

```
static LOG_LEVEL: u8 = 0;
```

- Use of mutable statics is unsafe →

```
static mut LOG_LEVEL: u8 = 0;

pub unsafe fn get_log_level() -> u8 {
    LOG_LEVEL
}
```

- The underlying problem is that a global variable is potentially visible from multiple threads.

- Solution without unsafe requires accessing static using a Mutex →
  - Lock is automatically released when `v` goes out of scope

```
use std::sync::Mutex;

static LOG_LEVEL: Mutex<u8> = Mutex::new(0);

pub fn get_log_file() -> u8 {
    let v = LOG_LEVEL.lock().unwrap();
    *v
}
```