Lecture 20

Returning References

Why Returning References Can Be Difficult?

- Assume we want to connect to an imaginary Postgres database server using a connection pool.
- The API for this imaginary library requires us to:

1- initialize a connection Pool with the connection string.

2- Then, once the pool is initialized, we call *connect()* on it to get access to a usable, owned Connection for queries.

won't compile!

```
fn connect(url: &str) -> &Connection {
    let pool: Pool = postgres::connection_pool::new(url);
    let connection: Connection = pool.connect();
    connection.set_database("startup.io");
    &connection
}
```

Why Returning References Can Be Difficult?

won't compile!

```
fn connect(url: &str) -> &Connection {
    let pool: Pool = postgres::connection_pool::new(url);
    let connection: Connection = pool.connect();
    connection.set_database("startup.io");
    &connection
}
```

- The compiler will complain about the lifetime of &Connection not lasting long enough.
- The compiler wants to deallocate the Connection object at the end of the function, but it knows that the &Connection reference we're trying to return points to that struct.

Memory Allocation

- As we write programs, all the processing data is represented in memory at some point during execution.
- In the simplest case, the newly allocated memory stores the data we assign to the variable.
- For other kinds of data, we store the memory address of another variable (pointers or references)
- Pointers exist mostly to:

  1- avoid duplicating data in memory.

  2- provide a predictably sized variable which represents unpredictably sized data (such as user input or a data from a network call).

Garbage Collectors

- Many languages use a “garbage collector” to decide when a process should free memory.
- Garbage collectors operate at “runtime,” while the program is running.
- This makes garbage collected languages easier and safer to write, but sometimes slower and with unpredictable memory usage.
- What makes Rust unique is that although it doesn’t use a garbage collector, it still guarantees that pointers will be safe to use.

Memory Safety without a Garbage Collector

- A variable owns memory when the programmer assigns data to it.
- When the owner goes out of scope and dies, that memory is deallocated.
- Because the compiler always knows when a variable goes out of scope, it always knows when to deallocate memory at compile time so Rust programs don't need to pause and garbage collect while running.
- Ownership can be transferred. Assigning one variable to another variable ***transfers*** ownership.
- We say that the value has been "***moved***." Moves in Rust copy data to a new place in memory. This is called "***move semantics***."

Memory Safety without a Garbage Collector

- You might think that if a large amount of data is moved then we're probably wasting RAM since the data must exist in at least two locations in memory.
- However, the Rust compiler heavily optimizes code when using the ***--release*** flag, so most of the time the compiler can see that we're about to waste memory and simply reuse the existing memory location instead of copying.

Memory Safety without a Garbage Collector

- Another problem introduced by move semantics is that you often want many variables to be able to access the same data from different places in your program.
- Rust also allows “borrowing” of memory by creating a pointer to it.
- If the owner goes out of scope and dies, pointers can no longer legally use that memory.
- This aspect of the Rust compiler is known as The Borrow Checker.

Back to the code sample

won't compile!

```
fn connect(url: &str) -> &Connection {
    let pool: Pool = postgres::connection_pool::new(url);
    let connection: Connection = pool.connect();
    connection.set_database("startup.io");
    &connection
}
```

- There's only one scope involved: the entire body of fn connect() {..}.
- So, at the end of connect(), all variables declared inside the function will be deallocated.
- This means that after we return the pointer to the Connection, there won't be a Connection anymore!
- That pointer will refer to freed memory after the end of the function.

Pattern 1: Return Owned Values

- Give up on references.
- The easiest way, and depending on the use case, this might be a good solution.

```
fn connect(url: &str) -> Connection {
    let pool: Pool = postgres::connection_pool::new(url);
    let connection: Connection = pool.connect();
    connection.set_database("startup.io");
    connection
}
```

Pattern 1: Return Owned Values

- Some types, like *str* and *Path* are only intended to be used with references, and they have a sibling type which is intended to be used only as an owned value.
- For these types, if we just try removing the & from our code, we'll still get compile errors. Here's an example that won't compile:

won't compile!

```
fn returns_a_path() -> Path {
    let path: Path = somehow_makes_path();
    path
}
```

Pattern 1: Return Owned Values

- For these types which can only be used as references, look for an *impl* of the trait *ToOwned*.
- *ToOwned* consumes a shared reference and copies the value into a new owned reference. Here's an example of leveraging the *ToOwned* trait:

```
fn returns_a_pathbuf() -> PathBuf {
    let path: &Path = somehow_makes_path();
    let pathbuf: PathBuf = path.to_owned();
    pathbuf
}
```

Pattern 1: Return Owned Values

Pros	Cons
• *Low Effort* - converting a return value from a shared reference to an owned value is easy. • *Widely Applicable* – whenever we can return a reference, we can return an owned copy. • *Safe* - a new copy of the value cannot corrupt memory elsewhere.	• *Synchronization* - this value won't be changed if we change the original value. • *Memory* - possibly wasting memory by making identical copies of some piece of data.

Pattern 2: Return Boxed Values

- Move the database connection off the function stack and into the heap.
- Explicitly heap allocate data using the Box struct.
- When the box itself goes out of scope, the heap allocated memory will be freed.

```
fn connect(url: &str) -> Box<Connection> {
    let pool: Pool = postgres::connection_pool::new(url);
    let connection: Connection = pool.connect();
    connection.set_database("startup.io");
    Box::new(connection)
}
```

Pattern 2: Return Boxed Values

Pros	Cons
• *Memory* - The only additional memory (beyond the boxed value) is the pointer to the heap. • *Applicability* - Almost any code which uses std can leverage this technique.	• *Indirection* - we'll need to write more code. • *Overhead* - It's more complicated to allocate memory on the heap.

Pattern 2: Return Boxed Values

- For types like *usize*, *bool*, *f32*, and other primitive types, it can be a code smell if we boxed these values (Return a copy of these types instead).
- For dynamically growable types like Vec, *String*, and *HashMap*, these types already use the heap internally so you're not necessarily gaining much by boxing them.

Pattern 3: Move Owned Values to a Higher Scope

- Reorganizing code to help us leverage references passed into functions.

```
fn setup_connection(connection: &Connection) -> &Connection
{
   connection.set_database("startup.io");
   connection
}

fn main() {
    let pool = postgres::connection_pool::new(url);
    for _ in 0..10 {
      let connection = setup_connection(&pool.connect());
      // Do something with connection here
    }
}
```

Pattern 3: Move Owned Values to a Higher Scope

- We reorganized the code by:
  - moving both the *Pool* and *Connection* objects to a scope outside the function.
  - changing the function signature to take in a reference to the connection we want to modify.
  - returning a reference to the same memory as the memory borrowed in the argument.

Pattern 3: Move Owned Values to a Higher Scope

- At the end of setup_connection(), no new structs were declared inside of it (so nothing needs to be deallocated).
- connection lives for one iteration of the loop. In the next iteration, a new connection is allocated.
- We can only get the lifetimes out of a function that we put into the function.
- No matter how many times we call setup_connection(), only one Pool object will be allocated, compared to having many pools allocated in the other patterns.

Pattern 3: Move Owned Values to a Higher Scope

Pros	Cons
• *Memory* - this pattern avoids heap allocating, so it's memory efficient and elegant. • *Aroma* - this pattern is often the natural result of good code organization that reduces unnecessary work.	• *Complexity* - requires deep understanding of the application and data flows. • *Applicability* - many times this pattern can't be used • *Rigidness* - might make refactoring the code more difficult.

Pattern 4: Use Callbacks not Return Values

- The basic idea is to avoid fighting the Borrow Checker by passing a closure into the function.

```
fn connect_and_attempt<F>(pool: &Pool, action: F) ->
Option<String>
    where F: Fn(Connection) -> Option<String>
{
    let connection: Connection = pool.connect();
    connection.set_database("startup.io");
    action(connection)
}
fn main() {
    let pool: Pool = postgres::connection_pool::new(url);
    let result = connect(&pool, |connection| {
        // do something with connection and return an
option
        Some(output)});
}
```

Pattern 4: Use Callbacks not Return Values

- The key difference over pattern 3 is that we pass an anonymous function |connection| { .. } into connect_and_attempt and we never return the connection object at all.
- This pattern can make unit testing easy because we can decouple blocks of code and test their logic without setting up the full environment. We can pass dummy callbacks into the system under test.

Pattern 4: Use Callbacks not Return Values

Pros	Cons
• *Elegant* - avoid fighting the borrow checker. • *Decoupled* - helps isolate application logic from I/O or dependencies.	• *Inflexible* - closures in Rust might require boxing in certain situations.