Intro to (a Subset of) Concepts of Rust PL

Jingqi Chen

April 1, 2024

Contents

This will not cover:

1. Syntax of Rust, e.g. `for`, `if`, `fn`, `let`, etc. You can read The Rust Programming Language.

This will cover:

1. Resource acquisition is initialization, RAII, comparing to C#.
2. Smart pointers.

Some things should be covered (but not today):

1. RTTI, comparing to C#.
2. Generic, comparing to C++.
3. Compile-time function execution, comparing to C++.

But to follow the tradition:

```rust
use std::io;

fn main() {
    let mut s = String::new();
    io::stdin().read_line(&mut s).unwrap();
    let a: i32 = s.trim().parse().unwrap();

    s.clear();

    io::stdin().read_line(&mut s).unwrap();
    let b: i32 = s.trim().parse().unwrap();

    let sum = a + b;
    println!("{}", sum);
}
```

Resource acquisition is initialization, RAII

1. A resource is anything that has to be acquired and released after use, regardless of explicitly or implicitly.

2. Examples are memory, locks, sockets, thread handles, and file handles.

3. A good resource management system handles all kinds of resources.

4. Leaks must be avoided in any long-running systems, but excessive resource retention can be almost as bad as a leak.

Some designs about resource management:

1. No abstraction at all: C, etc
2. RAII: C++, Rust, etc
3. GC: C#, Java, etc

Starting with no abstraction:

```
void func() {
    // allocate 100-size char array in stack
    char x[100];

    // allocate 100-size char array in heap
    char* y = malloc(sizeof(char) * 100);

    // free it before return
    free(y);
}
```

What if?

```
void func() {
    // allocate 100-size char array in stack
    char x[100];

    // this is just a compile warning for gcc 11.4
    // warning: 'free' called on unallocated object 'x'
    // segmentation fault (core dumped)
    // and there could be no warning at all for compilers
    free(x);

    // allocate 100-size char array in heap
    char* y = malloc(sizeof(char) * 100);

    // no free
    // memory leak
```

```
    // free(y);
}
```

In the real world:

```
// a very complex function
// in a deep function calling chain
// with early returns
// x and y are passed as pointer

// no info of whether in stack or heap (can or cannot free)
// no info of ownership (should or should not free)
void func(char* x, char* y) {

}
```

To solve this issue, ownership design would be straight forward.

As only the owner knows when it should be free (or destructed / garbage collected).

Specifically for Rust:

- 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.

And:

- There are move and reference(borrowing).
- There are smart pointers.

Move:

The ownership of a variable follows the same pattern every time: assigning a value to another variable moves it. When a variable that includes data on the heap goes out of scope, the value will be cleaned up by drop unless ownership of the data has been moved to another variable.

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

println!("{}, world!", s1);
// compile failure!
// as = is default to `move` in rust for complex types!
```

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

```rust
    // move the ownership to calculate_length
    let (s2, len) = calculate_length(s1);

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

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

    // move the ownership back
    (s, length)
}
```

Reference:

A reference is like a pointer in that it's an address we can follow to access the data stored at that address; that data is owned by some other variable. Unlike a pointer, a reference is guaranteed to point to a valid value of a particular type for the life of that reference.

```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()
}
```

Note: the closure works the same with function, which could capture the variable via ref, mutable ref, or move.

Some interesting tiny design choices:

1. Reference is immutable by default, which is opposite of C++.
2. Conpulsory compile-time check of lifetime(scope) & multi mutable ref, which could prevent issues (e.g. dangling pointers / data races), which is not conpulsory of C++. But the check can be bypassed via RefCell or Unsafe.

RAII:

1. There is almost zero overhead. The ideal situation is the releasing happens right after it is no longer needed.
2. It can goes wrong in runtime (and it is impossible to detect all the issues during compile-time), crash (dangling pointers), memory leak (cyclic ref), etc.
3. More complex for lock free concurrent environment, e.g. Hazard pointer.

Comparsions with GC:

1. GC is not deterministic, and there is much more overhead. (Recall: GC makes trade-off between footprint, throughput, latency).
2. GC can only collect managed resources. (Recall: `Dispose` of C#).

Further topics of performance difference when there is runtime or not:

1. Expected lifetime of each allocation. (or the ratio of IO/Compute)
2. Performance optimization methods brought by runtime:
   1. PGO, LTO.
3. How modern GC makes the trade-off:
   1. The Pauseless GC Algorithm
   2. JEP 439

Smart pointers

References only borrow data, in many cases, smart pointers own the data they point to.

- `Box<T>` for allocating values on the heap
- `Rc<T>`, a reference counting type that enables multiple ownership
- `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`, a type that enforces the borrowing rules at runtime instead of compile time

---

`Box<T>` is the Rust version of `unique_ptr`.

It represents exclusive ownership. Boxes allow you to store data on the heap rather than the stack.

```rust
enum List { Cons(i32, List), Nil, }

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
    // fail! recursive type `List` has infinite size
    // fail to put in stack
}

// ----- the following would do -----
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

Rc<T> is the Rust version of shared_ptr, but immutable.

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Note:

1. The `Rc::clone` only increments the reference count! It is different with `.clone()`.
2. `Rc<T>` allows only immutable ref, as "multiple mutable borrows to the same place can cause data races and inconsistencies". You may need `RefCell<T>` for multi mutable ref.
3. `Rc<T>` the increase / decrease of count is **NOT** thread safe (read: atomic)! If you need concurrency, use `Arc<T>`. This is a interesting design choice.

What is reference counting?

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}",
        Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}",
        Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}",
            Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}",
        Rc::strong_count(&a));
}
```

```
/*
$ cargo run
```

```
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
*/
```

`RefCell<T>` is used for *Interior mutability*, which is a design pattern in Rust that allows you to mutate data even when there are immutable references to that data.

With references and `Box<T>`, the borrowing rules' invariants are enforced at **compile time**. With `RefCell<T>`, these invariants are enforced at **runtime**. So, the program will `panic` rather than `compile failure`.

`RefCell<T>` is needed as, it is impossible to detect all the issues during compile-time (Recall: The halting problem is undecidable).

```rust
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;
}
```

Note:

1. Recall `const_cast` of C++.
2. Using `Rc<T>` with `RefCell<T>`, we finally get the full equivalent of `shared_ptr` of C++. So we can create the cyclic ref to leak the memory!

```rust
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("a next item = {:?}", a.tail());
}
```

But how can we break the cycle? `Weak<T>` would be the solution. Instead of creating the cycle, checking if the value has already been dropped in runtime is needed. (Recall: `weak_ptr` of C++).