Rust Bootcamp 2021

Day 4

Day 4 Activities

- ✓ Generic Types and Traits
- ✓ Smart Pointers
- ✓ Exercises and Homework

Generic Types - Structs

Generics is Rust's answer to templating in C++.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Here we have a Point class that can store ints, floats, or anything else.

https://doc.rust-lang.org/book/ch10-01-syntax.html#in-struct-definitions

Generic Types - Structs

Structs can also take multiple types.

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

This struct takes two types, allowing mixed values.

https://doc.rust-lang.org/book/ch10-01-syntax.html#in-struct-definitions

Struct Methods

Methods are like functions, but have the context of the struct.

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

Here we make an *area* method for *Rectangles*. Note: *self* doesn't need a type.

https://doc.rust-lang.org/book/ch05-03-method-syntax.html

Struct Methods

One other reason for choosing a method over a function is for referencing. Rust will automatically add &, &mut, or * as necessary. This makes the code cleaner and easier to read and write.

```
let p1 = Point { x: 0.0, y: 0.0 };
let p2 = Point { x: 5.0, y: 6.5 };

p1.distance(&p2); // use a method

(&p1).distance(&p2); // compiler-generated rust code
```

https://doc.rust-lang.org/book/ch05-03-method-syntax.html

Struct Associated Functions

Associated functions are methods without a *self*. They are similar to friend functions in C++, and commonly used for constructors.

```
impl Rectangle {
    fn new(width: u32, height: u32) -> Rectangle {
        Rectangle {
            width: width,
            height: height,
        }
    }
}
let r = Rectangle::new(1, 2);
```

https://doc.rust-lang.org/book/ch05-03-method-syntax.html#associated-functions

Generic Types for Struct Methods

For structs that take a generic type, these can also be referenced in the struct methods.

```
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

https://doc.rust-lang.org/book/ch10-01-syntax.html#in-method-definitions

Generic Types for Struct Methods

You can also have specialized methods, that only apply to a certain concrete type.

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

https://doc.rust-lang.org/book/ch10-01-syntax.html#in-method-definitions

Generic Types for Struct Methods

Note that you can use different types between the struct and the method.

```
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
```

https://doc.rust-lang.org/book/ch10-01-syntax.html#in-method-definitions

Generic Types - Enums

This is how several std library enums work

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Both enums are wrappers around a user-defined type.

https://doc.rust-lang.org/book/ch10-01-syntax.html#in-struct-definitions

Generic Types - Functions

The most basic way to make functions more generic is to take a slice.

```
fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

Here we take a slice of characters. This works on strings and vectors.

https://doc.rust-lang.org/book/ch10-01-syntax.html

Generic Types - Functions

We can make this more generic, to take any type.

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

We'll come back to PartialOrd and Copy, as these are Traits.

https://doc.rust-lang.org/book/ch10-01-syntax.html

Traits

Traits are like interfaces in other languages, describing functionality of a type.

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Here we declare a trait *Summary* which has a *summarize* function. Any type that implements this trait will also have a *summarize* function.

https://doc.rust-lang.org/book/ch10-02-traits.html

Trait Implementation

For types that use a trait, they need to implement it

```
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

https://doc.rust-lang.org/book/ch10-02-traits.html

Trait Implementation

One restriction on trait implementations is that the type must be local to the crate. We can implement std library traits for local types, but trying to implement local traits for std library types will fail.

This rule is designed to make sure other people's code can't break our code.

Without the rule, two or more implementations could exist in different crates, and Rust wouldn't know which one was the correct one.

https://doc.rust-lang.org/book/ch10-02-traits.html

Traits - Default Implementations

Sometimes we'd like to provide a default value for a trait.

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

impl Summary for Tweet {}
```

Note the empty implementation in *Tweet*.

It is not possible to call the default implementation from an overriding implementation of the same method.

Using Traits

Traits can be used as parameters to functions.

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

This function takes a single parameter *item* that must be a type that implements the *Summary* trait.

The function can call any function inside the *Summary* trait, but not other functions of that object.

https://doc.rust-lang.org/book/ch10-02-traits.html#traits-as-parameters

Using Traits

The previous slide was just syntax sugar for the longer, more flexible form:

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

This uses a generic type, with a trait bound.

The increased flexibility comes from using generic types.

https://doc.rust-lang.org/book/ch10-02-traits.html#traits-as-parameters

Using Traits

If you want to have two parameters that are guaranteed to have the same type, you must use the generic type syntax.

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
    println!("Breaking news! {} {}", item1.summarize(),
             item2.summarize());
}
```

https://doc.rust-lang.org/book/ch10-02-traits.html#traits-as-parameters

Using Traits

Specifying multiple trait bounds uses the + operator.

```
pub fn notify(item: &(impl Summary + Display)) {}

pub fn notify<T: Summary + Display>(item: &T) {}
```

https://doc.rust-lang.org/book/ch10-02-traits.html#traits-as-parameters

Using Traits

When using multiple types and traits, the function syntax can get cluttered.

The *where* clause allows this to be simplified.

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{}
```

This makes it easier to read the function signature.

https://doc.rust-lang.org/book/ch10-02-traits.html#clearer-trait-bounds-with-where-clauses

Returning Trait Types

A function return value can also use a trait instead of a type.

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

https://doc.rust-lang.org/book/ch10-02-traits.html#returning-types-that-implement-traits

Conditionally Implementing Methods

Rust trait implementations can be conditional on trait bounds.

```
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

https://doc.rust-lang.org/book/ch10-02-traits.html#using-trait-bounds-to-conditionally-implement-methods

Conditionally Implementing Methods

The std library does this a lot. For example, implementing *ToString*.

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Every type that implements the *Display* trait will automatically get the *ToString* trait implemented in a default way.

These are called blanket implementations.

https://doc.rust-lang.org/book/ch10-02-traits.html#using-trait-bounds-to-conditionally-implement-methods

Object Oriented Design

Rust has both objects (types) and encapsulation (with public methods).

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }
    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

Here we see a public *average* method and a private *update_average* method.
https://doc.rust-lang.org/book/ch17-01-what-is-oo.html

Object Oriented Design

Strictly speaking, Rust does not have object inheritance. But the concept of inheritance is used for two reasons:

1. Code reuse
2. Polymorphism (substituting child types for parent types)

Rust code can be reused by implementing default trait methods. And polymorphism can be achieved by using traits.

https://doc.rust-lang.org/book/ch17-01-what-is-oo.html

Traits as Struct Members

```
pub trait Draw {
    fn draw(&self);
}

pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

With the *Screen* struct, we would like to take multiple components that implement the *Draw* trait. Because we don't specify a fixed type or enum of types, we need a smart pointer called a *Box.* More on this in a few slides.

https://doc.rust-lang.org/book/ch17-02-trait-objects.html

Traits as Struct Members

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Say someone else creates a *Button* struct, which implements the *Draw* trait.

https://doc.rust-lang.org/book/ch17-02-trait-objects.html

Traits as Struct Members

Now they can create a *Screen* which contains a *Button*, or any other *Draw* type.

```
let screen = Screen {
    components: vec![
        Box::new(Button {
            width: 50,
            height: 10,
            label: String::from("OK"),
        }),
    ],
};
screen.run();
```

https://doc.rust-lang.org/book/ch17-02-trait-objects.html

Object Safety with Trait Objects

Trait objects must follow some rules in order to successfully be used. The most relevant rules are:

All default methods in the trait having these properties:

- The return type isn't *Self*
- There are no generic type parameters

The issue is that using a trait object means the concrete type is forgotten, and there is no way to know which types to use.

https://doc.rust-lang.org/book/ch17-02-trait-objects.html#object-safety-is-required-for-trait-objects

Dynamic Dispatch with Trait Objects

When trait objects are used, Rust must use dynamic dispatch, using the pointers inside the trait object to know which method to call. This incurs a runtime performance penalty, and prevents inlining of code.

Thus why this should be used rarely, and only as-needed.

https://doc.rust-lang.org/book/ch17-02-trait-objects.html#trait-objects-perform-dynamic-dispatch

An Object Oriented Example

https://doc.rust-lang.org/book/ch17-03-oo-design-patterns.html

Here is an example of constructing a blog post and changing its state, in an object oriented design pattern. The content of the post is kept hidden until it is published.

Also presented is a more Rust variant, where the states are encoded as independent types. In this case, the post types don't have a content function at all until the published post type, so asking for the content on unpublished posts is a compiler error.

These are two different styles of programming.

Smart Pointers

```
Box<T>
```

The *Box* tells the compiler to allocate a pointer to memory on the heap.

This is the most basic smart pointer in Rust. There is no real performance overhead, and it can be used for several reasons:

- A type with an undetermined size.
- A large amount of data and want to ensure move operations.
- Owning a value that implements a trait, rather than a type.

https://doc.rust-lang.org/book/ch15-01-box.html

Smart Pointers

Here is an example of creating a new *Box* for an integer.

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

This makes a *Box* on the stack and an *i32* on the heap.

This isn't very useful for a single value that would be allowed on the stack normally. Instead, a *Box* is used for things that can't or shouldn't be allocated on the stack.

https://doc.rust-lang.org/book/ch15-01-box.html

Recursive Types with Box

If we want a recursive type, we need to put the recursive part on the heap.

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
let list = Cons(1, Box::new(Cons(2, Box::new(Nil))));
```

Here we make a cons list, borrowed from Lisp as a construct function generating a list.

https://doc.rust-lang.org/book/ch15-01-box.html#enabling-recursive-types-with-boxes

Reference Counted Smart Pointer

If ownership of an object is unclear, such as in a graph, *RC<T>* can be used.

```
use std::rc::Rc;
enum List {
    Cons(i32, Rc<List>),
    Nil,
}
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 that *RC<T>* should only be used in single threaded scenarios, for multiple readers only.

https://doc.rust-lang.org/book/ch15-04-rc.html

Mutable RC Smart Pointer

RefCell<T> uses *unsafe* code to allow mutability while holding a reference. It does this by enforcing the borrowing rules at runtime instead of compile time.

```
struct MockMessenger {
    sent_messages: RefCell<Vec<String>>,
}
impl Messenger for MockMessenger {
    // works as long as no one else has a borrow open, otherwise panics
    fn send(&self, message: &str) {
        self.sent_messages.borrow_mut().push(String::from(message));
    }
}

let mock_messenger = MockMessenger::new();
// use mock_messenger in other code
assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
```

h

Mutable RC Smart Pointer

We can also combine *Rc* and *RefCell* to make multi-owner mutable pointers.

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

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;
println!("a after = {:?}", a);
println!("b after = {:?}", b);
println!("c after = {:?}", c);
```

Mutable RC Smart Pointer

But, we need to be wary with this of creating memory leaks.

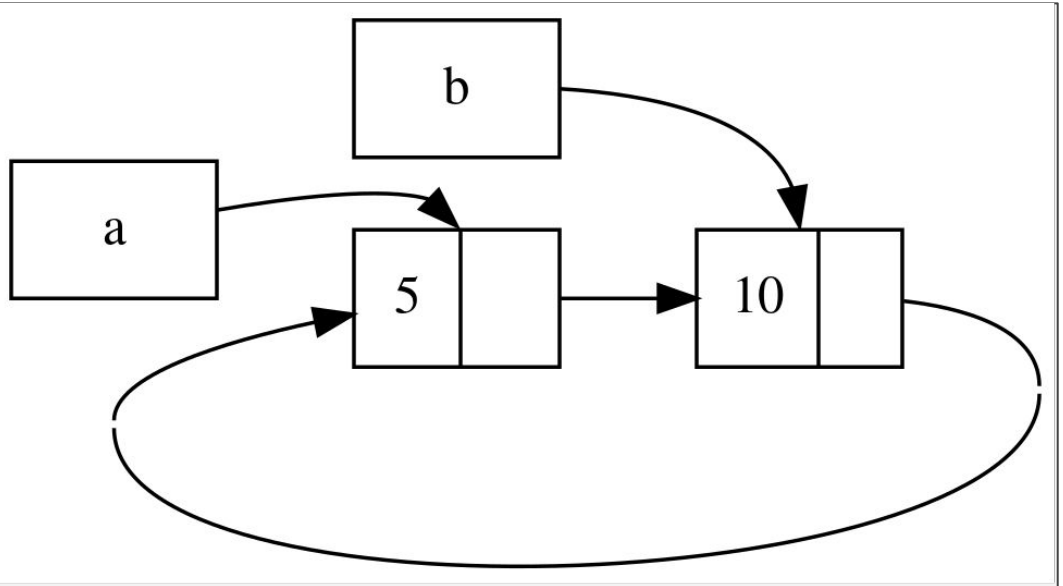
```
impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));
if let Some(link) = a.tail() {
    *link.borrow_mut() = Rc::clone(&b); // make a circular reference
}
// a and b link to each other, so won't be dropped at the end of scope
```

h

Mutable RC Smart Pointer

But, we need to be wary with this of creating memory leaks.

```
impl List {
    fn tail(&self) -> Option
        match self {
            Cons(_, item) =>
            Nil => None,
        }
    }
}
let a = Rc::new(Cons(5, RefC
let b = Rc::new(Cons(10, Ref
if let Some(link) = a.tail()
    *link.borrow_mut() = Rc:
}
// a and b link to each other, so won't be dropped at the end of scope
```



h

Preventing Smart Pointer Cycles

Weak references are the answer, as they do not increase the reference count.

Let's take an example of a Tree structure with references from child to parent:

```
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

https://doc.rust-lang.org/book/ch15-06-reference-cycles.html#preventing-reference-cycles-turning-an-rct-into-a-weakt

Preventing Smart Pointer Cycles

```
let leaf = Rc::new(Node {
    value: 3,
    parent: RefCell::new(Weak::new()),
    children: RefCell::new(vec![]),
});
let branch = Rc::new(Node {
    value: 5,
    parent: RefCell::new(Weak::new()),
    children: RefCell::new(vec![Rc::clone(&leaf)]),
});
*leaf.parent.borrow_mut() = Rc::downgrade(&branch);
```

Here, the parent pointer uses a *Weak* reference, which does not create a cycle.

https://doc.rust-lang.org/book/ch15-06-reference-cycles.html#preventing-reference-cycles-turning-an-rct-into-a-weakt

Preventing Smart Pointer Cycles

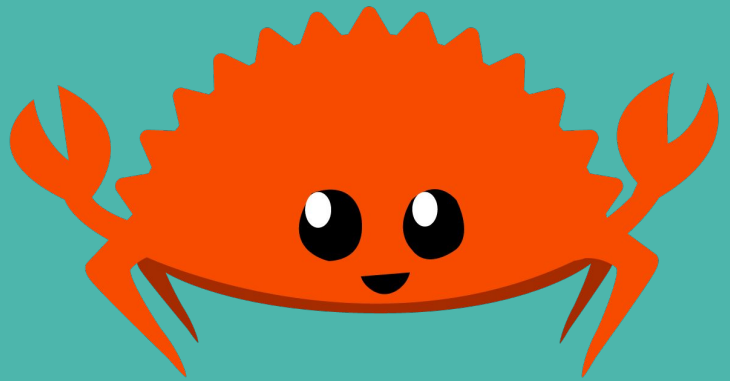
To use a Weak pointer, we need to check if it's still valid.

```
let parent = leaf.parent.borrow().upgrade();
println!("leaf parent = {:?}", parent);
```

The *upgrade* function returns an *Option<Rc<T>>*, which may be *None* if the parent no longer exists.

https://doc.rust-lang.org/book/ch15-06-reference-cycles.html#preventing-reference-cycles-turning-an-rct-into-a-weakt

End of Lecture



Homework

Create a binary search tree that can take arbitrary types for values. A stub has been created at homework/day4/bst.

Additional help is provided by a Python implementation in homework/day4/bst_python, which includes an example test for correctness.