

Rust programming

Module A3: Traits and generics

Last time...

- Rust references
- Structs & enums
- `Option`` and `Result``
- Advanced syntax
 - Pattern matching
 - Slices

Any questions?

In this module

Make your code more versatile with generics

Learning objectives

- Use traits and generics
- Use common traits from `std`
- Understand and use lifetime bounds

Module A3

Traits and generics

Content overview

- Introduction to generics
- Various traits from `std`
- Lifetime bounds

Introduction to generics

The problem

```
1  fn add_u32(l: u32, r: u32) -> u32 { /* -snip- */ }
2
3  fn add_i32(l: i32, r: i32) -> i32 { /* -snip- */ }
4
5  fn add_f32(l: f32, r: f32) -> f32 { /* -snip- */ }
6
7  /* ... */
```

We need generic code!

Generic code

An example

```
1 fn add<T>(lhs: T, rhs: T) -> T { /* - snip - */ }
```

Or, in plain English:

- `<T>` = "let `T` be a type"
- `lhs: T` "let `lhs` be of type `T`"
- `-> T` "let `T` be the return type of this function"

Some open points:

- What can we do with a `T`?
- What should the body be?

Bounds on generic code

We need to provide information to the compiler:

- Tell Rust what `T`` can do
- Tell Rust what `T`` is accepted
- Tell Rust how `T`` implements functionality

`trait`

Describe what the type can do

```
1 trait MyAdd {  
2     fn my_add(&self, other: &Self) -> Self;  
3 }
```

`impl trait`

Describe how the type does it

```
1  impl MyAdd for u32 {
2      fn my_add(&self, other: &Self) -> Self {
3          *self + *other
4      }
5  }
```

Using a `trait`

```
1 // Import the type and the trait
2 use my_mod::{MyAdd}
3
4 fn main() {
5     let left: u32 = 6;
6     let right: u32 = 8;
7     // Call trait method
8     let result = left.my_add(&right);
9     assert_eq!(result, 14);
10    // Explicit call
11    let result = MyAdd::my_add(&left, &right);
12    assert_eq!(result, 14);
13 }
```

- Trait needs to be in scope
- Call just like a method
- Or by using the explicit associated function syntax

Trait bounds

```
1 fn add_values<T: MyAdd>(this: &T, other: &T) -> T {
2     this.my_add(other)
3 }
4
5 // Or, equivalently
6
7 fn add_values<T>(this: &T, other: &T) -> T
8     where T: MyAdd
9 {
10     this.my_add(other)
11 }
```

Now we've got a *useful* generic function!

English: "For all types `T` that implement the `MyAdd` trait, we define..."

Limitations of `MyAdd`

What happens if...

- We want to add two values of different types?
- Addition yields a different type?

Making `MyAdd` itself generic

Add an 'Input type' `0`:

```
1 trait MyAdd<0> {  
2     fn my_add(&self, other: 80) -> Self;  
3 }  
4  
5 impl MyAdd<u16> for u32 {  
6     fn my_add(&self, other: &u16) -> Self {  
7         *self + (*other as u32)  
8     }  
9 }
```

We can now add a `u16` to a `u32`.

Defining output of `MyAdd`

- Addition of two given types always yields in one specific type of output
- Add *associated type* for addition output

```
1  trait MyAdd<O> {
2      type Output;
3      fn my_add(&self, other: O) -> Self::Output;
4  }
5
6  impl MyAdd<u16> for u32 {
7      type Output = u64;
8
9      fn my_add(&self, other: &u16) -> Self::Output {
10         *self as u64 + (*other as u64)
11     }
12 }
13
14 impl MyAdd<u32> for u32 {
15     type Output = u32;
16
17     fn my_add(&self, other: &u32) -> Self::Output {
18         *self + *other
19     }
20 }
```

`std::ops::Add`

The way `std` does it

```
1 pub trait Add<Rhs = Self> {  
2     type Output;  
3  
4     fn add(self, rhs: Rhs) -> Self::Output;  
5 }
```

- Default type of `Self` for `Rhs`

`impl std::ops::Add`

```
1 use std::ops::Add;
2 pub struct BigNumber(u64);
3
4 impl Add for BigNumber {
5     type Output = Self;
6
7     fn add(self, rhs: Self) -> Self::Output {
8         BigNumber(self.0 + rhs.0)
9     }
10 }
11
12 fn main() {
13     // Call `Add::add`
14     let res = BigNumber(1).add(BigNumber(2));
15 }
```

What's the type of `res`?

`impl std::ops::Add` (2)

```
1 pub struct BigNumber(u64);
2
3 impl std::ops::Add<u32> for BigNumber {
4     type Output = u128;
5
6     fn add(self, rhs: Self) -> Self::Output {
7         (self.0 as u128) + (rhs as u128)
8     }
9 }
10
11 fn main() {
12     let res = BigNumber(1) + 3u32;
13 }
```

What's the type of `res`?

Traits: Type Parameter vs. Associated Type

Type parameter (input type)

if trait can be implemented for many combinations of types

```
1 // We can add both a u32 value and a u32 reference to a u32
2 impl Add<u32> for u32 { /* */}
3 impl Add<&u32> for u32 { /* */}
```

Associated type (output type)

to define a type for a single implementation

```
1 impl Add<u32> for u32 {
2     // Addition of two u32's is always u32
3     type Output = u32;
4 }
```

`#[derive]` a `trait`

```
1  #[derive(Clone)]
2  struct Dolly {
3      num_legs: u32,
4  }
5
6  fn main() {
7      let dolly = Dolly { num_legs: 4 };
8      let second_dolly = dolly.clone();
9      assert_eq!(dolly.num_legs, second_dolly.num_legs);
10 }
```

- Some traits are trivial to implement
- Derive to quickly implement a trait
- For `Clone`: derived `impl` calls `clone` on each field

Orphan rule

Coherence: There must be at most one implementation of a trait for any given type

Trait can be implemented for a type iff:

- Either your crate defines the trait
- Or your crate defines the type

Or both, of course

Summary

- Traits describe functionality
- Generics allow writing code in terms of traits
- Traits can be generic, too

Questions?

Common traits from `std`

Operator overloading: `std::ops::Add<T>` et al.

- Shared behavior

```
1 use std::ops::Add;
2 pub struct BigNumber(u64);
3
4 impl Add for BigNumber {
5     type Output = Self;
6
7     fn add(self, rhs: Self) -> Self::Output {
8         BigNumber(self.0 + rhs.0)
9     }
10 }
11
12 fn main() {
13     // Now we can use `+` to add `BigNumber`s!
14     let res: BigNumber = BigNumber(1) + (BigNumber(2));
15 }
```

- Others: `Mul``, `Div``, `Sub``, ...

Markers: `std::marker::Sized`

- Marker traits

```
1  /// Types with a constant size known at compile time.  
2  /// [...]  
3  pub trait Sized { }
```

`u32` is `Sized`

Slice `[T]`, `str` is not `Sized`

Slice reference `&[T]`, `&str` is `Sized`

Others:

- `Sync`: Types of which references can be shared between threads
- `Send`: Types that can be transferred across thread boundaries

Default values: `std::default::Default`

```
1  pub trait Default: Sized {
2      fn default() -> Self;
3  }
4
5  #[derive(Default)] // Derive the trait
6  struct MyCounter {
7      count: u32,
8  }
9
10 // Or, implement it
11 impl Default for MyCounter {
12     fn default() -> Self {
13         MyCounter {
14             count: 1, // If you feel so inclined
15         }
16     }
17 }
```

Duplication: `std::clone::Clone` & `std::marker::Copy`

```
1 pub trait Clone: Sized {
2     fn clone(&self) -> Self;
3
4     fn clone_from(&mut self, source: &Self) {
5         *self = source.clone();
6     }
7 }
8
9 pub trait Copy: Clone { } // That's it!
```

- Both `Copy` and `Clone` can be `#[derive]`d
- `Copy` is a marker trait
- `trait A: B == "Implementor of A must also implement B"`
- `clone_from` has default implementation, can be overridden

Conversion: `Into<T>` & `From<T>`

```
1 pub trait From<T>: Sized {
2     fn from(value: T) -> Self;
3 }
4
5 pub trait Into<T>: Sized {
6     fn into(self) -> T;
7 }
8
9 impl <T, U> Into<U> for T
10 where U: From<T>
11 {
12     fn into(self) -> U {
13         U::from(self)
14     }
15 }
```

- Blanket implementation

Prefer `From` over `Into` if orphan rule allows to

Reference conversion: `AsRef<T>` & `AsMut<T>`

```
1 pub trait AsRef<T: ?Sized>
2 {
3     fn as_ref(&self) -> &T;
4 }
5
6 pub trait AsMut<T: ?Sized>
7 {
8     fn as_mut(&mut self) -> &mut T;
9 }
```

- Provide flexibility to API users
- `T` need not be `Sized`, e.g. slices `[T]` can implement `AsRef<T>`, `AsMut<T>`

Reference conversion: `AsRef<T>` & `AsMut<T>` (2)

```
1 fn print_bytes<T: AsRef<[u8]>>(slice: T) {
2     let bytes: &[u8] = slice.as_ref();
3     for byte in bytes {
4         print!("{:02X}", byte);
5     }
6     println!();
7 }
8
9 fn main() {
10    let owned_bytes: Vec<u8> = vec![0xDE, 0xAD, 0xBE, 0xEF];
11    print_bytes(owned_bytes);
12
13    let byte_slice: [u8; 4] = [0xFE, 0xED, 0xC0, 0xDE];
14    print_bytes(byte_slice);
15 }
```

Have user of `print_bytes` choose between stack local `[u8; N]` and heap-allocated `Vec<u8>`

Destruction: `std::ops::Drop`

```
1 pub trait Drop {  
2     fn drop(&mut self);  
3 }
```

- Called when owner goes out of scope

Destruction: `std::ops::Drop`

```
1  struct Inner;
2
3  impl Drop for Inner {
4      fn drop(&mut self) {
5          println!("Dropped inner");
6      }
7  }
8
9  struct Outer {
10     inner: Inner,
11 }
12
13 impl Drop for Outer {
14     fn drop(&mut self) {
15         println!("Dropped outer");
16     }
17 }
18
19 fn main() {
20     // Explicit drop
21     std::mem::drop(Outer { inner: Inner });
22 }
```

Output:

```
Dropped outer
Dropped inner
```

- Destructor runs *before* members are removed from stack
- Signature `&mut` prevents explicitly dropping `self` or its fields in destructor
- Compiler inserts `std::mem::drop` call at end of scope

```
1  // Implementation of `std::mem::drop`
2  fn drop<T>(_x: T) {}
```

Question: why does `std::mem::drop` work?

Compiling generic functions

```
1  impl MyAdd for i32 { /* - snip - */ }
2  impl MyAdd for f32 { /* - snip - */ }
3
4  fn add_values<T: MyAdd>(left: &T, right: &T) -> T
5  {
6      left.my_add(right)
7  }
8
9  fn main() {
10     let sum_one = add_values(&6, &8);
11     assert_eq!(sum_one, 14);
12     let sum_two = add_values(&6.5, &7.5);
13     println!("Sum two: {}", sum_two); // 14
14 }
```

Code is *monomorphized*:

- Two versions of `add_values` end up in binary
- Optimized separately and very fast to run (static dispatch)
- Slow to compile and larger binary

Lifetime bounds

What lifetime?

- References refer to variable
- Variable has a lifetime:
 - Start at declaration
 - End at drop

Question: Will this compile?

```
1  /// Return reference to longest of `&str`s
2  fn longer(a: &str, b: &str) -> &str {
3      if a.len() > b.len() {
4          a
5      } else {
6          b
7      }
8  }
```

```

1  /// Return reference to longest of `&str`s
2  fn longer(a: &str, b: &str) -> &str {
3      if a.len() > b.len() {
4          a
5      } else {
6          b
7      }
8  }

```

```

1      Compiling playground v0.0.1 (/playground)
2  error[E0106]: missing lifetime specifier
3      --> src/lib.rs:2:32
4      |
5  2 | fn longer(a: &str, b: &str) -> &str {
6      |           ----      ----      ^ expected named lifetime parameter
7      |
8      = help: this function's return type contains a borrowed value, but the signature does not say whether it is
borrowed from `a` or `b`
9      help: consider introducing a named lifetime parameter
10     |
11  2 | fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
12     |           +++++  ++          ++          ++
13
14  For more information about this error, try `rustc --explain E0106`.
15  error: could not compile `playground` due to previous error

```

Lifetime annotations

```
1 fn longer<'a>(left: &'a str, right: &'a str) -> &'a str {
2     if left.len() > right.len() {
3         left
4     } else {
5         right
6     }
7 }
```

English:

- Given a lifetime called `'a`,
- `longer` takes two references `left` and `right`
- that live for at least `'a`
- and returns a reference that lives for `'a`

Note: Annotations do NOT change the lifetime of variables! Their scopes do!

Just provide information for the borrow checker

Validating boundaries

- Lifetime validation is done within function boundaries
- No information of calling context is used

Question: Why?

Lifetime annotations in types

```
1  /// A struct that contains a reference to a T
2  pub struct ContainsRef<'r, T> {
3      reference: &'r T
4  }
```

Lifetime elision

Q: "Why haven't I come across this before?"

A: "Because of lifetime elision!"

Rust compiler has heuristics for eliding lifetime bounds:

- Each elided lifetime in input position becomes a distinct lifetime parameter.
- If there is exactly one input lifetime position (elided or annotated), that lifetime is assigned to all elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.
- Otherwise, annotations are needed to satisfy compiler

Lifetime elision examples

```
1 fn print(s: &str); // elided
2 fn print<'a>(s: &'a str); // expanded
3
4 fn debug(lvl: usize, s: &str); // elided
5 fn debug<'a>(lvl: usize, s: &'a str); // expanded
6
7 fn substr(s: &str, until: usize) -> &str; // elided
8 fn substr<'a>(s: &'a str, until: usize) -> &'a str; // expanded
9
10 fn get_str() -> &str; // ILLEGAL (why?)
11
12 fn frob(s: &str, t: &str) -> &str; // ILLEGAL (why?)
13
14 fn get_mut(&mut self) -> &mut T; // elided
15 fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded
```

Tutorial time!

- Exercises A2 recap
- Exercises A3 in 101-rs.tweede.golf

Don't forget to `git pull`!

