Generics and Traits

Performant Software Systems with Rust — Lecture 8

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Generics

- To remove duplication of code, we can replace specific types with a **placeholder** that represents multiple types
- We have seen them before: `Option<T>`, `Result<T, E>`, `Vec<T>`, `HashMap<K, V>`
- We will now define our own types, functions, and methods with generics
- Let’s start from a simple program that finds the largest number in a vector of numbers

Finding the large number in a vector

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");
}
```

Finding the largest number in two vectors

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
```

Obviously, we need to create an **abstraction** by defining a function that operates on any list of integers passed in as a parameter.

Defining the `largest` function

```rust
// list: any concrete slice of i32 values
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");
```

But what if we also wish to find the largest item in `&[char]`?

Writing two functions is silly!

```rust
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
```

Generics in Functions

```rust
// <T> is a type parameter placed between the name of the function
// and the parameter list
fn largest<T>(list: &[T]) -> &T {}
```

Defining `largest()` using the generic data type

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

```

Live Demo

Defining Structs using Generics

```rust
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 };
    let mixed = Point { x: 5, y: 4.0 };
}
```

Live Demo

Defining Structs using Multiple Generic Types

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

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Defining Enums using Generics

```rust
enum Option<T> {
    Some(T),
    None,
}

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

Defining Methods using Generics

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

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };
    println!("p.x = {}", p.x());
}
```

Defining Methods on Concrete Types

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

Mixing Generic Type Parameters

```rust
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

```

Advanced Topic: Zero-Cost Abstraction in Rust

- Rust boasts **zero-cost abstraction**
  - It does not add additional overhead when it introduces an abstraction to the language
  - Using generic types won’t make your code run any slower than it would with concrete types

Monomorphization

- Turning generic code into specific code by filling in the concrete types that are used when compiled

```rust
enum Option<T> {
    Some(T),
    None,
}

let integer = Some(5);
let float = Some(5.0);
```

Monomorphization

```rust
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Traits

- A **trait** defines the **functionality** (or **behaviour**) a particular type has and can share with other types
  - **Behaviour** implies **methods** that we define and call in a type
  - Different types share the same **behaviour** if we can call the same **methods** on all of those types
  - **Trait** definitions are used to **group** those behaviours

Example

```rust
// Both news articles and tweets can be summarized
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Implementing a Trait on a Type

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author,
            self.location)
    }
}
```

Implementing a Trait on a Type

```rust
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)
    }
}
```

Calling a Trait Method

```rust
// need to bring the Summary trait into scope
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("books"),
        content: String::from(
            "The Rust Programming Language",
        ),
        reply: false,
        retweet: false,
    };

    println!("New tweet: {}", tweet.summarize());
}
```

Default Implementations

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

To use the default behaviour on `NewsArticle`:

```rust
impl Summary for NewsArticle {}
```

Default Implementations Can Call Other Methods

```rust
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}

impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

Traits as Parameters: The `impl Trait` Syntax

```rust
// `item` is a reference to a type that implements the `Summary` trait
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Trait Bound

```rust
// `item` is a reference to a type that implements the `Summary` trait
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

is just **syntax sugar** to:

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

Why is Trait Bound a Good Idea?

- The compiler uses trait bound to check that all the concrete types provide the correct behavior
- Converts run-time errors to compile-time errors, improving performance when the code compiles

Trait Bound Is More Expressive Than `impl Trait`

```rust
// `item1` and `item2` are references to types that implement
// the `Summary` trait
pub fn notify(item1: &impl Summary, item2: &impl Summary) {}
```

vs.

```rust
// `item1` and `item2` must be of the same type
pub fn notify<T: Summary>(item1: &T, item2: &T) {}
```

Specifying Multiple Trait Bounds with the + Syntax

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

or

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

Clearer Trait Bounds with `where` Clauses

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

or

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

Returning Types That Implement Traits

```rust
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("books"),
        content: String::from(
            "The Rust Programming Language",
        ),
        reply: false,
        retweet: false,
    }
}
```

- Helpful with closures and iterators, to be covered later

Cannot Return Multiple Types

```rust
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Rust 1.82 Released",
            ),
            location: String::from("USA"),
            author: String::from("The Rust Release Team"),
            content: String::from(
                "October 17 2024",
            ),
        }
    } else {
        Tweet {
            username: String::from("books"),
            content: String::from(
                "The Rust Programming Language",
            )
```

Using Trait Bounds to Conditionally Implement Methods

```rust
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

// always implements the `new` function
impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

// only implements the `cmp_display` function if `T` implements
// `Display` and `PartialOrd`
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
```

Blanket Implementations

```rust
// used for conditionally implement a trait for any type that
// implements another trait
//
// this example implements the `ToString` trait for any type that
// implements the `Display` trait
impl<T: Display> ToString for T {
    // --snip--
}

let s = 3.to_string();
```

- used extensively in the Rust standard library

Associated Types in Traits as Placeholder Types

- **Associated types** in traits are type placeholders
- The trait method definitions can use them
- The implementor of a trait will need to specify the concrete type instead
- Allows us to define a function without specifying what types it can use

Associated Types in Traits as Placeholder Types

```rust
pub trait Iterator {
    type Item; // placeholder type

    fn next(&mut self) -> Option<Self::Item>;
}
```

Implementing the `Iterator` Trait on the `Counter` Type

```rust
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
    }
}
```

But why can't we just write the following for the `Iterator` trait?

```rust
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

- We have to provide type annotations to indicate which implementation of `Iterator`
  - `Iterator<String> for Counter` or `Iterator<i32> for Counter`?
- With associated types, there can only be one type of `Item`, because there can only be one `impl Iterator for Counter`
- We don't have to specify that we want an iterator of `u32` values everywhere we call `next()` on `Counter`
- The associate type becomes a part of the trait's contract

Supertraits: Requiring One Trait's Functionality within Another

```rust
use std::fmt::Display;

// `OutlinePrint` requires the `Display` trait to be implemented
trait OutlinePrint: Display {
    fn outline_print(&self) {
        // so that we can use the `Display` trait's functionality
        // including the `to_string` method
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {output} *");
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

```rust
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

```rust
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Required Additional Reading

The Rust Programming Language, Chapter 10.1, 10.2, 20.2