Tipos genéricos, traits e tempo de vida

Diogo Silveira Mendonça

Slides Baseados no Capítulo 10 do Rust Book
https://doc.rust-lang.org/book/ch10-00-generics.html

Introdução

- Cada linguagem de programação tem sua maneira de lidar com conceitos duplicados. No caso do Rust, é lidado com a utilização dos tipos genéricos.

- Tipos genéricos são substitutos abstratos para tipos concretos ou outras propriedades.

- As funções podem chamar um tipo genérico, em vez de um tipo concreto como u32 ou string.

- Lembrando que já fizemos isso antes, no capítulos 6, 8 e 9. Durante o estudo de Options, Vetores, HashMap e do Result.

Introdução

- Vamos ver como extrair uma função para reduzir a duplicação de código.
- Como criar uma função genérica.
- Como usar tipos genéricos em definições de structs e enums.
- Como usar traits para definir comportamentos de maneira genérica.
- Veremos o conceito de Lifetime (Tempo de vida).
- Os códigos de exemplo podem ser encontrados nesse link.

Extraindo uma função para remover a duplicação

- Primeiro, veremos como remover a duplicação sem utilizar o tipo genérico.
- Extraindo uma função que substitui valores específicos por um espaço reservado que representa múltiplos valores.
- O código ao lado, pega uma lista com números inteiros e referência o primeiro da lista em largest. Depois em um loop, procura-se o um número maior, se achar um maior, muda a referência. No final imprime o maior número da lista.

```
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);
}
```

Extraindo uma função para remover a duplicação

- Agora se quisermos verificar o maior número de duas listas?
- Uma possível solução seria duplicar o código e usar a mesma lógica em dois lugares diferentes. Como no exemplo abaixo.

```
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 {
        if number > largest {
            largest = number;
        }
    }

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

Extraindo uma função para remover a duplicação

- Apesar de funcionar, essa não é a solução mais adequada, já que é propensa a erros.

- Para eliminar essa duplicação, vamos criar uma abstração definindo uma função que opera em qualquer lista de inteiros passada como parâmetro.

- Link do código abaixo.

```
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);

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

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

Extraindo uma função para remover a duplicação

- A função largest possui como parâmetro, uma lista i32. Ela pode ser chamada, passando qualquer lista i32.

- Para realizar isso é preciso identificar a repetição de código, implementar uma nova função baseado no código duplicado e substituir a parte do código com a parte duplicada pela chamada da função.

```
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);

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

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

Tipos Genéricos em funções

- Veremos como simplificar o exemplo anterior com os tipos genéricos
- Os tipos genéricos são utilizados para criar itens como assinaturas de funções ou structs.
- Ao definir uma função, colocamos os tipos genéricos na assinatura da função. Onde normalmente colocaríamos os parâmetros da função e seu tipo de retorno.
- No exemplo, largest_i32 e largest_char foram criadas baseadas na função largest, elas se diferenciam apenas pelo parâmetro e o tipo de retorno.

```
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;
        }
    }
    largest
}
```

Tipos Genéricos em funções

- Podemos utilizar o tipo Genérico para criar uma função largest que aceite tanto i32 quanto char.

- Então a definição da função seria “fn largest<T>(list: &[T]) -> &T {}“.

- Você poderia nomear o parâmetro de tipo “T” de outra forma, é comum em Rust utilizar nomes curtos e com o formato CamelCase. O nome “T” de “Type” acabou se tornando um padrão entre os usuários do Rust.

- Lemos a definição da função como: a função largest é genérica em relação a algum tipo T. Essa função tem um parâmetro chamado list, que é uma fatia de valores do tipo T. A função largest retorna uma referência a um valor do mesmo tipo T.

Tipos Genéricos em funções

- No momento nosso código não irá funcionar, pois o corpo da função largest não funciona para todos os tipos de T.

- A nossa comparação neste código só irá permitir a utilização de tipos com valores ordenados. Para isso devemos utilizar o trait “std::cmp::PartialOrd” que pode ser implementado em tipos. Veremos mais para frente isso, ainda nesse capítulo.

```
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);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Tipos Genéricos em estruturas

- Também podemos definir para a estrutura utilizar o tipo genérico em um ou mais campos, utilizando “<>”.

- Como utilizamos somente um tipo genérico para definir Point<T>, os campos x e y precisarão ser do mesmo tipo.

- O exemplo 2, passa um integer e um float, logo o código terá o erro “error[E0308]: mismatched types”

```
#[derive(Debug)]
struct Point<T> {
        x: T,
        y: T,
}
```

Exemplo 1:

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

Exemplo 2:

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

Tipos Genéricos em estruturas

- Para definir uma estrutura onde x e y possam ser de tipos diferentes, será preciso passar dois tipos genéricos.
- No exemplo, definimos o tipo para a estrutura utilizar T e U. Onde x é do tipo T e y do tipo U.
- Podemos utilizar quantos tipos de tipos genéricos quisermos. Mas isso pode dificultar na leitura.

Exemplo:

```
#[derive(Debug)]
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 };

    println!("integer: {:?}", both_integer);
    println!("float: {:?}", both_float);
    println!("both: {:?}", integer_and_float);
}
```

Tipos Genéricos em enumerações

- Assim como em estruturas, podemos definir para a enumeração utilizar o tipo genérico.

- O exemplo ao lado, foi visto no capítulo 6, esse é o enum Option<T> da biblioteca padrão.

- Esse enum é genérico em relação ao tipo T e possui duas variantes, Some e None.
  - Some contém o valor do tipo T.
  - None não contém nenhum valor.

- Podemos criar um valor opcional com enum Option<T>, porque Option<T> é genérico e pode aceitar qualquer tipo.

```
#![allow(unused)]
fn main() {
    enum Option<T> {
        Some(T),
        None,
    }
}
```

Tipos Genéricos em enumerações

- Também vimos o enum Result<T, E>, no capítulo 9. Nesse caso o enum utiliza mais de um tipo genérico.

- O enum Result é genérico em relação ao tipo T e E. Possui duas variantes Ok, que tem o valor de T e Err, que tem o valor de E.

- Ela é útil em um caso onde temos um caso de sucesso que retorna o valor do tipo T e um caso de falha, que retorna um erro do tipo E.

- Como vimos na tentativa de abrir um arquivo no capítulo 9.

```
#![allow(unused)]
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Tipos Genéricos em métodos

- Podemos utilizar métodos em estruturas e enumerações e usar um tipo genérico em suas definições. Como fizemos no capítulo 5.

- Nesse exemplo temos a estrutura Point<T> e o método x. O método retorna uma referência do valor de x.

- Note que utilizamos impl<T>. Isso é para possamos usar T para especificar que estamos implementando métodos no tipo Point<T>.

- Com isso o Rust pode identificar que point é um tipo genérico e não um tipo concreto.

```
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());
}
```

Tipos Genéricos em métodos

- Podemos limitar Point a um tipo concreto, por exemplo, f32. Note que não precisamos utilizar <f32> logo após impl.

Exemplo:

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

- Nesse código os os tipos Point<f32> terão o método distance_from_origin.

- Outros casos, onde Point<T> possui o T do tipo f32, não terão o método distance_from_origin.

Tipos Genéricos em métodos

- A assinatura do método pode ser diferente dos parâmetros de tipo genérico em uma definição de struct. Link do código.

```
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' };
        let p3 = p1.mixup(p2);
        println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Tipos Genéricos em métodos

- No exemplo anterior, temos X1 e Y1 como o tipo genérico para a estrutura.

```
struct Point<X1, Y1>
```

- E X2 Y2 para a assinatura do método.

```
fn mixup<X2, Y2>
```

- O método cria uma nova instância de Point com o valor de x do próprio Point (do tipo X1) e o valor de y do Point passado como argumento (do tipo Y2).

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

Tipos Genéricos em métodos

- Na função main, temos a criação de p1, com um x do tipo i32 e um y do tipo f64.
- p2 tem x sendo uma string slice e y um char.
- Como todos os tipo são genéricos, todos os valores podem ser de tipos diferentes.
- Por fim p1 chama o método mixup e passa p2 como parâmetro, que tem como resultado x igual a x de p1 e y igual a y de p2.

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

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Performance utilizando tipos genéricos

- A utilização de tipos genéricos não fazer o programa ficar mais lento, em comparação a utilização de tipos concretos.

- Rust realiza a monomorfização do código usando genéricos, em tempo de compilação

- A monomorfização é o processo de transformar código genérico em código específico preenchendo os tipos concretos que são usados durante a compilação.

- O compilador do Rust examina todos os lugares onde o código genérico é chamado e gera código para os tipos concretos com os quais o código genérico é chamado.

Performance utilizando tipos genéricos

- Vejamos como isso funciona utilizando a enumeração Option<T>.

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

- Quando o Rust for compilar esse código ele vai fazer a monomorfização do código.

- Durante esse processo o compilador identifica o tipo genérico e identifica os tipos de Option<T>, no caso i32 e f64. Depois substitui a definição genérica pelas específicas de i32 e f64.

Performance utilizando tipos genéricos

- O compilador vai gerar um código semelhante ao código ao lado.
- A opção genérica Option<T> foi substituída pelas definições específicas criadas pelo compilador.
- Como o compilador do Rust realiza essa modificação, não há um aumento no custo de tempo de execução ao usar genéricos.

```
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: Definindo Comportamento Compartilhado

- Traits são semelhante a interfaces em outras linguagens de programação.

- Uma trait define a funcionalidade que um tipo específico possui e pode compartilhar com outros tipos.

- Podemos usar traits para definir comportamentos compartilhados de maneira abstrata.

- Podemos usar limites de trait para especificar que um tipo genérico pode ser qualquer tipo que tenha determinado comportamento.

Definindo Traits

- O comportamento de um tipo consiste nos métodos que podemos chamar nesse tipo.

- Tipos diferentes compartilham o mesmo comportamento se pudermos chamar os mesmos métodos em todos esses tipos.

- As definições de trait são uma maneira de agrupar assinaturas de métodos para definir um conjunto de comportamentos

- Para entender melhor isso, vamos ter como exemplo:
  - Uma estrutura NewsArticle, que vai armazenar notícias de algum local do mundo.
  - Uma estrutura Tweet, que pode ter no máximo 280 caracteres e com os metadados, dizendo se é um novo tweet, um retweet ou uma resposta a outro tweet.
  - Queremos criar uma biblioteca de mídia, com o nome “aggregator”, que pode exibir resumos de dados que podem estar armazenados em uma instância de NewsArticle ou Tweet.

Definindo Traits

- Para fazer isso, vamos precisar de um resumo de cada tipo, podemos fazer esse resumo chamando o método summarize em uma instância.

src/lib.rs

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

- Podemos declarar um trait utilizando a palavra-chave “trait”, depois disso podemos escrever o nome da trait, nesse caso Summary.
- Dentro das chaves, declaramos as assinaturas dos métodos que descrevem os comportamentos dos tipos que a trait implementa.
- Em vez de escrever uma implementação entre chaves, colocamos um “;”. Já que cada tipo que implementa este trait deve fornecer seu próprio comportamento personalizado para o corpo do método.
- O compilador vai garantir todo tipo que tenha Summary, terá o método summarize.

Implementando trait em um tipo

- Código implementado no arquivo src/lib.rs

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

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)
    }
}
```

```
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)
    }
}
```

Implementando trait em um tipo

- No exemplo de código, a trait Summary, é utilizado por NewsArticle e Tweet, cada um com um retorno de summarize diferente.
- Para NewsArticle, temos os valores do título, o autor e a localização (headline, author e location) para criar o valor de summarize.
- Para Tweet, temos os valores do como o nome de usuário e o conteúdo do tweet (username e content). Para tweet, estamos assumindo que já seja menor que 280 caracteres.

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

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

Implementando trait em um tipo

- Para implementar a trait, utilizamos:

```
impl NomeDaTrait for NomeDoTipo
```

- Dentro do bloco, colocamos as assinaturas dos métodos que trait definiu.

- Em vez de adicionar “;” após cada assinatura, usamos chaves e preenchemos o corpo do método com o comportamento específico

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

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

Implementando trait em um tipo

- A forma como chamamos a trait, é semelhante a forma como chamamos os métodos regulares.

- A única diferença é que o usuário deve trazer o trait para o escopo, assim como os tipos.

```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

Implementando trait em um tipo

- Crates que podem implementar traits ou tipos de outra outras crates, desde que um deles seja local a crate.
- Por exemplo, podemos utilizar a trait Display da biblioteca padrão, em um tipo personalizado como Tweet. Porque Tweet é local a nossa crate aggregator.
- Também podemos implementar a trait Summary no tipo Vec<T> em nossa crate aggregator. Porque a trait Summary é local a nossa crate.
- Mas não podemos implementar a trait Display em Vec<T>. Porque nenhuma delas é local a nossa crate.
- Essa restrição faz parte de uma propriedade chamada coerência, e mais especificamente a regra do órfão.
- Essa regra garante que o código de outras pessoas não possa quebrar o seu código e vice-versa. Evitando utilizar traits com mesmo nome para o mesmo tipo.

Definições padrões

- As vezes é interessante ter uma implementação padrão para alguns ou todos os métodos de uma trait.
- Nesse exemplo, estamos utilizando uma string padrão, em vez de apenas definir o método.

src/lib.rs:

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

- Para implementar Summary para NewsArticle, deixamos as chaves vazias. Exemplo: impl Summary for NewsArticle {}.
- Não precisamos modificar nada em Tweet, para ele continuar funcionando.

Definições padrões

- Podemos utilizar o método summarize de NewsArticle, como no exemplo abaixo.

```
use aggregator::{self, NewsArticle, Summary};

fn main() {
        let article = NewsArticle {
        headline: String::from("Penguins win the Stanley Cup Championship!"),
        location: String::from("Pittsburgh, PA, USA"),
        author: String::from("Iceburgh"),
        content: String::from(
                "The Pittsburgh Penguins once again are the best \
                hockey team in the NHL.",
        ),
        };

        println!("New article available! {}", article.summarize());
}
```

Definições padrões

- Implementações padrão podem chamar outros métodos na mesma trait, mesmo que esses outros métodos não tenham uma implementação padrão.

- Dessa forma, uma trait pode fornecer muita funcionalidade útil e exigir dos implementadores apenas a especificação de uma pequena parte dela.

- Por exemplo, podemos definir Summary para ter um método summarize_author que é necessário. Depois definir o método summarize que utiliza summarize_author.

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

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

Definições padrões

- Para utilizar essa versão de Summary, vamos precisar definir summarize_author quando implementarmos a trait de um tipo.

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

- Depois de definir summarize_author, podemos chamar summarize na estrutura Tweet. A implementação padrão de summarize chamará summarize_author, que fornecemos.

- Como implementamos summarize_author, o trait Summary nos deu o comportamento do método summarize sem exigir que escrevêssemos mais código.

Definições padrões

- O código abaixo vai imprimir: “ 1 new tweet: (Read more from @horse_ebooks...)”

```
use aggregator::{self, Summary, Tweet};

fn main() {
        let tweet = Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
              "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        };

        println!("1 new tweet: {}", tweet.summarize());
}
```

- Note que não é possível chamar a implementação padrão a partir de uma implementação que substitui o mesmo método.

Trait como parâmetro

- Usar traits para definir funções que aceitam muitos tipos diferentes.

- Vamos exemplificar isso, mudando o exemplo do slide 26 e 29.

- Vamos utilizar a trait Summary, para definir a função notify que chama o método summarize no seu parâmetro item, que é um tipo que implementa a trait Summary.

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

Trait como parâmetro

- Em vez de um tipo concreto para o parâmetro item, especificamos a palavra-chave impl e o nome do trait.

- No corpo do notify, podemos chamar qualquer método em item que venha do trait Summary, como summarize

- Podemos passar qualquer instância de NewsArticle ou Tweet para notify.

- O código não será compilado, caso chame a função com qualquer outro tipo, como string ou i32, já que esses tipos não implementam Summary.

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

Sintaxe de trait bound

- O “impl trait” é uma forma simplificada da sintaxe de trait bound.
- Na sintaxe de trait bound, colocamos as restrições de trait junto à declaração do parâmetro de tipo genérico após “:” e dentro de “<>”.
- A sintaxe de “impl trait” é mais interessante de ser utilizada em casos simples, enquanto trait bound, pode ser mais interessante em alguns casos mais complexos.

Exemplo de impl trait:

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

Exemplo de trait bound:

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

Sintaxe de trait bound

- Caso queiramos que a trait Summary tenha dois parâmetros, podemos escrever utilizando “impl trait”

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

- Nesse caso item1 e item2 podem ser tipos diferentes, caso queiramos que eles sempre sejam tipos iguais, teremos que escrever com trait bound.

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

- Já que especificamos o tipo genérico T para item1 e item2. Eles serão restringidos a serem do mesmo tipo.

Especificando múltiplos trait bound com a sintaxe +

- Podemos especificar mais de um trait bound.
- Nesse exemplo, queremos que notify apresente a formatação de exibição de summarize para item. Na definição de notify, item deve implementar Display e Summary.
- Podemos fazer isso utilizando a sintaxe +.

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

- Essa mesma sintaxe, também é válida para trait bound com tipos genéricos.

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

- Com as duas restrições de traço especificadas, o corpo de notify pode chamar summarize e usar {} para formatar item.

Trait bound mais claras com a cláusulas where

- O uso de muitas restrições de traço pode tornar a assinatura da função difícil de ler.

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

- Podemos melhorar a leitura disso utilizando a cláusula where.

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

- O nome da função, a lista de parâmetros e o tipo de retorno estão próximos, tornando o código mais fácil de ler

Tipos de retorno que implementam trait

- Podemos usar a sintaxe impl Trait na posição de retorno para retornar um valor de algum tipo que implementa um traço.

- No exemplo abaixo, impl Summary está especificado como tipo de retorno.

- Então a função retorna um tipo que implementa o traço Summary. Nesse caso o Tweet.

```
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,
        }
}
```

Tipos de retorno que implementam trait

- A capacidade de especificar um tipo de retorno apenas pelo traço que ele implementa é especialmente útil no contexto de closures e iteradores.

- Closures e iteradores criam tipos que apenas o compilador conhece ou tipos que são muito longos para serem especificados.

- No entanto, você só pode usar impl Trait se estiver retornando um único tipo.

Tipos de retorno que implementam trait

```
fn returns_summarizable(switch: bool) -> impl Summary {
        if switch {
                NewsArticle {
                        headline: String::from(
                                "Penguins win the Stanley Cup Championship!",
                        ),
                        location: String::from("Pittsburgh, PA, USA"),
                        author: String::from("Iceburgh"),
                        content: String::from(
                                "The Pittsburgh Penguins once again are the best \
                                hockey team in the NHL.",
                        ),
                }
        } else {
                Tweet {
                        username: String::from("horse_ebooks"),
                        content: String::from(
                                "of course, as you probably already know, people",
                        ),
                        reply: false,
                        retweet: false,
                }
        }
}
```

Tipos de retorno que implementam trait

- Devido a limitações de como “impl Trait” é implementada no compilador, não é possível retornar dois tipos.

- Caso queira saber como implementar uma função que retorne dois tipos, utilizando trait objects, recomendo a leitura do capítulo 17.

Utilizando trait bound para implementar métodos condicionalmente

- Podemos implementar métodos condicionalmente para tipos que atendem aos traços especificados, utilizando um limite de traço com um bloco impl que utiliza parâmetros de tipo genérico.

Exemplo do src/lib.rs

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

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

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

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);
        }
    }
}
```

Utilizando trait bound para implementar métodos condicionalmente

- O tipo Pair<T> sempre implementa a função new para retornar uma nova instância de Pair<T>.

- No próximo bloco impl, Pair<T> só implementa o método cmp_display se seu tipo interno T implementar o trait PartialOrd que possibilita comparação e o trait Display que permite a impressão.

```
impl<T> Pair<T> {
        fn new(x: T, y: T) -> Self {
                Self { x, y }
        }
}

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);
                }
        }
}
```

Utilizando trait bound para implementar métodos condicionalmente

- Também podemos implementar condicionalmente um trait para qualquer tipo que implementa outro trait.

- Implementações de um trait em qualquer tipo que satisfaça as restrições do trait são chamadas de implementações genéricas.

- Por exemplo, a biblioteca padrão implementa o traço ToString para qualquer tipo que implemente o traço Display. O bloco impl na biblioteca padrão se parece com este código:

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

Utilizando trait bound para implementar métodos condicionalmente

- Como a biblioteca padrão tem essa implementação genérica, podemos chamar o método to_string definido pelo trait ToString em qualquer tipo que implemente o trait Display.

- Por exemplo, podemos transformar números inteiros em strings, por conta que inteiros implementam Display.

```
#![allow(unused)]
fn main() {
    let s = 3.to_string();
}
```

Utilizando trait bound para implementar métodos condicionalmente

- Trait e bound trait permite escrever um código que usa parâmetros de tipo genérico para reduzir a duplicação.

- Também especificam ao compilador que queremos que o tipo genérico tenha um comportamento específico.

- No Rust, alguns erros que seriam em tempo de execução em outras linguagens, são tratados em tempo de compilação. Por exemplo a chamada de um método não definido.

Validando referências com Lifetime

- Lifetimes são outro tipo de genérico.
- Ao invés de garantir que um tipo tenha o comportamento desejado, lifetimes asseguram que referências sejam válidas pelo tempo que precisamos delas.
- Não discutimos sobre lifetime no capítulo 4 (Referências e Borrowing). A referência em Rust tem uma lifetime, que é o escopo para o qual aquela referência é válida.
- Maior parte do tempo são implícitos e inferidos.
- Precisamos anotar lifetimes quando as lifetimes de referências podem estar relacionadas de diferentes maneiras.
- Rust nos exige anotar os relacionamentos usando parâmetros genéricos de lifetime para garantir que as referências reais usadas em tempo de execução sejam definitivamente válidas.

Prevenindo referências soltas com lifetime

- O principal objetivo do lifetime é impedir referências soltas.
- O exemplo do código ao lado não compila com sucesso, pois “r” tem a referência de “x”, mas quando ele vai consultar o valor para imprimir, x já não está mais na memória.
- Teremos o error[E0597]: `x` does not live long enough.
- Rust utiliza um verificador de empréstimo (borrow check) para verificar isso.
- Nesse mesmo exemplo, podemos perceber que Rust não possibilita uso de valores nulos, caso tente utilizar r antes de ter um valor atribuído a ele.

Exemplo:

```
fn main() {
        let r;

        {
                let x = 5;
                r = &x;
        }

        println!("r: {}", r);
}
```

O verificador de empréstimo

- O Rust tem um verificador de empréstimo, que compara os escopo para determinar se todos os empréstimos são válidos.
- 'a é o tempo de vida de r.
- 'b é o tempo de vida de x.
- O exemplo 1 é a demonstração do tempo de vida no código anterior.
- O exemplo 2, é uma possível correção para o código.

Exemplo 1:

```
fn main() {
        let r;                  // ---------+-- 'a
                                //          |
        {                       //          |
                let x = 5;      // -+-- 'b  |
                r = &x;         //  |       |
        }                       // -+       |
                                //          |
        println!("r: {}", r);   //          |
}                               // ---------+
```

Exemplo 2:

```
fn main() {
    let x = 5;                  // ----------+-- 'b
                                //           |
    let r = &x;                 // --+-- 'a  |
                                //   |       |
    println!("r: {}", r);       //   |       |
                                // --+       |
}                               // ----------+
```

Tempo de vida genéricas em funções

- Neste exemplo, vamos tentar implementar um programa que recebe 2 string slice e retorna a maior.
- Note que está faltando a implementação da função longest.
- Esse código deveria printar “The longest string is abcd”
- Queremos utilizar string slice, que são referências, para que longest não tenha o ownership dos parâmetros.

```
fn main() {
        let string1 = String::from("abcd");
        let string2 = "xyz";

        let result = longest(string1.as_str(), string2);
        println!("The longest string is {}", result);
}
```

Tempo de vida genéricas em funções

- Adicionando a função longest ao código, vamos perceber que não compila e teremos o erro abaixo.

Exemplo:

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++         ++         ++
```

Tempo de vida genéricas em funções

- A mensagem, diz que o tipo de retorno precisa de um parâmetro de tempo de vida genérico.
- Pois o Rust não consegue determinar se a referência que está sendo retornada se refere a x ou y.
- Também não sabemos, pois o bloco if retorna uma referência de x e o else retorna uma referência de y.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Sintaxe de anotação de tempo de vida

- As anotações de tempo de vida não alteram a duração de nenhuma das referências.
- Elas descrevem as relações entre os tempos de vida de várias referências
- Funções podem aceitar referências com qualquer tempo de vida ao especificar um parâmetro de tempo de vida genérico.

Sintaxe de anotação de tempo de vida

- A sintaxe da anotação de tempo de vida é dada por: os nomes dos parâmetros de tempo de vida devem começar com um apóstrofo (').

- Normalmente os nomes são curtos e escritos com letras minúsculas. Sendo 'a a mais comum para a primeira anotação.

- Colocamos as anotações de parâmetro de tempo de vida após o & de uma referência.

```
&i32     // a reference
&'a i32      // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

Anotações de Tempo de Vida em Assinaturas de Funções

- Para usar anotações de tempo de vida em assinaturas de funções, precisamos declarar os parâmetros de tempo de vida genéricos dentro <>, entre o nome da função e a lista de parâmetros
- Queremos a assinatura expresse a seguinte restrição: A referência retornada será válida enquanto ambos os parâmetros forem válidos.
- Nomearemos o tempo de vida como 'a e o adicionaremos a cada referência.
- Com essa função já é possível rodar o código com sucesso.

Exemplo:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Anotações de Tempo de Vida em Assinaturas de Funções

- Agora a função informar para o Rust que algum tempo de vida 'a, a função recebe dois parâmetros, ambos os quais são fatias de string que vivem pelo menos tanto quanto o tempo de vida 'a. Também informa que a fatia de string retornada pela função viverá pelo menos tanto quanto o tempo de vida 'a

- Ou seja, Informa que os valores retornados por longest tem um tempo de vida igual ao menor dos tempos de vida dos valores referenciados pelos argumentos da função.

- Estamos somente especificando que o verificador de empréstimos deve rejeitar quaisquer valores que não se conformem a essas restrições.

Anotações de Tempo de Vida em Assinaturas de Funções

- Vamos ver como as anotações de tempo de vida restringem a função longest ao passar referências que têm tempos de vida concretos diferentes.

- Neste exemplo:
  - string1 é válida até o final do escopo externo.
  - string2 é válida até o final do escopo interno
  - result referência algo que é válido até o final do escopo interno.
  - println! usa result no escopo interno

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

Anotações de Tempo de Vida em Assinaturas de Funções

- Nesse novo exemplo que não compila, o tempo de vida da referência em result deve ser o menor tempo de vida entre os dois argumentos.
  - string1 é válida até o final do escopo externo.
  - string2 é válida até o final do escopo interno
  - result referência algo que é válido até o final do escopo interno.
  - println! usa result no escopo externo

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

Anotações de Tempo de Vida em Assinaturas de Funções

- O Rust informa na mensagem de erro, que string2 precisa ser válida para o println!.

- O Rust sabe disso porque anotamos os tempos de vida dos parâmetros da função e dos valores de retorno usando o mesmo parâmetro de tempo de vida 'a.

```
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^ borrowed value does not live long
enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
  |                                          ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
```

Anotações de Tempo de Vida em Assinaturas de Funções

- Podemos observar no código, que a string1 é maior que string2, então result deveria referenciar string1, que está disponível para o println!. Mas mesmo assim o código não compila.

- O compilador do Rust não consegue ver que a referência é válida neste caso. Pois, o tempo de vida da referência retornada pela função longest é a mesma de string2, que é a menor entre as duas.

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

Pensando em Termos de Tempos de Vida

- A maneira como você precisa especificar os parâmetros de tempo de vida depende do que sua função está fazendo.

- Por exemplo, agora vamos querer que a função sempre retorne a primeira string.

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

- 'a é somente um parâmetro de tempo de vida para x, pois y não tem relação a x e nem ao valor de retorno.

Pensando em Termos de Tempos de Vida

- O tipo de retorno precisa coincidir com o parâmetro de tempo de vida de um dos parâmetros.

- Se a referência retornada não se referir a um dos parâmetros, ela deve se referir a um valor criado dentro desta função. Porém isso iria criar uma referência pendurada, como mostra o código que não compila.

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

```
error[E0515]: cannot return reference to local variable
`result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^^ returns a reference to data owned by
the current function
```

Pensando em Termos de Tempos de Vida

- O problema é que o valor de result só é válido até o final da função, então o retorno da função é inválido.

- Neste caso, a melhor solução seria retornar um tipo de dado de propriedade em vez de uma referência. Para a função chamadora poder lidar com a limpeza da memória desse valor.

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Anotações de tempo de vida em definições de estruturas

- Para uma estrutura armazenar referências, é preciso adicionar uma anotação de tempo de vida para cada referência na definição da estrutura.
- Declaramos o nome do parâmetro genérico de tempo de vida dentro de <> após o nome da struct. Link do código.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Anotações de tempo de vida em definições de estruturas

- Aquele código tem um campo, part, que armazena um string slice, que é uma referência.
- Essa anotação, falar que uma instância de ImportantExcerpt não pode viver mais que a referência de seu campo part.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Elisão de Tempo de Vida

- Vimos que precisamos especificar o tempo de vida das referências, em funções e estruturas que utilizam referências.
- No capítulo 4, tinhamos o código que compilava sem essa anotação.

Exemplo:

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

```
fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices
    // already,

    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Elisão de Tempo de Vida

- A razão para isso começa na versões pré-1.0 do Rust, esse código não compila, porque toda referência precisava de um tempo de vida explícito.

- Nesse tempo, precisaria de uma assinatura:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

- A equipe, percebeu uma repetição de um padrão determinístico para essa assinatura, então programaram esse padrão no código do compilador.

- Então o verificador de empréstimos passou a inferir os tempos de vida nessas situações e passou a não precisar das anotações explícitas.

- Assim como esse padrão, outros foram adicionados, formando as regras de elisão de tempo de vida, que o compilador segue.

Elisão de Tempo de Vida

- Se o Rust seguir essas regras, mas continuar tendo ambiguidade, ele vai retornar uma mensagem de erro pedindo para adicionar as anotações de tempo de vida.

- Tempos de vida nos parâmetros de funções ou métodos são chamados de tempos de vida de entrada.

- Tempos de vida em valores de retorno são chamados de tempos de vida de saída.

Elisão de Tempo de Vida

- São utilizados 3 regras para determinar os tempos de vida das referências, quando não explícitos.
- A primeira regra se aplica ao tempo de vida de entrada.
- A segunda e terceira regra se aplica ao tempo de vida de saída.
- Essas regras se aplicam tanto a definições de fn quanto a blocos impl.

Elisão de Tempo de Vida

- A primeira regra é que o compilador atribui um parâmetro de tempo de vida para cada parâmetro que é uma referência.

  Por exemplo: `fn foo<'a>(x: &'a i32)`

- Uma função com dois parâmetros recebe dois parâmetros de tempo de vida. E assim por diante.

  Por exemplo: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`

Elisão de Tempo de Vida

- A segunda regra é que, se houver exatamente um parâmetro de tempo de vida de entrada, esse tempo de vida será atribuído a todos os parâmetros de tempo de vida de saída.

  Por exemplo: `fn foo<'a>(x: &'a i32) -> &'a i32`.

- A terceira regra é que, se houver vários parâmetros de tempo de vida de entrada, mas um deles for &self ou &mut self porque este é um método, o tempo de vida de self será atribuído a todos os parâmetros de tempo de vida de saída.

- Essa terceira regra reduz o uso de símbolos, deixando mais fácil de ler e de escrever métodos.

Elisão de Tempo de Vida

- Vamos verificar passo a passo, como o compilador segue essas regras. Dado a assinatura: `fn first_word(s: &str) -> &str {`

- O compilador aplica a primeira regra, que especifica que cada parâmetro recebe sua própria duração:

```
fn first_word<'a>(s: &'a str) -> &str {
```

- A segunda regra se aplica porque há exatamente um parâmetro de entrada.

- A segunda regra especifica que a duração do único parâmetro de entrada é atribuída à duração de saída:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

- Agora todas as referências possuem a duração e a assinatura é válida.

Elisão de Tempo de Vida

- Vamos olhar outro exemplo com a função longest, sem nenhum parâmetro de tempo de vida: `fn longest(x: &str, y: &str) -> &str {`
- Aplicando a primeira regra: `fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {`
- A segunda regra não se aplica porque há mais de uma duração de entrada.
- A terceira regra também não se aplica, porque longest é uma função em vez de um método, então nenhum dos parâmetros é self.
- Depois de passar por todas as três regras, ainda não conseguimos descobrir qual é a duração do tipo de retorno. Então o compilador retornar um erro.
- Porque a terceira regra só se aplica em assinaturas de métodos, vamos olhar tempos de vida nesse contexto agora

Anotações de Tempo de Vida em Definições de Métodos

- Os nomes de tempo de vida para campos de struct sempre precisam ser declarados após a palavra-chave “impl”. Em seguida, usados após o nome da struct, porque essas lifetimes fazem parte do tipo da struct.

- Em assinaturas de métodos dentro do bloco impl, as referências podem estar vinculadas à lifetime de referências nos campos da struct, ou podem ser independentes.

- Além disso, as regras de elisão de lifetime muitas vezes fazem com que as anotações de lifetime não sejam necessárias em assinaturas de métodos.

- Primeiro, usaremos um método chamado level cujo único parâmetro é uma referência a self e cujo valor de retorno é um i32, que não é uma referência a nada:

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

Anotações de Tempo de Vida em Definições de Métodos

- A declaração de parâmetro de lifetime após impl e seu uso após o nome do tipo são necessários, mas não somos obrigados a anotar a lifetime da referência a self devido à primeira regra de elisão.

- Aqui está um exemplo em que a terceira regra de elisão de lifetime se aplica:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

- Existem dois lifetimes de entrada, então o Rust aplica a primeira regra de elisão de lifetime e atribui a ambos &self e announcement seus próprios lifetimes. Em seguida, porque um dos parâmetros é &self, o tipo de retorno recebe o lifetime de &self, e todos os lifetimes foram considerados.

Um tempo de vida estático

- O tempo de vida estático é especial, pois ele denota que a referência afetada pode viver durante toda a duração do programa.

- Todas as literais de string têm um tempo de vida estático, que pode ser denotado por:

```
let s: &'static str = "I have a static lifetime.";
```

- O texto desta string é armazenado diretamente no binário do programa, o que está sempre disponível. Logo seu tempo de vida é estático.

Um tempo de vida estático

- Você pode ver sugestões de usar o tempo de vida 'static em uma mensagem de ajuda de erro.

- Mas antes de especificar 'static como o tempo de vida para uma referência, pense sobre se a referência que você tem é uma que vive todo o tempo de vida do seu programa ou não.

- Na maior parte do tempo, o problema no código é uma tentativa de criar uma referência solta ou uma incompatibilidade dos tempos de vida disponíveis, e a solução é consertar esses problemas, não especificar um tempo de vida 'static.

Parâmetros de tipos genéricos, trait bound e tempos de vida juntos

- Vamos ver um exemplo da sintaxe que especifica os parâmetros de tipo genérico, limites de trait e lifetimes em uma única função.

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

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result =
    longest_with_an_announcement(
        string1.as_str(),
        string2,
        "Today is someone's birthday!",
    );
    println!("The longest string is {}", result);
}
```

Parâmetros de tipos genéricos, trait bound e tempos de vida juntos

- Ela possui um parâmetro extra chamado ann do tipo genérico T, que pode ser preenchido por qualquer tipo que implemente o trait Display, conforme especificado pela cláusula where.
- Este parâmetro extra será impresso usando {}, razão pela qual é necessária a restrição do trait Display.
- Como as lifetimes são um tipo de genérico, as declarações do parâmetro de lifetime 'a e do parâmetro de tipo genérico T vão na mesma lista dentro dos colchetes angulares após o nome da função.

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

fn longest_with_an_announcement<'a, T>(
        x: &'a str,
        y: &'a str,
        ann: T,
) -> &'a str
where
        T: Display,
{
        println!("Announcement! {}", ann);
        if x.len() > y.len() {
                x
        } else {
                y
        }
}
```

Referências

Slides Baseados no Capítulo 10 do Rust Book
https://doc.rust-lang.org/book/ch10-00-generics.html