# Rust

## Concorrência e alta performance com segurança

Edição atualizada


Casa do Código | alura

MARCELO CASTELLANI
WILLIAN MOLINARI

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

*Edição*

Vivian Matsui

*Capa*

Design Grupo Alura

[2022]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

casadocodigo.com.br

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

grupo alura

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código é a editora do Grupo Alura, grupo que nasceu da vontade de criar uma plataforma de ensino com o objetivo de incentivar a transformação pessoal e profissional através da tecnologia.

Juntas, as empresas do Grupo constroem uma verdadeira comunidade colaborativa de aprendizado em programação, negócios, design, marketing e muito mais, oferecendo inovação na evolução dos seus alunos e alunas através de uma verdadeira experiência de encantamento.

Venha conhecer os cursos da Alura e siga-nos em nossas redes sociais.


alura.com.br


@casadocodigo


@casadocodigo

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

ISBN

Impresso e PDF: 978-85-5519-302-6

EPUB: 978-85-94188-34-2

MOBI: 978-85-94188-35-9

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

**A arte da capa deste livro faz uso do logo oficial do Rust, sob a licença CC BY 4.0.**

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

# DEDICATÓRIA

## MARCELO CASTELLANI

Este livro é dedicado a todos e a todas que gostam de escrever código e que sentem prazer em ver as coisas funcionando. Que colocam um sorriso honesto e sincero no rosto quando tudo vai bem, como meu filho Nicolas no dia em que fizemos juntos um estilingue.

Dedico-o a todas as pessoas que conseguem se divertir aprendendo algo novo, que sabem que o aprendizado de ontem pode não valer mais hoje e que sentem um orgulho enorme de suas conquistas – assim como o Nicolas, meu filho, que chegou em casa um dia e pediu um papel e uma caneta para me mostrar que havia aprendido a escrever seu nome e o meu.

Este livro é dedicado a todos que ainda estão descobrindo o mundo, não importa sua idade.

## WILLIAN MOLINARI (POTHIX)

Dedico este livro à minha esposa Jacqueline e aos meus filhos Abel e Anne. Eles estão tão presentes neste material quanto eu, pois todo o tempo que eu dediquei para escrever este livro foi uma troca do tempo que eu dedico a eles.

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

# AGRADECIMENTOS

## MARCELO CASTELLANI

Agradeço a todos da Casa do Código, aos envolvidos com a linguagem Rust aqui e no mundo, a toda minha família, em especial aos meus pais, Adelia e Nelson, por me apoiarem desde que eu era um jovem Marcelo.

Agradeço também ao meu amigo Willian Molinari, o PotHix, que se dispôs a me ajudar na revisão deste material, além de ter feito um trabalho incrível em outro livro da Casa do Código, o **Desconstruindo a web**, no qual mostra tudo o que acontece entre sua requisição ao browser e a página bonitinha desenhada na sua tela. Vale a pena dar uma olhada.

Para mais informações, visite o site: https://desconstruindoaweb.com.br.

## WILLIAN MOLINARI (POTHIX)

Agradeço à minha esposa por me ajudar a conseguir tempo o suficiente para me dedicar a este projeto nesse momento em que o tempo está bem escasso. Com a chegada dos nossos filhos, a minha admiração por ela só cresce. Esse projeto se torna pequeno perto de tamanho desafio que ela segura sozinha durante boa parte do dia.

Agradeço também à minha família, **Wilson**, **Evanilda** e **Lilian Molinari**, por me darem o alicerce para formar meu caráter e

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

alcançar minhas conquistas. Se cheguei aonde estou, devo isso a eles.

Agradeço ao Marcelo Castellani por me oferecer a chance de participar da escrita deste livro como coautor depois de participar da revisão da primeira edição.

Ambos os autores também agradecem a todas as pessoas envolvidas na linguagem Rust, principalmente ao Graydon e a Mozilla por terem iniciado o projeto.

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

# QUEM ESCREVEU ESTE LIVRO?

**Marcelo F. Castellani** é desenvolvedor de software há mais de 25 anos. Iniciou como estagiário na Companhia Telefônica da Borda do Campo, onde viu o surgimento da internet no país. Trabalhou por 14 anos com o desenvolvimento de software embarcado de alta performance, voltado à automação bancária e comercial (com C, C++, Java e Assembly) na Itautec S/A e, posteriormente, na OKI Brasil.

Desde 2007, é desenvolvedor Ruby e Rails, tendo participado da fundação do GURU-SP e palestrado em diversos eventos relacionados. Entusiasta de software livre, foi membro do projeto NetBeans, reconhecido pela Sun Microsystems, em 2008, como um dos principais colaboradores do Brasil.

Como escritor, foi colunista fixo da revista *Wap & Internet Móvel* e publicou artigos em diversas outras, como a *Java Magazine* e a *Internet.BR*. Também é autor do livro *Certificação SCJA - Guia de Viagem*.

Trabalhou profissionalmente com Rust entre os anos de 2018 e 2019 numa bolsa de criptomoedas em Oakland, na Califórnia. Participa eventualmente da organização do grupo de usuários de Rust de São Paulo.

No tempo livre toca bateria, guitarra e canta na banda Vingança Vil.

**Willian Molinari**, mais conhecido como **PotHix**, é formado em Sistemas de Informação e trabalha com desenvolvimento de

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

software há mais de 14 anos. Durante sua carreira, passou por diversas linguagens de programação: começou com ASP, investindo boa parte da carreira em Ruby e Python, brincou com outras linguagens, como Golang e Elixir, e é entusiasta de Rust. Ele trabalhou tanto em pequenas empresas e startups com cerca de 5 pessoas, dentro e fora do Brasil, quanto em grandes empresas com mais de 1.200 funcionários.

É o autor do livro *Desconstruindo a web* (https://desconstruindoaweb.com.br) que mostra tudo o que um desenvolvedor web precisa saber sobre uma requisição web. O livro explora o que acontece desde o primeiro "enter" no navegador, passando por sistema operacional, internet, servidor de aplicação e web, framework e retorno até a página estar completamente carregada.

Willian é um dos fundadores do grupo de usuários Ruby de São Paulo (**Guru-SP**), um dos principais organizadores do meetup de Rust em São Paulo (**Rust-SP**), organizador da Rust LATAM (https://rustlatam.org) Montevideo 2019 e Rust LATAM México (2020, cancelada devido à pandemia). Além de organizar meetups e conferências, é palestrante e já fez mais de 70 palestras pelo Brasil e exterior.

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

# MAIN()

Para a maioria dos programadores e programadoras, a função `main` é comumente o ponto de entrada para um complexo e fascinante universo: o seu código. Seja ela em Java, C ou Rust, `main` é a porta de entrada dos bits e bytes ordenados marotamente, capazes de tomar alguma decisão e gerar o resultado esperado, ou um erro qualquer inesperado.

Este é o ponto de partida deste livro. É aqui que vamos nos apresentar e mostrar nossos objetivos, dizer quem queremos atingir e contar nossos planos de dominação mundial.

Sim, dominação mundial. Não esperamos menos dessa poderosa linguagem chamada Rust. Desde o começo, ela não foi feita com objetivos menores do que aqueles dos vilões dos quadrinhos. Ela veio para mudar o jogo e mostrar que C é velha e que C++ é complicada demais para qualquer um.

## Por que um livro sobre Rust?

Quando falamos em linguagens de programação voltadas à construção de sistemas que exigem performance e agilidade, Rust tem ganhado muita notoriedade. Ela visa aposentar clássicos como *C* e *C++* na construção de aplicativos complexos, como browsers ou sistemas operacionais. Ao mesmo tempo, a linguagem também bate de frente com outras linguagens como *Go* para desenvolvimento de CLIs (*Command Line Interfaces*, ou interfaces de linha de comando), APIs, frameworks de cryptomoedas e muito mais.

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Para mostrar um pouco dessa versatilidade do uso do Rust atualmente, podemos citar alguns projetos:

- **Servo** (https://github.com/servo/servo): um motor multiplataforma para navegadores.
- **RedoxOS** (https://www.redox-os.org/): um sistema operacional.
- **ripgrep** (https://github.com/BurntSushi/ripgrep): um substituto do `grep` mais moderno e com mais funcionalidades.
- **Starship** (https://starship.rs/): um prompt para terminais.
- **Rocket** (https://rocket.rs): um framework web simples parecido com o Sinatra (Ruby), Flask (Python) e outros.
- **Yew** (https://yew.rs): uma biblioteca para desenvolvimento front-end usando Rust e Web Assembly.

Rust permite a criação de programas que rodem no metal, criando código de máquina nativo, mas isso não quer dizer que ele serve apenas para resolver problemas de baixo nível. Rust é uma linguagem muito versátil e atende bem muitos tipos de demandas.

O motivo de a linguagem fazer muito sucesso para tarefas de baixo nível é a sua capacidade de criar código seguro de verdade (e vamos falar muito sobre isso ao longo deste livro). Desenvolver um código seguro em C ou C++ não é tão simples, pois sempre acaba sobrando um ponteiro maroto por aí que, na pior hora, vai dar alguma dor de cabeça. É notório o vídeo do Bill Gates tendo de lidar com uma tela azul da morte na apresentação de uma versão do Microsoft Windows (https://youtu.be/IW7Rqwwth84).

Tudo isso somado com a escassez de material completo em português já são ótimos motivos para termos um livro sobre Rust,

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

mas acho que você precisa de mais para se convencer. Por vários anos, Rust foi considerada a linguagem mais amada pelos desenvolvedores de software de todo o mundo na pesquisa anual do Stack Overflow (veja o link disponível no capítulo *Primeiros passos*). Se amor não for motivo suficiente para lhe convencer, não sabemos mais o que pode ser.

Brincadeiras à parte, este livro não busca ser uma versão final e definitiva de Rust. Ele serve como um bom ponto de partida, apresentando aplicações simples e autossuficientes, mas satisfatórias, para exemplificar os conceitos apontados. Por ser uma linguagem de propósito geral, o limite para criação com Rust é a sua imaginação. De jogos a sistemas embarcados, ela veio para ser a próxima geração de linguagem que todos devem conhecer, assim como o C é hoje.

## Para quem é este livro?

Este livro definitivamente não é para iniciantes. Se você nunca escreveu um código em sua vida, você provavelmente vai encontrar na leitura deste material algo demasiadamente complexo. O foco deste livro é mostrar como a linguagem Rust ajuda você a criar um bom software e, para isso, você precisa ter ao menos experiência básica fazendo isso.

Se você já conhece C, C++, D, Go, Pascal ou outra linguagem compilada, ele provavelmente será fácil para você. Apesar da sintaxe diferente, Rust é uma linguagem de programação de propósito geral, então ela tem condicionais, variáveis, palavras-chaves e tudo mais que outras linguagens também apresentam. Por ser compilada, ela possui todo o fluxo de codificar, compilar e

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

executar, e todas as facilidades e dificuldades que esse modelo de programação suporta.

Se você vem de Java ou .NET, saiba que não há o conceito de IDE principal (ou mais conhecidas, no caso de Java) para Rust, e várias pessoas ainda usam apenas um editor de texto bem customizado. Com isso, você não verá exemplos envolvendo coisas específicas de IDEs como acontece no Eclipse ou no Visual Studio. Este livro não se baseia em nenhum editor de texto ou IDE, a forma de programar deve funcionar em qualquer editor, então cabe a você escolher a sua ferramenta preferida para o trabalho.

Se você vem do Ruby ou do Python, acredito que, mesmo com as diferenças entre o processo de lidar com linguagens interpretadas e compiladas, você tenha tudo para achar Rust uma linguagem muito divertida. A linguagem faz uso de alguns conceitos parecidos, como *crates*, que são como as `gems` ou os pacotes `pip` , e também faz uso de estruturas bem definidas de namespaces.

Apesar da linguagem não vir "com baterias inclusas", podemos dizer que as ferramentas inclusas na linguagem conseguem prover essa sensação de uma forma bem simples e configurável. Diria que, assim como Ruby e Python, você consegue ter essa sensação de um ecossistema completo (ou "linguagem com baterias inclusas") e se você entende o que isso quer dizer, tenha certeza de que este livro é para você.

## O que preciso para acompanhar a leitura?

Todo o material aqui apresentado foi compilado com o **Rustc 1.48.0**, portanto você precisará das ferramentas da linguagem

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

instaladas, preferencialmente nessa versão, mas versões mais novas também são bem-vindas. Recomendamos que você use o editor de textos de sua preferência, com alguma extensão para destaque da sintaxe do Rust ou até mesmo alguma IDE, se assim se sentir mais confortável.

Rust é uma linguagem multiplataforma, ou seja, você pode utilizá-la no *MacOS* ou no *Microsoft Windows*, mas todo o processo de instalação apresentado neste livro é para *GNU/Linux*. Existem referências para os outros sistemas operacionais, que serão citadas em um momento mais oportuno.

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

# Sumário

**1 Primeiros passos** **1**

1.1 Por que uma nova linguagem de programação? 5

1.2 Um pouco de história 8

1.3 O que é Rust? 11

1.4 O que preciso instalar em meu computador? 11

1.5 Pronto para o Alô Mundo? 18

**2 Começando no Cargo** **25**

2.1 Preludes 25

2.2 Crates, cargo e outras ferramentas 27

2.3 Criando um projeto com o Cargo 28

2.4 Utilizando extensões 37

2.5 Outros utilitários do Cargo 43

**3 Mergulhando no oceano Rust** **50**

3.1 Atribuição e vinculação de variáveis 51

3.2 Funções 59

3.3 Tipos de dados em Rust 65

3.4 Agrupando em módulos 84

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

3.5 Comentários 86
3.6 O bom e velho if 87
3.7 Busca de padrões com match 89
3.8 While 96
3.9 Loop 97
3.10 For e ranges 99

**4 Traits e estruturas 101**
4.1 Derivando 107
4.2 PartialEq e Eq 110
4.3 PartialOrd e Ord 117
4.4 Operações aritméticas e de bit 122

**5 Vetores e iteradores 126**

**6 Strings e Slices 143**

**7 Tipos de dados genéricos 158**

**8 Alocação e gerenciamento de memória 165**
8.1 Gerenciamento de memória 165
8.2 Escopo de variáveis 166
8.3 Casting 168
8.4 Ponteiros, box e drop 171
8.5 Ownership 178
8.6 Borrowing 181

**9 Processamento paralelo e assíncrono 186**
9.1 Processamento paralelo e threads 187
9.2 Criando uma thread 189

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

9.3 Movendo o contexto 190
9.4 Entendendo channels 193
9.5 Processamento assíncrono e o padrão async/await 197

**10 Macros 208**
10.1 Por que macros? 208
10.2 Macros declarativas 209
10.3 Recursividade 212
10.4 Exemplo usando uma árvores de tokens 217
10.5 Macros procedurais 222
10.6 Por que usamos macros? 226

**11 Testar, o tempo todo 228**
11.1 A macro panic! 228
11.2 Macros de asserção 234
11.3 Desempacotamento e o operador ? 236
11.4 Escrevendo testes 244

**12 Alguns ponteiros para Rust 254**
12.1 Rust para a web 254
12.2 Rust vs. Go 256
12.3 Rust e Webassembly 257
12.4 Rust para software embarcado 260

**13 O começo de uma jornada 265**
13.1 Material online 266

Versão: 26.7.18

E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 1

# PRIMEIROS PASSOS

Aprender uma nova linguagem de programação envolve, entre outras coisas, abrir a mente para novos conceitos, paradigmas e maneiras de fazer algo que já fazemos. É um processo cíclico de repensar, enquanto aplicamos o que já sabemos durante o processo de assimilação.

Rust é um desses casos. O que mais nos chamou a atenção sobre a linguagem foi quando percebemos que sua performance era comparável à do C++. Tudo isso além de ser mais segura, concorrente e fácil de usar do que o seu predecessor. O Marcelo trabalhou por muito tempo com C++ e pode dizer que a clássica frase do Tio Ben não se aplica tanto a nenhuma outra situação do mundo quanto esta: "Com grandes poderes vêm grandes responsabilidades".

Trabalhar com C++ lhe dá poderes quase infinitos. Entretanto, a capacidade de fazer uma bobagem e ficar por horas investigando até achar o problema é um brinde que você não vai querer.

Claro que em Rust, como em qualquer outra linguagem de programação, é possível fazer bobagens. A parte interessante é que Rust foi desenvolvida pensando em evitar que quem está programando cometa alguns tipos de erros. O bom compilador é



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

aquele que não só transforma código legível em código de máquina, mas aquele que também ajuda quem programa a fazer o melhor código e evitar erros que podem ser evitados.

A linguagem faz um gerenciamento de memória manual, que não usa coletor de lixo (*garbage collector*) mas também não deixa as amarras soltas. Esse processo evita aquelas exceções de ponteiro nulo que nos fazem ligar para casa e avisar que a noite será longa. Isso é ainda mais importante quando pensamos em concorrência. Felizmente podemos dizer que ela foi feita para ser segura a ponto de possibilitar uma concorrência limpa sem a famigerada *race condition* a nível de memória.

Rust possui um livro mantido pela comunidade, o *Rustonomicon* (https://doc.rust-lang.org/nomicon/), que ensina as artes ocultas para criar código não seguro. Pode parecer uma piada, mas o objetivo é mostrar que, com Rust, a criação de um código que vai explodir na sua cara talvez seja mais difícil do que escrever um código que não explodirá. Isso, quando comparado ao desenvolvimento com C++, é como acordar em um paraíso em dia de sol.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**O Rustonomicon**

O nome Rustonomicon é uma piada com o *Necronomicon*, o livro fictício criado por H. P. Lovecraft e usado em sua literatura de terror fantástico como um documento real banido pelo Papa Gregório IX, em 1232. Citado em diversas obras de ficção, o Necronomicon é personagem central dos filmes *Evil Dead*, de Sam Raimi, e da série *Ash versus Evil Dead*.

Ao começarmos a ler sobre Rust e seu ecossistema, em meados de 2015, nós conhecemos o projeto Servo (https://servo.org/), autointitulado um browser engine moderno e de alta performance para aplicações e uso em sistemas embarcados. O Servo, mantido pela Mozilla Research por muitos anos e que hoje é parte da Linux Foundation, é totalmente escrito em Rust e usa o `cargo`, o gerenciador de pacotes do Rust, como ponto de partida. O projeto atualmente funciona em Linux, macOS, Windows, Android e em outras plataformas que possuem o navegador Firefox ou sua engine interna (até mesmo o finado Firefox OS/Gonk).

O fato de ter Rust sendo executado em um projeto desse tipo nos chamou muito a atenção. Um browser engine é o responsável por pegar aquele código HTML que vem do servidor e interpretar, transformando-o em algo visualmente agradável e fácil de ler (se o responsável pelo site ajudar, claro). Além de processar o HTML, ele ainda deve baixar os arquivos relacionados (imagens, estilo etc.), processá-los, desenhá-los na tela no lugar certo e mantê-los lá



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

quando você mover a barra de rolagem.

> **Nota sobre browser engines**
>
> Um dos autores deste livro (PotHix) é um grande curioso sobre como as coisas funcionam na internet e escreveu outro livro pela Casa do Código que descreve todo o processo de uma requisição web, incluindo como a página é renderizada no navegador. Não é do escopo deste livro ir mais a fundo sobre o tanto de trabalho que um browser engine tem, mas se você tiver interesse em saber mais sobre esse processo e outros mais, considere ler o livro *Desconstruindo a web*. Mais informações em https://desconstruindoaweb.com.br.

Enfim, é um projeto realmente complicado, todo feito em Rust. Não só complicado pelo tamanho da especificação do HTML e CSS recente, mas pela necessidade de ser veloz e ágil, de forma que a navegação seja fluida e constante. Com isso em mente, pensamos: "Ou o pessoal da Mozilla estava maluco, querendo promover uma nova linguagem de programação, ou haviam desenvolvido algo realmente capaz de ser comparado ao C++, sem seus brindes.".

Nós dois começamos separadamente a desenvolver uma série de pequenos projetos usando Rust apenas por diversão, e ficávamos mais animados a cada compilação bem-sucedida. Começamos a frequentar os fóruns, criar encontros presenciais, organizar conferências nacionais e internacionais e ler freneticamente sobre essa nova linguagem, com uma euforia que



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

só havíamos sentido quando aprendemos Ruby.

Neste primeiro capítulo, vamos mostrar algumas das características dessa linguagem e contar um pouco de sua história, desde quando era apenas um projeto pessoal de Graydon Hoare até ser celebrada como a tecnologia mais amada pela pesquisa do Stack Overflow por vários anos consecutivos (https://insights.stackoverflow.com/survey).

## 1.1 POR QUE UMA NOVA LINGUAGEM DE PROGRAMAÇÃO?

Talvez a melhor pergunta seja: e por que não?

Existem centenas de linguagens de programação por aí, das mais famosas, como Java, Ruby, C, Python, C++ e C# às mais desconhecidas como a Parla – uma linguagem escrita pelo Marcelo Castellani apenas para entender como isso funciona. Muitas são de propósito geral e servem para você fazer praticamente qualquer coisa, e muitas são focadas em resolver um problema específico, como o SQL – uma linguagem para manipulação de bancos de dados.

Quando uma nova linguagem aparece, a primeira pergunta que alguém faz é: por que precisamos de mais uma? Por que mais um complexo arcabouço de palavras-chave e definições apenas para transformar uma lógica etérea em algo repetível e verificável?

Rust surgiu para preencher um espaço em aberto. Seu objetivo é ser uma linguagem que gere código compilado, rápido e seguro, e que possa ser usada para o desenvolvimento de ferramentas que exigem performance. Isso se aplica principalmente a sistemas que


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

precisam ser bem enxutos e sem custo de execução (sem coletor de lixo, por exemplo), como sistemas operacionais ou browser engines.

Alguns podem dizer que esse espaço, na verdade, não existia. Temos C++ e mais de trinta anos de melhorias, desde o seu surgimento em 1979 até os dias de hoje. Temos C e seus mais de quarenta anos de melhorias, desde seu surgimento em 1972 aos dias atuais. Temos Go e, claro, D, que possuem características bem parecidas com Rust. Mas todas elas ainda possuem algum ponto que a linguagem Rust vem para cobrir.

Em um post interessante sobre quem vai realmente substituir a C no futuro, Andrei Alexandrescu, o arquiteto por trás da linguagem D, enumera pontos positivos e negativos de cada uma das candidatas (no caso D, Go e Rust). O que ele fala sobre Rust é realmente interessante, leia a seguir um trecho traduzido e ligeiramente adaptado para fazer sentido, mas mantendo o ponto de vista original, sem distorções.

> "Deixe-me voltar a lembrar que esta é apenas a minha opinião. Acho que Rust está enfrentando alguns desafios interessantes:
>
> **Uma personalidade desarmônica** – Olhar para qualquer quantidade de código Rust evoca a piada 'amigos não deixam amigos pularem o dia de malhar a perna' e as imagens em quadrinhos dos homens com torsos do Hulk descansando em pernas finas. Rust coloca a segurança e o gerenciamento de memória no centro de tudo. Infelizmente, esse quase nunca é o domínio do problema, o que significa que uma grande parte



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

do pensamento e da codificação é dedicada essencialmente a um trabalho clerical (que linguagens com um coletor de lixo automatizam por baixo dos panos). Segurança e recuperação de memória determinista são problemas complicados, mas não são os únicos, nem mesmo os mais importantes em um projeto. [...]

**Uma sintaxe diferente** – A sintaxe de Rust é diferente, e não há nenhuma vantagem evidente para a diferença. Isso é irritante para as pessoas que vêm de linguagens compatíveis com o tradicional estilo Algol e precisam lidar com uma sintaxe diferente sem nenhum ganho aparente para isso.

Já as vantagens de Rust são:

**Ter os melhores teóricos** – Dos três, Rust é a única linguagem com teóricos de classe mundial na lista. Isso pode ser visto na definição precisa da linguagem e na profundidade da sua abordagem técnica.

**Uma melhor segurança do que outras linguagens de programação de sistemas** – Claro que tinha de estar aqui, visto que acabamos de discutir o custo disso.

**O melhor PR** – Houve um longo período pré 1.0 em que Rust foi o queridinho da comunidade e não podia errar: qualquer problema que houvesse, Rust tinha uma solução para isso, ou teve até a versão 1.0. Ocorreu que o release da versão 1.0 terminou com a lua de mel e resultou (por minhas medidas e estimativas) em uma redução gritante do interesse geral, mas esses efeitos devem se atenuar. Além disso, Rust é uma linguagem decente com muito acontecendo com e para ela, e



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

está bem posicionada para converter essa campanha publicitária persistente em um marketing sólido."

Fonte: https://www.quora.com/Which-language-has-the-brightest-future-in-replacement-of-C-between-D-Go-and-Rust-And-Why.

Mesmo com pouco tempo de vida, em comparação com a C++ e a D, Rust tem feito muito barulho. Fez tanto barulho que é citada como potencial substituta do C por uma personalidade referência quando falamos em linguagens de programação. Entre os motivos principais está a sua segurança.

Talvez essa seja a melhor resposta para nosso questionamento: por que uma nova linguagem? Rust encaixa-se perfeitamente em um momento no qual está em alta a busca por tecnologias capazes de efetuar processamento rápido e paralelo de uma forma robusta. Temos toneladas de dados para processar, temos recursos computacionais para usar e, mesmo com o custo adicional envolvido no aprendizado de uma nova sintaxe que preze pela segurança e performance, Rust ainda é a melhor opção.

## 1.2 UM POUCO DE HISTÓRIA

Rust originou-se como um projeto pessoal de Graydon Hoare, em 2006. Seu interesse em criar uma linguagem surgiu devido ao seu trabalho: ele desenvolvia compiladores e outras ferramentas para linguagens já existentes. Então, por que não criar uma? Isso fez com que ele se autodenominasse “um engenheiro de linguagens criado pelo mercado”.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

O projeto, desenvolvido por Graydon como hobby, começou a amadurecer até que ele o apresentou para seu coordenador na Mozilla, que gostou muito do que viu. Para refazer seu browser, a Mozilla procurava por opções que fossem rápidas como C++, mas mais simples, que tivessem uma concorrência melhor e fossem mais seguras. E em todos esses pontos, Rust se encaixou como uma luva.

A Mozilla montou uma equipe interna para cuidar da linguagem em 2010 e, em janeiro de 2012, conseguiram lançar uma versão 0.1 do compilador, escrito em OCaml e posteriormente reescrito em Rust. Essa versão inicial tinha diversos problemas de performance e a biblioteca padrão era precária.

Tudo isso mudou na versão 0.2, lançada em março do mesmo ano, com mais de 1.500 correções e mudanças. Entre as principais modificações, estava a possibilidade de realizar chamadas a partir de código C, o que ampliou muito o leque de opções disponíveis.

Foram lançados 12 releases entre março de 2012 e janeiro de 2015, quando a versão 1.0.0-alpha foi liberada. Em fevereiro, uma versão alpha.2 veio à tona e, em maio de 2015, a versão 1.0.0 foi finalmente disponibilizada ao público. Essa infinidade de mudanças deixou algumas coisas para trás, como o Graydon Hoare, que em 30 de agosto de 2013 abandonou a liderança do projeto, com o seguinte e-mail na lista Rust-Dev (em tradução livre):



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

"Olá,

Tenho certeza de que muitos de vocês que me conhecem sabem o quanto meu papel como líder técnico em Rust me drenou bastante durante os anos. Tanto para mim como para aqueles que trabalharam comigo, e isso não é algo que eu desejo.

Reconhecendo isso, estou deixando o projeto para trabalhar em outras partes da organização, e o Brian Anderson assumirá o papel de líder técnico da linguagem Rust.

Brian é um dos mais qualificados, criteriosos, profissionais e produtivos desenvolvedores com quem já trabalhei. Junto à habilidade excepcional e à dedicação do restante da equipe Rust, tenho certeza de que continuarão o restante do caminho da versão 1.x da mesma forma bem-sucedida que moldou Rust como uma excelente linguagem.

Tem sido um raro prazer e privilégio trabalhar com a equipe de Rust, bem como com aqueles de dentro da Mozilla e de toda a comunidade.

Obrigado,

– Graydon."

Com a saída de Graydon, a Mozilla continuou a patrocinar o desenvolvimento da linguagem e a manter um time focado nisso, mas a linguagem continua a ser um projeto aberto, mantido pela



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

empresa e por uma comunidade enorme. No momento em que escrevo esta linha, consigo contar mais de cinco mil e seiscentos colaboradores únicos no projeto do `rustc`, o compilador Rust, incluindo membros da equipe Mozilla e de outras empresas.

## 1.3 O QUE É RUST?

Um erro comum que ouço por aí é que Rust é uma linguagem *funcional*. Rust é uma linguagem multiparadigma, compilada, funcional, imperativa, procedural, estruturada e genérica. Seu principal objetivo é ser uma alternativa simples ao C++, de forma que seja possível escrever sistemas complexos, de alta performance, robustos e seguros. Rust é *open source*, mas é mantida pela Mozilla, a mesma empresa que faz o navegador da internet, o Firefox.

Um código Rust é muito parecido com um código C ou C++. Ele faz uso de chaves `{}` para delimitar blocos de código e possui palavras-chave já bem conhecidas, como `if`, `while` e `for`. Algumas de suas características, porém, não são tão familiares a quem programa em C ou C++, como o uso de busca por padrões ou o fato de não ser necessário o uso de `return`.

## 1.4 O QUE PRECISO INSTALAR EM MEU COMPUTADOR?

Para trabalhar com Rust é necessário instalar algumas ferramentas em sua máquina, como o gerenciador de pacotes `cargo` e o compilador `rustc`. Um compilador é uma ferramenta que analisa o código-fonte escrito em uma linguagem compreensível a seres humanos e converte-o para uma linguagem



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

de computadores.

Além disso, o compilador faz um **pré-processamento** de seu código, avaliando se o que está escrito ali é um código válido e que executará sem problemas. Você também pode testar o Rust online, como será mostrado mais à frente neste capítulo.

## Instalando Rust no GNU/Linux ou no macOS

A maneira mais simples de instalar o `rustc` e afins é com o comando a seguir (para macOS e GNU/Linux):

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Não é recomendado trazer coisas direto da internet e executar via `sh` (que é o que o comando acima está fazendo), mas nesse caso estamos fazendo diretamente do site oficial do rustup (https://rustup.rs/) e confiamos na idoneidade deles, portanto aceitamos os riscos.

O script vai baixar o instalador e fazer a instalação. Sua senha de root é necessária. A saída do comando vai ser:

```
info: downloading installer

Welcome to Rust!

This will download and install the official compiler for the Rust
programming language, and its package manager, Cargo.

Rustup metadata and toolchains will be installed into the Rustup
home directory, [..]

Current installation options:


   default host triple: x86_64-unknown-linux-gnu
     default toolchain: stable (default)
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
               profile: default
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
```

Adicionamos `[...]` pois há várias informações que são dependentes do computador e sistema operacional. São coisas pouco relevantes aqui, apenas dizem os locais em que a linguagem será instalada.

Ao fim, o processo de instalação pergunta como você quer seguir com a instalação. Nós recomendamos a opção 1, que é a opção padrão. Basta digitar 1 ou só apertar *enter* para prosseguir.

Com isso, o processo prossegue:

```
info: profile set to 'default'
info: default host triple is x86_64-unknown-linux-gnu
info: syncing channel updates for 'stable-x86_64-unknown-linux-gn
u'
679.5 KiB / 679.5 KiB (100 %) 159.9 KiB/s in  4s ETA:  0s
info: latest update on 2022-02-24, rust version 1.59.0 (9d1b2106e
 2022-02-23)
info: downloading component 'cargo'
  6.1 MiB /   6.1 MiB (100 %) 108.8 KiB/s in  1m  8s ETA:  0s
info: downloading component 'clippy'
  2.4 MiB /   2.4 MiB (100 %) 128.0 KiB/s in  1m 18s ETA:  0s
info: downloading component 'rust-docs'
 17.0 MiB /  17.0 MiB (100 %) 203.3 KiB/s in  1m  4s ETA:  0s
info: downloading component 'rust-std'
 22.3 MiB /  22.3 MiB (100 %)   1.0 MiB/s in  1m  1s ETA:  0s
info: downloading component 'rustc'
 51.0 MiB /  51.0 MiB (100 %) 293.2 KiB/s in  2m 29s ETA:  0s
info: downloading component 'rustfmt'
  3.7 MiB /   3.7 MiB (100 %) 582.4 KiB/s in  8s ETA:  0s
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
 17.0 MiB /  17.0 MiB (100 %)   6.7 MiB/s in  2s ETA:  0s
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
info: installing component 'rust-std'
 22.3 MiB /  22.3 MiB (100 %)   7.4 MiB/s in  2s ETA:  0s
info: installing component 'rustc'
 51.0 MiB /  51.0 MiB (100 %)   8.4 MiB/s in  6s ETA:  0s
info: installing component 'rustfmt'
info: default toolchain set to 'stable-x86_64-unknown-linux-gnu'

  stable-x86_64-unknown-linux-gnu installed - rustc 1.59.0 (9d1b2
106e 2022-02-23)


Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, run:
source $HOME/.cargo/env
```

Na última linha, o compilador recomenda fazer o `source` do arquivo `$HOME/.cargo/env` (para a instalação em um Linux). Portanto, precisamos executar o seguinte comando antes de usar os comandos do Rust:

```
source $HOME/.cargo/env
```

Se tudo correr bem, já podemos usar o Rust e pedir a versão do compilador e do cargo:

```
$ rustc --version
rustc 1.59.0 (9d1b2106e 2022-02-23)

$ cargo --version
cargo 1.59.0 (49d8809dc 2022-02-10)
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

> **Instalação nativa em distribuições Linux**
>
> Cada distribuição Linux possui seu próprio gerenciador de pacotes. Algumas já entregam o compilador do Rust e o cargo por lá. Se você preferir usar o pacote oficial da sua distribuição, vale procurar entre os pacotes oficiais (via `apt-get`, `pacman`, `yum` etc.) por `rust`, `cargo` ou até mesmo `rustup`, que é nosso método preferido.

## Instalando Rust no Microsoft Windows

Para Windows, existem diferentes formas de fazer. A forma recomendada por nós é também instalar o rustup por lá. Para mais informações sobre isso, veja a seção "Outros métodos de instalação" do site oficial: https://www.rust-lang.org/pt-BR/tools/install.

### Problema com o linker.exe

Vários leitores das versões anteriores deste livro entraram em contato relatando um problema que ocorre na compilação de código Rust. Ao compilar, o processo para no seguinte erro:

```
error: `linker.exe` not found
  |
  = note: The system cannot find the file specified. (os error 2)

note: the msvc targets depend on the msvc linker but `link.exe` w
as not found

note: please ensure that VS 2013, VS 2015, VS 2017 or VS 2019 was
 installed with the Visual C++ option
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
error: aborting due to previous error
```

Esse problema ocorre com frequência no Windows 10 e é relacionado à não existência do executável `linker.exe` no sistema. Para resolver isso, é necessário baixar o `Build Tools for Visual Studio 2022` do endereço:

https://visualstudio.microsoft.com/downloads/

Links como esse em livros ficam atualizados rapidamente, mas basta procurar por *Build Tools for Visual Studio* e instalar o mais recente para resolver esse problema.

Ao instalar, lembre-se de selecionar os seguintes items em *C++ build tools*:

- MSVC v165 - VS 2022 C++ x64/x86 build tools
- Windows 10 SDK
- C++ CMake tools for Windows
- Testing tools core features

As versões vão mudar, mas se atente para que os items acima em suas versões mais atualizadas estejam selecionados.

Após a instalação dessas dependências, o Rust está pronto para ser usado em sua máquina com Windows.

## Rust Playground

Você também pode utilizar o **Rust Playground** para testar seus códigos. Acesse o endereço https://play.rust-lang.org, e terá acesso a uma página com um editor e alguns botões, como você pode ver na imagem a seguir.


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com


Figura 1.1: Os botões do Playground.

O botão `Run` é responsável por fazer seu código ser executado. Os dois itens ao lado do `Run` são botões de configuração. O primeiro é referente ao modo de compilação e vem configurado como `Debug` por padrão e exibe o máximo de informações possíveis sobre a compilação do código. O segundo é sobre a versão do compilador que será utilizada, configurada como `Stable` (a última versão estável do compilador), que também é nossa recomendação.

Basta digitar seu código no editor e clicar no botão `Run`, e o resultado da execução será exibido logo abaixo da caixa de texto.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

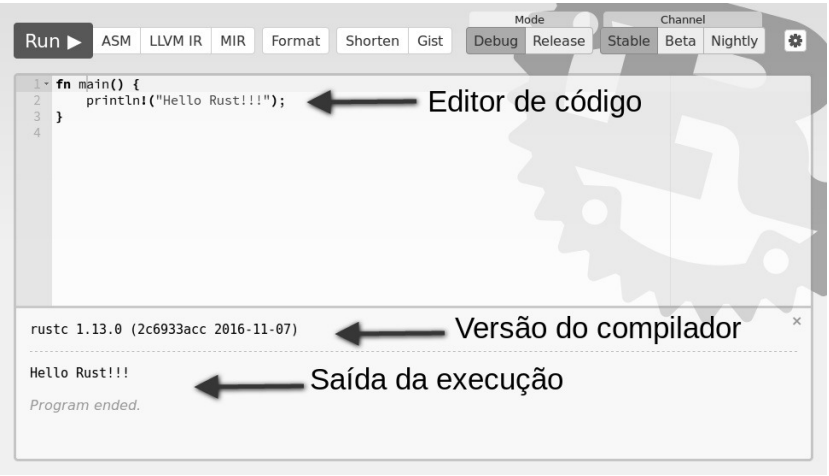


Figura 1.2: Editor, resultado da execução e mais.

Apesar da facilidade do Rust Playground, recomendo que você utilize o compilador em seu console e um editor de textos ou IDE de sua preferência (ex.: **VScode**, **Atom**, **Sublime Text**, **Vim**, **Emacs**, **CLion**). Isso lhe dará um ambiente de desenvolvimento de verdade e também contato com o compilador em si.

## 1.5 PRONTO PARA O ALÔ MUNDO?

*Bom, mas este é um livro sobre programação. E até agora eu não vi nenhum código por aqui. Quando vou ver a cara desse tal de Rust?*

Como diria o Chapolin Colorado: "Palma, palma, não priemos cânico". Vamos dar uma olhada na linguagem e em alguns de seus conceitos, que podem inicialmente causar algum estranhamento, mas que serão mais detalhados nos próximos capítulos. Por enquanto, vamos apresentar algumas das características que nos


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

fizeram sofrer quando começamos a usar Rust.

Como todo bom desenvolvedor sabe, para escrever código de boa qualidade, é necessário sempre seguir alguns protocolos – entre eles, o uso de um programa *Hello World* como primeiro contato com a linguagem.

Nosso primeiro código pega uma `String` (nesse caso, o tipo `&str` no Rust) e a exibe na tela, simples e eficaz. Para isso, utilizamos a famigerada função `main`. Quem programa em Java ou C sabe que a função `main` é o ponto de partida de qualquer programa, mas ela não existe em algumas linguagens. Se você veio de uma dessas, que não faz uso do `main`, é importante saber que tudo começa aí.

Também vamos usar o `println!`, que funciona como o `System.out.println` do Java ou o `puts` do Ruby. Ele não é similar ao `printf` do C, pois insere uma quebra de linha na sentença.

**Compilação, em poucas palavras**

Algumas linguagens populares hoje em dia não contam com o conceito de compilação, como Ruby ou Python. Compilar é o processo de pegar o código-fonte escrito em uma linguagem, como Rust ou C, e transformá-lo em código de máquina, possível de ser executado pelo computador.

Eis o código:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    println!("Hello World!");
}
```

Abra seu editor, digite o código passado e salve-o como `hello-world.rs`. A extensão `rs` é a extensão padrão de código-fonte Rust, mas entendemos que algumas pessoas possam achar que todos os arquivos têm "risos" como extensão 😂.

Vamos compilá-lo:

```
$ rustc hello-world.rs
```

Se tudo foi compilado com sucesso, nenhuma mensagem será mostrada, mas você verá que agora um novo arquivo foi criado. Ele será um arquivo binário com o mesmo nome do nosso arquivo `.rs`. Para testar o resultado do nosso programa, devemos executá-lo:

```
$ ./hello-world
Hello World!
```

*Wow!* Funciona. Vamos fazer algumas mudanças. A primeira será nossa mensagem dentro de uma variável:

```
fn main() {
    let greeting = "Hello World!";
    println!(greeting);
}
```

Pegamos nossa saudação, jogamos dentro de uma variável chamada `greeting` e passamos para o `println!`. Vamos compilar e ver o que rola?

```
$ rustc hello-world.rs
error: format argument must be a string literal
 --> hello-world.rs:3:14
  |
3 |     println!(greeting);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
  |              ^^^^^^^^
  |
help: you might be missing a string literal to format with
  |
3 |     println!("{}", greeting);
  |              ^^^^^

error: aborting due to previous error
```

Não foi tão fácil dessa vez. A macro `println!` não aceita simplesmente pegar o conteúdo de uma variável e jogá-lo na saída padrão. Ela exige que qualquer tipo enviado para impressão seja um literal, para tornar seu código seguro.

A parte interessante é que, diferente de outras linguagens, o compilador faz o possível para ser claro quanto ao problema e ainda dá a você sugestões do que pode resolvê-lo.

Nesse caso, precisamos deixar nosso código como a seguir:

```
fn main() {
    let greeting = "Hello World!";
    println!("{}", greeting);
}
```

O primeiro parâmetro que passamos ao `println!` é um formatador de saída. Ele pega os parâmetros posteriores e substitui as `{}` (chaves) pelo conteúdo que está na variável, convertendo-os em uma literal. Dessa forma, nosso código passa a funcionar. Você pode ter várias `{}` dentro do formatador, desde que possua a mesma quantidade de parâmetros.

Veja o código:

```
fn main() {
    let first_greeting = "Hello";
    let second_greeting = "World!";
    println!("{} {}", first_greeting, second_greeting);
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Agora, vamos utilizar uma parte de nossa biblioteca padrão. Qualquer código Rust inicia alocando o mínimo necessário para ele funcionar, o que é chamado de prelúdio (ou `prelude` no inglês). Qualquer função necessária, além da que está no prelúdio, não será carregada automaticamente e deve ser explicitamente informada.

O objetivo é modificar nossa saudação e exibir "Olá, seu nome" em vez do clássico "Hello World!". Assim sendo, vamos pegar a biblioteca `io`, que nos permite trabalhar com entrada e saída em nosso código.

Precisaremos de uma segunda variável em nosso código, do tipo `String`. Ao contrário do que possa parecer, a variável `greeting` é um ponteiro e não uma `String`. A sentença "Hello World!" foi salva em nosso código compilado, e a variável `greeting` aponta para ela. Para criarmos uma `String`, precisamos de algo um pouco mais complexo, apresentado a seguir:

```
let mut name = String::new();
```

Em nossa declaração, temos um prefixo `mut`, indicando ao compilador que a variável possui uma referência mutável. Por padrão, ao definirmos uma variável em Rust com o `let`, ela será imutável, ou seja, não poderá ter seu valor modificado. Em outras palavras, isso quer dizer que o que é referenciado por `let` sempre será imutável, a não ser que você defina uma referência que permita a modificação – no caso utilizando o `mut`.

Referências são um dos pontos mais complexos de se entender em qualquer linguagem de programação. Muitos não gostam da linguagem C por causa de seus ponteiros e referências por todos os


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

lados. Rust torna referências algo simples e poderoso, acessível em qualquer lugar. Usaremos essa referência mutável futuramente.

Definimos nossa variável `name` recebendo o tipo `String`, criada através do método `new()`. Vamos também mudar nossa saudação para não ter o " World!" implícito nela.

Nosso código precisa receber um nome para exibir a saudação `Hello, nome`, dessa forma leremos o nome a partir da entrada padrão. Para isso, usaremos o método `read_line` implementado em `Stdin`, que é retornado pela função `stdin()`, do módulo `io`. O código fica assim:

```
io::stdin().read_line(&mut name);
```

Veja nossa referência ao "mutable" `name`. Isso quer dizer que `read_line` acessa a referência que permite a modificação do `name`, ou seja, conseguimos colocar um valor dentro dela. Juntando tudo, nosso código ficará assim:

```
use std::io;

fn main() {
    let greeting = "Hello,";
    let mut name = String::new();

    io::stdin().read_line(&mut name);

    println!("{} {}", greeting, name);
}
```

Se você compilá-lo agora, receberá uma mensagem de aviso, ou *warning*, do `rustc`. Um warning é uma informação de que seu código poderia ser melhorado para ficar mais seguro e robusto, evitando problemas para você mesmo. Veja a seguir:

```
$ rustc hello-world.rs
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
warning: unused `std::result::Result` that must be used
 --> hello-world.rs:7:5
  |
7 |     io::stdin().read_line(&mut name);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: this `Result` may be an `Err` variant, which should be
handled

warning: 1 warning emitted
```

Mais para a frente, falaremos sobre `result`. Por enquanto, apenas ignoramos o warning. Ao executar o programa gerado, você terá a solicitação para informar o seu nome na primeira linha e, ao pressionar `[ENTER]`, será exibida a mensagem:

```
$ ./hello-world
leitor
Hello, leitor
```

## Conclusão

Neste capítulo, falamos sobre a história da linguagem Rust, suas vantagens e desvantagens, e por que ela é uma das candidatas a virar padrão na programação de sistemas de alta performance no futuro, aposentando concorrentes de peso como C e C++.

No próximo capítulo, vamos nos aprofundar mais na linguagem, falando sobre sua *biblioteca padrão* e sobre o que é um *prelúdio*, além de começarmos a utilizar o `cargo` para criar e gerenciar nossos projetos.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

CAPÍTULO 2

# COMEÇANDO NO CARGO

Agora que já demos uma primeira olhada em Rust, chegou a hora de entender alguns dos conceitos relacionados ao ecossistema e às ferramentas disponíveis, bem como o fluxo de execução de um código Rust. Neste capítulo, vamos entender como usar o `cargo` para a criação de nossos projetos. Também veremos como funcionam as importações de bibliotecas no ecossistema Rust.

## 2.1 PRELUDES

As linguagens de programação são, por si só, pobres. Elas contêm o mínimo necessário para você tratar *loops*, condicionais e outros conceitos básicos, mas ações mais complexas dependem geralmente de extensões, como a biblioteca padrão da C ou os frameworks pesados.

De fato, uma linguagem pode optar por carregar um pacote enorme de ferramentas na inicialização, como fazem Ruby ou Python, ou não carregar nada, como faz a linguagem C. Veja a seguir o exemplo de um programa *Hello World* feito em C puro. Ele simplesmente exibe a mensagem *Hello World* na saída padrão:

```
#include <stdio.h>

main()
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
{
    printf("Hello World");
}
```

A primeira linha do código pede para o pré-compilador incluir o `stdio`, pacote da biblioteca padrão que inclui o `printf`, responsável por pegar os parâmetros e imprimi-los na saída padrão. A linguagem C não inclui nada em sua inicialização além do básico da linguagem.

Já Ruby, por exemplo, traz inúmeros utilitários, incluindo o `print`, que é o equivalente ao `printf` da C. Essa característica facilita a vida de quem programa, mas tem um custo: o consumo de memória.

Em Rust, ficamos em um meio-termo, nem tudo o que é utilitário é carregado automaticamente, mas também não é carregada apenas a sintaxe básica da linguagem. Essa decisão de design do Rust permite que os executáveis gerados na compilação sejam mais leves, pois, quando o código é compilado, todo o básico da linguagem é vinculado ao seu executável.

> **Sem biblioteca padrão**
>
> Também é possível fazer o mesmo que a linguagem C utilizando `#![no_std]`. Ao utilizar `no_std`, Rust não vai inserir nada da biblioteca padrão e vai deixar a sua biblioteca com o mínimo possível de código incluído.

Isso é possível graças ao conceito chamado `prelude`, já citado no capítulo anterior, que diz ao Rust o que deve ser carregado. Em



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Rust, temos os `crates`, que são muito parecidos com os pacotes `pip` do Python ou as `gems` do Ruby, ou outra forma de empacotamento de código, como os `jars` do Java.

Um `crate` é um pacote de código que pode ser referenciado em seu projeto Rust para adicionar funcionalidades. O `prelude` diz quais são os `crates`, ou pacotes, que serão usados por padrão em um projeto Rust. Você pode adicionar `crates` no seu projeto e gerenciá-los com o programa chamado `cargo`, que faz parte da instalação do Rust.

## 2.2 CRATES, CARGO E OUTRAS FERRAMENTAS

Desde os primórdios, os desenvolvedores buscaram uma forma de reutilizar código. Um dos modelos adotados é o de criar as chamadas **bibliotecas**, ou seja, um pacote de código que pode ser acoplado ao seu próprio código sem muita dor de cabeça.

Algumas linguagens possuem mecanismos robustos para empacotar e reaproveitar código, e talvez os melhores exemplos sejam os pacotes `pip`, `gems`, as `libs` e `jars`. Estes são formatos de código autocontido, possuem uma finalidade específica e podem ser adicionados ao seu projeto com uma declaração simples em um arquivo de manifesto. O `cargo` é a ferramenta do Rust para lidar com isso e as `crates` são os pacotes no Rust.

Um `crate` é um pacote de código com alguma funcionalidade específica, por exemplo, uma biblioteca OAuth ou um driver de conexão ao banco de dados MySQL. *Crate* quer dizer


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

"engradado" ou "caixote", o que diz muito a respeito de sua utilidade dentro do ecossistema Rust.

Atualmente, os `crates` são publicados no site https://crates.io/ e são gerenciados com uma ferramenta chamada `cargo`. Este é o responsável por baixar os pacotes necessários pela sua aplicação e mantê-los em seu ambiente de desenvolvimento.

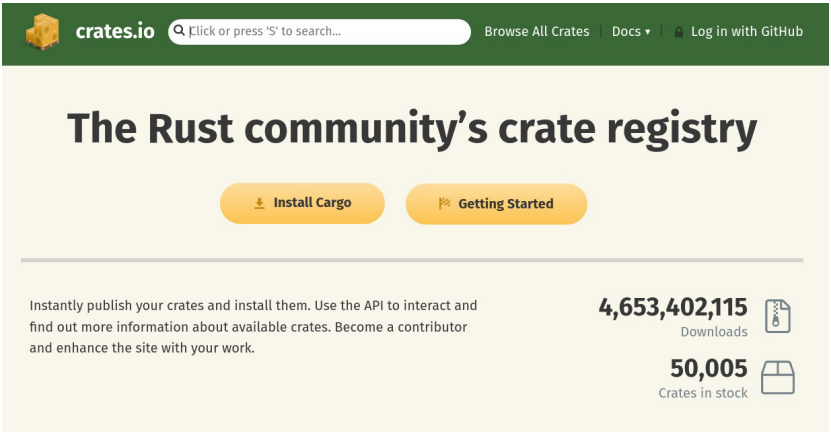


Figura 2.1: O site crates.io

Além disso, o `cargo` automatiza o seu processo de build, bem como o de criação de um projeto Rust. Vamos criar um projeto simples usando o `cargo` e adicionar uma dependência a ele.

## 2.3 CRIANDO UM PROJETO COM O CARGO

Vamos criar um novo projeto Rust do zero, usando o `cargo`. Para isso, utilize o comando:

```
$ cargo new hello_world --bin --vcs none
     Created binary (application) `hello_world` package
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Usando o `new`, dizemos ao `cargo` para criar um novo projeto chamado `hello_world`. Ele vai criar um novo diretório chamado `hello_world` com o código-fonte básico de nosso projeto e também um arquivo de configuração. A opção `--bin` diz que estamos criando um projeto binário, ou seja, que vai gerar um executável.

Já a opção `--vcs none` diz ao `cargo` para ele não adicionar o projeto em um controle de versões no momento. O `cargo` inicializa o seu projeto configurado para o Git, mas não nos preocuparemos com isso por ora.

> **Controle de versões**
>
> Utilizar um sistema de controle de versões é uma prática fundamental no desenvolvimento de software. Optamos por não o utilizar agora por questões puramente didáticas, mas é importante que você conheça e use um sistema de controle de versões em qualquer código que escreva.

A estrutura criada pelo `cargo` é:

```
$ tree hello_world/
hello_world/
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Começaremos dando uma olhada no arquivo `Cargo.toml`, também chamado de **manifest** ou **arquivo de manifesto**. Ele



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

possui algumas configurações básicas de nosso projeto no formato TOML (*Tom's Obvious, Minimal Language*), que é um formato de arquivo de configuração cujo objetivo é ser fácil de ler e com uma semântica óbvia. Seu conteúdo é apresentado a seguir:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Livro de Rust <livroderust@gmail.com>"]
edition = "2021"

# See more keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Nessa versão bem básica, temos o *nome* de nosso projeto, a sua *versão*, a *edição* do Rust que vamos usar para o projeto (no caso, a edição 2021) e uma *lista de autores*. Também temos uma *lista de dependências*, que no momento está vazia. Mais detalhes de configuração do manifesto de um projeto `cargo` podem ser encontrados em https://doc.crates.io/manifest.html.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

### Edições do Rust

Novas versões da linguagem Rust e seu ferramental são liberadas a cada ciclo de seis semanas. Dado esse curto espaço de tempo cada versão inclui apenas um pequeno número de novidades no ecossistema. Isso possibilita que os desenvolvedores tenham acesso mais rapidamente ao que está sendo desenvolvido (ao invés de se liberar um grande pacote de mudanças de uma só vez, como outras linguagens fazem normalmente).

Essas mudanças são agrupadas em uma **edição** a cada três anos mais ou menos. No momento do lançamento desta nova edição do livro, a documentação e o ferramental relacionado são totalmente atualizados, de forma que novos usuários possam ter um ponto de partida totalmente novo, porém ainda compatível com as versões anteriores da linguagem, mantendo os usuários e sistemas antigos felizes.

Este livro é compatível com a versão 2021 do Rust, que no momento de publicação é a edição mais atual disponível para a linguagem.

Mais detalhes sobre as edições podem ser encontrados em https://doc.rust-lang.org/edition-guide/editions/index.html

Agora, vamos dar uma olhada no código do arquivo `main.rs` que foi gerado pelo `cargo`:

```
fn main() {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Hello world!");
}
```

Simples, o projeto imprime na saída padrão a mensagem **Hello world!** como vimos no capítulo anterior. Apesar de parecer mais do mesmo, essa é a forma padrão para a criação de um projeto Rust, em vez da criação manual de arquivos.

Vamos compilar nosso projeto, desta vez com o `cargo build`. Ele deve ser executado no diretório raiz do projeto, onde temos nosso `Cargo.toml`.

```
$ cargo build
   Compiling hello_world v0.1.0 (/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
```

O `cargo` vai criar um diretório `target` e, dentro dele, outro chamado `debug`, onde temos nosso executável. Além disso, vai adicionar um arquivo `Cargo.lock`, responsável por manter as dependências travadas na versão que usamos em nosso projeto – semelhante ao que o `Gemfile.lock` faz no Ruby, o `package-lock.json` no `Node` ou o `pom.xml`, do `Maven`.

Nossa nova estrutura de arquivos fica assim, após a compilação com o `cargo`:

```
$ tree hello_world/
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    ├── CACHEDIR.TAG
    └── debug
        ├── build
        ├── deps
        │   ├── hello_world-2ad19aa2de21faea
        │   └── hello_world-2ad19aa2de21faea.d
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
├── examples
├── hello_world
├── hello_world.d
└── incremental
    └── hello_world-2ht72ghppvbu0
        ├── s-ftcza7xq02-7b296s-2627df5uzzhgc
        │   ├── 1ceylbolbsvmfywa.o
        │   ├── 1o4566agbrge8x5m.o
        │   ├── 256ee9uv33b1rxn7.o
        │   ├── 26vuzpqg1cmxmjzu.o
        │   ├── 2n5an8hwciayqsly.o
        │   ├── 4ye8wxg7ootzyu4w.o
        │   ├── 5236apqljgdxepjg.o
        │   ├── 5g5son6gojf5ccn5.o
        │   ├── dep-graph.bin
        │   ├── query-cache.bin
        │   └── work-products.bin
        └── s-ftcza7xq02-7b296s.lock

9 directories, 20 files
```

Nesse caso, o `cargo` está utilizando compilação incremental para que as próximas compilações fiquem mais rápidas, por isso temos vários pequenos arquivos `.o` no caminho. Com o tempo esses arquivos podem vir a ocupar bastante espaço em seu projeto, para isso você, caso deseje, pode os apagar com o comando `cargo clean`. Apenas lembre-se de que, caso use o `clean`, sua próxima compilação demorará mais, pois todas as dependências serão novamente baixadas e compiladas.

A compilação incremental é muito útil em projetos grandes feitos com Rust pois diminui consideravelmente o tempo de compilação. Para todo os projetos, incluindo dependências e seu código fonte, são gerados esses arquivos `.o`, que só serão gerados novamente se o trecho de código relacionado a eles for modificado.

O `cargo` não faz nenhuma mágica. Se você quiser ver o que está acontecendo por baixo dos panos, você pode executar o



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

`cargo build` utilizando o parâmetro `--verbose`, que ele vai lhe mostrar exatamente o que está sendo executado via `rustc`. No momento de escrita deste livro, temos o seguinte comando durante a compilação do nosso exemplo:

```
rustc --crate-name hello_world
      --edition=2021 src/main.rs
      --error-format=json
      --json=diagnostic-rendered-ansi
      --crate-type bin
      --emit=dep-info,link
      -C embed-bitcode=no
      -C debuginfo=2
      -C metadata=2ad19aa2de21faea
      -C extra-filename=-2ad19aa2de21faea
      --out-dir /hello_world/target/debug/deps
      -C incremental=/hello_world/target/debug/incremental
      -L dependency=/hello_world/target/debug/deps
```

O comando pode mudar devido a melhorias no `cargo` e no `rustc`, mas você sempre pode ver o que está acontecendo via `--verbose`.

Vamos executar o nosso binário:

```
$ ./target/debug/hello_world
Hello world!
```

Outra maneira de executar nosso projeto é utilizando o `cargo run`. Se ele identificar modificações no código, vai recompilar o projeto e executá-lo. Para ver como isso ocorre, modifique seu código:

```
fn main() {
    println!("Hello world via cargo run!");
}
```

E agora compile-o e execute-o com o `cargo run`:

```
$ cargo run
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
   Compiling hello_world v0.1.0 (file:...)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
     Running `target/debug/hello_world`
Hello world via cargo run!
```

Perceba que nosso código é compilado como uma versão de `debug`. Isso quer dizer que o binário gerado possui um *overhead* de código relacionado a pontos de depuração, dos quais falaremos com mais detalhes à frente. É possível gerar um código sem esses pontos, mais otimizado e que executará mais rápido com a opção `--release`, que funciona tanto para o `run` como para o `build`.

O modo de compilação padrão é o modo `debug`, pois ele compila mais rápido nosso código-fonte, visto que não são feitas otimizações.

```
$ cargo run --release
   Compiling hello_world v0.1.0 (file:...)
    Finished release [optimized] target(s) in 0.34 secs
     Running `target/release/hello_world`
Hello world via cargo run!
```

Neste caso, o `cargo` vai criar um diretório `release` dentro do diretório `target`, com o binário compilado sem os pontos de depuração:

```
$ tree hello_world/
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    ├── CACHEDIR.TAG
    ├── debug
    │   ├── build
    │   ├── deps
    │   ├── examples
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    |   └── incremental
    └── release
        ├── build
        ├── deps
        ├── examples
        └── incremental
```

Para executar o binário do diretório de `release`, use:

```
$ cargo run --release
    Finished release [optimized] target(s) in 0.0 secs
     Running `target/release/hello_world`
Hello world via cargo run!
```

E o binário do diretório de `debug`:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_world`
Hello world via cargo run!
```

Caso não queira ver as informações de compilação como o nome do arquivo, o nome do binário, quantidade de tempo que demorou para compilar ou qual crates estão compilando, você pode usar o parâmetro `--quiet` do cargo:

```
$ cargo run --quiet
Hello world via cargo run!
```

Esse parâmetro funciona tanto para o `cargo run` como para o `cargo build`. Vamos utilizar o `--quiet` sempre que fizer sentido a partir de agora para deixar o resultado dos nossos códigos mais fácil de ler. Se quiser ver as informações de compilação ao testar os exemplos no seu computador, basta remover esse parâmetro ao testar os exemplos.

Agora vamos inspecionar o conteúdo do nosso arquivo `Cargo.lock`:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
# This file is automatically @generated by Cargo.
# It is not intended for manual editing.
[[package]]
name = "hello_world"
version = "0.1.0"
```

A única dependência que temos é a do próprio projeto, mas na sequência vamos adicionar uma dependência externa e as coisas ficarão um pouco mais interessantes.

## 2.4 UTILIZANDO EXTENSÕES

Vamos adicionar uma extensão a partir da internet. Ela permite consultar a data e a hora do sistema em nosso código.

No arquivo `Cargo.toml`, adicionaremos a dependência:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Livro de Rust <livroderust@gmail.com>"]
edition = "2021"

[dependencies]
chrono = "0.4.19"
```

Adicionamos ao projeto a biblioteca `chrono`, com a versão 0.4.19. Essa versão é fixa, ou seja, o `cargo` tentará encontrar exatamente a que passamos.

Podemos generalizar um pouco qual versão desejamos utilizando os modificadores `~` e `^`. O sinal til ( `~` ) indica que desejamos uma versão qualquer entre o release especificado como mínimo e a versão imediatamente superior, considerando o valor para *major*, *minor* e *patch*. Já o acento circunflexo ( `^` ) ignora o minor e o patch, usando apenas o major como referência.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**VERSIONAMENTO DE SOFTWARE**

Major, minor e patch são partes de um padrão de versionamento de software largamente usado na indústria. Em uma versão como **1.2.3**, o **1** é o major, o **2** é o minor e o **3** é o patch.

Esse padrão é estabelecido pelo https://semver.org/, em que os incrementos em major, minor e patch são definidos assim:

- MAJOR version – quando você faz mudanças que tornam algo incompatível;
- MINOR version – quando você faz mudanças que mantenham a compatibilidade;
- PATCH version – quando você corrige algo que estava com problemas.

Vamos rodar novamente o `cargo build` para que ele verifique e atenda às versões definidas em nosso arquivo `Cargo.toml`. Essa execução demorará um pouco, pois o `cargo` vai baixar algumas coisas da internet.

```
$ cargo build
    Updating crates.io index
  Downloaded time v0.1.44
  Downloaded num-traits v0.2.14
  Downloaded libc v0.2.98
  Downloaded num-integer v0.1.44
  Downloaded autocfg v1.0.1
  Downloaded chrono v0.4.19
  Downloaded 6 crates (778.7 KB) in 1.66s
   Compiling autocfg v1.0.1
   Compiling libc v0.2.98
   Compiling num-traits v0.2.14
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
   Compiling num-integer v0.1.44
   Compiling time v0.1.44
   Compiling chrono v0.4.19
   Compiling hello_world v0.1.0 (/tmp/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 3.68s
```

Vamos analisar nosso log, parte a parte, para entender o que aconteceu aqui. Primeiro, o `cargo` atualizou as suas definições de dependências a partir do banco de referência do `crates`. Isso possibilita ao `cargo` saber quais são os `crates` disponíveis, quais as versões para cada e outros dados mais.

```
    Updating crates.io index
```

A seguir, ele busca e instala o pacote que nós solicitamos (`chrono`), então avalia e baixa as suas dependências:

```
  Downloaded time v0.1.44
  Downloaded num-traits v0.2.14
  Downloaded libc v0.2.98
  Downloaded num-integer v0.1.44
  Downloaded autocfg v1.0.1
  Downloaded chrono v0.4.19
  Downloaded 6 crates (778.7 KB) in 1.66s
```

Com o download concluído, é hora de compilar o código baixado:

```
   Compiling autocfg v1.0.1
   Compiling libc v0.2.98
   Compiling num-traits v0.2.14
   Compiling num-integer v0.1.44
   Compiling time v0.1.44
   Compiling chrono v0.4.19
   Compiling hello_world v0.1.0 (/tmp/hello_world)
```

Tudo disponível, vamos compilar o código-fonte de nosso projeto:

```
$ cargo build
   Compiling hello_world v0.1.0 (/tmp/hello_world)
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    Finished dev [unoptimized + debuginfo] target(s) in 3.68s
```

Se você der uma olhada na estrutura de seu projeto, agora verá que temos as suas dependências no diretório `deps` de `debug`:

```
$ find target/debug/deps/ | wc -l
21
```

Com a nossa dependência já compilada, precisamos editar nosso código para exibir a data atual, a partir do `crate chrono`. Para isso, vamos usar a função `now()` do módulo `Local`:

```
use chrono::Local;

fn main() {
    let t = Local::now();
    println!("{}", t);
}
```

Na primeira linha de nosso código, referenciamos o uso do módulo `Local` do crate `chrono`:

```
use chrono:;Local;
```

Depois, usamos a função `println!` para imprimir a data atual usando o formato de acordo com a ISO 8601. Quando executamos o arquivo, conseguimos um resultado parecido com o seguinte (dependendo da sua data e hora):

```
$ cargo run --quiet
2020-02-02 20:20:20.220200202 -03:00
```

Essa data é só um exemplo, mas nós adoraríamos ter executado esse exemplo em um palíndromo tão perfeito (até em microssegundos!).

Por último, mas não menos importante, vale a pena dar uma olhada em nosso arquivo `Cargo.lock`, após termos adicionado



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

uma dependência no projeto. Repare que ele adiciona a origem, a versão usada e um `checksum` para validação, no caso de ser necessário o download do pacote ou uma nova compilação.

É importante ressaltar que as dependências da biblioteca `chrono` também são adicionadas ao nosso projeto:

```
# This file is automatically @generated by Cargo.
# It is not intended for manual editing.
version = 3

[[package]]
name = "autocfg"
version = "1.0.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "cdb031dd78e28731d87d56cc8ffef4a8f36ca26c38fe2de700543
e627f8a464a"

[[package]]
name = "chrono"
version = "0.4.19"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "670ad68c9088c2a963aaa298cb369688cf3f9465ce5e2d4ca10e6
e0098a1ce73"
dependencies = [
 "libc",
 "num-integer",
 "num-traits",
 "time",
 "winapi",
]

[[package]]
name = "hello_world_chrono"
version = "0.1.0"
dependencies = [
 "chrono",
]

[[package]]
name = "libc"
version = "0.2.98"
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "320cfe77175da3a483efed4bc0adc1968ca050b098ce4f2f1c13a
56626128790"

[[package]]
name = "num-integer"
version = "0.1.44"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d2cc698a63b549a70bc047073d2949cce27cd1c7b0a4a862d08a8
031bc2801db"
dependencies = [
 "autocfg",
 "num-traits",
]

[[package]]
name = "num-traits"
version = "0.2.14"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9a64b1ec5cda2586e284722486d802acf1f7dbdc623e2bfc57e65
ca1cd099290"
dependencies = [
 "autocfg",
]

[[package]]
name = "time"
version = "0.1.44"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6db9e6914ab8b1ae1c260a4ae7a49b6c5611b40328a735b218625
67685e73255"
dependencies = [
 "libc",
 "wasi",
 "winapi",
]

[[package]]
name = "wasi"
version = "0.10.0+wasi-snapshot-preview1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1a143597ca7c7793eff794def352d41792a93c481eb1042423ff7
ff72ba2c31f"

[[package]]
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
name = "winapi"
version = "0.3.9"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "5c839a674fcd7a98952e593242ea400abe93992746761e3864140
5d28b00f419"
dependencies = [
 "winapi-i686-pc-windows-gnu",
 "winapi-x86_64-pc-windows-gnu",
]

[[package]]
name = "winapi-i686-pc-windows-gnu"
version = "0.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ac3b87c63620426dd9b991e5ce0329eff545bccbbb34f3be09ff6
fb6ab51b7b6"

[[package]]
name = "winapi-x86_64-pc-windows-gnu"
version = "0.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "712e227841d057c1ee1cd2fb22fa7e5a5461ae8e48fa2ca79ec42
cfc1931183f"
```

## 2.5 OUTROS UTILITÁRIOS DO CARGO

Neste capítulo, falamos sobre as opções `clean`, `build` e `run` do Cargo. Com esses subcomandos, você já consegue trabalhar com seu código fonte Rust numa boa. Essas, porém, não são as únicas opções do nosso canivete suíço chamado `cargo`. Ao executar `cargo help`, você verá diversas outras opções, além das já estudadas:

```
$ cargo help
Rust's package manager

USAGE:
    cargo [+toolchain] [OPTIONS] [SUBCOMMAND]

OPTIONS:
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    -V, --version                  Print version info and exit
        --list                     List installed commands
        --explain <CODE>           Run `rustc --explain CODE`
    -v, --verbose                  Use verbose output (-vv very v
erbose/build.rs output)
    -q, --quiet                    No output printed to stdout
        --color <WHEN>             Coloring: auto, always, never
        --frozen                   Require Cargo.lock and cache a
re up to date
        --locked                   Require Cargo.lock is up to da
te
        --offline                  Run without accessing the netw
ork
        --config <KEY=VALUE>...    Override a configuration value
 (unstable)
    -Z <FLAG>...                   Unstable (nightly-only) flags
to Cargo, see 'cargo -Z help' for details
    -h, --help                     Prints help information

Some common cargo commands are (see all commands with --list):
    build, b    Compile the current package
    check, c    Analyze the current package and report errors, bu
t don't build object files
    clean       Remove the target directory
    doc         Build this package's and its dependencies' docume
ntation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directo
ry
    run, r      Run a binary or example of the local package
    test, t     Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/
.cargo/bin
    uninstall   Uninstall a Rust binary

See 'cargo help <command>' for more information on a specific com
mand.
```

Além disso, o `cargo` pode ter suas funcionalidades estendidas com o uso de extensões. Duas bem interessantes são o `clippy` e o



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

`rustfmt`.

## Instalando um novo subcomando

Com o `rustup` é bem simples instalar um novo subcomando no `cargo`:

```
$ rustup component add rustfmt
```

Essa linha vai instalar o `rustfmt` como um subcomando do `cargo`. Se você receber algum erro, há grandes chances de você estar usando uma versão antiga do rustup, portanto, faça o seguinte:

```
$ rustup self update
```

Isso vai atualizar a versão do `rustup` para ter certeza que você tem a última versão instalada, para assim executar novamente o `rustup component add`.

Você pode ver uma lista de extensões disponíveis em https://github.com/rust-lang/cargo/wiki/Third-party-cargo-subcommands. Algumas delas, como o `rustfmt` e o `clippy`, já são instaladas por padrão pelo `rustup`, então você pode fazer uso delas sem a necessidade de instalar nada mais.

## Usando o rustfmt

O `rustfmt` é um utilitário que formata seu código baseado em convenções da comunidade. Essa formatação pode ocorrer automaticamente através da integração do `rustfmt` com o seu editor de textos preferido (por exemplo, Vim, Emacs, Sublime Text 3, Atom, VS Code e o CLion são suportados) ou através do utilitário de linha de comando.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Para apresentar de uma forma bem simples a ferramenta, crie um novo projeto binário e bagunce o código. No nosso caso, removemos espaços e adicionamos quebras de linha em lugares onde, por convenção, não deveriam existir.

```
fn main()
{
println!("Hello, world!");
}
```

Perceba que esse código, apesar de estar com a formatação péssima, ainda compila e executa:

```
$ cargo run --quiet
Hello, world!
```

Após instalar o `rustfmt`, para formatar nosso arquivo corretamente, usamos:

```
$ rustfmt src/main.rs
```

E o resultado passa a ser:

```
fn main() {
    println!("Hello, world!");
}
```

Como mencionamos, normalmente essa integração ocorre diretamente com seu editor de textos favorito, com o `rustfmt` rodando antes ou durante o salvamento de arquivos. Não é uma ação que será executada na linha de comando.

Você pode encontrar mais detalhes sobre como configurar o `rustfmt` com seu editor favorito aqui: https://github.com/rust-lang/rustfmt.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## Usando o Clippy

O `clippy` é um utilitário que possui uma coleção de *linters* para Rust. Um *linter* é uma ferramenta que analisa o código-fonte em busca de erros de estilo, possíveis bugs e outros problemas de sintaxe. Muitos deles são pegos na compilação, mas existem diversos linters que podem melhorar a legibilidade de seu código, algo que o `rustc` não verifica.

Durante o desenvolvimento de seus códigos Rust, os autores deste livro sempre fazem uso do `clippy` antes da compilação. O resultado da execução do `clippy` é um conjunto de dicas para melhorar o código, o que, dado o tempo suficiente, vai fazer de você um programador ou programadora Rust melhor.

Vamos pegar um exemplo simples de código para demonstrar como o `clippy` pode ser útil e instrutivo em nosso dia a dia. O código a seguir compila e executa sem problemas:

```
fn main() {
    let x = true;

    if x == true {
        println!("X é verdadeiro");
    }
}
```

O resultado da execução:

```
$ cargo run --quiet
X é verdadeiro
```

Porém a linha `if x == true` contém uma expressão que pode ser otimizada. Sabemos que o `if` avalia o resultado da expressão e executa uma ação se o resultado for verdadeiro ( `true` ) ou falso ( `false` ). Ora, nosso `x` já é um booleano, não



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

precisamos comparar com `true`.

Vamos ver como o `clippy` trata isso. Para isso, execute `cargo clippy`:

```
$ cargo clippy
    Checking test-rust v0.1.0
warning: equality checks against true are unnecessary
 --> src/main.rs:4:8
  |
4 |     if x == true {
  |        ^^^^^^^^^ help: try simplifying it as shown: `x`
  |
  = note: `#[warn(clippy::bool_comparison)]` on by default
  = help: for further information visit
          https://rust-lang.github.io/rust-clippy/master/index.ht
ml#bool_comparison

warning: 1 warning emitted

    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
```

Veja o que o `clippy` fala aqui: `warning: equality checks against true are unnecessary`. Ou seja, exatamente o que falamos: não é necessário comparar `true` com `true`. E, como de costume em Rust, o `cargo` não apenas dá um alerta perdido no meio de um monte de linhas, ele mostra onde está o problema e sugere uma solução:

```
4 |     if x == true {
  |        ^^^^^^^^^ help: try simplifying it as shown: `x`
```

Basicamente, o que ele diz aqui é: use apenas o `x`. Além disso, ele diz qual linter foi usado na verificação e dá um endereço com mais informações sobre o porquê de esse ponto de código ter sido apontado como problemático.

```
  = note: `#[warn(clippy::bool_comparison)]` on by default
  = help: for further information visit
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
          https://rust-lang.github.io/rust-clippy/master/index.ht
ml#bool_comparison
```

Aplicando a melhoria sugerida pelo `clippy`, nosso código fica assim:

```
fn main() {
    let x = true;

    if x {
        println!("X é verdadeiro");
    }
}
```

Existem vários linters disponíveis hoje no `clippy`. A lista completa pode ser encontrada em https://rust-lang.github.io/rust-clippy/master/. Além disso, os linters do `clippy` podem ser configurados ou ativados/desativados conforme a sua necessidade. É uma ferramenta bem complexa e poderosa, e recomendamos que você dê uma olhada em sua documentação em https://github.com/rust-lang/rust-clippy.

## Conclusão

Neste capítulo, vimos como usar uma biblioteca do Rust (conhecida como `crate`) em nosso projeto Rust. Conhecemos o `cargo` e vimos como criar um projeto com ele, bem como gerenciar dependências com um arquivo `Cargo.toml`.

No próximo capítulo, vamos nos aprofundar nos tipos de dados padrão do Rust, e também dar uma olhada em alguns pontos fundamentais da linguagem, como condicionais e relacionados.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

CAPÍTULO 3

# MERGULHANDO NO OCEANO RUST

Neste capítulo, veremos alguns dos principais elementos da linguagem Rust, como seus tipos de dados e suas estruturas para controle de fluxo. Esses elementos fazem parte da **biblioteca padrão**.

Antes, vamos falar rapidamente de um conceito importante que está disponível em toda a linguagem: os *traits*. Vamos vê-los com profundidade no próximo capítulo; por enquanto, basta saber que funcionam como uma classe abstrata ou interface genérica a ser implementada por qualquer tipo de dados existentes ou que você venha a criar.

Um exemplo é *Veículo*, que pode conter métodos como *andar*, *parar*, *buzinar*, *acender faróis*, *piscar seta* e muito mais. Já a implementação seria os tipos de veículos, como *Caminhão*, *Carro* e outros clássicos da indústria automobilística nacional.

Um *trait* nada mais é do que uma coleção de métodos, definida para um tipo que não sabemos qual é, denominado `Self` . Como em outras linguagens, `Self` faz referência ao contexto atual. Esse contexto geralmente é associado a uma estrutura, que é a



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

implementação do nosso trait.

Uma estrutura é um dos principais tipos de dados disponíveis em linguagens de programação. Ela possibilita agrupar informações em um único objeto em memória, e acessá-las facilmente por um ponteiro direto.

A biblioteca padrão do Rust é formada por diversos *crates*. Vamos dar uma olhada nos tipos de dados básicos disponíveis que podemos chamar de *core* da linguagem: o *crate* `std`, acessível em todos os *crates* Rust por padrão.

O crate `std` é composto por uma grande quantidade de módulos. Não vamos explorar todos neste livro, mas vamos ver os principais módulos para que você consiga entender o básico de Rust. Caso queira conhecer todos os módulos que compõem o crate `std`, você pode ver a lista na íntegra aqui: https://doc.rust-lang.org/std/#modules

Vamos começar vendo a sintaxe básica e os principais módulos do `std`.

## 3.1 ATRIBUIÇÃO E VINCULAÇÃO DE VARIÁVEIS

### Atribuição com inferência de tipo

Em Rust, usamos `let` para atribuir um valor a uma variável. Essa atribuição é **imutável**, ou seja, não é possível modificar um valor já atribuído.

No nosso projeto criado no capítulo anterior, atribuímos o


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

retorno da função `Local::now()` a uma variável chamada `t`.

```
let t = Local::now();
```

Rust possui um conceito chamado **inferência de tipos**. Ela, no momento da atribuição, identifica que o retorno da função `Local::now()` é uma instância do tipo `DateTime` e atribui o tipo correto (instância de `DateTime`) à nossa variável `t`. Isso é importante, pois Rust é uma linguagem de tipos estáticos, isto é, um inteiro será sempre um inteiro e, se você tentar atribuir a uma variável de um determinado tipo um valor que não seja da mesma categoria, terá um erro de compilação.

Vamos modificar nosso código, tentando atribuir um inteiro à nossa variável `t`.

```
use chrono::Local;

fn main() {
    let t = Local::now();
    println!("{}", t);

    t = 2;
}
```

Ao compilar, teremos um erro **mismatched types**, que indica que estamos atribuindo um valor que não pode ser colocado na variável. Nossa definição infere que a variável `t` é do tipo `DateTime<Local>`, e a atribuição do inteiro `2` espera que `t` seja do tipo `{integer}`.

```
   Compiling hello_world v0.1.0 (file:...)
error[E0308]: mismatched types
 --> src/main.rs:7:9
  |
7 |     t = 2;
  |         ^ expected struct `DateTime`, found integer
  |
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
  = note: expected struct `DateTime<Local>`
              found type `{integer}`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `hello_world`

To learn more, run the command again with --verbose.
```

**WARNING E ERRORS**

O processo de compilação vai gerar alertas de atenção (ou *warnings*) se algo potencialmente errado estiver definido em nosso código, como também gerará alertas de erro se existir algo realmente errado nele. *Warnings* não impedem a compilação, mas *errors* impedem.

Para fins de comparação, algo potencialmente errado seria utilizar uma variável não inicializada. Definir um valor para uma variável na sua declaração é uma boa prática para evitar que o seu código retorne lixo direto da memória. Já algo realmente errado seria um código que vai quebrar, como a ausência de um ponto e vírgula no final de uma linha, quando necessária.

## Atribuição múltipla

Podemos atribuir valores a diversas variáveis em uma mesma execução de `let` , basta encadeá-las dentro de `()` (parênteses). Vamos modificar nosso código para termos uma variável com o



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

dia, uma com o mês e outra com o ano.

```
use chrono::prelude::*;

fn main() {
    let t = Local::now();
    let (year, month, day) = (t.year(), t.month(), t.day());

    println!("Hoje é {}-{}-{}", year, month, day);
}
```

> **Traits em ação**
>
> Lembra das `traits` que falamos no começo deste capítulo? Aqui estamos utilizando elas na prática. Os métodos `year`, `month` e `day` são implementados para cada fuso-horário dentro da `chrono` via `Datelike` trait. Esse foi o motivo de importarmos `chrono::prelude::*` aqui. O *crate* `chrono` sugere importar o `prelude` deles para deixar as coisas mais fáceis, mas poderíamos somente importar a trait `Datelike` se quiséssemos.

## Atribuição sem inferência de tipo

Em vez de deixar o Rust usar a inferência de tipos e descobrir o que queremos colocar em nossa variável, podemos definir o seu tipo na declaração:

```
let x: i32;
```

Nesse exemplo, nossa variável é um inteiro de 32 bits. Vale ressaltar que esse código vai resultar em um warning quando a compilação do código ocorrer, e se a variável `x` for usada em


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

algum lugar, vai impedir a compilação com um erro.

Rust precisa que você inicialize algum valor em sua variável. Para definir o tipo e o valor, use a sintaxe:

```
let x: i32 = 0;
```

Isso é parte das garantias do Rust para evitar que variáveis sejam utilizadas com valores inesperados, geralmente com "lixo de memória". A linguagem deixa o desenvolvedor ciente do que pode acontecer e reforça boas práticas.

Continuando no nosso exemplo, podemos conseguir ano, mês e dia com tipos dessa forma:

```
use chrono::prelude::*;

fn main() {
    let t = Local::now();

    let year: i32 = t.year();
    let month: u32 = t.month();
    let day: u32 = t.day();

    println!("Hoje é {}-{}-{}", year, month, day);
}
```

Em Rust, isso é chamado **anotação de tipo**, e é particularmente interessante quando falamos de inteiros devido aos diversos formatos que eles podem assumir. Um exemplo disso é a variável `year`, que representa o ano. O ano é o único da lista que pode ser negativo devido à forma como organizamos o calendário, que começou a ser contado a partir de um evento na história.

## Declarando variáveis mutáveis

No capítulo *Primeiros passos*, vimos rapidamente como criar


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

variáveis mutáveis utilizando o `mut`. Por padrão, em Rust, qualquer variável criada usando o `let` é imutável, isto é, você terá um erro de compilação se tentar modificar o seu valor após a sua definição.

```
fn main() {
    let a = 1;
    a = 2;

    println!("{}", a);
}
```

Ao tentarmos compilar o código anterior, o resultado será um belo erro:

```
$ cargo run --quiet
warning: value assigned to `a` is never read
 --> arst.rs:2:9
  |
2 |     let a = 1;
  |         ^
  |
  = note: `#[warn(unused_assignments)]` on by default
  = help: maybe it is overwritten before being read?

error[E0384]: cannot assign twice to immutable variable `a`
 --> arst.rs:3:5
  |
2 |     let a = 1;
  |         -
  |         |
  |         first assignment to `a`
  |         help: make this binding mutable: `mut a`
3 |     a = 2;
  |     ^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error; 1 warning emitted

For more information about this error, try `rustc --explain E0384
`.
```

Você pode dizer: *mas isso é uma péssima ideia, pois vou ter de*


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

*escrever mais código para declarar uma variável*. Rust é sobre segurança, e imutabilidade de variáveis é o mesmo caso. Usando o `let`, você garante que não terá mudanças em suas variáveis se não as desejar; e se quiser que elas sejam mutáveis, é só usar o `mut`.

```
fn main() {
    let mut a = 1;
    a = 2;

    println!("{}", a);
}
```

Quem vem de linguagens funcionais sabe o quanto ter coisas imutáveis ajuda no desenvolvimento de software. Em Rust, isso não é obrigatório, mas pelo menos é assim por padrão. Adicionar o `mut` deixa explícito que uma variável pode mudar, e como já dizia o Zen do Python:

> Explícito é melhor que implícito.

## Constantes

Rust possui suporte a um tipo de declaração muito usada em programação: as constantes. Uma **constante** é um valor declarado e nomeado que nunca muda.

Para criar constantes em Rust, utilizamos `const` no lugar de `let`. Perceba que, neste caso, a inferência de tipos não se aplica, ou seja, é obrigatório informar o tipo de dado na declaração:

```
fn main() {
    const Y: i32 = 3;
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Const: {}", Y);
}
```

Veja que usamos um caractere maiúsculo para nomear nossa constante. Caso não o façamos, Rust retornará um *warning*:

```
$ cargo build --quiet
error[E0425]: cannot find value `Y` in this scope
 --> arst.rs:4:35
  |
2 |         const y: i32 = 3;
  |         ----------------- similarly named constant `y` define
d here
3 |
4 |             println!("Const: {}", Y);
  |                                   ^ help: a constant with a s
imilar name exists (notice the capitalization): `y`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0425
`.
```

Utilizando o nosso código de exemplo que imprime o dia de hoje, podemos fazer algo que ninguém nunca vai querer: podemos transformar o ano de 2020 em uma constante:

```
use chrono::prelude::*;

fn main() {
    const THAT_YEAR: i32 = 2020;
    let t = Local::now();
    let (month, day) = (t.month(), t.day());

    println!("Hoje é {}-{}-{}", THAT_YEAR, month, day);
}
```

Isso seria o pesadelo de qualquer usuário usando esse sistema e sinceramente esperamos que em breve ninguém entenda mais a referência do caos que foi o ano de 2020.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**LET E CONST**

Você pode estar se perguntando neste momento: "Se as variáveis são imutáveis, por que existem as constantes? Qual a vantagem?".

As constantes são muito mais limitadas do que simples variáveis. Você deve usar uma constante se você quiser uma ou mais das seguintes garantias:

- Nunca possa ser mutável, nem via `mut` (que não existe para constantes).
- O tipo precise ser declarado (não existe inferência de tipos).
- Que o tamanho seja sabido em tempo de compilação.

## 3.2 FUNÇÕES

O menor elemento de um código Rust é uma função. Tudo se inicia na função `main`, já que a execução do código começa efetivamente a partir dela. O que acontece dentro dela basicamente é: chamar funções, pegar o resultado e utilizá-lo em novas chamadas de funções.

### Declarando funções

Vamos ignorar o exemplo da constante no ano (ufa!) e usar nosso código de exemplo novamente, agora modificando-o para que a impressão do dia de hoje ocorra dentro de uma função `print_today`, chamada em `main`.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
use chrono::prelude::*;

fn print_today() {
    let t = Local::now();
    let (year, month, day) = (t.year(), t.month(), t.day());

    println!("Hoje é {}-{}-{}", year, month, day);
}

fn main() {
    print_today();
}
```

Em Rust, uma função pode retornar um valor, mas sempre um único valor, e o retorno pode ser o resultado da última linha executada caso não seja usado o `return`. É possível encontrar esse `return` em diversas linguagens, mas falaremos mais dele adiante.

Imagine uma função que calcule a soma de dois valores, como:

```
fn sum(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let x = 3;
    let y = 5;
    println!("{} + {} = {}", x, y, sum(x, y));
}
```

```
$ cargo run --quiet
3 + 5 = 8
```

Vamos dar uma olhada no código da nossa função `sum`. Perceba que ela recebe dois valores do tipo `i32`, `a` e `b`. Além disso, ela retorna um valor do tipo `i32`, que é declarado com o sinal de "flecha" ( `->` ) seguido do tipo.

```
fn sum(a: i32, b: i32) -> i32 { a + b }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Repare também que a última linha da nossa função não possui um ponto e vírgula ( `;` ). Se ele for colocado, teremos um erro de compilação:

```
$ cargo run --quiet
error[E0308]: mismatched types
 --> src/main.rs:1:27
  |
1 | fn sum(a: i32, b: i32) -> i32 {
  |    ---                    ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return
` expression
2 |     a + b;
  |           - help: consider removing this semicolon

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `sum`

To learn more, run the command again with --verbose.
```

Rust considera tanto o `;` quanto o `}` como finalizadores de uma expressão quando colocados ao final da definição de uma função. É como se ele entendesse que, após a expressão `a + b;` e antes do `}`, existisse uma outra expressão vazia. Nesse caso, faz-se necessário a remoção do `;` (ponto e vírgula), ou o compilador retorna um erro dizendo que está sendo retornado o tipo vazio `()` e não o tipo `i32` que é o tipo esperado.

Como dito, Rust possui a clássica expressão `return`, para retornar o valor antes do final da função. No código a seguir, modifiquei nossa função de soma para que ela sempre retorne `0` (zero).

```
fn sum(a: i32, b: i32) -> i32 {
    return 0;
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    a + b
}

fn main() {
    let x = 3;
    let y = 5;
    println!("{} + {} = {}", x, y, sum(x, y));
}
```

Execute o código e veja que o Rust identifica que as variáveis `a` e `b` não são usadas e somos informados com `warn(unused_variables)`. Ela também mostra que a linha onde a soma ocorre nunca será alcançada, pois temos um `return` logo acima, nos informando com `warn(unreachable_code)`. Mas nenhum desses casos impede nosso código de ser executado, como pode ser visto ao compilá-lo.

```
$ cargo run --quiet
warning: unreachable expression
 --> src/main.rs:3:5
  |
2 |     return 0;
  |     -------- any code following this expression is unreachabl
e
3 |     a + b
  |     ^^^^^ unreachable expression
  |
  = note: `#[warn(unreachable_code)]` on by default

warning: unused variable: `a`
 --> src/main.rs:1:8
  |
1 | fn sum(a: i32, b: i32) -> i32 {
  |        ^ help: if this is intentional, prefix it with an unde
rscore: `_a`
  |
  = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `b`
 --> src/main.rs:1:16
  |
1 | fn sum(a: i32, b: i32) -> i32 {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
  |                ^ help: if this is intentional, prefix it with
 an underscore: `_b`

warning: 3 warnings emitted
3 + 5 = 0
```

## Ponteiro para função

Rust suporta o conceito de ponteiros para função, que possibilita alocar uma variável que representa o endereço de memória de uma função. Isso é muito útil para situações nas quais é necessário passar uma função como parâmetro de outra.

Vamos modificar nosso código para usar um ponteiro com a definição completa do tipo, ou seja, sem inferência:

```
use chrono::prelude::*;

fn print_today() {
    let t = Local::now();
    let (year, month, day) = (t.year(), t.month(), t.day());

    println!("Hoje é {}-{}-{}", year, month, day);
}

fn run_function(function: fn()) {
    function()
}

fn main() {
    let function: fn() = print_today;
    run_function(function);
}
```

Embora isso pareça complicado, é simples de entender. Criamos uma função chamada `print_today`, que imprime o dia de hoje, e outra, chamada `run_function`, que recebe como parâmetro uma variável chamada `function` do tipo `fn()`.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Em nossa função `main`, criamos uma variável chamada `function` que aponta para `print_today`. Na sequência, chamamos a função `run_function` e passamos como parâmetro nosso ponteiro para a função `print_today` (que não é chamada explicitamente em nenhum momento em nosso código). E é aí que a mágica acontece.

Dentro de `run_function`, temos `function()`, que é o parâmetro recebido pela função. Perceba que ele não é uma função declarada em nosso código, mas na verdade recebe o ponteiro definido em `main`, que aponta para `print_today`. Sendo assim, ao executar a linha `function()`, o que ocorre é uma chamada à função `print_today()`.

A inferência de tipo da função também funciona:

```
use chrono::prelude::*;

fn print_today() {
    let t = Local::now();
    let (year, month, day) = (t.year(), t.month(), t.day());

    println!("Hoje é {}-{}-{}", year, month, day);
}

fn run_function(function: fn()) {
    function()
}

fn main() {
    let function: fn() = print_today;
    run_function(function);
}
```

Agora que já vimos o básico da criação de funções e atribuição de valores a variáveis, vamos falar dos tipos de dados que existem em Rust.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

# 3.3 TIPOS DE DADOS EM RUST

## Caracteres

Rust possui um tipo para tratar caracteres únicos, o `char`. Para definir um `char`, basta usar aspas simples ( `'` ).

```
let x = 'x';
```

Em Rust, um `char` representa um caractere unicode, dessa forma, um `char` não corresponde a um único byte, mas a um conjunto de quatro bytes – diferentemente do C.

O código a seguir imprimirá um coração na tela:

```
fn main() {
    let a: char = '\u{2764}';
    println!("{}", a)
}
```

O tipo `char` possui diversos métodos interessantes, entre eles, o `is_digit()`, que indica se o caractere é um dígito válido dentro de uma base específica. Uma base de 2 representa base binária, um 10 representa base decimal e um 16 representa a base hexadecimal.

Vamos definir três variáveis do tipo `char`, e depois as avaliar na base binária e na base decimal. Para a base binária, são esperados apenas os bytes 0 e 1, e, para a base decimal, qualquer caractere de 0 a 9. Já representações diferentes disso, como um caractere unicode, não são nem binárias nem decimais.

```
fn main() {
    let a: char = '\u{2764}';
    let b: char = '9';
    let c: char = '0';

    println!("{} é um dígito? {}", a, a.is_digit(10));
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("{} é binário? {}", a, a.is_digit(2));

    println!("{} é um dígito? {}", b, b.is_digit(10));
    println!("{} é binário? {}", b, b.is_digit(2));

    println!("{} é um dígito? {}", c, c.is_digit(10));
    println!("{} é binário? {}", c, c.is_digit(2));
}
```

Como esperado na execução, vemos que ❤ não é nem binário nem decimal, que 0 é binário e que 9 é um dígito válido na base decimal.

```
$ cargo run --quiet
❤ é um dígito? false
❤ é binario? false
9 é um dígito? true
9 é binario? false
0 é um dígito? true
0 é binario? true
```

Outro método bem útil de `char` é o `escape_unicode()`, que possibilita pegar um caractere unicode e obter a sua representação numérica. O seu retorno é um struct do tipo `EscapeUnicode`, que pode ser colocado em uma `String` através do método `collect()`. Isso possibilita a interoperabilidade com sistemas que não suportam unicode, ou em protocolos de comunicação.

Vamos definir uma variável com um caractere unicode nela (no caso, →), e depois pegar a sua representação (no caso, `\u{2192}`).

```
fn main() {
    let a: char = '→';
    let repr: String =  a.escape_unicode().collect();
    println!("{}", repr);
}
```

```
$ cargo run --quiet
\u{2192}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Caso você não saiba o que é unicode ou como ele funciona por debaixo dos panos, este artigo da **Better Explained** (em inglês) detalha tudo o que você precisa saber, de maneira bem básica. Vale uma lida, em https://betterexplained.com/articles/unicode/.

É possível verificar se o caractere faz parte de algum alfabeto com a função `is_alphabetic()`.

```
fn main() {
    println!("{}", 'a'.is_alphabetic());
    println!("{}", '' .is_alphabetic());
    println!("{}", '→'.is_alphabetic());
}
```

```
$ cargo run --quiet
true
true
false
```

Também podemos identificar se o caractere é maiúsculo, com o `is_uppercase()`, ou minúsculo, com o `is_lowercase()`. Além disso, é possível detectar se é um espaço em branco, com o `is_whitespace()`; um alfanumérico, com o `is_alphanumeric()`; ou um numeral, com o `is_numeric()`.

```
fn main() {
    println!("Maiúsculo -> {}", 'a'.is_uppercase());
    println!("Minúsculo -> {}", 'a'.is_lowercase());
    println!("Espaço em branco -> {}", 'a'.is_whitespace());
    println!("Alfanumérico -> {}", 'a'.is_alphanumeric());
    println!("Numérico -> {}", 'a'.is_numeric());
}
```

```
$ cargo run --quiet
Maiúsculo -> false
Minúsculo -> true
Espaço em branco -> false
Alfanumérico -> true
Numérico -> false
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

A documentação completa do tipo `char` pode ser encontrada no seguinte link: https://doc.rust-lang.org/std/primitive.char.html.

## Booleanos

Rust possui um tipo para tratar booleanos, o `bool`:

```
let x: bool = false;
let y: bool = true;
```

O uso mais comum de booleanos é em condicionais, como o `if`. Veja o código:

```
fn main() {
    let x = false;
    let y: bool = true;
    if x {
        println!("X é true!");
    }
    if y {
        println!("Y é true!");
    }
}
```

A condicional `if` avalia se o resultado da operação recebida é verdadeiro (`true`) ou falso (`false`). Caso seja verdadeiro, o conteúdo entre chaves (`{}`) é executado; do contrário, não. Como `y` era a nossa variável com valor `true`, a execução gera a saída:

```
$ cargo run --quiet
Y é true!
```

O `if` espera um booleano, diferente de linguagens como C, em que **0** corresponde a `false`, e o que for diferente de zero


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

corresponde a `true`. O código a seguir não vai compilar, apesar de parecer válido para quem programa em C.

```
fn main() {
    let a = 1;

    if a {
        println!("A é igual a 1");
    }
}
```

Ao tentarmos compilar, obtemos o erro `error[E0308]: mismatched types`, que indica que é esperado um `bool`, e não um inteiro:

```
$ cargo run --quiet
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if a {
  |        ^ expected `bool`, found integer

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `boolean`

To learn more, run the command again with --verbose.
```

Rust também possui operadores booleanos, como a maioria das linguagens: o **and** é representado por `&&`; o **or** representado por `||`; e o **not** pelo sinal de exclamação (`!`).

```
fn main() {
    println!("true AND false é {}", true && false);
    println!("true OR false é {}", true || false);
    println!("NOT true é {}", !true);
}
```

O resultado é:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
true AND false é false
true OR false é true
NOT true é false
```

> A documentação completa do tipo `bool` pode ser encontrada no seguinte link https://doc.rust-lang.org/std/primitive.bool.html.

## Valores numéricos

Rust possui diversos primitivos para o tratamento de números. Esses primitivos estão divididos entre tipos com sinal e sem sinal. Por exemplo, o tipo `u32` é um inteiro de 32 bits sem sinal. Tipos com e sem sinal diferem entre si, pois o sinal (positivo ou negativo) ocupa um bit na memória, diminuindo o valor máximo que pode ser armazenado naquela variável.

O código a seguir tenta armazenar um valor que não cabe nele, em um inteiro com sinal. Rust armazenará na variável o valor **-1** e dará um aviso na compilação. Veja:

```
fn main() {
    let a: i32 = 4294967295;
    println!("{}", a)
}
```

Ao compilarmos, recebemos um erro dizendo que nosso literal estoura o tamanho máximo que a variável suporta:

```
$ cargo run --quiet
error: literal out of range for `i32`
 --> src/main.rs:2:15
  |
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
2 |     let a: i32 = 4294967295;
  |                  ^^^^^^^^^^
  |
  = note: `#[deny(overflowing_literals)]` on by default
  = note: the literal `4294967295` does not fit into the type `i3
2` whose range is `-2147483648..=2147483647`

error: aborting due to previous error

error: could not compile `overflowing_literals`

To learn more, run the command again with --verbose.
```

Ao convertermos nosso tipo para um tipo sem sinal, o problema deixa de existir:

```
fn main() {
    let a: u32 = 4294967295;
    println!("{}", a)
}
```

```
$ cargo run --quiet
4294967295
```

É importante saber quando será necessário usar uma variável que pode ter valores positivos e negativos, ou quando ela receberá apenas valores positivos **ou** negativos. Isso otimiza nosso consumo de memória, deixando a execução mais leve.

Os tipos primitivos inteiros possuem duas funções que podem nos ajudar aqui, o `min_value` e o `max_value`. Com eles, conseguimos ver de onde até onde podemos ir em cada um de nossos tipos:

```
fn main() {
    println!("i8 = {} a {}", i8::min_value(), i8::max_value());
    println!("i16= {} a {}", i16::min_value(), i16::max_value());
    println!("i32= {} a {}", i32::min_value(), i32::max_value());
    println!("i64= {} a {}", i64::min_value(), i64::max_value());
    println!("u8 = {} a {}", u8::min_value(), u8::max_value());
    println!("u16= {} a {}", u16::min_value(), u16::max_value());
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("u32= {} a {}", u32::min_value(), u32::max_value());
    println!("u64= {} a {}", u64::min_value(), u64::max_value());
}
```

Vamos dar uma olhada na execução.

```
$ cargo run --quiet
i8 = -128 a 127
i16= -32768 a 32767
i32= -2147483648 a 2147483647
i64= -9223372036854775808 a 9223372036854775807
u8 = 0 a 255
u16= 0 a 65535
u32= 0 a 4294967295
u64= 0 a 18446744073709551615
```

Os tipos inteiros de Rust possuem alguns métodos interessantes para trabalhar com dados binários, como o `count_ones()` e o `count_zeros()`, que retornam o número de uns e zeros na sua representação binária, respectivamente. Também há o `trailing_zeros()` e o `leading_zeros()`, que retornam a quantidade de zeros no começo ou no fim dessa representação. Eles são muito úteis quando falamos de desenvolvimento de hardware ou protocolos de comunicação.

Vamos ver um exemplo com o número **1**, cuja representação binária é **00000001** para um inteiro de 8 bits.

```
fn main() {
    let a: i8 = 1;
    println!("Uns: {}", a.count_ones());
    println!("Zeros: {}", a.count_zeros());
}
```

Com isso, temos:

```
$ cargo run --quiet
Uns: 1
Zeros: 7
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Você pode rotacionar os bits de seu inteiro em *n* bits, com o `rotate_left(n)` ou o `rotate_right(n)`, ou invertê-los com o `swap_bytes()`.

```
fn main() {
    let a: i8 = 1;
    println!(">>: {}", a.rotate_left(7));
    println!(">>: {}", a.rotate_right(8));
    println!(">>: {}", a.swap_bytes());
}
```

```
$ cargo run --quiet
>>: -128
>>: 1
>>: 1
```

Outros métodos dos tipos inteiros podem ser vistos na documentação oficial:

- i8 – https://doc.rust-lang.org/std/primitive.i8.html
- i16 – https://doc.rust-lang.org/std/primitive.i16.html
- i32 – https://doc.rust-lang.org/std/primitive.i32.html
- i64 – https://doc.rust-lang.org/std/primitive.i64.html
- u8 – https://doc.rust-lang.org/std/primitive.u8.html
- u16 – https://doc.rust-lang.org/std/primitive.u16.html
- u32 – https://doc.rust-lang.org/std/primitive.u32.html
- u64 – https://doc.rust-lang.org/std/primitive.u64.html

Além disso, Rust possui variáveis de ponto flutuante, onde são armazenados valores com decimais. Tais variáveis são definidas desta forma:

```
fn main() {
    let a: f64 = 42949.67295;
    println!("{}", a)
}
```

O retorno desse código é como o esperado:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
42949.67295
```

Novamente, o tamanho em bits faz a diferença. Se a variável anterior fosse definida como `f32`, a execução do código mostraria apenas duas casas decimais:

```
fn main() {
    let a: f32 = 42949.67295;
    println!("{}", a)
}
```

```
$ cargo run --quiet
42949.67
```

Assim como os tipos inteiros, os tipos de ponto flutuante possuem métodos para nos ajudar a lidar com eles. Temos o método `floor()`, que retorna o inteiro anterior; o `ceil()`, que retorna um inteiro acima; o `round()`, que arredonda; o `truncate()`, que retorna a parte inteira do número; e o `fract()`, que retorna a parte fracionária.

Também temos como verificar se o número é finito, com `is_finite()` e `is_infinite()`, ou mesmo se ele é um **NaN**, com `is_nan()`. Veja o exemplo:

```
fn main() {
    let a: f32 = 3.549236;
    println!("Piso: {}", a.floor());
    println!("Teto: {}", a.ceil());
    println!("Arredondado: {}", a.round());
    println!("Truncado: {}", a.trunc());
    println!("Fracionário: {}", a.fract());

    println!("É finito?: {}", a.is_finite());
    println!("É infinito?: {}", a.is_infinite());
    println!("É NaN?: {}", a.is_nan());
}
```

Esse código traz o seguinte resultado:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
Piso: 3
Teto: 4
Arredondado: 4
Truncado: 3
Fracionário: 0.54923606
É finito?: true
É infinito?: false
É NaN?: false
```

> A documentação completa para `f32` pode ser encontrada em https://doc.rust-lang.org/std/primitive.f32.html. Já a para `f64`, acesse https://doc.rust-lang.org/std/primitive.f64.html.

## Arrays e slices

Como não poderia deixar de ser, Rust possui o indispensável tipo array. Ele é declarado utilizando colchetes ( `[]` ), e seus itens podem ser acessados por meio do índice, que inicia em 0. Veja o exemplo:

Vamos definir um array de `string` chamado `a`, e depois acessaremos seu conteúdo pelo índice de cada uma das `strings` dentro do array.

```
fn main() {
    let a = ["livro", "de", "Rust"];
    println!("Programando em {}, acompanhando o {}", a[2], a[0]);
}
```

```
$ cargo run --quiet
Programando em Rust, acompanhando o livro
```

Você pode definir um array em Rust utilizando também a sintaxe:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
let a = [0; 5];
```

Teremos um array imutável com cinco elementos, todos inicializados com o valor

1. Ele não difere em nada do inicializado item a item, que corresponde ao código:

```
let a = [0, 0, 0, 0, 0];
```

Vamos modificar nosso exemplo. Em vez de um array de `string`, criaremos um com inteiros cujo valor inicial será `0`.

```
fn main() {
    let a = [0; 5];
    println!("Programando em {}, acompanhando o {}", a[2], a[0]);
}
```

```
$ cargo run --quiet
Programando em 0, acompanhando o 0
```

Você pode obter o número de elementos de um array usando o método `len`:

```
fn main() {
    let a = [0; 5];

    println!("{}", a.len());
}
```

```
$ cargo run --quiet
5
```

Em Rust, arrays têm o tipo `[T; N]`, em que `T` é o tipo dos elementos do array e `N` é o seu tamanho. Isso quer dizer que ele é formado por vários elementos do mesmo tipo.

Vamos modificar nosso exemplo: tornar o nosso array mutável e tentar modificar um dos seus elementos.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    let mut a = [0; 5];
    a[2] = "Rust";

    println!("Programando em {}, acompanhando o {}", a[2], a[0]);
}
```

```
$ cargo run --quiet
error[E0308]: mismatched types
 --> src/main.rs:3:12
  |
3 |     a[2] = "Rust";
  |            ^^^^^^ expected integer, found `&str`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `array`

To learn more, run the command again with --verbose.
```

Esse erro ocorre porque nosso array foi inicializado como um conjunto de inteiros, e estamos atribuindo uma string estática a um de seus elementos. Para que isso funcione, precisamos modificar a declaração de nosso array, de forma que o valor inicial de cada elemento seja do tipo `str`. Usaremos uma string vazia para isso, `""`.

```
fn main() {
    let mut a = [""; 3];
    a[0] = "livro";
    a[1] = "de";
    a[2] = "Rust";

    println!("Programando em {}, acompanhando o {}", a[2], a[0]);
}
```

E conseguimos o mesmo resultado de antes:

```
$ cargo run --quiet
Programando em Rust, acompanhando o livro
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Em Rust, podemos pegar pedaços de nosso array e utilizá-los em novas variáveis, sem modificar o array original. Isso chama-se *slice*. Veja o próximo exemplo, no qual defino um array `a` de 10 posições e, a partir dele, crio dois outros: `b`, com a lista completa de `a`, e `c`, que possui dois elementos de `a`.

```
fn main() {
    let a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
    let b = &a[..];
    let c = &a[3..5];

    println!("a tem {} elementos", a.len());
    println!("b tem {} elementos", b.len());
    println!("c tem {} elementos", c.len());
}
```

```
$ cargo run --quiet
a tem 10 elementos
b tem 10 elementos
c tem 2 elementos
```

Para percorrer os elementos de um array, usamos uma construção bem conhecida de quem já programa: `for i in array`. Isso vai executar o código entre chaves para cada elemento do array, colocando-o dentro da variável `i`.

```
fn main() {
    let array = ['a', 'b', 'c', 'd'];

    for i in array {
        println!("{}", i);
    }
}
```

```
$ cargo run --quiet
a
b
c
d
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

A documentação completa do tipo `array` pode ser vista em https://doc.rust-lang.org/std/primitive.array.html.

## Tuplas

Tupla é uma lista de valores fixos independente de tipos. Comumente, ela é utilizada como uma dupla chave e valor, mas em Rust podemos ter vários elementos em uma tupla:

```
let a = ('a', "char");
let b = (42, "integer");
let c = ("Nome", "Marcelo");
let d = ("Ru", 5, 't');
```

Você pode acessar os dados de uma tupla pelo índice do elemento:

```
fn main() {
  let a = ("Ru", 5, 't');
  let a0 = a.0;
  let a1 = a.1;
  let a2 = a.2;

  println!("a0:{}, a1:{}, a2:{}", a0, a1, a2);
}
```

Outro meio seria por um `let` encadeado, também conhecido como **let destrutivo**:

```
fn main() {
  let a = ("Ru", 5, 't');
  let (a0, a1, a2) = a;

  println!("a0:{}, a1:{}, a2:{}", a0, a1, a2);
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Ambos vão retornar o mesmo resultado:

```
$ cargo run --quiet
a0:Ru, a1:5, a2:t
```

E, claro, você pode jogar tuplas dentro de um array desde que ele tenha o mesmo formato. No exemplo a seguir, temos três tuplas com um inteiro e uma string estática ( `str` ) e, ao colocá-las em um array, tudo funciona corretamente:

```
let a = (1, "um");
let b = (2, "dois");
let c = (3, "tres");

let d = [a, b, c];
```

No próximo exemplo, temos três tuplas totalmente diferentes entre si: a primeira tem um inteiro e um ponto flutuante; a segunda, dois `str` ; e a terceira, um `str` e um `char` .

```
fn main() {
    let a = (1, 45.44);
    let b = ("Numero", "dois");
    let c = ("Letra", 'c');

    let d = [a, b, c];
}
```

Ao tentarmos compilar o código, o resultado não será o esperado:

```
$ cargo run --quiet
error[E0308]: mismatched types
 --> src/main.rs:6:17
  |
6 |     let d = [a, b, c];
  |                 ^ expected integer, found `&str`
  |
  = note: expected type `({integer}, {float})`
             found tuple `(&str, &str)`
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
error: aborting due to previous error

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `tuple`

To learn more, run the command again with --verbose.
```

Temos um erro do tipo `error[E0308]: mismatched types`, que indica que os tipos informados não correspondem aos esperados. Como dito anteriormente, elementos de um array devem ser do mesmo tipo e, no caso, temos três tipos diferentes:

```
let a = (1, 45.44);         // tipo `(i32, f32)`
let b = ("Numero", "dois"); // tipo `(str, str)`
let c = ("Letra", 'c');     // tipo `(str, char)`
```

## Enums

**Enum** é um tipo de dado, útil para criar listas de elementos constantes organizados. Em Rust, ele é definido com a palavra-chave `enum`, e o acesso aos seus elementos é feito com a expressão `::`.

Vamos usar como exemplo as raças do povo livre da terra média de Tolkien como exemplo. Para isso, vamos criar um `enum` chamado `Race`, no qual teremos uma lista de valores representando as raças do povo livre da terra média. A seguir, criaremos uma estrutura chamada `Creature`, que possui os atributos `name` e `race`, em que `name` é uma string estática e `race` é um item de nosso `enum Race`.

Dentro de `main`, vamos criar duas instâncias de `Person`, e atribuir nome e gênero a elas. Veja o código.

```
enum Race {
    Dwarf,
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    Elf,
    Ent,
    Hobbit,
    Men
}

struct Creature {
    name: &'static str,
    gender: Race
}

fn main() {
    let elrond = Creature {
        name: "Elrond",
        gender: Race::Elf
    };
    let treebeard = Creature {
        name: "Treebeard",
        gender: Race::Ent
    };
}
```

Ao ser compilado, esse código gerará diversos alertas, pois não estamos usando as variáveis, como pode ser visto no exemplo a seguir.

```
$ cargo run --quiet
warning: unused variable: `elrond`
  --> src/main.rs:15:9
   |
15 |     let elrond = Creature {
   |         ^^^^^^ help: if this is intentional, prefix it with
an underscore: `_elrond`
   |
   = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `treebeard`
  --> src/main.rs:19:9
   |
19 |     let treebeard = Creature {
   |         ^^^^^^^^^ help: if this is intentional, prefix it wi
th an underscore: `_treebeard`
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
warning: variant is never constructed: `Dwarf`
 --> src/main.rs:2:5
  |
2 |     Dwarf,
  |     ^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

warning: variant is never constructed: `Hobbit`
 --> src/main.rs:5:5
  |
5 |     Hobbit,
  |     ^^^^^^

warning: variant is never constructed: `Men`
 --> src/main.rs:6:5
  |
6 |     Men,
  |     ^^^

warning: field is never read: `name`
  --> src/main.rs:10:5
   |
10 |     name: &'static str,
   |     ^^^^^^^^^^^^^^^^^^

warning: field is never read: `gender`
  --> src/main.rs:11:5
   |
11 |     gender: Race,
   |     ^^^^^^^^^^^

warning: 7 warnings emitted
```

Esses alertas podem ser ignorados por enquanto, já que o objetivo é apenas demonstrar como declaramos e usamos um `enum`. Em capítulos posteriores, vamos ver mais exemplos de uso do `enum` na prática.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 3.4 AGRUPANDO EM MÓDULOS

Uma boa prática em programação é agrupar seu código a algo que faça sentido em vez de deixar os métodos soltos pelo caminho. Rust possui um conceito comum a outras linguagens de programação, que é agrupar funções, traits etc. em módulos.

Rust foi projetada para ser uma linguagem de programação multiparadigma. Ela possui conceitos de Programação Funcional, como a imutabilidade de dados por padrão ou o uso de avaliação de funções. Porém, também é possível encontrar Programação Imperativa em suas estruturas de decisão. O agrupamento em módulos é encontrado nesses dois paradigmas.

Um módulo é definido pela palavra-chave `mod`, e tudo o que for definido dentro dele é visível apenas neste lugar. Ou seja, o que é definido dentro de um módulo é sempre privado. Para tornar algo público, é necessário utilizar a palavra-chave `pub`.

O acesso aos métodos de um módulo é feito pela notação `nome_do_modulo::metodo`. A seguir, criarei um módulo que possui métodos públicos para a realização das funções matemáticas `add`, `divide`, `subtract` e `multiply`. Nele, terei o método privado `is_zero` e, a partir de `main`, chamarei esses métodos.

```
mod compute {
    // private function
    fn is_zero(number: i32) -> bool {
        if number == 0 { return true };
        false
    }

    pub fn add(a: i32, b: i32) -> i32 {
        a + b
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    }

    pub fn divide(a: i32, b: i32) -> i32 {
        if is_zero(b) { return 0 };
        a / b
    }

    pub fn subtract(a: i32, b: i32) -> i32 {
        a - b
    }

    pub fn multiply(a: i32, b: i32) -> i32 {
        a * b
    }
}

fn main() {
    let a: i32 = 10;
    let b: i32 = 4;
    println!("{} + {} = {}", a, b, compute::add(a, b));
    println!("{} / {} = {}", a, b, compute::divide(a, b));
    println!("{} - {} = {}", a, b, compute::subtract(a, b));
    println!("{} * {} = {}", a, b, compute::multiply(a, b));
}
```

Ao compilar o código, podemos ver o resultado de cada execução de nossas funções dentro do módulo.

```
 $ cargo run --quiet
10 + 4 = 14
10 / 4 = 2
10 - 4 = 6
10 * 4 = 40
```

Você pode utilizar o método `use` para criar apelidos para métodos de módulos, tornando sua chamada menos verbosa. Veja o exemplo a seguir, no qual crio o apelido `my_add` para `compute::add`.

```
mod compute {
    pub fn add(a: i32, b: i32) -> i32 {
        a + b
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    }
}

use compute::add as my_add;

fn main() {
    let a: i32 = 10;
    let b: i32 = 4;
    println!("{} + {} = {}", a, b, my_add(a, b));
}
```

```
$ cargo run --quiet
10 + 4 = 14
```

## 3.5 COMENTÁRIOS

Um recurso importante das linguagens de programação é o comentário. Comentar um código é uma arte que vem sendo esquecida com o decorrer do tempo, graças ao mantra de que um bom código não precisa de comentários. Isso é besteira; um bom código com comentários é muito melhor do que termos apenas um bom código.

Lembre-se da máxima do Chuck Norris: codifique sempre como se a próxima pessoa a mexer em seu código fosse o Chuck. Você realmente não gostará se ele não apreciar seu código sem comentários.

Basicamente, comentários são linhas ignoradas pelo compilador, nas quais você pode escrever o que der na telha, mas de preferência algo que explique o que seu código faz. Em Rust, eles utilizam a barra vertical duplicada ( `//` ) no começo da linha, então será ignorado o que estiver depois, até o seu final.

Veja no código a seguir exemplos de comentários:


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    // esta variável é um inteiro sem
    // sinal de oito bits
    let a: u8 = 167;

    if a > 100 {
        // se ela for maior do que 100
        // será impresso "maior que 100"
        println!("A é maior do que 100");
    } else {
        // se ela não for maior do que 100
        // será impresso "menor ou igual
        // a 100"
        println!("A é menor ou igual a 100");
    }
}
```

## 3.6 O BOM E VELHO IF

Já demos uma rápida olhada no `if` quando falamos sobre booleanos e, como você viu, o `if` do Rust não é muito diferente do que você vai encontrar por aí, em outras linguagens. Na verdade, ele é bem parecido com o `if` do Ruby ou do Python, em que o uso de parênteses é opcional. Você pode encadear vários `if` utilizando o `else if`, bem como usá-lo para condicionar a inicialização de uma variável, o que é algo bem comum em Rust.

Veja o exemplo a seguir. Nele, definimos uma variável `x` que receberá o resultado da verificação feita com `if`. Se `10 + 5` for igual a `15`, o conteúdo de `x` será `10 + 5 is 15`; caso contrário, será `10 + 5 is not 15`.

```
fn main() {
    let x = if 10 + 5 == 15 {
        "10 + 5 é 15"
    } else {
        "10 + 5 não é 15"
    };
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("{}", x);
}
```

```
$ cargo run --quiet
10 + 5 é 15
```

No próximo exemplo, temos uma construção usando vários `if` aninhados. Ela permite várias condições em um mesmo `if`, possibilitando a melhor decisão dentre várias opções.

```
fn check_grade(grade: f32) -> () {
    if grade >= 0.0 && grade < 4.9 {
        println!("Não aprovado");
    } else if grade >= 4.9 && grade < 6.0 {
        println!("De exame");
    } else if grade >= 6.0 {
        println!("Aprovado");
    } else {
        println!("Nota inválida!");
    }
}

fn main() {
    let grade_a = 0.0;
    let grade_b = 3.2;
    let grade_c = 5.1;
    let grade_d = 8.3;

    check_grade(grade_a);
    check_grade(grade_b);
    check_grade(grade_c);
    check_grade(grade_d);
}
```

Repare que o retorno da nossa função `check_grade` é `()`, ou seja, nenhuma operação. Isso seria equivalente a uma função C que retornasse `void`.

```
$ cargo run --quiet
Não aprovado
Não aprovado
De exame
Aprovado
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 3.7 BUSCA DE PADRÕES COM MATCH

Uma opção bem mais poderosa do que o `if` é o `match`, que possibilita um operador condicional através da busca de padrões (*pattern matching*).

> **PATTERN MATCHING**
>
> Busca por padrões (ou *pattern matching*) é escrever código que, ao receber um valor, identifique qual atende ao padrão dentre diversas opções. Esse padrão pode ser um valor exato, como um número qualquer ou uma `string`, ou pode ser um intervalo em que o valor informado se encaixe.
>
> O uso de busca por padrões é um dos principais motivos da adoção recente de várias linguagens que utilizam o paradigma de Programação Funcional, por permitir analisar dados com estruturas complexas de forma simples e concisa, como é o `match` do Rust. Dificilmente você verá um código escrito em Haskell, Scala, Elixir ou OCaml que não faça uso de busca por padrões.

O código apresentado anteriormente poderia ser substituído pelo seguinte. Usaremos o `match` para buscar dentro de `ranges` inclusivos.

```
fn check_grade(grade: f32) -> () {
    match grade {
        0.0...4.8 => println!("Não aprovado"),
        4.9...5.9 => println!("De exame"),
        6.0...10.0 => println!("Aprovado"),
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
        _ => println!("Nota inválida!"),
    }
}

fn main() {
    check_grade(0.0);
    check_grade(3.2);
    check_grade(5.1);
    check_grade(8.3);
}
```

O resultado da execução continuará sendo o mesmo:

```
$ cargo run --quiet
Não aprovado
Não aprovado
De exame
Aprovado
```

Em nosso exemplo, usamos `ranges` inclusivos para nossa busca por padrão. Um `range` inclusivo é definido por um valor inicial, três pontos e um valor final. Repare que tanto o valor inicial quanto o final fazem parte do intervalo. Isso quer dizer que `1...5` inclui os números 1, 2, 3, 4 e 5.

Outro padrão usado foi o `_` (underline), indicando que nenhuma das opções apresentadas anteriormente foi atendida. O underline deve ser usado com cuidado, pois ele funciona como um *catch-all*, ou seja, um padrão que vai pegar qualquer coisa que não tenha sido atendida pelos outros padrões definidos. Se você colocá-lo no começo de sua lista de padrões, não será executado nenhum dos outros após o `_`. Você pode usar outros padrões para o match, como valores individuais ou valores diversos separados por `|` (pipe). Veja o exemplo:

```
fn check_number(number: i16) -> () {
    match number {
        0 => println!("Zero"),
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
        2 | 3 | 5 | 7 | 11 => println!("Primo"),
        _ => println!("Qualquer número"),
    }
}

fn main() {
    let number_a = 0;
    let number_b = 3;
    let number_c = 8;

    check_number(number_a);
    check_number(number_b);
    check_number(number_c);
}
```

O resultado dessa execução seria:

```
$ cargo run --quiet
Zero
Primo
Qualquer número
```

## Match e enums

Em muitas situações cotidianas, você precisa classificar um dado de acordo com uma categoria predefinida. O uso de `match` junto ao `enum` possibilita a busca de um status predefinido ou o gênero de uma pessoa, como em nosso próximo exemplo. Mas, antes, faremos uma pausa para discutirmos diretivas de compilação e código não utilizado.

O compilador do Rust valida se existe código escrito por você, mas que não serve para nada. Essa verificação é útil, pois, no final das contas, esse código ocupará espaço em seu programa. Apesar de vivermos na época dos terabytes, na qual espaço muitas vezes não é um problema, Rust foi pensada para ser uma linguagem usada também na programação de sistemas embarcados. E, nesse universo, cada bit extra conta.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Sendo assim, ao compilar, seu código Rust vai lhe dizer: "Ei, este código aqui está perdido, ninguém o chama. Não seria uma boa removê-lo?". E é aí que entram as diretivas de compilação. Elas nada mais são do que instruções que deixamos em nosso código para que o compilador faça (ou não) alguma de suas funções.

Neste código, vamos ter um `enum` com o gênero *masculino*, o *feminino* e a opção *outro*. Nosso `match` receberá uma dessas opções e imprimirá o selecionado. As outras opções nunca serão acessadas em nosso programa de exemplo, e Rust sabe disso. Por isso, vamos usar uma diretiva de compilação chamada `allow`, que indica ao compilador que ele pode permitir algo que normalmente geraria um aviso. Com o `#[allow(dead_code)]`, dizemos a ele: "Ei, Rust, tudo bem deixar esse código, que nunca será usado, jogado aí".

Voltemos ao nosso exemplo:

```
#[allow(dead_code)]
enum Gender {
    Male,
    Female,
    Other,
}

fn main() {
    let gender = Gender::Male;
    match gender {
        Gender::Other => println!("Outro"),
        Gender::Male => println!("Homem"),
        Gender::Female => println!("Mulher")
    }
}
```

E o resultado da execução:

```
$ cargo run --quiet
Homem
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## Match e tuplas

Vimos como o uso de `match` facilita nossa vida e é poderoso. Mas e em situações nas quais precisamos comparar não apenas um, mas dois valores? Por exemplo, imagine que eu queira encontrar algum zero em um par de valores.

No próximo exemplo, defino uma função `check_tuple` que valida se uma tupla com dois inteiros possui um valor zero. Além disso, ela identifica onde está o zero, se no primeiro ou no segundo elemento. Veja o código:

```
#[allow(unused_variables)]
fn check_tuple(t: (i32, i32)) -> () {
    match t {
        (x, 0) => println!("O segundo é zero"),
        (0, x) => println!("O primeiro é zero"),
        _ => println!("Nenhum zero"),
    }
}

fn main() {
    check_tuple((0, 10));
    check_tuple((33, 0));
    check_tuple((8, 12));
}
```

E o resultado será:

```
$ cargo run --quiet
O primeiro é zero
O segundo é zero
Nenhum zero
```

Perceba que temos outra diretiva de compilação aqui, o `#[allow(unused_variables)]`. Ele diz ao Rust que não será um problema termos variáveis que nunca serão utilizadas (no caso, `x`) em nosso código. Outra forma de declarar variáveis que



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

efetivamente não serão usadas é colocar um `_` antes de seu nome, como a seguir.

```
fn check_tuple(t: (i32, i32)) -> () {
    match t {
        (_x, 0) => println!("O segundo é zero"),
        (0, _x) => println!("O primeiro é zero"),
        _ => println!("Nenhum zero"),
    }
}

fn main() {
    check_tuple((0, 10));
    check_tuple((33, 0));
    check_tuple((8, 12));
}
```

## Vinculação de match

No exemplo a seguir, temos um método chamado `is_vowel_or_consonant` que utiliza o `match` para identificar se um caractere passado é uma vogal ou não. Esse método retorna o caractere `c`, caso seja uma consoante, e `v`, caso seja uma vogal.

Depois, dentro da função `main`, usamos o `match` para ver se o retorno da função é `c` ou `v`, e incrementamos em um contador para imprimir posteriormente a quantidade de vogais e consoantes na string. Vale a pena lembrar que o `()`, utilizado com nosso padrão `_`, é equivalente ao `void` da C.

```
fn is_vowel_or_consonant(c: char) -> char {
    match c {
        'a' | 'e' | 'i' | 'o' | 'u' => 'v',
        'A' | 'E' | 'I' | 'O' | 'U' => 'v',
        _ => 'c'
    }
}

fn main() {
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let name: &'static str  = "Marcelo";
    let mut vowel_count = 0;
    let mut consonant_count = 0;

    for a in name.chars() {
        match is_vowel_or_consonant(a) {
            'v'  => vowel_count += 1,
            'c'  => consonant_count += 1,
            _ => ()
        }
    }

    println!("O nome {} tem {} vogais e {} consoantes",
             name, vowel_count, consonant_count);
}
```

E o resultado de nossa execução será:

```
$ cargo run --quiet
O nome Marcelo tem 3 vogais e 4 consoantes
```

Até aí, nada demais. Mas se quisermos usar o valor retornado pelo método `is_vowel_or_consonant` dentro de nosso segundo `match`, precisamos utilizar um mecanismo chamado *vinculação de match*. Com ele, é possível vincular o `match` ao retorno de um método, usando o sinal de `@` (arroba). Assim, o resultado da execução do método é colocado na variável. No exemplo a seguir, no segundo `match`, vou vincular através do `@` o retorno da função `is_vowel_or_consonant` a uma variável `r` e imprimir seu valor.

A princípio, a sintaxe pode ser um pouco estranha e não se parecer com nada que você tenha feito antes, mas é simples de entender. Antes do sinal `@`, você passa o nome da variável a que deseja vincular o retorno da função passada para o `match`, e ele usa-a para comparar com os padrões.

```
fn is_vowel_or_consonant(c: char) -> char {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    match c {
        'a' | 'e' | 'i' | 'o' | 'u' => 'v',
        'A' | 'E' | 'I' | 'O' | 'U' => 'v',
        _ => 'c'
    }
}

fn main() {
    let name: &'static str  = "Marcelo";

    for a in name.chars() {
        match is_vowel_or_consonant(a) {
            r @ 'v'  => println!("{}", r),
            r @ 'c'  => println!("{}", r),
            _ => ()
        }
    }
}
```

E o resultado da execução passará a:

```
c
v
c
c
v
c
v
```

## 3.8 WHILE

Rust também possui o bom e velho `while` , que executa uma ação enquanto a condição passada for verdadeira.

```
fn main() {
    let mut a = 0;

    while a < 10 {
        a += 1;
        println!("Agora a é {}", a);
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
Agora a é 1
Agora a é 2
Agora a é 3
Agora a é 4
Agora a é 5
Agora a é 6
Agora a é 7
Agora a é 8
Agora a é 9
Agora a é 10
```

Perceba que usamos um formato para incrementar nossa variável, semelhante ao que temos em Ruby, o `a += 1;`. Essa linha equivale a `a = a + 1;`.

Se você programar em C, pode se perguntar se o operador `++` funciona em Rust. Mas a resposta é não. Essa é uma decisão de design da linguagem, cujo objetivo é evitar entendimentos equivocados na leitura de um bloco de código, como pode ocorrer com pessoas que não são habituadas a usar `++a` ou `a++`.

Uma forma fácil de visualizar o problema de falta de legibilidade, com os sinais de incremento disponíveis em C ou Java, é o código seguinte. Você saberia dizer qual será o resultado da operação apenas olhando para ela?

```
#include <stdio.h>

void main() {
    int i = 0;
    printf(">> %d\n", i++ + ++i);
}
```

## 3.9 LOOP

O `loop` é uma outra forma de escrever a expressão `while`



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

`true` , que executará para sempre, e é muito empregado em sistemas embarcados. Veja o código:

```
fn main() {
    loop {
        println!("Executando para sempre...");
    }
}
```

Ao executá-lo, você verá a mensagem *Executando para sempre...* ser impressa indefinidamente em seu console. Para interromper a execução, pressione `CTRL+C` . Essa combinação de teclas envia um sinal `SIGINT` para o processo em execução, e ele é encerrado pelo sistema operacional.

Você pode utilizar o `break` para encerrar um `loop` . Veja o exemplo a seguir; ele imprimirá apenas onze vezes nossa mensagem (e não para sempre, como no anterior):

```
fn main() {
    let mut a = 0;
    loop {
        println!("Executando para sempre...");
        a += 1;
        if a > 10 { break; }
    }
}
```

O uso do `break` nesse exemplo pode ser facilmente substituído por um `while` , como vemos em seguida. A escolha é sua.

```
fn main() {
    let mut a = 0;
    while a < 10  {
        println!("Executando para sempre...");
        a += 1;
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 3.10 FOR E RANGES

Demos uma rápida olhada no `for` quando falamos do método `iter()` do tipo `array`. O `for` é utilizado para percorrer cada um dos itens de uma coleção, como o já citado array, ou os ranges – intervalos numéricos separados por pontos.

O `range` é um tipo de dado bem interessante do Rust, definido em `std::ops::Range`. Com ele você pode criar intervalos entre valores de uma forma bem simples. Imagine que precise percorrer os números de 1 a 5. Em linguagens que não possuem ranges, teríamos uma construção como a seguinte:

```
let mut contador = 1;
while contador <= 5 {
    println!("Valor atual: {}", contador);
    contador += 1
}
```

A cada execução do nosso loop imprimimos o valor e incrementamos uma variável auxiliar mutável para a próxima interação. Com o uso de `range`, isso fica bem mais simples:

```
for contador in 1..=5 {
    println!("Valor atual: {}", contador);
}
```

Em Rust, temos dois tipos de `range`, os **inclusivos** e os **exclusivos**. O `range` com três pontos `inicio...fim` chama-se **range inclusivo**, pois inclui o valor definido como *fim*. Ele funciona apenas quando trabalhamos com busca por padrões, dentro de um bloco `match`. O `range` com o formato `inicio..=fim` também é um `range inclusivo` e que pode ser usado com o bloco `for`.

```
for contador in 1..=5 {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Valor atual: {}", contador);
}
```

```
Valor atual: 1
Valor atual: 2
Valor atual: 3
Valor atual: 4
Valor atual: 5
```

Os `ranges` exclusivos diferem-se dos inclusivos por usarem dois pontos em vez de três. Enquanto um `range` inclusivo `1..=5` inclui os números 1, 2, 3, 4 e 5, um `range` exclusivo `1..5` inclui os números 1, 2, 3 e 4.

```
for contador in 1..5 {
    println!("Valor atual: {}", contador);
}
```

```
Valor atual: 1
Valor atual: 2
Valor atual: 3
Valor atual: 4
```

## Conclusão

Neste capítulo, abordamos mais conceitualmente os tipos de dados básicos disponíveis no *crate* `std` do Rust, bem como outras características da linguagem, como comentários e módulos. Além disso, vimos os operadores condicionais e de repetição do Rust.

Eles são a base do nosso sistema de decisão em fluxos de execução de um código Rust. Alguns deles são velhos conhecidos nossos, como o `if`, mas também nos deparamos com o poderoso `match`. No próximo capítulo, vamos nos aprofundar no conceito de traits.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 4

# TRAITS E ESTRUTURAS

Interfaces são um dos conceitos da linguagem de programação Java de que mais gostamos. Com elas, podemos criar um contrato básico de definição de uma estrutura e, posteriormente, implementá-lo. Isso significa que, nesse contrato, dizemos quais ações devem ser feitas e, na implementação, declaramos como fazê-las.

Vamos pegar como exemplo um sistema padrão de login. Temos normalmente diversos perfis, como *admin* ou *operador*, que implementam uma interface padrão com as ações básicas para autenticação. Em Java, definiríamos isso em uma interface:

```
interface User {
   public void logout();
   public void login();
   public boolean isLoggedIn();
}
```

Em Rust, um trait é similar a uma interface em Java, mas com algumas diferenças. É possível codificar métodos no trait, algo que não podemos fazer na interface Java. Veja a seguir um trait para regras de autenticação:

```
trait User {
    // Temos um construtor onde vamos receber um nome de
    // usuário (login)
    fn new(username: &'static str) -> Self;
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    // retorna o login definido em new
    fn username(&self) -> &'static str;

    // logar-se no sistema
    fn login(&self) -> &'static str;

    // deslogar-se no sistema
    fn logout(&self) -> &'static str;

    // verificar se está logado
    fn is_logged_in(&self) -> bool {
        false
    }
}
```

Um trait é definido com a palavra reservada `trait`, e seu nome deve ser iniciado com uma letra em maiúscula, indicando que é uma constante. O primeiro método que definimos foi o `new`, chamado na inicialização de uma instância que implementa nosso trait. Ele recebe uma string estática, nomeada como `username`, e retorna um tipo genérico `Self`.

```
fn new(username: &'static str) -> Self;
```

Definimos também um método para retornar o nome, anteriormente definido, bem como outros três que vão simular ações de login, logout e uma validação para ver se o usuário está logado. Todos recebem como referência o contexto atual `self`, e os métodos `username`, `login` e `logout` não têm definições ainda, apenas uma assinatura dizendo que retornam uma string estática. Já o método `is_logged_in`, que retorna se o usuário está ou não logado, está implementado, porém sempre devolvendo `false`.

```
fn username(&self) -> &'static str;
fn login(&self) -> &'static str;
fn logout(&self) -> &'static str;
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn is_logged_in(&self) -> bool {
    false
}
```

Esse código não faz nada de útil neste momento, apenas define o que é um usuário. Vamos criar alguns tipos de usuários, mas primeiro precisamos definir uma estrutura na qual agruparemos os métodos e as variáveis relativos a ela.

Em Rust, a definição de uma estrutura é feita com a palavra reservada `struct`, e seu nome também deve ser uma constante (ou seja, iniciado com uma letra maiúscula). Veja o código:

```
struct Admin { username: &'static str }
struct Operador { username: &'static str }
struct BasicUser { username: &'static str }
```

Perceba que cada um de nossos usuários possui uma variável `username` do tipo referência para uma string estática. Essa variável vai conter os nomes dos usuários. Vamos agora implementar o trait para a estrutura `Admin`.

A implementação de um trait para uma estrutura é feita com a palavra reservada `impl`, seguida do nome do trait, da palavra reservada `for` e do nome da `struct`. Depois, implementamos cada um dos métodos definidos no trait, que ainda não foram implementados.

```
impl User for Admin {
    fn new(username: &'static str) -> Admin {
        Admin { username: username }
    }

    fn username(&self) -> &'static str {
        self.username
    }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    fn login(&self) -> &'static str {
        "Usuário do tipo ADMIN entrou no sistema"
    }

    fn logout(&self) -> &'static str {
        "Usuário do tipo ADMIN saiu do sistema"
    }
}
```

Agora já podemos instanciar um usuário `admin` a partir de `Admin` e usar os métodos definidos nele. Para tanto, criamos uma variável chamada `admin` do tipo `Admin`, inicializada com um novo `User`. Veja:

```
fn main() {
    let admin: Admin = User::new("Corleone");

    println!("Bem-vindo usuário {}", admin.username());
    println!("{}", admin.login());
    println!("{}", admin.logout());
}
```

O resultado da execução será:

```
$  cargo run --quiet
warning: struct is never constructed: `Operador`
 --> src/main.rs:4:8
  |
4 | struct Operador {
  |        ^^^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

warning: struct is never constructed: `BasicUser`
 --> src/main.rs:7:8
  |
7 | struct BasicUser {
  |        ^^^^^^^^^

warning: 2 warnings emitted

Bem-vindo usuário Corleone
Usuário do tipo ADMIN entrou no sistema
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
Usuário do tipo ADMIN saiu do sistema
```

Perceba que o compilador nos avisa através do warning `#[warn(dead_code)]` que temos estruturas não utilizadas em nosso código, no caso, `Operador` e `BasicUser`. Vamos implementá-las:

```
trait User {
    // Temos um construtor onde vamos receber um nome de
    // usuário (login)
    fn new(username: &'static str) -> Self;

    // retorna o login definido em new
    fn username(&self) -> &'static str;

    // logar-se no sistema
    fn login(&self) -> &'static str;

    // deslogar-se do sistema
    fn logout(&self) -> &'static str;

    // verificar se está logado
    fn is_logged_in(&self) -> bool {
        false
    }
}

struct Admin { username: &'static str }
struct Operador { username: &'static str }
struct BasicUser { username: &'static str }

impl User for Admin {
    fn new(username: &'static str) -> Admin {
        Admin { username: username }
    }

    fn username(&self) -> &'static str {
        self.username
    }

    fn login(&self) -> &'static str {
        "Usuário do tipo ADMIN entrou no sistema"
    }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    fn logout(&self) -> &'static str {
        "Usuário do tipo ADMIN saiu do sistema"
    }
}

impl User for Operador {
    fn new(username: &'static str) -> Operador {
        Operador { username: username }
    }

    fn username(&self) -> &'static str {
        self.username
    }

    fn login(&self) -> &'static str {
        "Usuário do tipo OPERADOR entrou no sistema"
    }

    fn logout(&self) -> &'static str {
        "Usuário do tipo OPERADOR saiu do sistema"
    }
}

impl User for BasicUser {
    fn new(username: &'static str) -> BasicUser {
        BasicUser { username: username }
    }

    fn username(&self) -> &'static str {
        self.username
    }

    fn login(&self) -> &'static str {
        "Usuário do tipo BÁSICO entrou no sistema"
    }

    fn logout(&self) -> &'static str {
        "Usuário do tipo BÁSICO saiu do sistema"
    }
}


fn main() {
    let admin: Admin = User::new("Corleone");
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Bem-vindo usuário {}", admin.username());
    println!("{}", admin.login());
    println!("{}", admin.logout());

    let operador: Operador = User::new("PessoaQualquer");

    println!("Bem-vindo usuário {}", operador.username());
    println!("{}", operador.login());
    println!("{}", operador.logout());


    let basic_user: BasicUser = User::new("PessoaQualquer2");

    println!("Bem-vindo usuário {}", basic_user.username());
    println!("{}", basic_user.login());
    println!("{}", basic_user.logout());
}
```

E o resultado do nosso código será:

```
$ cargo run --quiet
Bem-vindo usuário Corleone
Usuário do tipo ADMIN entrou no sistema
Usuário do tipo ADMIN saiu do sistema
Bem-vindo usuário PessoaQualquer
Usuário do tipo OPERADOR entrou no sistema
Usuário do tipo OPERADOR saiu do sistema
Bem-vindo usuário PessoaQualquer2
Usuário do tipo BÁSICO entrou no sistema
Usuário do tipo BÁSICO saiu do sistema
```

## 4.1 DERIVANDO

Se essa palavra lhe deu dor de cabeça só de lembrar de derivadas e integrais que você viu em algum momento da sua vida, pode se tranquilizar que as coisas aqui são bem mais simples.

É possível utilizar traits do Rust pela diretiva `derive`, que funciona como uma espécie de implementação do trait em seu



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

código atual. Isso possibilita adicionar sem muito esforço mecanismos poderosos e funcionais a suas classes, como ordenação, inspeção de estruturas, igualdade e outros.

Veja alguns traits disponíveis para uso:

- **Debug**: permite inspecionar estruturas por meio da formatação da saída e do uso de `{:?}`.
- **Eq**, **PartialEq**, **Ord** e **PartialOrd**: para comparação e ordenação.
- **Hash**: para criar uma hash de tipo `&T`.
- **Default**: para criar instâncias vazias de um tipo de dados.

Voltando ao nosso exemplo dos usuários, imagine que precisamos inspecionar o que há dentro da nossa variável `admin`. Para isso, basta implementar o trait `debug` em nosso código:

```
trait User {
    // Temos um construtor onde vamos receber um nome de
    // usuário (login)
    fn new(username: &'static str) -> Self;

    // retorna o login definido em new
    fn username(&self) -> &'static str;

    // logar-se no sistema
    fn login(&self) -> &'static str;

    // deslogar-se do sistema
    fn logout(&self) -> &'static str;

    // verificar se está logado
    fn is_logged_in(&self) -> bool {
        false
    }
}

#[derive(Debug)]
struct Admin { username: &'static str }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
impl User for Admin {
    fn new(username: &'static str) -> Admin {
        Admin { username: username }
    }

    fn username(&self) -> &'static str {
        self.username
    }

    fn login(&self) -> &'static str {
        "Usuário do tipo ADMIN entrou no sistema"
    }

    fn logout(&self) -> &'static str {
        "Usuário do tipo ADMIN saiu do sistema"
    }
}

fn main() {
    let admin: Admin = User::new("Corleone");

    println!("{:?}", admin);
}
```

Perceba que usamos o `{:?}` em vez de simplesmente `{}`. O `{:?}` é provido pelo trait `debug` e permite-nos inspecionar nossos objetos. A execução desse código ficaria assim:

```
$  cargo run --quiet
Admin { username: "Corleone" }
```

Temos o nome de nossa `struct` e os valores dos campos disponíveis nela (no caso `username`). Uma recomendação é sempre manter o trait `debug` disponível em suas estruturas para facilitar a inspeção de código.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**A MACRO DBG!**

Para esse caso onde estamos utilizando `println!` com `{:?}` podemos usar a macro `dbg!` no lugar. A macro `dbg!` foi criada exclusivamente para fazer impressão de debug e deixar este processo bem mais simples. O mesmo código anterior, apenas trocando `println!("{:?}", admin);` por `dbg!(admin);` imprime o seguinte resultado:

```
$ cargo run --quiet
[src/main.rs:47] admin = Admin {
    username: "Corleone",
}
```

Veja que além das informações do `struct`, temos o nome do arquivo e a linha que estamos debugando! Tudo isso além de ser mais ergonômico de usar por não precisar escrever `"{:?}"`.

Os tipos de dados básicos, apresentados no capítulo anterior, possuem alguns traits padrões implementados, como o `PartialEq`, o `BitXor`, o `Eq` e outros mais. A seguir, vamos dar uma olhada em cada um deles.

## 4.2 PARTIALEQ E EQ

O trait `PartialEq` implementa uma série de funções para definir a relação de equivalência entre valores. Essa relação de equivalência parcial é definida por meio de **simetria** e **transitividade**.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Dizemos que algo é **simétrico** quando, na comparação de dois ou mais elementos (ou partes de um mesmo elemento), eles correspondam ponto a ponto, como os dois lados de uma figura do teste de Rorschach.

Figura 4.1: Um teste de Rorschach

Dizemos que algo é **transitivo** quando ele possui uma relação indireta com um elemento através de outro. Por exemplo, um tio e um sobrinho possuem uma relação transitiva por conta do elemento pai da criança, irmão do tio.

Dessa forma, podemos afirmar que, em um ambiente computacional de Orientação a Objetos, duas instâncias da mesma classe possuem uma relação transitiva, e que duas instâncias da mesma classe que possuam algum elemento igual são simétricas, devido ao peso desse elemento. Duas instâncias de uma classe `Pessoa`, cujo CPF seja o mesmo, são iguais por esses conceitos, e é aí que entra o `PartialEq`.

Já o trait `Eq` é usado em conjunto com o `PartialEq`, para incluir na comparação também a **reflexividade**, que indica a exata igualdade dos valores comparados, como se fossem refletidos em um espelho. Ou seja, em uma comparação entre **a** e **b**, o valor de **a** deve ser exatamente o mesmo valor de **b**.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Os traits `PartialEq` e `Eq` são responsáveis pelas operações de comparação em nossos tipos básicos, visto que esses tipos derivam deles. Quando executamos uma comparação com o sinal de `==` (igualdade), Rust utiliza a função `eq` do tipo básico, da mesma forma quando comparamos com `!=` (diferença), ela utiliza a função `ne`. Podemos fazer um paralelo com a sobrecarga de operadores quando um toma mais de uma ação, dependendo do contexto ao qual ele é chamado. Como podemos sobrescrever os métodos que Rust invoca na execução de um operador, isso torna-se muito simples.

Basicamente, os sinais `==` e `!=` são *syntax sugar*, ou seja, uma forma de tornar a leitura de um código mais simples. No exemplo a seguir, temos uma comparação usando o *syntax sugar* e as funções que ele usa por baixo dos panos para nossos tipos básicos. A igualdade pode ser verificada pelo uso de `==` ou de `eq`, já a diferença, pelo `!=` ou o `ne` (*not equal*). Cada tipo possui sua implementação de `ne` ou `eq`, e esses métodos são chamados quando utilizamos o sinal de `!=` ou de `==`. Caso o tipo não possua o `eq` implementado, o uso de `==` gerará um erro.

```
fn main() {
    let bol1: bool = true;
    let bol2: bool = false;
    println!("bol1 == bol2   -> {}", bol1 == bol2);
    println!("bol1.eq(&bol2) -> {}", bol1.eq(&bol2));
    println!("bol1 != bol2   -> {}", bol1 != bol2);
    println!("bol1.ne(&bol2) -> {}\n", bol1.ne(&bol2));

    let char1: char = 'a';
    let char2: char = 'b';
    println!("char1 == char2   -> {}", char1 == char2);
    println!("char1.eq(&char2) -> {}", char1.eq(&char2));
    println!("char1 != char2   -> {}", char1 != char2);
    println!("char1.ne(&char2) -> {}\n", char1.ne(&char2));
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let int1: i32 = 1;
    let int2: i32 = 2;
    println!("int1 == int2   -> {}", int1 == int2);
    println!("int1.eq(&int2) -> {}", int1.eq(&int2));
    println!("int1 != int2   -> {}", int1 != int2);
    println!("int1.ne(&int2) -> {}\n", int1.ne(&int2));

    let float1: f32 = 1.3;
    let float2: f32 = 2.4;
    println!("float1 == float2   -> {}", float1 == float2);
    println!("float1.eq(&float2) -> {}", float1.eq(&float2));
    println!("float1 != float2   -> {}", float1 != float2);
    println!("float1.ne(&float2) -> {}\n", float1.ne(&float2));

    let str1: &'static str = "Marcelo";
    let str2: &'static str = "Willian";
    println!("str1 == str2  -> {}", str1 == str2);
    println!("str1.eq(str2) -> {}", str1.eq(str2));
    println!("str1 != str2  -> {}", str1 != str2);
    println!("str1.ne(str2) -> {}", str1.ne(str2));
}
```

O resultado da execução será:

```
$ cargo run --quiet
bol1 == bol2   -> false
bol1.eq(&bol2) -> false
bol1 != bol2   -> true
bol1.ne(&bol2) -> true

char1 == char2   -> false
char1.eq(&char2) -> false
char1 != char2   -> true
char1.ne(&char2) -> true

int1 == int2   -> false
int1.eq(&int2) -> false
int1 != int2   -> true
int1.ne(&int2) -> true

float1 == float2   -> false
float1.eq(&float2) -> false
float1 != float2   -> true
float1.ne(&float2) -> true
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
str1 == str2  -> false
str1.eq(str2) -> false
str1 != str2  -> true
str1.ne(str2) -> true
```

Quando falamos em estruturas, a derivação de `Eq` ou `PartialEq` deve considerar esses princípios matemáticos. Uma instância de `Pessoa` é parecida com outra instância de `Pessoa`, mas o valor de um campo de uma instância de `Pessoa` pode ou não ser igual ao valor do mesmo campo em outra instância de `Pessoa`. Podemos verificar esse caso em nosso dia a dia, por exemplo, as carteiras de habilitação. Todas possuem um padrão por serem feitas a partir de um modelo, e incluem campos para nome e data de nascimento. Logo, todas as pessoas habilitadas possuem um nome e uma data de nascimento.

É possível existir outra carteira de habilitação por aí cujo nome seja *Marcelo Fontes Castellani*, afinal, homônimos existem. Assim como pode existir outra habilitação cuja data de nascimento seja *30/06/1977*. Contudo, ainda que exista outra pessoa com a mesma data e o mesmo nome que os do Marcelo, ela ainda não será um dos autores deste livro.

Então, vamos derivar o `PartialEq` em nossas estruturas. Nesse caso, ele vai comparar os membros de ambas e, se todos forem iguais, ele vai considerá-las estruturas iguais. Veja o próximo exemplo, no qual criamos uma estrutura `MyStruct` com um membro chamado `member`. Depois, definimos duas instâncias de `MyStruct` e as comparamos.

```
#[derive(PartialEq)]
struct MyStruct { member: i32 }

fn main() {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let a = MyStruct { member: 1 };
    let b = MyStruct { member: 1 };

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

Veja a execução:

```
$ cargo run --quiet
true
```

A implementação do trait `PartialEq` em nossa estrutura, por meio de derivação, possibilita o uso do *syntax sugar* ( `==` ) e a fácil comparação das nossas instâncias de estruturas em nosso código. Como o `PartialEq` lhe dá de graça a derivação de `Eq` , conseguimos comparar se um valor de uma instância é igual ao de outra, ou seja, se seus membros possuem os mesmos valores. Para tanto, vamos modificar o método `eq` do trait `PartialEq` , responsável pela comparação. Dessa forma, nem todos os membros precisam ser iguais. Veja um exemplo:

```
enum BookFormat {
    Paperback,
    Ebook
}

struct Book {
    isbn: i32,
    #[allow(dead_code)]
    title: &'static str,
    #[allow(dead_code)]
    format: BookFormat
}

impl PartialEq for Book {
    fn eq(&self, other: &Book) -> bool {
        self.isbn == other.isbn
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    let b1 = Book {
        isbn: 1234567890,
        title: "O Senhor dos Anéis",
        format: BookFormat::Paperback
    };
    let b2 = Book {
        isbn: 1234567890,
        title: "O Senhor dos Anéis",
        format: BookFormat::Paperback
    };
    let b3 = Book {
        isbn: 1234567810,
        title: "O Hobbit",
        format: BookFormat::Ebook
    };

    println!("{}", b1 == b2);
    println!("{}", b2 == b3);
    println!("{}", b1 == b3);
}
```

Em nosso exemplo, sobrescrevemos o método `eq` do `PartialEq` para considerar apenas o campo `isbn` quando comparar duas instâncias de `Book`. Veja novamente o uso de `#[allow(dead_code)]` para fins didáticos aqui. O resultado da execução é:

```
$ cargo run --quiet
true
false
false
```

Um exemplo interessante do uso de `PartialEq` ocorre quando comparamos dois valores `NaN` (ou **Not a Number**), um indicador usado em linguagens de programação para representar um valor indefinido ou impossível de ser representado como número. Sendo assim, dois valores `NaN` nunca serão iguais ( `NaN != NaN` ).



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Veja o código a seguir, no qual criamos dois floats de 64 bits e dois floats de 32 bits. Os tipos `f32` e `f64` implementam o trait `PartialEq` para comparação, como pode ser visto na documentação do tipo, em https://doc.rust-lang.org/std/primitive.f32.html.

```
fn main() {
    let a = 0.0f64 / 0.0f64;
    let b = 0.0f64 / 0.0f64;

    let c = 0.0f32 / 0.0f32;
    let d = 0.0f32 / 0.0f32;

    println!("a == b -> {}", a == b);
    println!("c == d -> {}", c == d);

    println!("a: {}", a);
    println!("b: {}", b);
    println!("c: {}", c);
    println!("d: {}", d);
}
```

Veja no resultado da execução que as comparações retornam `false`, mas que todas são `NaN`.

```
$  cargo run --quiet
a == b -> false
c == d -> false
a: NaN
b: NaN
c: NaN
d: NaN
```

## 4.3 PARTIALORD E ORD

Os traits `PartialOrd` e `Ord` são usados para ordenação de dados. Eles são os responsáveis pelo *syntax sugar* que permite o uso dos sinais de maior ( `>` ), menor ( `<` ), maior ou igual ( `>=` ) e


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

menor ou igual ( `<=` ). A diferença entre eles é que o `PartialOrd` faz uma comparação utilizando o `PartialEq` , enquanto o `Ord` faz a comparação usando o `Eq` . Para exemplificar, implementaremos uma classe `Person` que terá nome e idade. Vamos criar suas instâncias e, usando o `Ord` , ver quem é mais velho, pela comparação da data pelo método `cmp` .

Para implementar o `Ord` para `Person` , vamos derivar os traits dos quais ele depende, no caso, `PartialOrd` , `PartialEq` e `Eq` . Vamos também derivar os traits `Clone` e `Copy` , que serão explicados a seguir. Além disso, declararemos o uso da estrutura `Ordering` , que é o tipo de retorno usado em uma comparação de ordenação em Rust.

```
use std::cmp::Ordering;

#[derive(PartialOrd, PartialEq, Eq, Clone, Copy)]
struct Person {
    age: i32,
    name: &'static str
}

impl Ord for Person {
    fn cmp(&self, other: &Person) -> Ordering {
        (self.age).cmp(&(other.age))
    }
}

fn older(p1: Person, p2: Person) {
    if p1 > p2 {
        println!("{} tem mais anos de vida do que {}",
        p1.name, p2.name);
    } else {
        println!("{} tem mais anos de vida do que {}",
        p2.name, p1.name);
    }
}

fn main() {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let p1 = Person {
        age: 6,
        name: "Nicolas"
    };
    let p2 = Person {
        age: 32,
        name: "Willian"
    };
    let p3 = Person {
        age: 40,
        name: "Marcelo"
    };

    older(p1, p2);
    older(p2, p3);
    older(p1, p3);
}
```

Veja o resultado dessa execução:

```
$ cargo run --quiet
Willian tem mais anos de vida do que Nicolas
Marcelo tem mais anos de vida do que Willian
Marcelo tem mais anos de vida do que Nicolas
```

Perceba que derivamos dois outros traits em nosso código, `Copy` e `Clone`. Eles são responsáveis pela duplicação e pelo uso de cópias de objetos quando passamos eles como parâmetros do método `older`. Quando ocorre essa passagem de parâmetro, o Rust utiliza o trait `Clone` para gerar um *clone* de nossa estrutura, garantindo a imutabilidade. Por baixo dos panos, quando derivamos `Clone`, a trait faz uma chamada de `.clone()` para todos os atributos do struct.

Por padrão, Rust utiliza uma semântica de *move*, indicando que, quando algo é mexido na memória, seu valor é movido para o resultado da execução. No nosso código, ao compararmos `p1` com `p2`, o valor de `p2` deixa de existir.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**Gerenciamento de memória**

O gerenciamento de memória do Rust é uma de suas maiores qualidades. Diferente de outras linguagens com gerenciamento automático de memória – nas quais ações como cópia de valores são inferidas, ou utiliza-se o máximo de memória disponível para garantir o dado –, em Rust o gerenciamento é manual, ou seja, você tem que explicitamente dizer o que precisa manter e o que não precisa.

Os traits `Copy` e `Clone` são o jeito Rust de ter os dados persistindo entre ações destrutivas, como uma chamada de função. No exemplo, ao passarmos `p2` para `older`, o compilador identifica que `p2` saiu de `main` e foi para outro contexto, no caso `older`, e que seu valor não estará mais disponível na chamada subsequente.

A diferença entre `Copy` e `Clone` é que o `Copy` é extremamente barato em termos de processamento e ocorre de forma implícita. Já o `Clone` é sempre explícito (através do `.clone()`, que o `derive` do trait `Clone` facilita, como mencionamos a pouco) e pode ou não ser custoso. De forma a reforçar essas características, o Rust não permite que o `Copy` seja reimplementado, diferente do `Clone`, que pode ser implementado e executar qualquer código.

Se você remover os traits `Copy` e `Clone` da linha que contém o `#[derive]` e compilar o código, verá uma explicação detalhada disso:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
error[E0382]: use of moved value: `p2`
  --> src/main.rs:38:11
   |
28 |     let p2 = Person {
   |         -- move occurs because `p2` has type `Person`, which
 does not implement the `Copy` trait
...
37 |     older(p1, p2);
   |               -- value moved here
38 |     older(p2, p3);
   |           ^^ value used here after move

error[E0382]: use of moved value: `p1`
  --> src/main.rs:39:11
   |
24 |     let p1 = Person {
   |         -- move occurs because `p1` has type `Person`, which
 does not implement the `Copy` trait
...
37 |     older(p1, p2);
   |           -- value moved here
38 |     older(p2, p3);
39 |     older(p1, p3);
   |           ^^ value used here after move

error[E0382]: use of moved value: `p3`
  --> src/main.rs:39:15
   |
32 |     let p3 = Person {
   |         -- move occurs because `p3` has type `Person`, which
 does not implement the `Copy` trait
...
38 |     older(p2, p3);
   |               -- value moved here
39 |     older(p1, p3);
   |               ^^ value used here after move

For more information about this error, try `rustc --explain E0382
`.
error: could not compile `trait_person_copy_without_clone` due to
 3 previous errors
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 4.4 OPERAÇÕES ARITMÉTICAS E DE BIT

Cada operação aritmética possui um trait específico, bem como as operações de bit. Elas estão detalhadas na tabela a seguir:

| Operação | Trait |
|---|---|
| `a + b` | `std::ops::Add` |
| `a - b` | `std::ops::Sub` |

| Operação | Trait |
|---|---|
| `-a` | `std::ops::Neg` |
| `a * b` | `std::ops::Mul` |
| `a / b` | `std::ops::Div` |
| `a % b` | `std::ops::Rem` |
| `!a` | `std::ops::Not` |
| `a & b` | `std::ops::BitAnd` |
| `a \| b` | `std::ops::BitOr` |
| `a ^ b` | `std::ops::BitXor` |
| `a << b` | `std::ops::Shl` |
| `a >> b` | `std::ops::Shr` |
| `a == b` | `std::cmp::PartialEq::eq` |
| `a != b` | `std::cmp::PartialEq::ne` |
| `a > b` | `std::cmp::PartialOrd::gt` |
| `a < b` | `std::cmp::PartialOrd::lt` |
| `a >= b` | `std::cmp::PartialOrd::ge` |
| `a <= b` | `std::cmp::PartialOrd::le` |

Com exceção do trait `Rem`, todos os outros são facilmente


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

identificados pelo nome. Na maior parte das linguagens de programação, operações de resto são definidas como `mod` ou algo parecido, mas em Rust usa-se `Rem`, de *Remainder*.

As operações matemáticas aplicam-se aos tipos inteiros e de ponto flutuante, e as operações de bit, aos tipos inteiros e booleanos. Já as operações implementadas por `Shl` e `Shr` aplicam-se apenas aos inteiros. Veja o exemplo:

```
fn main() {
    println!("1 + 2: {}", 1 + 2);
    println!("1.3 + 2.4: {}", 1.3 + 2.4);

    println!("1 - 2: {}", 1 - 2);
    println!("1.3 - 2.4: {}", 1.3 - 2.4);

    println!("1 * 2: {}", 1 * 2);
    println!("1.3 * 2.4: {}", 1.3 * 2.4);

    println!("1 / 2: {}", 1 / 2);
    println!("1.3 / 2.4: {}", 1.3 / 2.4);

    println!("1 % 2: {}", 1 % 2);
    println!("1.3 % 2.4: {}", 1.3 % 2.4);

    println!("!false: {}", !false);
    println!("!3: {}", !3);

    println!("true & false: {}", true & false);
    println!("2 & 6: {}", 2 & 6);

    println!("true | false: {}", true | false);
    println!("12 | 5: {}", 12 | 5);

    println!("true ^ false: {}", true ^ false);
    println!("12 ^ 5: {}", 12 ^ 5);

    println!("12 << 5: {}", 12 << 5);
    println!("12 >> 5: {}", 12 >> 5);
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Veja a execução:

```
$ cargo run --quiet
1 + 2: 3
1.3 + 2.4: 3.7
1 - 2: -1
1.3 - 2.4: -1.0999999999999999
1 * 2: 2
1.3 * 2.4: 3.12
1 / 2: 0
1.3 / 2.4: 0.5416666666666667
1 % 2: 1
1.3 % 2.4: 1.3
!false: true
!3: -4
true & false: false
2 & 6: 2
true | false: true
12 | 5: 13
true ^ false: true
12 ^ 5: 9
12 << 5: 384
12 >> 5: 0
```

Como você já deve imaginar, podemos implementar esses traits em nossas estruturas. Vamos criar uma classe `Person`, que, quando somada a um inteiro, incrementa a idade da pessoa.

```
use std::ops::Add;

#[derive(Copy, Clone)]
struct Person {
    name: &'static str,
    age: i32
}

impl Add<i32> for Person {
    type Output = i32;

    fn add(self, b: i32) -> i32 {
        self.age + b
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    let p1 = Person { name: "Connor MacLeod", age: 502 };
    let x = p1 + 10;

    println!("A nova idade de {} é {}", p1.name, x);
}
```

Repare que temos uma nova linha em nosso código:

```
type Output = i32;
```

Essa linha define o tipo de dado que o trait vai retornar. Veja a execução:

```
$ cargo run --quiet
A nova idade de Connor MacLeod é 512
```

## Conclusão

Trait é um dos mecanismos mais interessantes da linguagem Rust. Ele proporciona uma forma elegante e prática de criar métodos genéricos que podem ser facilmente usados em suas estruturas. A partir da derivação de um trait já definido, sua estrutura passa a ter recursos, como comparadores, iteradores ou mesmo ferramentas de inspeção. Todos são passíveis de serem expandidos via implementação customizada. Mais informações sobre quais traits são implementados por cada um dos tipos básicos de Rust podem ser encontradas na documentação oficial: https://doc.rust-lang.org/stable/std/#primitives.

No próximo capítulo, falaremos sobre algumas das estruturas mais poderosas de Rust, como strings, vetores e programação genérica.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 5

# VETORES E ITERADORES

Em Rust, vetores são arrays redimensionáveis, cujos elementos possuem o mesmo tipo e podem ser livremente modificados para possuírem mais ou menos elementos. Uma das formas mais simples de inicializar um vetor é pelo uso da macro `vec!`. Ela recebe como parâmetro um array de dados do mesmo tipo e converte-o em um vetor.

```
fn main() {
    let vector = vec![1, 2, 3];

    println!("{:?}", vector);
}
```

```
$ cargo run --quiet
[1, 2, 3]
```

Outra forma de inicializar um vetor, com o conteúdo apresentado anteriormente, é por meio do uso de um `range` e do método `collect()`.

```
fn main() {
    let vector: Vec<i32> = (1..4).collect();

    println!("{:?}", vector);
}
```

Você pode inicializar um vetor com uma quantidade específica de um mesmo dado, como no exemplo a seguir, que inicializa um



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

vetor com cinco zeros. Veja:

```
fn main() {
    let vector = vec![0; 5];

    println!("{:?}", vector);
}
```

```
$ cargo run --quiet
[0, 0, 0, 0, 0]
```

Para trabalhar com nosso vetor, vamos criar uma lista telefônica, com nomes e números de contato. Para fazermos essa lista, precisamos de uma estrutura para adicionar os dados do nosso contato, para então criarmos duas instâncias e adicioná-las a um vetor.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];
    println!("{:?}", agenda);
}
```

Ao compilarmos, teremos:

```
$ cargo run --quiet
[Contact { name: "Marcelo", phone_number: "+55(11) 9.1234.5678" }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
, Contact { name: "Willian", phone_number: "+55(11) 9.8765.4321"
}]
```

Vamos utilizar o método `push()` para adicionar um novo contato à agenda. Ele adiciona um novo elemento ao final do vetor. Repare que nosso vetor, agora, é mutável, pois adicionamos o `mut` à sua definição.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let mut agenda = vec![contact_1, contact_2];
    println!("Agenda com 2 contatos \n {:?}", agenda);

    let contact_3 = Contact {
        name: "Nobody",
        phone_number: "+55(11) 9.9999.0000"
    };

    agenda.push(contact_3);
    println!("\nAgenda com 3 contatos \n {:?}", agenda);
}
```

O resultado é:

```
$ cargo run --quiet
Agenda com 2 contatos
 [Contact { name: "Marcelo", phone_number: "+55(11) 9.1234.5678"
},
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
  Contact { name: "Willian", phone_number: "+55(11) 9.8765.4321"
}]

Agenda com 3 contatos
 [Contact { name: "Marcelo", phone_number: "+55(11) 9.1234.5678"
},
  Contact { name: "Willian", phone_number: "+55(11) 9.8765.4321"
},
  Contact { name: "Nobody", phone_number: "+55(11) 9.9999.0000" }
]
```

Pelo método `pop()`, podemos remover um membro de nosso vetor. O valor removido via `pop()` é retornado para o código que o invocou.

Como em Rust você não pode simplesmente descartar o retorno de uma chamada, ele deverá ser atribuído a uma variável (para ser usado posteriormente) ou a `_`. A atribuição para `_` indica que não nos importamos com o retorno daquela operação e que ele não nos será útil.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let mut agenda = vec![contact_1, contact_2];
    println!("Agenda com 2 contatos \n {:?}", agenda);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let _ = agenda.pop();
    println!("\nAgenda com 1 contato \n {:?}", agenda);
}
```

O resultado é:

```
$ cargo run --quiet
Agenda com 2 contatos
 [Contact { name: "Marcelo", phone_number: "+55(11) 9.1234.5678"
}, Contact { name: "Willian", phone_number: "+55(11) 9.8765.4321"
 }]

Agenda com 1 contato
 [Contact { name: "Marcelo", phone_number: "+55(11) 9.1234.5678"
}]
```

Suponha que você deseja acessar um contato qualquer de sua agenda diretamente. Isso pode ser feito sem complicações em Rust, pois os membros de um vetor podem ser acessados através de seu índice, iniciando por 0 e indo até o tamanho do vetor menos um ( `len() - 1` ).

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    println!("Contato 1: {:?}", agenda[0]);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Contato 2: {:?}", agenda[1]);
}
```

Com isso, temos:

```
$ cargo run --quiet
Contato 1: Contact { name: "Marcelo", phone_number: "+55(11) 9.12
34.5678" }
Contato 2: Contact { name: "Willian", phone_number: "+55(11) 9.87
65.4321" }
```

Devemos ter atenção ao acessar um item de um vetor pelo seu índice. Se você acessar um índice que não existe, terá uma exceção de pânico. Isso pode ser evitado com o `try!`, que veremos mais à frente neste livro, no capítulo sobre testes.

Veja o código a seguir. Nele, tento acessar um item que não existe em nosso `vector`.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    println!("Contato 50: {:?}", agenda[50]);
}
```

```
$ cargo run --quiet
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
thread 'main' panicked at 'index out of bounds: the len is 2 but
the index is 50', src/main.rs:20:34
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace
```

## Iterando em vetores

Até agora, vimos como criar, adicionar ou remover elementos de um vetor em Rust – operações elementares que facilitam nosso dia a dia, mas não mostram o poder de trabalharmos com vetores.

Por serem coleções de qualquer tipo de dados, os vetores possibilitam agrupar instâncias e trabalhar com elas a partir da iteração sobre a coleção. Isso quer dizer que podemos agrupar dados em vetores e depois acessá-los um a um.

Uma forma de percorrer os dados de um vetor é utilizando o `for` . Vamos imprimir cada um dos contatos de nossa agenda.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    for contact in agenda {
        println!("{:?}", contact);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    }
}
```

```
$ cargo run --quiet
Contact { name: "Marcelo", phone_number: "+55(11) 9.1234.5678" }
Contact { name: "Willian", phone_number: "+55(11) 9.8765.4321" }
```

Nesse caso, o `for` é um açúcar sintático para um iterador de vetor, gerado a partir do método `iter()`. Esse iterador permite percorrer os itens de um vetor pelo método `next()`. O código apresentado no exemplo anterior poderia ser reescrito como a seguir, obtendo o mesmo resultado na execução.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];
    let mut agenda_iter = agenda.iter();

    println!("{:?}", agenda_iter.next().unwrap());
    println!("{:?}", agenda_iter.next().unwrap());
}
```

Mas e se eu tiver mais uma linha com `agenda_iter.next`, como a seguir?

```
#[derive(Debug)]
struct Contact {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];
    let mut agenda_iter = agenda.iter();

    println!("{:?}", agenda_iter.next().unwrap());
    println!("{:?}", agenda_iter.next().unwrap());
    println!("{:?}", agenda_iter.next().unwrap());
}
```

Ao chegar à terceira linha de `println!("{:?}", agenda_iter.next().unwrap());` vamos ter um erro `thread 'main' panicked at 'called Option::unwrap() on a None value'` , pois não temos três elementos em nossa agenda. Para evitar isso, o Rust retorna os dados de nosso iterador encapsulados dentro de um `Option` :

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

O que o método `unwrap` faz aqui é retornar o valor que supostamente está dentro de `Some(valor)` . O erro que vimos acima ocorre porque chamamos o `unwrap` para "desempacotar" um pacote que está vazio. Daí o `called Option::unwrap() on a None value` .



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Como pode perceber, já estávamos lidando com um `Option` em nosso código, e chamamos o método `unwrap` em uma operação que sabíamos que retornaria um `Option`. Tínhamos dois itens em nossa coleção, e com duas linhas de `agenda_iter.next().unwrap()` desempacotamos ambos. Ao chamar uma terceira vez, tomamos um erro, por termos apenas dois elementos.

Vamos percorrer nossa coleção de outra forma, dessa vez com um `loop` e o iterador. Vamos verificar se foi retornado um `Some`, quando temos um contato na agenda, ou um `None`, se acabaram nossos itens.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];
    let mut agenda_iter = agenda.iter();

    loop {
        let item = agenda_iter.next();

        match item {
            None => break,
            Some(contact) => println!("{:?}", contact),
        }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    }
}
```

Agora a cada iteração em nossa agenda verificamos se o resultado foi um `None` ou um `Some`. Caso seja um `None`, não há mais itens para iterar e o `loop` é encerrado.

Uma outra forma de escrever este código sem utilizar o `match` é com a construção `if let`, que pode ser mais legível em alguns cenários.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];
    let mut agenda_iter = agenda.iter();

    loop {
        let item = agenda_iter.next();

        if let Some(contact) = item {
            println!("{:?}", contact);
        } else {
            break;
        }
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## Map, blocos e collect

Um dos recursos dos iteradores é o uso de *mapping*. Em programação, um `map` pega uma coleção de objetos e executa um bloco de código para cada um de seus elementos, retornando uma nova coleção com o resultado de cada operação.

A sintaxe do `map` em Rust é apresentada a seguir.

```
meu_vetor.iter().map(|elemento| { bloco a ser executado })
```

Pegamos o iterador de `meu_vetor` por meio do método `iter()` e chamamos o método `map` a partir dele. O `map` recebe um bloco de código, no qual nomeamos o item usado atualmente como `elemento` e passamos um trecho de código a ser executado para cada `elemento`.

Outro método interessante é o `collect()`, que coleta o resultado da execução do `map` e gera um novo vetor, utilizando `generics`. No exemplo da agenda, tínhamos dois campos em nossa estrutura, `name` e `phone_number`. Usando os métodos `map` e `collect`, podemos facilmente gerar novos vetores apenas com os nomes ou os telefones, como no exemplo a seguir.

Perceba que é importante utilizar o `collect()`, pois é ele quem efetivamente vai pegar o retorno dos itens criados pelo `map()` e atribuí-lo ao novo vetor.

```
#[derive(Debug)]
struct Contact {
    name: &'static str,
    phone_number: &'static str
}

fn main() {
    let contact_1 = Contact {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
        name: "Marcelo",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Willian",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    let names = agenda.iter()
        .map(|contact| { contact.name })
        .collect::<Vec<_>>();

    println!("Nomes: {:?}", names);

    let phone_numbers = agenda.iter()
        .map(|contact| { contact.phone_number })
        .collect::<Vec<_>>();

    println!("Telefones: {:?}", phone_numbers);
}
```

```
$ cargo run --quiet
Nomes: ["Marcelo", "Willian"]
Telefones: ["+55(11) 9.1234.5678", "+55(11) 9.8765.4321"]
```

Vamos analisar o que fizemos. Para cada membro de nosso vetor, pegamos o nome por meio de `contact.name` e o jogamos em um novo vetor, pelo `collect`.

O `collect` sabe que tem de agrupar os retornos de `map` em um novo vetor, pois passamos o tipo `Vec` na chamada do método. Dizemos que o nosso vetor será uma coleção de quaisquer dados de um mesmo tipo quando usamos o sinal `_`.

```
let names = agenda.iter()
    .map(|contact| { contact.name })
    .collect::<Vec<_>>();
```

Fizemos o mesmo para o número do telefone:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
let phone_numbers = agenda.iter()
    .map(|contact| { contact.phone_number })
    .collect::<Vec<_>>();
```

Perceba que o bloco passado ao `map` pode ser simplesmente o retorno de um membro da estrutura ou qualquer código válido em Rust. Isso possibilita um processamento de cada elemento de sua coleção, assim temos como resultado uma nova coleção com esses dados. As possibilidades são infinitas.

Por exemplo, imagine que nossa agenda tem mais um campo, a data de nascimento dos contatos. Pelo `map`, poderíamos calcular a idade de cada membro, ou então a média das idades. Seria possível verificar quais nasceram neste século ou no século passado, entre outros recursos.

## Ordenação de vetores

Vetores permitem a ordenação de elementos por meio da implementação do trait `Ord` e do método `sort()`. No capítulo *Traits e estruturas*, vimos como implementar o `Ord`. Vamos modificar nosso código, criando quatro contatos e ordenando-os por nome. Para isso, vamos implementar a comparação de nomes para ordenação.

Chamaremos o método `sort()` para reordenar alfabeticamente o vetor, utilizando o campo `name`. É importante ressaltar que o vetor deve ser mutável para possibilitar essa reordenação.

```
use std::cmp::Ordering;

#[derive(PartialOrd, PartialEq, Eq, Clone, Copy, Debug)]
struct Contact {
    name: &'static str,
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    phone_number: &'static str
}

impl Ord for Contact {
    fn cmp(&self, other: &Contact) -> Ordering {
        (self.name).cmp(&(other.name))
    }
}

fn main() {
    let c1 = Contact {
        name: "Kwame",
        phone_number: "+55(11) 9.1234.5678"
    };

    let c2 = Contact {
        name: "Gi",
        phone_number: "+55(11) 9.8765.4321"
    };

    let c3 = Contact {
        name: "Joey Wheeler",
        phone_number: "+55(11) 9.9999.8888"
    };

    let c4 = Contact {
        name: "Linka",
        phone_number: "+55(11) 9.4567.7654"
    };

    let c5 = Contact {
        name: "Ma-Ti",
        phone_number: "+55(11) 9.4567.7654"
    };

    let mut agenda = vec![c1, c2, c3, c4, c5];

    agenda.sort();

    println!("{:?}", agenda);
}
```

O resultado da execução será o vetor ordenado por nome:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
[Contact { name: "Gi", phone_number: "+55(11) 9.8765.4321" },
 Contact { name: "Joey Wheeler", phone_number: "+55(11) 9.9999.88
88" },
 Contact { name: "Kwame", phone_number: "+55(11) 9.1234.5678" },
 Contact { name: "Linka", phone_number: "+55(11) 9.4567.7654" },
 Contact { name: "Ma-Ti", phone_number: "+55(11) 9.4567.7654" }]
```

## First e last

Para finalizar, não poderíamos deixar de falar de outros dois métodos bastante úteis, o `first()` e o `last()`. Eles apresentam o primeiro e o último membro de um vetor, respectivamente.

```
fn main() {
    let numbers = vec![134, 12, 2, 45, 6];

    println!("{:?}", numbers.first());
    println!("{:?}", numbers.last());
}
```

```
$ cargo run --quiet
Some(134)
Some(6)
```

O `first` pode ser facilmente representado por `numbers[0]`, e o `last` poderia ser substituído por `numbers[numbers.len() - 1]`, o que é um tanto quanto verboso. Portanto, `last` é uma maneira mais simples e concisa de acessarmos o último elemento sem nos preocupar com o tamanho do vetor.

Os vetores e os seus iteradores possibilitam a criação de poderosas coleções que podem ser manipuladas livremente, obtendo resultados complexos com o uso do `map` e do `collect`. Os recursos aqui apresentados cobrem apenas a parte mais básica do uso de vetores. Para mais detalhes, consulte a documentação em: https://doc.rust-lang.org/std/vec/.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## Conclusão

Neste capítulo, falamos sobre como construir coleções de dados e usar iteradores para percorrer seus valores.

O uso de vetores facilita muito o dia a dia de um desenvolvedor, pois permite agrupar dados em coleções além de possuir métodos poderosos para a manipulação dos seus dados.

No próximo capítulo, falaremos sobre string, um dos tipos de dados mais usados em qualquer linguagem de programação.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

CAPÍTULO 6

# STRINGS E SLICES

Sequências de caracteres (ou strings) estão entre os recursos mais populares das linguagens de programação existentes. Trabalhar com uma cadeia de caracteres – seja ela o nome de uma pessoa, seu CPF ou um bloco de um protocolo de comunicação – é parte fundamental de qualquer linguagem que deseja ser considerada para aplicações de grande porte.

Há várias formas de se criar uma string. As mais comuns são: utilizar o método `to_string()` disponível no tipo básico `str`; explicitar o tipo `String` no seu `let` junto ao método `into()`; e parsear uma constante com o construtor `from()`. Vamos ver exemplos dos três métodos:

```
fn main() {
    let editora = "Casa do Código".to_string();
    let autor = String::from("Marcelo Castellani e Willian Molina
ri");
    let livro: String = "A linguagem Rust".into();

    println!("Este é o livro {} da {},\nescrito por {}",
             livro, editora, autor);
}
```

```
$ cargo run --quiet
Este é o livro A linguagem Rust da Casa do Código,
escrito por Marcelo Castellani e Willian Molinari
```

Também é possível criar uma instância de `String` a partir de



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

um vetor de bytes, pelo método `from_utf8()`, como no exemplo a seguir. Repare no uso do `unwrap()`, visto que o `from_utf8()` retorna um `Result`.

```
fn main() {
    let vec = vec![82, 117, 115, 116];
    let a = String::from_utf8(vec).unwrap();

    println!("{}", a);
}
```

```
$ cargo run --quiet
Rust
```

Strings em Rust possuem poderosos mecanismos de concatenação por meio do sinal de `+` (mais) ou do método `push()`. O uso do sinal `+` espera que o valor a ser concatenado seja uma string estática, obtida a partir de uma instância de `String` usando o sinal `&`. Veja a seguir:

```
fn main() {
    let publisher = "Casa do Código".to_string();
    let authors = String::from("Marcelo Castellani e Willian Moli
nari");
    let book: String = "A linguagem Rust".into();

    let mut sentence = String::from("Este é o livro ");
    sentence += &book;
    sentence += " da ";
    sentence += &publisher;
    sentence += ",\nescrito por ";
    sentence += &authors;

    println!("{}", sentence);
}
```

```
$ cargo run --quiet
Este é o livro A linguagem Rust da Casa do Código,
escrito por Marcelo Castellani e Willian Molinari
```

Podemos ainda utilizar o `push_str()`, de funcionamento



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

semelhante ao `+`. Veja o mesmo exemplo, agora apresentado com o `push_str()` no lugar do `+`.

```
fn main() {
    let publisher = "Casa do Código".to_string();
    let authors = String::from("Marcelo Castellani e Willian Moli
nari");
    let book: String = "A linguagem Rust".into();

    let mut sentence = String::from("Este é o livro ");
    sentence.push_str(&book);
    sentence.push_str(" da ");
    sentence.push_str(&publisher);
    sentence.push_str(",\nescrito por ");
    sentence.push_str(&authors);

    println!("{}", sentence);
}
```

Além do `push_str()`, é possível anexar instâncias de `char` usando o `push()`, como vemos a seguir.

```
fn main() {
    let mut sentence = String::from("Oi ");
    sentence.push('R');
    sentence.push('u');
    sentence.push('s');
    sentence.push('t');
    sentence.push('!');

    println!("{}", sentence);
}
```

```
$ cargo run --quiet
Oi Rust!
```

Uma string ocupará o espaço necessário para o armazenamento dos dados que foram designados para ela alocar. Você pode verificar seu tamanho com o método `capacity()`. Também é possível alocar esse espaço sem definir uma sentença, utilizando o método `with_capacity()`. Isso reservará memória



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

para um uso posterior, o que é muito útil em sistemas embarcados.

No exemplo seguinte, alocamos as strings `a` (com tamanho 255) e `b`, a partir de uma constante com três caracteres. A alocação de `a` mostra como reservar memória para uma `String` que será populada posteriormente.

```
fn main() {
    let a = String::with_capacity(255);
    let b = String::from("ABC");

    println!("a: {} -> {}", a.capacity(), a);
    println!("b: {} -> {}", b.capacity(), b);
}
```

```
$ cargo run --quiet
a: 255 ->
b: 3 -> ABC
```

Vale ressaltar que `capacity()` retorna o quanto podemos colocar naquela instância, mas o tamanho alocado é obtido através do método `len()`.

```
fn main() {
    let a = String::with_capacity(255);

    println!("a: {}", a.capacity());
    println!("a: {}", a.len());
}
```

```
$ cargo run --quiet
a: 255
a: 0
```

Também é possível alocar espaço adicional para uma instância mutável já inicializada, por meio do `reserve()`. Ele alocará espaço de memória adicional para a instância criada, concatenando-o.

```
fn main() {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let mut a = String::from("Marcelo");
    a.reserve(20);

    println!("a: {}", a.capacity());
    println!("a: {}", a.len());
}
```

```
$ cargo run --quiet
a: 27
a: 7
```

Você pode remover o espaço adicional de uma instância de `String` com o método `shrink_to_fit()`. Ele redimensionará a instância para o tamanho exato do seu conteúdo.

```
fn main() {
    let mut a = String::from("Rust");
    a.reserve(10);

    println!("capacity sem shrink: {}", a.capacity());
    println!("len sem shrink: {}", a.len());

    a.shrink_to_fit();

    println!("capacity com shrink: {}", a.capacity());
    println!("len com shrink: {}", a.len());
}
```

```
$ cargo run --quiet
capacity sem shrink: 14
len sem shrink: 4
capacity com shrink: 4
len com shrink: 4
```

Rust provê o método `truncate()`, que possibilita truncar os dados de sua instância sem afetar a capacidade de armazenamento. Também existe o método `clear()`, que limpa a sua `String`, mas não afeta a sua capacidade.

No exemplo a seguir, criamos uma instância e adicionamos mais cinquenta caracteres de espaço de armazenamento.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Adicionamos a ele uma segunda sentença, e ajustamos nossa instância para o tamanho dela. Depois, truncamos a sentença para 4 caracteres, novamente ajustamos o espaço, e a limpamos. Veja a evolução da capacidade de armazenamento a cada operação.

```
fn main() {
    let mut a = String::from("Rust");
    a.reserve(50);

    a.push_str(" rules");
    println!("a: {} -> {}", a.capacity(), a);

    a.shrink_to_fit();
    println!("a: {} -> {}", a.capacity(), a);

    a.truncate(4);
    println!("a: {} -> {}", a.capacity(), a);

    a.shrink_to_fit();
    println!("a: {} -> {}", a.capacity(), a);

    a.clear();
    println!("a: {} -> {}", a.capacity(), a);
}
```

```
$ cargo run --quiet
a: 54 -> Rust rules
a: 10 -> Rust rules
a: 10 -> Rust
a: 4 -> Rust
a: 4 ->
```

Outra forma de remover caracteres de sua `String` é pelo método `remove()`. Ele possibilita a remoção de qualquer caractere de uma sentença, através do índice.

```
fn main() {
    let mut a = String::from("Rust");

    println!("Antes da remoção: {}", a);

    a.remove(2);
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Depois da remoção: {}", a);
}
```

```
$ cargo run --quiet
Antes da remoção: Rust
Depois da remoção: Rut
```

Pode-se também remover caracteres de uma sentença por meio do método `pop()`. Ele retorna um `Option`, sendo que o `Some` representa o caractere removido; caso contrário, retorna `None`.

```
fn main() {
    let mut a = String::from("Rust");
    for _x in 0..a.len() {
        let ret = a.pop();
        match ret {
            Some(char) => println!("Pop -> {}", char),
            None => println!("Sem mais caracteres..."),
        }
    }
}
```

```
$ cargo run --quiet
Pop -> t
Pop -> s
Pop -> u
Pop -> R
```

## Chars e bytes

A `String` possui duas representações que possibilitam iterar sobre os membros da sentença: `chars()` e `bytes()`. Ambas retornam uma coleção de dados que pode facilmente ser iterada com um loop `for`.

```
fn main() {
    let a = String::from("Rust");

    for chr in a.chars() {
        println!("Char: {}", chr)
    }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    for bte in a.bytes() {
        println!("Byte: {}", bte)
    }
}
```

```
$ cargo run --quiet
Char: R
Char: u
Char: s
Char: t
Byte: 82
Byte: 117
Byte: 115
Byte: 116
```

Como vimos, é possível iterar em cada um dos caracteres de uma `String` através das coleções `Chars` e `Bytes`. Já com o método `enumerate`, podemos iterar sobre a coleção utilizando seu índice. Veja um exemplo:

```
fn main () {
    let a = String::from("Rust");

    for (index, charac) in a.chars().enumerate() {
        println!("{} -> {}", index, charac);
    }
}
```

```
$ cargo run --quiet
0 -> R
1 -> u
2 -> s
3 -> t
```

Iterar sobre coleções de caracteres é particularmente útil para o tratamento de protocolos de comunicação. Converter uma `String` para um array de bytes permite que protocolos complexos sejam desenvolvidos e parseados, byte a byte ou caractere a caractere.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## Casting de String

Existem situações em que precisamos converter uma `String` para outros tipos de dados – transformar a `String` "42" no inteiro 42, ou a `String` "C" no caractere C, por exemplo. É possível converter strings em instâncias de outros tipos com o método `parse`.

No exemplo a seguir, vamos converter uma instância de `String` com o número `42` para um `i8`. Neste caso, o `parse` devolve uma instância de `Result`, portanto, precisamos verificar se a nossa conversão ocorreu sem problemas. Se tudo correr bem, somamos 1 ao valor parseado; caso contrário, retornamos zero. Veja uma conversão de sucesso:

```
fn main() {
    let a = String::from("42");

    let b = match a.parse::<i8>() {
        Ok(c) => c + 1,
        Err(_d) => 0,
    };
    println!("{} + 1 = {}", a, b);
}
```

```
$ cargo run --quiet
42 + 1 = 43
```

Caso a `String` a ser parseada não seja convertida para o formato definido, recebemos um `Err`, que determinamos como retorno `0`. Veja a seguir:

```
fn main() {
    let a = String::from("ERRO");

    let b = match a.parse::<i8>() {
        Ok(c) => c + 1,
        Err(_d) => 0,
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    };
    println!("{} + 1 = {}", a, b);
}
```

```
$ cargo run --quiet
ERRO + 1 = 0
```

## String ou str?

Em Rust, um dos pontos de confusão quando falamos em sequências de caracteres é a existência de dois tipos *built-in* na linguagem: `String` e `str`. Apesar de ambos manipularem o mesmo tipo de dados, os dois são tipos diferentes, e seu uso requer muita atenção. Veja o exemplo a seguir, que parece ser válido, mas não compila.

```
fn say_my_name(name: String) {
    println!("Seu nome é {}", name);
}

fn main() {
    let name = "Heisenberg";
    say_my_name(name);
}
```

```
$ cargo run --quiet
error[E0308]: mismatched types
 --> src/main.rs:7:17
  |
7 |     say_my_name(name);
  |                 ^^^^
  |                 |
  |                 expected struct `String`, found `&str`
  |                 help: try using a conversion method: `name.to
_string()`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `parse`
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
To learn more, run the command again with --verbose.
```

Isso ocorre porque usamos o tipo literal `str`, e não uma instância de `String` – que era o esperado pelo método. Pode parecer que `"Heisenberg"` é uma `String` válida, mas não é. Podemos facilmente resolver o problema convertendo o `str` em `String` com o método `to_string()`.

```
fn say_my_name(name: String) {
    println!("Seu nome é {}", name);
}

fn main() {
    let name = "Heisenberg".to_string();
    say_my_name(name);
    // You're Goddamn Right.
}
```

Uma `String` é um tipo de dados **dinâmico** alocado no *heap* (que veremos mais a fundo no capítulo sobre memória). Já uma `str` é uma sequência **estática** de bytes UTF-8 em algum lugar na memória. Ela sempre é acessada por empréstimo através de uma espécie de ponteiro, ou seja, normalmente você a encontrará sendo acessada como `&str`, o que podemos chamar de uma referência a um grupo de bytes UTF-8 na memória. Chamamos isso de `slice`, ou de `string slice`.

Um `slice` é uma visualização de algum dado e pode ser acessado de qualquer lugar dentro do contexto onde é definido. Sabendo disso podemos trabalhar diretamente com uma referência. Para isso, modificamos a função `say_my_name` para receber uma string estática, e não uma instância de `String`.

```
fn say_my_name(name: &'static str) {
    println!("Seu nome é {}", name);
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    let name = "Heisenberg";
    say_my_name(name);
}
```

Perceba que usamos o sinal de `&`, indicando ao Rust que estamos *pedindo emprestado* a variável `name` (passada como parâmetro) ao escopo que chama o método `say_my_name`. Isso chama-se **borrowing**, e é como Rust lida com variáveis de fora do escopo. Também temos outro símbolo no código, o `'` (apóstrofo). Ele indica ao Rust que devemos criar um **lifetime**, ou seja, que a variável emprestada deve existir dentro do nosso escopo atual, vinda de um escopo externo.

Repare que ainda estamos trabalhando com uma instância de `str`, que não apresenta todos os métodos e todo o poder que temos em `String`. Por isso, recomendamos o uso de `to_string()` e de `String` sempre que possível. Claro, se você estiver utilizando Rust para sistemas embarcados ou qualquer outro contexto onde a memória é limitada, o ideal é empregar instâncias de `str`. Elas são bem mais leves, porém mais limitadas.

## Buscando e dividindo

Um recurso bastante útil de string são os métodos para buscar e dividir sua sequência de caracteres. O `find` permite encontrar um caractere em uma string facilmente, retornando a sua posição.

A seguir, definimos uma `String` e utilizamos o `find` para localizar um espaço em branco. Usamos o método `unwrap_or` para evitar a necessidade de tratarmos um `Option`. Ou seja, se ele achar um espaço em branco, considerará o resultado em `Some`; caso contrário, levará em conta o valor entre parênteses – no caso,


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

`0`.

```
fn main() {
    let name = String::from("Luke Skywalker");
    let space = name.find(" ").unwrap_or(0);

    println!("Espaço na posição {}", space);
}
```

```
$ cargo run --quiet
Espaço na posição 4
```

Você pode pensar: "mas o quarto caractere é um "e", não um espaço em branco". Na verdade, strings em Rust iniciam como o índice `0`, sendo assim:

```
Luke Skywalker
01234
```

O espaço está na posição `4`. Agora que sabemos onde está o caractere que buscamos, que tal utilizá-lo para dividir a `String`?

Rust possui um método chamado `drain`, que recebe como parâmetro um intervalo. Este indica o que deve ser removido da `String` original. Imagine que você deseja pegar o primeiro nome, no caso *Luke*, em nossa `String`. Podemos utilizar o `drain` para removê-lo da `String` original, passando um intervalo que vai do começo até o primeiro espaço da `String` – já localizado graças ao `find` e à nossa variável `space`. Isso resultará no código `0..space`.

O `drain` é esperto o suficiente para que você não precise informar o `0` como início da `String`. Você poderá utilizar `..space` para pegar do começo até o espaço, ou mesmo `space..` para partir do espaço até o final da `String`.

Vamos jogar o resultado da execução do `drain` em uma nova



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

variável, a `first_name`. Para fazê-lo, usamos o `collect`, responsável por pegar o resultado de um iterador e transformar em uma coleção, que no nosso exemplo é uma `String`. Nosso código ficará assim:

```
fn main() {
    let mut name = String::from("Luke Skywalker");
    let space = name.find(" ").unwrap_or(0);

    let first_name: String = name.drain(..space).collect();

    println!("Primeiro nome: {}", first_name);
}
```

```
$ cargo run --quiet
Primeiro nome: Luke
```

Repare que adicionamos um `mut` em nossa `String`. Isso ocorre porque o `drain` vai remover a parte que corresponde ao intervalo passado de nossa `String` original. Faça um teste e altere o código para imprimir o conteúdo original de `name`, como a seguir. Veja o resultado:

```
fn main() {
    let mut name = String::from("Luke Skywalker");
    let space = name.find(" ").unwrap_or(0);

    let first_name: String = name.drain(..space).collect();

    println!("Primeiro nome: {}", first_name);
    println!("Nome original: {}", name);
}
```

```
$ cargo run --quiet
Primeiro nome: Luke
Nome original:  Skywalker
```

Esse comportamento de modificar a `String` original indica que você precisa ter cuidado caso deseje executar o `drain` mais de uma vez na mesma `String`, visto que ela será modificada. Veja


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

a seguir um exemplo no qual pego o primeiro nome e o sobrenome a partir da `String`. Nele, modifiquei a variável `space` para ser mutável, pois precisarei chamar duas vezes o método `find`. Também adicionei uma verificação para ver se o nome foi totalmente limpo, por meio do método `is_empty`.

```
fn main() {
    let mut name = String::from("Luke Skywalker");
    let mut space = name.find(" ").unwrap_or(0);

    let first_name: String = name.drain(..space).collect();

    space = name.find(" ").unwrap_or(0);
    let last_name: String = name.drain(space..).collect();

    println!("Primeiro nome: {}", first_name);
    println!("Último nome: {}", last_name);

    println!("A string original está vazia? {}", name.is_empty())
;
}
```

```
$ cargo run --quiet
Primeiro nome: Luke
Último nome:  Skywalker
A string original está vazia? true
```

## Conclusão

Neste capítulo falamos de strings, um dos tipos de dados mais usados em qualquer linguagem de programação. Em Rust, strings são tipos complexos, com inúmeros métodos prontos para serem usados e facilitar o nosso dia a dia. A seguir vamos falar sobre tipos de dados genéricos.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 7

# TIPOS DE DADOS GENÉRICOS

*Generics* (ou Programação Genérica) é uma forma de escrever código que pode ser reutilizado sem que o tipo dos dados informados seja explicitado em sua declaração. Isso quer dizer que um código pode receber parâmetros de um tipo X ou Y, e eles serão entendidos e respeitados como tais.

A vantagem do uso de programação genérica é que podemos criar assinaturas de métodos e estruturas, que podem ser utilizados em situações diversas, com diferentes tipos de dados, possibilitando um melhor aproveitamento de código evitando repetição.

Um exemplo é a declaração de um método genérico de soma, no qual podemos passar dois parâmetros, informar seu tipo e realizar a soma de ambos. Sem o uso da programação genérica, seria necessário escrever um método para cada um dos tipos Rust:

```
fn sum_i32(x: i32, y: i32) -> i32 {
    x + y
}

fn sum_f32(x: f32, y: f32) -> f32 {
    x + y
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    println!("{}", sum_i32(1, 2));
    println!("{}", sum_f32(1.3, 2.24));
}
```

```
$ cargo run --quiet
3
3.54
```

Perceba que a lógica é a mesma nos dois métodos, `sum_i32` e `sum_f32`. Apesar de a lógica ser a mesma, a assinatura do método é diferente. Em um caso, usamos o tipo `i32` como parâmetros e retorno, já no outro, usamos o tipo `f32`.

## Traits genéricos

Podemos utilizar programação genérica para criarmos traits que podem ser utilizados em diversas situações, como as coordenadas cartesianas, e que receba tanto valores inteiros como de ponto flutuante. Essa definição de tipo genérico é feita com o uso de `<T>`, em que `T` indica um tipo qualquer.

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

fn main() {
    let point_int = Point { x: 10, y: 12 };
    let point_flt = Point { x: 1.4, y: 12.3 };

    println!("{:?}", point_int);
    println!("{:?}", point_flt);
}
```

```
$ cargo run --quiet
Point { x: 10, y: 12 }
Point { x: 1.4, y: 12.3 }
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

No `point_int` declaramos uma instância de `Point` que recebe dois valores inteiros, e em `point_flt` declaramos uma instância de `Point` que recebe dois valores de ponto flutuante. Perceba que ambos os valores devem ter o mesmo tipo, ou então teremos um erro de compilação:

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

fn main() {
    let point_int = Point { x: 10, y: 12.0 };

    println!("{:?}", point_int);
}
```

```
$ cargo run --quiet
error[E0308]: mismatched types
 --> src/main.rs:8:39
  |
8 |     let point_int = Point { x: 10, y: 12.0 };
  |                                       ^^^^ expected integer,
found floating-point number
```

Para isso devemos informar que nosso trait vai receber não um, mas dois tipos de dados quaisquer quando for instanciado.

```
#[derive(Debug)]
struct Point<T1, T2> {
    x: T1,
    y: T2,
}

fn main() {
    let point_mixed_1 = Point { x: 10, y: 12.0 };
    let point_mixed_2 = Point { x: 10.0, y: 12 };
    let point_int = Point { x: 10, y: 12 };
    let point_float = Point { x: 10.0, y: 12.0 };

    println!("{:?}", point_mixed_1);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("{:?}", point_mixed_2);
    println!("{:?}", point_int);
    println!("{:?}", point_float);
}
```

Dessa forma podemos tratar combinações como dois inteiros, dois valores de ponto flutuante ou qualquer combinação entre os dois.

```
$ cargo run --quiet
Point { x: 10, y: 12.0 }
Point { x: 10.0, y: 12 }
Point { x: 10, y: 12 }
Point { x: 10.0, y: 12.0 }
```

Repare, porém, que nossa estrutura `Point` é realmente genérica e aceita qualquer tipo. Por exemplo, podemos criar um `Point` passando o nome dos dois maravilhosos autores deste livro, que talvez não seja o esperado (tanto os autores serem maravilhosos quanto receber strings no struct).

```
#[derive(Debug)]
struct Point<T1, T2> {
    x: T1,
    y: T2,
}

fn main() {
    let point = Point { x: "Marcelo", y: "Willian" };

    println!("{:?}", point);
}
```

```
$ cargo run --quiet
Point { x: "Marcelo", y: "Willian" }
```

## Funções genéricas

Ao utilizar Programação Genérica podemos reescrever nosso exemplo de método de soma de uma forma a não precisarmos de



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

dois métodos diferentes apenas para somar valores.

```
fn generic_sum<T>(x: T, y: T) -> T {
    x + y
}

fn main() {
    println!("{}", generic_sum::<i32>(1, 2));
    println!("{}", generic_sum::<i16>(3, 4));
    println!("{}", generic_sum::<i8>(5, 6));

    println!("{}", generic_sum::<f32>(1.3, 2.24));
}
```

```
cargo run --quiet
error[E0369]: cannot add `T` to `T`
 --> src/main.rs:2:7
  |
2 |     x + y
  |     - ^ - T
  |     |
  |     T
  |
help: consider restricting type parameter `T`
  |
1 | fn generic_sum<T: std::ops::Add<Output = T>>(x: T, y: T) -> T
 {
  |                 ++++++++++++++++++++++++++
```

Veja que temos um erro ao definir um método genérico para somar, pois o sinal `+` é definido em `std::ops::Add`, e apenas quem o implementa pode ser somado. Ou, seja, você precisa limitar os tipos genéricos a serem aceitos por esse método, garantindo que todos os tipos que possam ser usados implementem `std::ops::Add`.

Para facilitar nossa vida neste caso usaremos um crate externo chamado `num`, disponível em: https://github.com/rust-num/num. Ele provê uma série de tipos numéricos e traits para trabalhar com programação genérica.


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Para adicionar o crate a seu projeto, edite o arquivo `Cargo.toml` e adicione a dependência ao crate `num`, como a seguir.

```
[package]
name = "my_generic_sum"
version = "0.0.1"
authors = ["Marcelo Castellani <marcelo@linux.com>", "PotHix <pot
hix@pothix.com>"]

[dependencies]
num = "0.3"
```

Com o `num`, podemos criar um método `generic_sum` que possibilita somar um par de qualquer tipo numérico.

Em nosso caso, `T` será qualquer um dos tipos numéricos de `num`. O uso de `T` é opcional. A constante que representa o tipo a ser utilizado em nosso método pode ser qualquer uma, mas `T` é uma convenção da comunidade Rust.

Além de o método receber dois parâmetros, como vimos anteriormente, precisamos passar também o tipo genérico `T`. Nossa declaração ficará assim:

```
fn generic_sum<T: num::Num>(x: T, y: T) -> T {
```

No código, `<T: num::Num>` indica que o método `generic_sum` poderá receber qualquer tipo válido dentro de `num::Num`. Esse tipo `T` será o mesmo dos parâmetros `x` e `y` em `(x: T, y: T)`, bem como do retorno do método em `-> T`.

Isso possibilita chamarmos nosso método para um inteiro de 32 bits `i32`, um número de ponto flutuante de 32 bits `f32` ou suas variantes com menor quantidade de bits, como vemos a seguir.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn generic_sum<T: num::Num>(x: T, y: T) -> T {
    x + y
}

fn main() {
    println!("{}", generic_sum::<i32>(1, 2));
    println!("{}", generic_sum::<i16>(3, 4));
    println!("{}", generic_sum::<i8>(5, 6));

    println!("{}", generic_sum::<f32>(1.3, 2.24));
}
```

```
$ cargo run --quiet
3
7
11
3.54
```

## Conclusão

Programação genérica nos dá a capacidade de reaproveitar nosso código de uma forma inteligente, sem a necessidade de repetição. A programação genérica é encontrada em diversos traits padrão do Rust, bem como em diversos crates de terceiros.

No próximo capítulo, vamos ver como o Rust gerencia a memória por debaixo dos panos.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 8

# ALOCAÇÃO E GERENCIAMENTO DE MEMÓRIA

Neste capítulo, vamos falar sobre como Rust gerencia memória. Esse é um ponto importante, pois a linguagem faz um misto de coletor de lixo e gerenciamento manual de memória, que pode confundir quem programa em outras linguagens.

Quem veio de linguagens como Ruby, Python ou Java já se acostumou a usar instâncias de classes e não se preocupar em liberar espaço quando elas deixam de ser necessárias. Já quem programa em C e C++ sofre com o uso de alocação e desalocação de memória.

## 8.1 GERENCIAMENTO DE MEMÓRIA

Este é um dos aspectos mais interessantes de Rust. Hoje em dia, a maior parte das linguagens de programação permite um dos dois extremos: você gerencia sua memória ou ela gerencia tudo para você. Rust está no meio-termo.

Rust não possui um coletor de lixo como muitas outras. O



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

gerenciamento é feito de maneira eficiente por meio de um maior controle de como a memória vai ser usada, assim a linguagem sabe o momento exato para a desalocação do recurso.

Uma das estratégias bem conhecidas para implementação dos coletores de lixo usa o conceito de contagem de referências a uma instância. Isso quer dizer que um coletor não se preocupa com o contexto no qual o objeto será usado, mas se existe alguma referência a ele ainda no sistema.

Esse modelo funciona bem em projetos em que não é necessário um controle fino de memória. Porém, quando o projeto tem o uso de memória como um fator crucial (como serviços rodando em servidores com restrições ou software embarcado), uma gestão eficiente de memória torna-se um fator crucial.

## 8.2 ESCOPO DE VARIÁVEIS

Quando falamos em escopo, falamos sobre o espaço onde algo estará disponível para ser acessado. Veja o código a seguir.

```
fn magic_number(b: i32) -> i32 {
    let c: i32 = 123;
    b + c
}

fn main() {
    let a: i32 = 2048;
    println!("{}", magic_number(a));
}
```

Temos dois escopos nesse código, `main` e `magic_number`. Você pode interpretar `main` como o escopo que vai existir durante toda a execução de seu código, visto que é dentro desse método que as coisas acontecem. E quando `main` finaliza, o


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

programa se encerra.

Já o escopo `magic_number` existe apenas quando o método é chamado, e isso quer dizer que as variáveis definidas nele existem apenas nesse momento. Se você tentar acessar a variável `c` dentro de `main`, ela não estará disponível, já que faz parte de outro escopo. O código a seguir, por exemplo, não compila.

```
fn magic_number(b: i32) -> i32 {
    let c: i32 = 123;
    b + c
}

fn main() {
    let a: i32 = 2048;
    println!("{}", magic_number(a));

    println!("{}", c);
}
```

```
$ cargo run --quiet
error[E0425]: cannot find value `c` in this scope
  --> src/main.rs:10:20
   |
10 |     println!("{}", c);
   |                    ^ help: a local variable with a similar n
ame exists: `a`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0425
`.
error: could not compile `magic_number`

To learn more, run the command again with --verbose.
```

Por `c` pertencer ao escopo `magic_number`, na linha onde tentamos imprimi-lo, Rust já o eliminou da memória e não faz a menor ideia de quem ele seja.

Outro exemplo seria o uso de blocos dentro do mesmo



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

método. A memória alocada dentro de um bloco pertence apenas àquele bloco no qual foi criado, não existindo fora dele. Veja o código a seguir. Ele não compila, pois `b` não existe fora do bloco.

```
fn main() {
    let mut a: i32 = 2048;
    {
        let b = a + 1;
        println!("{}", b);
    }
    println!("{}", b);
}
```

```
$ cargo run --quiet
error[E0425]: cannot find value `b` in this scope
 --> src/main.rs:7:20
  |
7 |     println!("{}", b);
  |                    ^ help: a local variable with a similar na
me exists: `a`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0425
`.
error: could not compile `magic_number`

To learn more, run the command again with --verbose.
```

## 8.3 CASTING

Casting (ou conversão de valores) é uma maneira de tornar tipos incompatíveis em um tipo compatível com o que você precisa. O casting é interessante quando falamos de alocação de memória, porque ele possibilita a conversão de valores para tamanhos menores do que o original.

No exemplo a seguir, declaro um `i32` chamado `a` com valor 8. Isso é um exagero e um desperdício de memória, pois o 8 ocupa


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

bem menos bits do que o `i32`. Então, vamos pegar `a` e convertê-lo para um `i8` chamado `b`, utilizando o método `as`.

O método `as` realiza uma conversão segura (*safe casting*) do nosso valor, possibilitando uma conversão transparente.

```
fn main() {
    let a: i32 = 8;
    let b: i8 = a as i8;
    println!("a: {}", a);
    println!("b: {}", b);
}
```

```
$ cargo run --quiet
a: 8
b: 8
```

É importante ressaltar que convertemos um valor definido em um inteiro de 32 bits para um valor definido em um inteiro de 8 bits. Como o número 8 cabe em nosso inteiro de 8 bits, o resultado da conversão é o mesmo número, ocupando menos espaço de memória.

```
i32: 00000000000000000000000000001000
i8 : 00001000
```

Vamos pegar agora um número que não cabe dentro de um i8, por exemplo, 2055. Sua representação binária é:

```
2055 -> 0000100000000111
```

Modifique seu código como a seguir e veja o resultado.

```
fn main() {
    let a: i32 = 2055;
    let b: i16 = a as i16;
    let c: i8 = a as i8;
    println!("a: {}", a);
    println!("b: {}", b);
    println!("c: {}", c);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
}
```

```
$ cargo run --quiet
a: 2055
b: 2055
c: 7
```

Nosso `2055` virou `7`. Isso porque, durante a conversão, os bits excedentes foram desprezados. Veja a seguir.

```
2055 -> 0000100000000111
7    ->         00000111
```

Vale lembrar que estamos falando de casting aqui. O compilador não deixa você cometer esse erro de overflow na execução do seu código. O código a seguir não compila:

```
fn main() {
    let mut _c: i8 = 127 as i8;
    _c += 1;
}
```

O compilador nos alerta deste problema:

```
$ cargo run --quiet
error: this arithmetic operation will overflow
 --> src/main.rs:3:5
  |
3 |     _c += 1;
  |     ^^^^^^^ attempt to compute `i8::MAX + 1_i8`, which would
overflow
  |
  = note: `#[deny(arithmetic_overflow)]` on by default

error: aborting due to previous error

error: could not compile `arithmetic_overflow`

To learn more, run the command again with --verbose.
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 8.4 PONTEIROS, BOX E DROP

Por padrão, tipos primitivos que têm seu tamanho conhecido em tempo de compilação em Rust são alocados no bloco de memória conhecido como **stack** ou **pilha**. Esse bloco "empilha" os valores na sequência em que eles são alocados, possibilitando que sejam desalocados quando a pilha estiver cheia, para dar lugar a outros valores.

Uma pilha usa o conceito de **LIFO**, ou seja, *Last In, First Out* (o último a chegar é o primeiro a sair). Valores são empilhados conforme usados, e removidos quando não são mais necessários em um efeito cascata.

Vejamos um exemplo simples:

```
struct RaceCar {
    number: i32,
}

fn main() {
    let car_a = RaceCar { number: 3 };
}
```

Nesse exemplo, temos uma variável chamada `car_a`. Se olhássemos nossa pilha, ela seria algo como:

```
|       |
| car_a |
|-------|
| Pilha |
```

Vamos adicionar mais um método em nosso código, que instancia duas variáveis:

```
struct RaceCar {
    number: i32,
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn two_cars() {
    let car_b = RaceCar { number: 12 };
    let car_c = RaceCar { number: 8 };
}

fn main() {
    let car_a = RaceCar { number: 3 };
    two_cars();
}
```

Agora vamos simular a execução do nosso código e ver como a pilha se comporta. Ao iniciarmos a execução do projeto, a primeira linha a ser chamada é a definição da função `main()`.

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
      let car_b = RaceCar { number: 12 };
      let car_c = RaceCar { number: 8 };
  }

> fn main() {
      let car_a = RaceCar { number: 3 };
      two_cars();
  }
```

Neste momento, nossa pilha está vazia.

```
|       |
|-------|
| Pilha |
```

Teremos então a execução da linha que define a variável `car_a`.

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
      let car_b = RaceCar { number: 12 };
      let car_c = RaceCar { number: 8 };
  }

  fn main() {
>     let car_a = RaceCar { number: 3 };
      two_cars();
  }
```

Ela passará a ocupar nossa pilha:

```
|       |
| car_a |
|-------|
| Pilha |
```

Agora, teremos a chamada da função que define duas variáveis, a `two_cars()`:

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
      let car_b = RaceCar { number: 12 };
      let car_c = RaceCar { number: 8 };
  }

  fn main() {
      let car_a = RaceCar { number: 3 };
>     two_cars();
  }
```

A primeira linha declara uma variável `car_b`, e a segunda, uma variável `car_c`. Veja como a pilha se comporta em cada um dos casos.

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
>     let car_b = RaceCar { number: 12 };
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
      let car_c = RaceCar { number: 8 };
  }

  fn main() {
      let car_a = RaceCar { number: 3 };
      two_cars();
  }
```

```
|       |
| car_b |
| car_a |
|-------|
| Pilha |
```

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
      let car_b = RaceCar { number: 12 };
>     let car_c = RaceCar { number: 8 };
  }

  fn main() {
      let car_a = RaceCar { number: 3 };
      two_cars();
  }
```

```
|       |
| car_c |
| car_b |
| car_a |
|-------|
| Pilha |
```

As variáveis entram na pilha na ordem em que são instanciadas. Isto é, `car_c`, que foi instanciada após `car_b`, passa ao topo da pilha, pois foi a última a ser instanciada.

Agora, vamos à próxima linha, o fechamento de chaves do nosso método `main`. Como saímos do escopo da função `two_cars`, suas variáveis definidas são descartadas.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
      let car_b = RaceCar { number: 12 };
      let car_c = RaceCar { number: 8 };
  }

  fn main() {
      let car_a = RaceCar { number: 3 };
      two_cars();
> }
```

```
|       |
| car_b |
| car_a |
|-------|
| Pilha |
```

E depois:

```
  struct RaceCar {
      number: i32,
  }

  fn two_cars() {
      let car_b = RaceCar { number: 12 };
      let car_c = RaceCar { number: 8 };
  }

  fn main() {
      let car_a = RaceCar { number: 3 };
      two_cars();
> }
```

```
|       |
| car_a |
|-------|
| Pilha |
```

Como você deve imaginar, após o fim da execução do programa, `car_a` também deixa de existir.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

A pilha possui um tamanho fixo, alocado no início da execução do seu código e, em alguns casos, ela pode estourar. Daí surgiu o nome do famoso site **Stack Overflow**, que literalmente quer dizer *estouro de pilha*. Isso ocorre quando alocamos algo muito grande. Para isso, existe um outro tipo de memória, chamado **heap**.

O uso do heap é mais lento do que o da pilha, mas permite que aloquemos mais memória do que temos disponível nela. Tipos como o `Vec`, usado para criar vetores, utilizam o heap para prover um tipo extensível em tamanho. Outra forma de se alocar algo no heap em Rust é usando o tipo `Box<T>`.

Vamos ver um exemplo de alocação no heap:

```
fn main() {
    let a = 10;
    let b = Box::new(10);
}
```

Ao alocar no heap, Rust também adiciona um valor na pilha, o que chamamos de ponteiro. Nossa pilha ficará assim:

```
|       |       |
|   b   | xxxxx |
|   a   |  10   |
|-------|-------|
| Pilha | Valor |
```

O valor `xxxxx` é um endereço de memória que aponta para onde nosso valor, apontado por `b`, se encontra. Isso quer dizer que, dentro da nossa memória, existem dois valores `b`: o ponteiro, que indica onde encontrar o valor; e o valor efetivamente.

Rust gerencia o heap automaticamente e com segurança. Em C, por exemplo, a gestão de valores do heap é feita por meio de chamadas de métodos para sua alocação e desalocação.


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Isso permite que você mate um valor e tente usá-lo posteriormente, gerando uma exceção do tipo *ponteiro nulo* (*NullPointerException* em Java, por exemplo). Como Rust cuida disso para você, nada sai do controle.

**EXCEÇÃO DE PONTEIRO NULO**

Uma exceção de ponteiro nulo ocorre quando você tenta acessar um objeto que não existe mais, mas cuja referência no código ainda está disponível. Imagine que você tem uma classe `Pessoa` e, a partir dela, cria uma instância referenciada por `minha_pessoa`, como em: `minha_pessoa = Pessoa.new`.

A referência `minha_pessoa` serve apenas para dizer em qual lugar da memória estão os dados dessa nova instância, `Pessoa`. Se por algum motivo a instância for destruída, `minha_pessoa` passará a referenciar algo nulo. Se você tentar chamar algum método dessa instância, terá uma exceção e seu programa morrerá.

Além disso, Rust possui um trait chamado `Drop`, que permite realizar ações ao fazer a limpeza da memória. Esse trait possui um método denominado `drop()`, chamado quando o valor é retirado da memória.

```
struct RaceCar {
    number: i32,
}

impl Drop for RaceCar {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    fn drop(&mut self) {
        println!("Carro {} terminou a corrida", self.number);
    }
}

fn main() {
    let car_a = RaceCar { number: 3 };
    let car_b = RaceCar { number: 5 };
    let car_c = RaceCar { number: 8 };
}
```

Ao executarmos nosso código, os tipos são desalocados do heap no esquema LIFO, pois eles estão referenciados em nossa pilha a partir de ponteiros.

```
$ cargo run --quiet
Carro 8 terminou a corrida
Carro 5 terminou a corrida
Carro 3 terminou a corrida
```

## 8.5 OWNERSHIP

Rust se tornou popular por sua gestão extremamente eficiente de memória, e isso se deve a dois conceitos bem importantes da linguagem, `ownership` e `borrowing`, que podemos traduzir como, `propriedade` e `empréstimo`, respectivamente.

Todos os dados alocados podem ter apenas uma referência, ou seja, um único dono. Veja o exemplo a seguir para entender melhor:

```
fn main() {
    let v = vec![1, 2, 3];
    let v2 = v;

    println!("{:?}", v);
}
```

Nós criamos um Vector que é referenciado por `v`, e depois


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

dizemos que esse vetor deve ser referenciado também por `v2`. O compilador do Rust sabe que isso pode dar um problema de concorrência, ou seja, temos duas formas de acessar o mesmo dado na memória, e quando a linha `let v2 = v` é executada `v` deixa de ser dono do Vector. Qualquer tentativa de usar `v` após essa linha dará um erro de compilação. Dizemos que neste momento `v2` passa a ter a propriedade do Vector, em vez de `v`.

```
error[E0382]: borrow of moved value: `v`
 --> src/main.rs:5:22
  |
2 |     let v = vec![1, 2, 3];
  |         - move occurs because `v` has type `Vec<i32>`, which
does not implement the `Copy` trait
3 |     let v2 = v;
  |              - value moved here
4 |
5 |     println!("{:?}", v);
  |                      ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args
_nl` (in Nightly builds, run with -Z macro-backtrace for more inf
o)
```

O compilador nos diz que tentamos emprestar um valor que foi movido, e que o nosso tipo `Vec<i32>` não implementa o trait `Copy`. Isso quer dizer que Rust não sabe como duplicar nosso Vector, e que ele não permite dois ponteiros apontando pra mesma coisa na memória, ou seja, não permite dois donos para o mesmo dado, então ele não compila esse código.

Caso o Vector implementasse o trait `Copy` Rust duplicaria nosso Vector na memória na linha `let v2 = v`, e assim `v` apontaria para um Vector e `v2` apontaria para outro Vector. Como ele não sabe, ele move a referência ao Vector de `v` para `v2`.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Isso pode causar um pouco de estranhamento ou confusão para quem está iniciando em Rust, pois o código a seguir compila e funciona:

```
fn main() {
    let v = 123;
    let v2 = v;

    println!("{:?}", v);
    println!("{:?}", v2);
}
```

```
$ cargo run --quiet
123
123
```

Isso ocorre porque o tipo `i32` implementa o `Copy`. Ou seja, tanto `v` quanto `v2` possuem o seu próprio `123`. Se eu modificar `v2` para `456`, por exemplo, `v` ainda valerá `123`. Para resolver a questão com o Vector podemos explicitamente dizer ao Rust para clonar o dado.

```
fn main() {
    let v = vec![1, 2, 3];
    let v2 = v.clone();

    println!("{:?}", v);
    println!("{:?}", v2);
}
```

```
$ cargo run --quiet
[1, 2, 3]
[1, 2, 3]
```

Agora temos em memória dois Vectors, um sendo referenciado por `v` e outro sendo referenciado por `v2`. Para vermos a diferença, vamos tornar nossos vetores mutáveis e, após o clone, mudar os dados.

```
fn main() {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let mut v = vec![1, 2, 3];
    let mut v2 = v.clone();

    println!("v:  {:?}", v);
    println!("v2: {:?}", v2);

    v[1] = 5;
    v2[1] = 88;

    println!("v:  {:?}", v);
    println!("v2: {:?}", v2);
}
```

```
$ cargo run --quiet
v:  [1, 2, 3]
v2: [1, 2, 3]
v:  [1, 5, 3]
v2: [1, 88, 3]
```

Nosso código funciona, pois respeitamos o princípio de propriedade do Rust, segundo o qual o dado em memória só pode ter uma única referência. Isto é, um dado em memória tem um único dono apenas.

Uma dica antes de seguirmos em frente, para saber quais traits um determinado tipo do Rust implementa basta olhar a documentação oficial. Por exemplo, em https://doc.rust-lang.org/std/primitive.i32.html vemos que o tipo `i32` implementa o trait `Copy`. Já em https://doc.rust-lang.org/std/vec/struct.Vec.html vemos que o tipo `Vec` não o implementa, apenas o trait `Clone`.

## 8.6 BORROWING

Quando compilamos nosso exemplo anterior pela primeira vez o compilador nos informou o erro `borrow of moved value`, ou seja, *empréstimo de valor movido*. Mas o que ele quis dizer com



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

empréstimo?

Em Rust, o conceito de empréstimo quer dizer que não damos a propriedade de um dado para outro contexto, mas o emprestamos por algum tempo. Imagine uma situação onde definimos um vetor e o passamos para parâmetro para outro método:

```
fn print_v(pv: Vec<i32>) {
    println!("v em print_v:  {:?}", pv);
}

fn main() {
    let v = vec![1, 2, 3];
    print_v(v);
    println!("v em main:  {:?}", v);
}
```

Ao chamarmos o método `print_v` e passarmos nosso Vector `v` como parâmetro, estamos dando ao `print_v` a propriedade de `v`. Isso quer dizer que qualquer tentativa de acesso a `v` posterior vai nos dar um erro.

```
error[E0382]: borrow of moved value: `v`
 --> src/main.rs:8:34
  |
6 |     let v = vec![1, 2, 3];
  |         - move occurs because `v` has type `Vec<i32>`, which
does not implement the `Copy` trait
7 |     print_v(v);
  |             - value moved here
8 |     println!("v em main:  {:?}", v);
  |                                  ^ value borrowed here after
move
  |
  = note: this error originates in the macro `$crate::format_args
_nl` (in Nightly builds, run with -Z macro-backtrace for more inf
o)
```

O compilador nos diz novamente que tentamos emprestar o



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

nosso Vector após o mover. E isso não é permitido pois Rust leva muito a sério a questão de quem é dono do que. Para isso, podemos utilizar o empréstimo de nosso Vector, em vez de passar sua propriedade de `main` para `print_v`. Usamos o sinal de `&` para indicar que estamos emprestando algo, não passando sua propriedade.

```
fn print_v(pv: &Vec<i32>) {
    println!("v em print_v:  {:?}", pv);
}

fn main() {
    let v = vec![1, 2, 3];
    print_v(&v);
    println!("v em main:  {:?}", v);
}
```

```
$ cargo run --quiet
v em print_v:  [1, 2, 3]
v em main:  [1, 2, 3]
```

Observe que dados emprestados, caso sejam mutáveis, podem ser modificados dentro do contexto que recebeu o empréstimo. Vamos alterar nosso código para que o método `print_v` altere o nosso Vector após o imprimir.

```
fn print_v(pv: &mut Vec<i32>) {
    println!("v em print_v:  {:?}", pv);
    pv[1] = 44;
}

fn main() {
    let mut v = vec![1, 2, 3];
    print_v(&mut v);
    println!("v em main:  {:?}", v);
}
```

```
$ cargo run --quiet
v em print_v:  [1, 2, 3]
v em main:  [1, 44, 3]
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Uma curiosidade: quando a macro `println!` é expandida (vamos ver mais sobre isso no capítulo 10, sobre macros), o código gerado faz o empréstimo do valor que passamos como parâmetro. Isso quer dizer que este código:

```
fn main() {
    let v = vec![1, 2, 3];
    println!("{:?}", v);
}
```

Vira este código quando é compilado (repare no `&v`, indicando o empréstimo):

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2021::*;
#[macro_use]
extern crate std;
fn main() {
    let v = <[_]>::into_vec(box [1, 2, 3]);
    {
        ::std::io::_print(::core::fmt::Arguments::new_v1(
            &["", "\n"],
            &match (&v,) {
                _args => [::core::fmt::ArgumentV1::new(
                    _args.0,
                    ::core::fmt::Debug::fmt,
                )],
            },
        ));
    };
}
```

Não se preocupe muito com o funcionamento e expansão da macro em si, pois, como já falamos, vamos falar sobre isso mais à frente. O importante no momento é saber que o `println!` e outras macros de impressão do Rust não pegam para si a propriedade dos dados que você passa.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## Conclusão

Neste capítulo, falamos sobre como Rust gerencia a memória, como são feitas a alocação de espaço e a conversão de um objeto para outro, e também como trabalhar com a pilha e o heap. Falamos ainda do `Box`, um recurso simples, mas poderoso, que facilitou muito o uso do heap na linguagem Rust, sem sinais esotéricos, como `~`.

Também detalhamos com exemplos os conceitos de propriedade e empréstimo, que são base fundamental para entender a linguagem.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 9

# PROCESSAMENTO PARALELO E ASSÍNCRONO

A maior parte do processamento que executamos em nossos códigos é síncrona, ou seja, você pede que algo seja feito, aguarda, pega o resultado dessa execução e usa em um próximo grupo de instruções, e assim sucessivamente até que tenha o resultado desejado e o retorne ao usuário. Porém, existem ações que não precisam necessariamente ser executadas naquele momento.

Um exemplo do mundo real é a compra de comida para comer em casa. No modo síncrono você teria que pegar seu carro, ir até o *drive thru* da loja e comprar a comida. No modo assíncrono você pede a comida em um aplicativo e faz outras coisas até ela chegar.

Assim como na vida real, vamos encontrar diversas situações em que podemos delegar nossas ações para algo que vai ser processado dessa forma, ou seja, enquanto você faz suas tarefas síncronas o processamento assíncrono está ocorrendo em paralelo e você será informado quando ele for concluído. Chamamos esse tipo de execução de processamento paralelo, assíncrono ou mesmo processamento cooperativo.

Neste capítulo vamos falar sobre como paralelizar o



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

processamento em Rust. A linguagem possui mecanismos poderosos como as clássicas `threads` e também inovadores como o `async/await`.

## 9.1 PROCESSAMENTO PARALELO E THREADS

Thread é um recurso computacional que possibilita a execução paralela de código no mesmo processo. Um processo possui sempre uma thread principal e pode apresentar subthreads, criadas no momento em que duas situações ocorrem lado a lado – como enviar um e-mail de confirmação.

Se você vem de linguagens como C ou Java, sabe que trabalhar com threads pode ser bem complexo e trabalhoso. Isso advém do compartilhamento de um estado mutável, ou seja, duas partes do código acessando concomitantemente o mesmo grupo de valores e variáveis.

Rust tem um modo particular de lidar com processamento paralelo que nos ajuda a gerenciar o compartilhamento dos valores. Já falamos sobre contexto de responsabilidade anteriormente, e ele se aplica ao modo como Rust trabalha com threads. Isso evita que elas acessem dados que não foram explicitamente enviados para elas, bem como a famigerada **race condition**.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**RACE CONDITION**

Uma race condition (ou condição de corrida) acontece quando o código possui processamento paralelo (por exemplo: múltiplas threads) e retorna um valor inesperado quando o processamento paralelo não retorna o valor esperado. Um exemplo clássico disso é quando dois recursos acessam o mesmo valor compartilhado, mas não conseguem pegar o valor e atualizá-lo de forma atômica.

Exemplo:

```
+----------------------------------+
|       PROCESSAMENTO PRINCIPAL    |
+----------------------------------+
|             estado = 0           |
+----------------------------------+
| PROCESSAMENTO 1 | PROCESSAMENTO 2 |
+-----------------|-----------------+
| pega estado     |                 |
| meu estado = 0  | pega estado     | <= antes do incremento
| incrementa      | meu estado = 0  |
| meu estado = 1  | incrementa      |
| escreve estado  | meu estado = 1  |
| estado = 1      | escreve estado  |
|                 | estado = 1      |
+----------------------------------+
|             estado = 1           |
+----------------------------------+
```

Nesse caso, dois processamentos fazendo o incremento resultariam no estado igual a 1, não 2. Tudo isso acontece devido à race condition/condição de corrida.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 9.2 CRIANDO UMA THREAD

Em Rust, criar uma thread utilizando o pacote `std::thread` é simples. A partir de `thread`, usamos o método `spawn`, que recebe um bloco de código para execução e retorna uma referência imutável à thread criada.

No exemplo a seguir, criaremos duas threads, `a` e `b`, e cada uma vai imprimir uma mensagem diferente na tela.

```
use std::thread;

fn main() {
    let _a = thread::spawn(|| {
        println!("Oi vindo da thread A!");
    });

    let _b = thread::spawn(|| {
        println!("Oi vindo da thread B!");
    });
}
```

Esse exemplo funciona, mas há grandes chances de termos nenhuma mensagem impressa na saída padrão ao executar o nosso corriqueiro `cargo run --quiet`. Isso ocorre porque nossa thread principal encerrou antes que as subthreads tenham sido executadas. Para sincronizar as threads, usamos o método `join()`, que as anexará à principal thread do código, garantindo que esta finalize apenas após a execução das outras.

```
use std::thread;

fn main() {
    let a = thread::spawn(|| {
        println!("Oi vindo da thread A!");
    });

    let b = thread::spawn(|| {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
        println!("Oi vindo da thread B!");
    });

    let _ = a.join();
    let _ = b.join();
}
```

Execute algumas vezes o código e veja como não existe garantia de qual thread executará antes.

```
$ cargo run --quiet
Oi vindo da thread B!
Oi vindo da thread A!

$ cargo run --quiet
Oi vindo da thread B!
Oi vindo da thread A!

$ cargo run --quiet
Oi vindo da thread A!
Oi vindo da thread B!
```

## 9.3 MOVENDO O CONTEXTO

Uma thread possui um contexto totalmente novo, que pode assumir a responsabilidade por aquele onde originalmente foi criada. Isso deve ser feito com o método `move` de maneira explícita. O código a seguir, por exemplo, não compila.

```
use std::thread;

fn main() {
    let value = 10;

    let a = thread::spawn(|| {
        println!("{}", value);
    });

    let _ = a.join();
}
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
error[E0373]: closure may outlive the current function, but it bo
rrows `value`, which is owned by the current function
 --> src/main.rs:6:27
  |
6 |     let a = thread::spawn(|| {
  |                           ^^ may outlive borrowed value `valu
e`
7 |         println!("{}", value);
  |                        ----- `value` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:13
  |
6 |       let a = thread::spawn(|| {
  |  _____________^
7 | |         println!("{}", value);
8 | |     });
  | |_____^
help: to force the closure to take ownership of `value` (and any
other referenced variables), use the `move` keyword
  |
6 |     let a = thread::spawn(move || {
  |                           ^^^^^^^

error: aborting due to previous error

For more information about this error, try `rustc --explain E0373
`.
error: could not compile `thread`

To learn more, run the command again with --verbose.
```

Vamos modificar o nosso código e adicionar o `move`.

```
use std::thread;

fn main() {
    let value = 10;

    let a = thread::spawn(move || {
        println!("{}", value);
    });
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let _ = a.join();
}
```

```
$ cargo run --quiet
10
```

Ao transferirmos a responsabilidade do contexto para uma thread, o contexto original pode ser livremente alterado dentro dela, mantendo-se inalterado na que o possuía originalmente. Vamos criar duas threads que acessam o contexto da thread principal e alteram nossa variável `value`. Depois, imprimiremos o valor original.

```
use std::thread;

fn main() {
    let mut value = 10;

    let a = thread::spawn(move || {
        value = value + 123;
        println!("Thread A {}", value);
    });

    let b = thread::spawn(move || {
        value = value + 1;
        println!("Thread B {}", value);
    });

    let _ = a.join();
    let _ = b.join();

    println!("Thread principal {}", value);
}
```

```
$ cargo run --quiet
Thread A 133
Thread B 11
Thread principal 10

$ cargo run --quiet
Thread B 11
Thread A 133
Thread principal 10
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Perceba que o valor de `value` foi alterado nas duas threads, `a` e `b`, mas manteve-se inalterado dentro da principal. Rust faz uma gestão robusta do contexto, evitando que ocorram conflitos no acesso aos dados como ocorrem em outras linguagens.

Essa proteção evita problemas de *race condition*, mas dificulta o envio de resultados da execução dentro de uma thread para outra. Os *channels* entram nessas situações, em que precisamos comunicar o resultado. São eles que possibilitam comunicação assíncrona entre as threads em execução.

## 9.4 ENTENDENDO CHANNELS

Programação com concorrência geralmente apresenta alguns problemas, como a *race condition*. Nessa situação, a thread principal inicia inúmeras outras e infere que elas vão terminar o processamento em uma sequência esperada, o que pode não ocorrer e, assim, gerar resultados indesejados.

Outro problema comum é o *deadlock*, quando duas ou mais threads tentam acessar o mesmo recurso ao mesmo tempo. Imagine um cenário no qual temos uma *thread A* que inicia duas outras, B e C. A *thread A* possui dois objetos, X e Y, acessíveis globalmente. O ciclo inicia, e a *thread B* começa o seu processamento obtendo X para si e bloqueando o acesso para outras threads. Já a *thread C* obtém Y para si, e também o bloqueia. A *thread B* agora precisa usar Y, mas ele está bloqueado por *C*; e a *thread C* precisa usar X, bloqueado por *B*. Temos um deadlock e uma jornada de trabalho que vai durar até o dia seguinte.

É aí que entra um documento de 1977, chamado



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**Communicating Sequential Processes**, ou simplesmente **CSP** (https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf). Inicialmente elaborado por Tony Hoare, ele define boas práticas para especificação e verificação de aspectos de concorrência em diversos sistemas.

Um dos conceitos-chaves desse material é o de *channels*, que define um mecanismo para intercomunicação entre processos, por meio do envio de mensagens. Um channel possui um canal de envio e um para recebimento de mensagens entre contextos diferentes. Dessa forma, não precisamos de uma variável fora do contexto da thread para armazenar o resultado da sua execução. Ele pode ser enviado diretamente por um canal de comunicação para outra thread, que o recebe e o trata da melhor forma.

Para utilizarmos channels, Rust provê o pacote `std::sync::mpsc`. Ele cria uma tupla com dois elementos, cujo primeiro é uma referência ao canal de envio, ou `Sender`, e o segundo é uma referência ao canal de recebimento, ou `Receiver`.

Vamos escrever um exemplo no qual teremos uma constante estática chamada `CARS`; ela apresentará uma quantidade predefinida de carros que participarão de uma corrida imaginária. Nosso programa vai imprimir na tela a mensagem *Carro xx terminou a corrida* – em que `xx` será um `id` gerado pela thread principal – dentro de uma thread específica para cada um de nossos carros.

Entretanto, antes de imprimirmos essa mensagem na saída padrão, a thread que representa nosso carro vai enviar o seu `id` para a thread principal, por meio de um `Sender`. Assim, ela poderá verificar se a ordem em que as mensagens são impressas


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

corresponde à sua real disposição.

Depois de todos os carros chegarem ao final – ou seja, após todas as threads finalizarem sua execução –, a principal vai coletar os `ids` recebidos em um vetor e exibi-los. Um ponto importante é que cada carro deverá ter seu próprio canal de comunicação. Logo, cada thread vai copiar o contexto da thread principal, mas receberá um clone do `sender` original.

A definição do nosso channel ocorrerá na linha a seguir. Ela foi dividida em duas para não quebrar a formatação do livro, mas é uma única instrução.

```
let (sender, receiver): (Sender<i32>,
   Receiver<i32>) = mpsc::channel();
```

Nessa linha, criamos uma tupla `(sender, receiver)` com uma referência ao canal de envio chamado `sender`, e outra ao canal de recebimento, `receiver`. Ambos foram definidos no contexto da thread principal, ou seja, eles pertencem a ela.

Na sequência, teremos um loop no qual vamos disparar várias threads, mas em cada uma delas passaremos um clone do `sender` original, chamado `thread_sender`. Isso é necessário para que cada thread tenha o seu próprio canal para se comunicar com a principal.

A thread é criada com o `spawn` e envia seu `id` pelo `thread_sender`. Perceba que ignoramos o resultado do método `send`, pois não precisamos tratar o retorno. Durante a sua execução, a thread ainda imprime na saída padrão quando finaliza a corrida.

```
for id in 0..CARS {
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    let thread_sender = sender.clone();

    thread::spawn(move || {
        let _ = thread_sender.send(id);

        println!("Carro {} terminou a corrida", id);
    });
}
```

Na sequência, criamos um vetor `ids` para receber os dados enviados ao nosso `receiver`, e eles são guardados na sequência em que chegam. Para ler um dado dessa fila, usamos o `recv`.

```
let mut ids = Vec::with_capacity(CARS as usize);
for _ in 0..CARS {
    ids.push(receiver.recv());
}
```

Nosso código completo ficará assim:

```
use std::sync::mpsc::{Sender, Receiver};
use std::sync::mpsc;
use std::thread;

static CARS: i32 = 5;

fn main() {
    let (sender, receiver): (Sender<i32>,
       Receiver<i32>) = mpsc::channel();

    for id in 0..CARS {
        let thread_sender = sender.clone();

        thread::spawn(move || {
            let _ = thread_sender.send(id);

            println!("Carro {} terminou a corrida", id);
        });
    }

    let mut ids = Vec::with_capacity(CARS as usize);
    for _ in 0..CARS {
        ids.push(receiver.recv());
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    }

    println!("Ordem final -> {:?}", ids);
}
```

Algumas execuções do código mostram que nem todas as corridas são iguais.

```
$ cargo run --quiet
Carro 0 terminou a corrida
Carro 2 terminou a corrida
Carro 1 terminou a corrida
Carro 4 terminou a corrida
Carro 3 terminou a corrida
Ordem final -> [Ok(0), Ok(2), Ok(1), Ok(4), Ok(3)]

$ cargo run --quiet
Carro 2 terminou a corrida
Carro 1 terminou a corrida
Carro 4 terminou a corrida
Carro 3 terminou a corrida
Carro 0 terminou a corrida
Ordem final -> [Ok(2), Ok(1), Ok(4), Ok(3), Ok(0)]

$ cargo run --quiet
Carro 2 terminou a corrida
Carro 1 terminou a corrida
Carro 0 terminou a corrida
Carro 4 terminou a corrida
Carro 3 terminou a corrida
Ordem final -> [Ok(2), Ok(1), Ok(0), Ok(4), Ok(3)]
```

Esse exemplo foi livremente adaptado do disponível em: https://doc.rust-lang.org/stable/rust-by-example/std_misc/channels.html

## 9.5 PROCESSAMENTO ASSÍNCRONO E O PADRÃO ASYNC/AWAIT

O processamento assíncrono é um formato de programação


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

assíncrona, não blocante, que por padrão retorna uma *promessa de retorno*, e não o retorno efetivamente. Ela se difere da programação com threads, por exemplo, por fazer uso de processamento cooperativo e de seu retorno poder não ocorrer (por exemplo, o download de algo que não está mais disponível).

Rust implementa o modelo de programação assíncrono através do conceito de `futures`, ou seja, uma execução que vai retornar um valor no futuro. Exemplos do dia a dia incluem operações que podem ser blocantes no modo síncrono, como o download de arquivos longos ou cálculos complexos.

No modelo de programação síncrona o download de um longo arquivo pode levar bastante tempo, e isso bloquearia nossas ações enquanto o arquivo não é baixado por completo. E definitivamente, em um universo onde temos processadores de sobra, isso não é algo esperado ou desejável.

Dada a visão da linguagem Rust de abstrações de custo zero, as funcionalidades de async que usamos são implementadas por bibliotecas que chamamos de *runtime*. Vamos falar dessas bibliotecas em breve, mas primeiramente vamos mostrar o conceito de `future` em Rust, implementado através do trait `Future`, e como isso funciona por baixo dos panos. A definição da trait `Future` é a seguinte:

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self:
:Output>;
}
```

Temos um tipo associado chamado `Output`, que representa o tipo do dado que vamos tratar assincronamente. No caso da leitura


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

de um arquivo grande, por exemplo, o tipo associado seria um `File`.

Já o método `poll` possibilita verificar se o valor esperado já está disponível. O retorno é do tipo `Poll<Self::Output>`, onde `Poll` é representado dessa forma:

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Ao chamarmos `poll`, receberemos um `enum` que nos diz se o resultado do processamento ainda está pendente com o `Pending`, ou se já está disponível, com o `Ready`. Um exemplo bem simples seria:

```
let future = async_download("http://arquivo.enorme.com/arquivo.tx
t");
let file_content = loop {
    match future.poll(...) {
        Poll::Ready(value) => break value,
        Poll::Pending => {},
    }
}
```

Obviamente, esse não é o melhor caso de uso de um `future`, pois ainda estamos bloqueando a continuidade de nosso código dentro do `loop`, mas ele serve para exemplificar o funcionamento do `poll` e seu retorno. Repare que usamos `...` como parâmetro de `poll` nesse exemplo. Isso é apenas para fins didáticos, pois neste momento o objetivo é explicar como o `future` se comporta, então os parâmetros não importam por enquanto (e esse código atualmente não compila).

Para evitar que o código fique bloqueado, o Rust possui uma palavra-chave que indica que o método deve ser executado de


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

forma assíncrona, no caso, o `async`. Crie um novo projeto Rust e adicione no `Cargo.toml` a dependência a seguir:

```
[dependencies]
futures = "0.3.0"
```

Agora vamos ao código. Primeiro precisamos declarar que vamos utilizar o crate `futures`.

```
extern crate futures;
```

Na sequência, vamos definir um método que faz um cálculo assíncrono, ou seja, ele só fará a operação (sequência de Fibonacci) quando efetivamente for necessário. Repare na palavra-chave `async` no começo da linha, antes do `fn`.

```
async fn async_calculation(n: u32) -> u32 {
    if n == 0 {
        panic!("Zero não é um argumento válido!");
    } else if n == 1 {
        return 1;
    }

    let mut sum = 0;
    let mut last = 0;
    let mut curr = 1;
    for _i in 1..n {
        sum = last + curr;
        last = curr;
        curr = sum;
    }
    sum
}
```

A palavra reservada `async` vai, em tempo de compilação, deixar nosso código assim: `fn async_calculation(num1: u32, num2: u32) -> impl Future<Output = u32>`. É nesse momento que dizemos ao Rust: *quando esse método for chamado, não precisa parar tudo pra executar ele não, pode deixá-lo na memória aí e*


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

*quando eu precisar desse resultado eu te aviso, ou faz assim que você puder*.

Obviamente, nosso código depende em algum momento da execução desse método, então precisamos dizer em algum momento para o Rust: *agora eu preciso que você aguarde a execução do método*. E fazemos isso com o método `await`. Para entender melhor vamos criar um novo método que vai chamar nosso cálculo assíncrono:

```
async fn example_task() {
    let num1 = 10;
    let number = async_calculation(num1).await;
    println!("{} => {}", num1, number);
}
```

Veja que ao chamar o método `async_calculation` passamos o `await`, informando ao Rust que o momento de executar aquele pedaço de código é ali. Agora é só criar um `future` e, no momento certo, bloquear nosso código. Adicionamos um loop de "Nada acontece por aqui" para que fique claro em qual momento a execução de nossa soma assíncrona ocorreu:

```
fn main() {
    let future = example_task();

    for _ in 0..10 {
        println!("Nada acontece por aqui");
    }

    futures::executor::block_on(future);
}
```

O código completo fica assim.

```
extern crate futures;

async fn async_calculation(n: u32) -> u32 {
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    if n == 0 {
        panic!("Zero não é um argumento válido!");
    } else if n == 1 {
        return 1;
    }

    let mut sum = 0;
    let mut last = 0;
    let mut curr = 1;
    for _i in 1..n {
        sum = last + curr;
        last = curr;
        curr = sum;
    }
    sum
}

async fn example_task() {
    let num1 = 10;
    let number = async_calculation(num1).await;
    println!("{} => {}", num1, number);
}

fn main() {
    let future = example_task();

    for _ in 0..10 {
        println!("Nada acontece por aqui");
    }

    futures::executor::block_on(future);
}
```

Repare que a execução do método ocorre apenas quando dizemos para o `executor` do `futures` bloquear em nosso `future`.

```
$ cargo run --quiet
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
10 => 55
```

O que queremos com esse exemplo é dar uma ideia de como a linguagem lida com async para que você tenha essa ideia ao programar, mas não se preocupe, que as coisas vão ser bem mais automáticas que isso.

Nós queremos mesmo é só delegar a execução para um tempo no futuro e que o código seja executado em paralelo, e para isso vamos explorar o papel do tal do *runtime* nisso tudo.

## Processamento multitarefa cooperativo

Quando falamos em multitarefa imaginamos várias tarefas sendo executadas ao mesmo tempo, como um time de futebol com cada um correndo para um lado do campo de forma a se posicionar para o grande momento de finalização do lance. No mundo real é fácil visualizar isso porque cada indivíduo que realiza uma tarefa tem o seu próprio controle, ou seja, ele quem define o que vai fazer.

Quando falamos de computadores temos a questão de o controle ser único. Ou é o sistema operacional ou o programa em execução que controla quem faz o quê naquele momento. Ou seja, não existe uma independência entre tarefas. Voltando ao exemplo do time, é como se cada jogador aguardasse uma ordem do treinador para onde ir.

As linguagens de programação que hoje implementam o


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

processamento multitarefa cooperativo através de conceitos como *corotinas* ou *async/await* possuem uma peça de software importante chamada `runtime`. Se você já programou em Ruby usando o Sidekiq ou em Python com o Celery esses conceitos talvez sejam mais familiares.

O `runtime` é uma peça de software que dá a liberdade para os nossos jogadores se posicionarem em campo, liberando o treinador do time para cuidar de outras coisas. É como se o técnico do time tivesse um grupo de auxiliares que falasse "vai lá e faz o seu melhor", sem sobrecarregar o técnico. Chamamos esse tipo de processamento paralelo de cooperativo, pois ele depende da cooperação desse runtime para existir.

## Tokio

Tokio é o runtime mais popular em todo o ecossistema Rust. Ele provê exatamente o que precisamos: um modo de executar nosso código em paralelo sem demandar do nosso processo principal.

Por ser bem complexo e extenso, não vamos nos aprofundar no Tokio neste capítulo. Você pode ler mais sobre ele no tutorial disponível em https://tokio.rs/tokio/tutorial, porém é importante entender o que o Tokio faz. Vamos reescrever nosso exemplo de soma assíncrona com ele. Para isso, adicione em seu `Cargo.toml` a dependência:

```
[dependencies]
tokio = { version = "1.16.1", features = ["full"] }
```

Veja que, além de definirmos a versão que queremos, dizemos ao cargo quais são as funcionalidades que vamos usar. No caso,


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

aqui coloquei `full`, ou seja, todas as funcionalidades do Tokio serão carregadas. Repare também que removemos a dependência a `futures`. Eles passam a ser responsabilidade do Tokio.

A primeira mudança em nosso projeto ocorre em nossa função `main`. Ela agora será `async` e terá um retorno. Isso ocorre porque ela passará a ser gerenciada pelo Tokio, através da diretiva `#[tokio::main]`.

Nossa função `example_task`, que é quem faz a chamada a nossa soma assíncrona, será lançada pelo Tokio através do método `tokio::spawn`. Além disso, para que seja possível ver a execução do código em paralelo colocamos nosso `println!` dentro de um loop, que será executado a cada meio segundo. Nosso `main` ficará assim:

```
#[tokio::main]
async fn main() -> () {
    tokio::spawn(async { example_task().await });

    loop {
        sleep(Duration::from_millis(500)).await;
        println!("Nada acontece por aqui");
    }
}
```

Da mesma forma modificamos a função `async_calculation` para realizar nossa operação demorando um pouco a cada interação do cálculo de Fibonacci. Juntando tudo, nosso código completo ficará assim:

```
use std::time::Duration;
use tokio::time::sleep;

async fn async_calculation(n: u64) -> u64 {
    if n == 0 {
        panic!("Zero não é um argumento válido!");
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    } else if n == 1 {
        return 1;
    }

    let mut sum: u64 = 0;
    let mut last: u64 = 0;
    let mut curr: u64 = 1;
    for _i in 1..n {
        sleep(Duration::from_millis(100)).await;
        sum = last + curr;
        last = curr;
        curr = sum;
    }
    sum
}

async fn example_task() {
    let num1 = 75;
    let number = async_calculation(num1).await;
    println!("{} => {}", num1, number)
}

#[tokio::main]
async fn main() -> () {
    tokio::spawn(async { example_task().await });

    loop {
        sleep(Duration::from_millis(500)).await;
        println!("Nada acontece por aqui");
    }
}
```

Ao executarmos esse código veremos a cada meio segundo a mensagem *Nada acontece por aqui* ser impressa. E em algum momento o cálculo será finalizado, pois ele ocorre em paralelo:

```
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
Nada acontece por aqui
75 => 2111485077978050
Nada acontece por aqui
```


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
Nada acontece por aqui

[...]
```

Esse processamento vai ocorrer para sempre, então para encerrar a execução basta pressionar o CTRL+C.

## Conclusão

Neste capítulo, vimos como criar threads em Rust, como trabalhar com o contexto dentro delas, e também como realizar a intercomunicação entre elas por meio dos channels.

Além disso falamos sobre um novo e popular recurso da linguagem, o conjunto `async/await` . No próximo capítulo, discutiremos sobre macros.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 10
# MACROS

Neste capítulo, daremos uma olhada no poderoso mecanismo de macros da linguagem Rust. Seu uso é fundamental para estender a linguagem e reduzir a quantidade de código escrito.

## 10.1 POR QUE MACROS?

Geralmente, uma macro parece com uma função, mas uma diferença fundamental entre as duas é que o código da macro é inserido nos pontos em que ela é chamada, em vez de ser um ponto único no projeto.

Isso pode dar a falsa impressão de que o uso de macros é ruim. Pelo contrário, por ocorrer em tempo de análise léxica, uma macro pode acessar diretamente o código que será gerado na compilação, algo que não estará mais disponível quando ele for compilado.


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**TEMPO DE COMPILAÇÃO E AFINS**

Em linguagens compiladas, o código passa por várias etapas quando pedimos ao compilador para transformá-lo em código binário. Da análise léxica do código – na qual a sintaxe é validada e erros são informados, como a ausência de um ponto e vírgula no final de uma linha — até a escrita do binário executável em si, o código-fonte escrito passa por diversas transformações, análises e otimizações.

Na prática, isso quer dizer que macros podem ser implementadas como uma forma de substituição textual simples, como é o caso da macro `println!`, que usamos exaustivamente até agora.

## 10.2 MACROS DECLARATIVAS

Mencionamos a macro `println!` na seção anterior e repare que, em sua definição seu nome não leva a exclamação ( `!` ) e que ela é feita a partir de outra macro, chamada `macro_rules!`, na qual definimos as regras de execução da macro em nosso código.

```
macro_rules! println {
    () => { ... };
    ($($arg : tt) *) => { ... };
}
```

Pode parecer um código estranho a princípio, mas cada linha define uma regra de acordo com os parâmetros recebidos. Você pode chamar `println!` sem nenhum ou com vários parâmetros,



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

ou até com uma string fixa.

Vamos criar uma macro simples, que imprime o *hello world*. Como não receberemos nenhum parâmetro em nossa execução, vamos definir a regra apenas para a chamada em que não recebemos nenhum parâmetro, ou `()`.

```
macro_rules! hello {
    () => { println!("hello world de macros"); }
}

fn main() {
    hello!();
}
```

A execução desse código vai imprimir a mensagem "hello world de macros" em sua saída padrão, mas essa não é a parte interessante. O nosso código compilado não possui duas funções, apenas uma, a função `main`, o que podemos identificar ao usarmos o utilitário `nm`, que está disponível no pacote *binutils* da GNU (https://www.gnu.org/software/binutils/).

Esse utilitário lista os símbolos de um binário em conjunto com o `grep`, e este filtra o resultado. Perceba que a saída do comando `nm` retorna todos os símbolos de nosso binário, e não é o que queremos. Vamos usar como parâmetro do `grep`, o nome de nosso binário (no caso, `hello_macro_world`), para que ele mostre apenas os símbolos dele. Veja a compilação e a execução:

```
$ cargo run --quiet
hello world de macros
```

Veja os símbolos impressos com o `nm`:

```
$ nm target/debug/hello_macro_world | grep hello_macro_world
0000000000005160 t _ZN17hello_macro_world4main17he479c8f0378dbc5d
E
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Vamos agora reescrever nosso código utilizando um método, como já vimos antes:

```
fn hello() {
    println!("hello world");
}

fn main() {
    hello();
}
```

Ao executar vemos o seguinte retorno:

```
$ cargo run --quiet
hello world
```

Podemos ver que o resultado da execução do comando `nm` é diferente:

```
$ nm target/debug/hello_world | grep hello_world
00000000000057a0 t _ZN11hello_world4main17h77d9d462919d3604E
0000000000005750 t _ZN11hello_world5hello17h3a6ae60c6fadbc60E
```

No momento da compilação de nosso código, o analisador sintático da linguagem Rust identifica que temos uma definição de macro e, quando a encontra, ele expande o código para seu resultado. Podemos dizer que o nosso código com a macro `hello` é equivalente ao:

```
fn main() {
    println!("hello world de macros");
}
```

Por isso, o `nm` não encontra outra função em nosso código, já que ela não existe. Rust identifica que chamamos a macro `hello` sem nenhum argumento e a substitui no código pela regra estabelecida. As regras são definidas com o uso de `=>`, sendo a primeira parte o padrão e a segunda, o que deve ser feito para



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

atendê-la. Em nosso caso, o padrão era `()`, ou seja, nenhum argumento.

## 10.3 RECURSIVIDADE

Vamos tornar as coisas um pouco mais complexas. A seguir, teremos uma macro que calcula a distância entre dois números. O resultado desse cálculo será armazenado em uma variável que será inicializada também nessa macro.

```
macro_rules! distance {
    ($a: ident, $b: expr, $c: expr) => {
        let $a = {
            if $b >= $c {
                $b - $c
            } else {
                $c - $b
            }
        };
    }
}

fn main() {
    distance!(x, 3, 5);
    distance!(y, 5, 3);
    println!("{}, {}", x, y);
}
```

```
$ cargo run --quiet
2, 2
```

Agora, recebemos três argumentos em nossa macro. Eles são atribuídos a metavariáveis, devem ser nomeados com o símbolo `$` e receber como tipo uma definição da linguagem específica para a criação de macros (no caso, `ident` e `expr`).


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**METAVARIÁVEIS**

Na definição de variáveis em macros, é comum a utilização do termo metavariável para identificadores que correspondem a partes de nosso código que serão substituídas posteriormente.

Na Wikipédia, o termo é definido como:

*"Em lógica, uma metavariável (também conhecida como variável metalinguística ou variável sintática) é um símbolo ou string de símbolos que pertence a uma metalinguagem e se aplica a elementos de alguma linguagem objeto. Por exemplo, na sentença: Sejam A e B duas sentenças de uma linguagem ℒ.*

*Os símbolos A e B são parte de uma metalinguagem na qual a afirmação sobre a linguagem objeto ℒ é formulada".*

Fonte: https://pt.wikipedia.org/wiki/Metavariável.

A metavariável `$a` é um identificador de uma variável externa à macro (definido pelo tipo `ident`) – em nosso caso, `x` e `y`. Já as metavariáveis `$b` e `$c` são expressões (definidas pelo tipo `expr`) – no caso, os números 3 e 5. Outros valores podem incluir: uma definição de tipo como `i32` ou `char` (definido pelo tipo `ty`); um bloco de código dentro de chaves, como `{ return 0; }` (definido pelo tipo `block`); entre outros.

Perceba que não declaramos as variáveis `x` e `y` com o `let`. A nossa macro recebe-as como parâmetro e atribui o código da macro a elas. É correto dizer que o código anterior é equivalente a:



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn main() {
    let x = if 3 >= 5 {
        3 - 5
    } else {
        5 - 3
    };
    let y = if 3 >= 5 {
        3 - 5
    } else {
        5 - 3
    };
    println!("{}, {}", x, y);
}
```

Em macros, parâmetros do tipo `expr` recebem qualquer expressão válida em Rust. Dessa forma, o código a seguir é perfeitamente válido. Ele realiza o cálculo da distância entre dois números – por exemplo, 2 e 5, cuja distância é 3. Porém, em vez de receber valores inteiros, passamos expressões como *10 + 3*.

```
macro_rules! distance {
    ($a: ident, $b: expr, $c: expr) => {
        let $a = {
            if $b >= $c {
                $b - $c
            } else {
                $c - $b
            }
        };
    }
}

fn main() {
    distance!(x, 42 + 54, 112 + 33);
    distance!(y, 99 - 12, 1024 + 1);
    println!("{}, {}", x, y);
}
```

```
$ cargo run --quiet
49, 938
```

Poderíamos ainda modificar nossa macro e nomear as



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

metavariáveis, como a seguir:

```
macro_rules! distance {
    ($a: ident,
     v1 => $b: expr,
     v2 => $c: expr) => {
        let $a = {
            if $b >= $c {
                $b - $c
            } else {
                $c - $b
            }
        };
    }
}

fn main() {
    distance!(x, v1 => 2 + 4, v2 => 2 + 3);
    distance!(y, v1 => 9 - 2, v2 => 4 + 1);
    println!("{}, {}", x, y);
}
```

```
$ cargo run --quiet
1, 2
```

Também é possível utilizar uma lista variável de parâmetros. Para exemplificar, criaremos uma macro que faz a somatória de uma quantidade de elementos e atribui o resultado a uma variável. A definição de um método de somatória de valores pode ser complexa em linguagens como C, mas, em Rust, torna-se simples quando usamos macros.

O nosso primeiro parâmetro será do tipo `ident`, como vimos anteriormente, e o segundo, uma lista variável de parâmetros. Essa lista é definida pelo identificador `$*`. Veja:

```
macro_rules! sum {
    ($a: ident, $($x: expr), *) => {
        let $a = {
            let mut temp = 0;
            $(
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
                temp = temp + $x;
            )*
            temp
        };
    }
}

fn main() {
    sum!(x, 1, 2, 3, 4, 5, 6, 7);
    println!("{}", x);
}
```

```
$ cargo run --quiet
28
```

Nesse caso, dissemos à nossa macro que nosso segundo parâmetro é uma lista variável de `expr`, com a sintaxe `$($x:expr), *`. Na definição da macro, criamos uma variável interna temporária, chamada `temp`, e lhe atribuímos o valor inicial zero. Depois, percorremos cada um dos elementos da lista de tamanho variável e vamos adicionando um a um à variável temporária, que será retornada posteriormente e atribuída à metavariável `$a`.

O responsável pela iteração entre os elementos é o bloco `$(...)*`, que percorre cada elemento da lista, executando o código entre parênteses. Em nosso caso, a linha `temp = temp + $x;` será executada para cada elemento dentro dela, sendo que o elemento atual está em `$x`.

```
$(
    temp = temp + $x;
)*
```

Outra grande vantagem da macro é que ela não faz a definição de tipos. Assim, com uma pequena modificação em nossa macro, para que receba um valor inicial, podemos utilizá-la para realizar a somatória de qualquer tipo numérico.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Vamos adicionar um novo parâmetro, o `$b` , que pegará o primeiro valor de nossa lista de parâmetros para podermos usar qualquer conjunto de tipos. No exemplo anterior, inicializávamos nossa variável temporária com 0 (um inteiro), limitando o nosso código.

```
macro_rules! sum {
    ($a: ident,
     $b: expr,
     $($x: expr), *) => {
        let $a = {
            let mut temp = $b;
            $(
                temp = temp + $x;
            )*
            temp
        };
    }
}

fn main() {
    sum!(x, 1, 2, 3, 4, 5, 6, 7);
    println!("{}", x);

    sum!(x, 1.3, 2.4, 3.1, 4.98);
    println!("{}", x);
}
```

O resultado dessa execução é:

```
$ cargo run --quiet
28
11.780000000000001
```

## 10.4 EXEMPLO USANDO UMA ÁRVORES DE TOKENS

Nossa macro de soma mostrou como iterar em uma coleção de parâmetros. Nesta seção, veremos como iterar sobre um tipo de


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

dados conhecido como *token tree* (ou `tt`).

Este exemplo é baseado em um código que foi usado em uma versão antiga do livro oficial de Rust (https://doc.rust-lang.org/book). Ele recebe uma estrutura de dados e, a partir dela, gera código HTML.

A estrutura que vamos usar é simples. Definimos uma seção (por exemplo, *head*) e, entre colchetes, colocamos seus dados. É possível ter subseções dentro de uma seção indefinidamente, de forma a criar uma estrutura de documento HTML real. Veja a nossa estrutura:

```
html[
    head[title["Livro de Rust"]]
        body[
            h1["Autores"]
             p["Marcelo Castellani"]
             p["Willian 'PotHix' Molinari"]]
]
```

Para parsear a estrutura, usaremos o tipo `tt`. Ele é usado em metavariáveis de macros e pode ser traduzido como uma espécie de *catch-all*. Qualquer dado que não se encaixe nas outras metavariáveis da nossa declaração de macro será pego por ele.

Por exemplo, note a definição da macro `println!`. Ela possui duas partes, uma que indica o que deve ser impresso, e outra apontando os dados que serão processados e colocados na sentença a ser impressa. Sua definição `($fmt:expr, $($arg:tt)*)` indica que o primeiro item é o dado a ser impresso e interpolado, e o que vier a seguir é o que deve ser processado e encaixado no lugar certo do primeiro parâmetro.

Um `tt` é processado recursivamente até que todos os


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

argumentos passados sejam processados. A definição de argumento para um elemento do tipo `tt` é qualquer informação que esteja dentro de `()`, `[]` ou `{}`. Assim, podemos decompor nossa estrutura como a seguir:

```
html[
    head[title["Livro de Rust"]]
        body[
            h1["Autores"]
            p["Marcelo Castellani"]
            p["Willian 'PotHix' Molinari"]]
]

head[title["Livro de Rust"]]
    body[
        h1["Autores"]
        p["Marcelo Castellani"]
        p["Willian 'PotHix' Molinari"]]

body[
    h1["Autores"]
    p["Marcelo Castellani"]
    p["Willian 'PotHix' Molinari"]]

h1["Autores"]
p["Marcelo Castellani"]
p["Willian 'PotHix' Molinari"]

p["Marcelo Castellani"]

p["Willian 'PotHix' Molinari"]
```

Precisamos chamar nossa macro recursivamente para que o código HTML seja gerado corretamente para cada um dos conjuntos decompostos. Necessitamos também que ela identifique três conjuntos de dados: uma expressão solitária, uma seguida de um token tree e uma para a expressão completa. Essa última vai efetivamente criar a tag HTML de abertura (com o `<tag>`), adicionar o seu conteúdo através de outra chamada à nossa macro



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

e fechá-la (com o `</tag>`).

Usaremos uma outra macro, chamada `write!`, para nos auxiliar no processo. Ela é parecida com a `println`, mas, em vez de imprimir o resultado do processamento na saída padrão, ela pega e escreve o resultado em uma variável mutável. Como `write!` retorna um resultado (`Result`), precisamos verificá-lo para ter certeza de que a escrita aconteceu com sucesso. Vamos usar o `expect` para gerar um erro mais fácil de entender caso o `write!` não funcione e dê `panic`. Vejamos como fica nosso código:

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e).expect("Não foi pos
sível escrever o HTML"));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag)).expect("Não foi poss
ível escrever o HTML");
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag)).expect("Não foi pos
sível escrever o HTML");
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
    html[
        head[title["Livro de Rust"]]
            body[
                h1["Autores"]
                    p["Marcelo Castellani"]
                    p["Willian 'PotHix' Molinari"]]
    ]);
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("{}", out);
}
```

```
$ cargo run --quiet
<html><head><title>Livro de Rust</title></head><body><h1>Autores<
/h1><p>Marcelo Castellani</p><p>Willian 'PotHix' Molinari</p></bo
dy></html>
```

Fantástico, não? Token tree é um formato de dados muito simples e poderoso, que possibilita a criação de macros complexas com poucas linhas de código. Essas macros podem ser difíceis de se entender devido à sua necessidade de recursividade. Contudo, para facilitar a nossa vida, temos uma extensão para o `cargo` que realiza a expansão da macro e exibe o código-fonte gerado. Ela está disponível em: https://github.com/dtolnay/cargo-expand.

Instale a extensão com o comando `cargo install cargo-expand` . Para facilitar a leitura do código gerado. Uma vez instalado, basta executar o comando `cargo expand` para gerar o código da macro expandida.

Como resultado, temos cerca de trezentas linhas, porém vamos reproduzir apenas uma pequena amostra para referência, já que a maior parte delas é a repetição da chamada a seguir, que pega uma string de nossa estrutura e formata-a para gerar o código HTML. Vale a pena testar em seu ambiente.

```
(&mut out)
    .write_fmt(
        match match (&"html",) {
            (arg0,) => [::core::fmt::ArgumentV1::new(
                arg0,
                ::core::fmt::Display::fmt,
            )],
        } {
            ref args => unsafe { ::core::fmt::Arguments::new_v1(&
"<", ">"], args) },
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    },
)
.expect("Não foi possível escrever o HTML");
```

## 10.5 MACROS PROCEDURAIS

Além da versão declarativa de macros via padrões que descrevemos anteriormente, também temos as macros procedurais, que possuem uma finalidade parecida (substituir código em tempo de compilação) mas um tipo de operação diferente, operando em cima de tokens.

Não pretendemos ir a fundo no assunto de macros procedurais e como trabalhar com `TokenStream` (mais sobre isso em breve) neste livro porque esses são assuntos mais avançados e com vários pontos de atenção, mas vamos tentar mostrar as vantagens que esse tipo de macro traz e como discernir a macro que explicamos acima de uma `proc_macro`.

As macros procedurais são divididas em **Macros de função**, **Macros de Derive** e **Macros de atributos**.

### Resumo rápido de como funcionam

As macros procedurais (também conhecidas como `proc_macros`) operam em cima da AST (*Abstract Syntax Tree*, ou Árvore de Sintaxe Abstrata) em tempo de compilação e requerem alguns cuidados para evitar loops infinitos durante esse processo, já que o compilador não vai conseguir detectá-los.

Elas tem acesso às mesmas coisas que o compilador naquele dado momento da compilação e o código gerado é adicionado diretamente junto ao código que a invocou. Por esse motivo, deve-


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

se usar nomes únicos (ex.: `\_\_minhaproc_variavel` ao invés de `variavel`) e caminhos completos para os tipos (ex.: `::std::option::Option` em vez de só `Option`).

São utilizadas em conjunto com o crate `proc_macro`, que provê o tipo `TokenStream` e já vem junto com o compilador do Rust. Além disso, geralmente também fazem uso dos crates `syn` (para fazer parse de uma string de código para uma estrutura que pode ser manipulada) e `quote` (para transformar uma estrutura em uma string e um `TokenStream`).

## Macros de função

As macros de função são aquelas que geram funções com `!`. Elas recebem um `TokenStream` com o que há delimitado como parâmetros da macro e retornam um `TokenStream` que vai substituir a chamada da macro.

Um exemplo simples de uma macro de função seria uma que ignora o que recebe e imprime algo:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn i_am_leet(_item: TokenStream) -> TokenStream {
    "fn print_who_am_i() -> u32 { 1337 }".parse().unwrap()
}
```

Como uso na prática, podemos ver que estamos chamando a função `print_who_am_i()` que não aparece em nenhum lugar, pois é definida dentro de `i_am_leet`:

```
extern crate proc_macro_examples;
use proc_macro_examples::i_am_leet;
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
make_i_am_leet!();

fn main() {
    println!("{}", print_who_am_i());
}
```

## Macros de derive

As macros de derive, como você já deve imaginar, dão a habilidade de definir mais funções via `derive`. Com elas, você pode ter um código que faz algo como:

```
extern crate proc_macro_examples;
use proc_macro_examples::LeetDefinition;

#[derive(LeetDefinition)]
struct Struct;

fn main() {
    assert_eq!(42, print_who_am_i());
}
```

Nesse código, estamos fazendo o mesmo que fizemos no exemplo anterior, apenas ignorando o input do `TokenStream` (para mostrar a utilidade sem a complexidade de explicar o `TokenStream`, para meios didáticos) e adicionando a função `print_who_am_i` via `proc_macro`:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(LeetDefinition)]
pub fn derive_leet_fn(_item: TokenStream) -> TokenStream {
    "fn print_who_am_i() -> u32 { 1337 }".parse().unwrap()
}
```

A função recebe um `TokenStream` do item que está recebendo o `derive`, que no nosso exemplo é o `struct`, e



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

retorna um `TokenStream` com os itens que serão adicionados ao módulo ou bloco do `TokenStream` recebido.

Esse é um exemplo bem simples, mas o nosso derive macro `LeetDefinition` poderia, por exemplo, criar novos itens no struct em questão baseando-se no `TokenStream` recebido, caso assim quiséssemos.

## Macros de atributos

As macros de atributos funcionam de uma forma similar às macros de derive, só que de uma forma mais poderosa. Ao invés de dar a possibilidade de adicionar `derive` apenas em `struct` e `enum`, você pode aplicar os atributos em outros items (como funções!).

Como exemplo, vamos implementar uma macro de atributos que retorna o item inalterado:

```
#[proc_macro_attribute]
pub fn noop_attribute_macro(_attr: TokenStream, item: TokenStream
) -> TokenStream {
    item
}
```

Diferente da macro de derive, as macros de atributos recebem dois parâmetros de `TokenStream`. O primeiro parâmetro recebe os parâmetros da macro em si e o segundo é o corpo do item que a macro está "anotando".

Podemos exemplificar utilizando um exemplo de uma rota `get` da página principal do framework web Rocket (https://rocket.rs/):

```
#[get("/hello/<name>/<age>")]
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
fn hello(name: &str, age: u8) -> String {
    format!("Hello, {} year old named {}!", age, name)
}
```

Nesse exemplo, o parâmetro `_attr` vai ser `/hello/<name>/<age>`, enquanto o parâmetro `item` vai ser a função `hello` completa. Em vez de desestruturar o `item` (via crate `syn`), fazer modificações, reestruturar (possivelmente usando a crate `quote`), e devolver o `TokenStream`, estamos apenas retornando `item` inalterado.

## 10.6 POR QUE USAMOS MACROS?

O uso de macros nos possibilita olhar para o nosso através de uma camada mais profunda e gerar a versão final através de condições previamente definidas, sem repetição. Por ser uma linguagem compilada, a definição de uma simples função (como `println`) que não use o mecanismo de macros seria um trabalho hercúleo.

Para cada situação passível de impressão de dados, seria necessário um processamento que verificasse o tipo de dados a ser impresso, onde ele se encaixa e como formatá-lo. Ao usarmos uma macro como `println!`, esse trabalho é todo feito em tempo de compilação. O compilador identifica qual padrão deve ser usado para aquela situação específica e gera um código exclusivo para aquele ponto – isso para todos os usos de `println!` em nosso sistema.

Ao se usar macros as decisões são tomadas em tempo de compilação. Isso deixa a compilação um pouco mais lenta, mas garante que o código gerado no fim seja mais rápido, além, é claro,


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

da ergonomia ao escrever código, como mencionamos há pouco.

## Conclusão

Macros são um ponto bem *sexy* da linguagem Rust. Poderosas e robustas, possibilitam escrever código que automaticamente gera outro código para você, evitando repetições.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 11

# TESTAR, O TEMPO TODO

Escrever testes, o tempo todo, tornou-se um dos mantras da programação moderna. Por meio de recursos como macros e metaprogramação, é possível escrever código que valide o seu código e seja seu aliado para que novas implementações não quebrem o que já funcionava anteriormente.

Com o recurso de macros do Rust, podemos inserir em nosso código trechos de teste para validar o que esperamos que ocorra em determinadas situações, sem sermos surpreendidos por valores inesperados.

Neste capítulo, vamos apresentar as macros que possibilitam escrever testes em Rust, e mostrar o trait `Error` e sobre como utilizar o `Debug` para inspecionar suas instâncias durante a execução.

## 11.1 A MACRO PANIC!

Antes de discutirmos asserções e testes, é importante falarmos sobre a macro `panic!`. Ela é responsável por finalizar o processamento atual e disparar uma mensagem.

```
fn main() {
    panic!("PALMA PALMA, NÃO PRIEMOS CANICO");
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    println!("Esse código não será usado!");
}
```

No exemplo, a linha com o `println!` nunca será executada. Veja:

```
$ cargo run --quiet
 --> src/main.rs:4:5
  |
2 |     panic!("PALMA PALMA, NÃO PRIEMOS CANICO");
  |     ------------------------ any code following this express
ion is unreachable
3 |
4 |     println!("Esse código não será usado!");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ unreachable st
atement
  |
  = note: `#[warn(unreachable_code)]` on by default
  = note: this warning originates in a macro (in Nightly builds,
run with -Z macro-backtrace for more info)

warning: 1 warning emitted

thread 'main' panicked at 'THIS ENDS HERE', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace
```

Na saída, temos a informação de que podemos acessar o *backtrace* de nosso código ao definirmos a variável de ambiente `RUST_BACKTRACE` com o valor 1. Esse valor indica que o backtrace deve ser impresso na execução.

O backtrace é a lista de execuções que ocorreram desde o momento do início da execução de nosso código até o momento em que ele foi parado, pelo fim da execução ou por um erro.

Vamos dar uma olhada no backtrace:

```
$ RUST_BACKTRACE=1 cargo run --quiet
warning: unreachable statement
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
 --> src/main.rs:4:5
  |
2 |     panic!("PALMA PALMA, NÃO PRIEMOS CANICO");
  |     ------------------------ any code following this express
ion is unreachable
3 |
4 |     println!("Esse código não será usado!");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ unreachable st
atement
  |
  = note: `#[warn(unreachable_code)]` on by default
  = note: this warning originates in a macro (in Nightly builds,
run with -Z macro-backtrace for more info)

warning: 1 warning emitted

thread 'main' panicked at 'PALMA PALMA, NÃO PRIEMOS CANICO', src/
main.rs:2:5
stack backtrace:
   0: std::panicking::begin_panic
             at /[caminho da versão do toolchain do Rust]/lib/rus
tlib/src/rust/library/std/src/panicking.rs:519:12
   1: macro_panic::main
             at ./src/main.rs:2:5
   2: core::ops::function::FnOnce::call_once
             at /[caminho da versão do toolchain do Rust]/lib/rus
tlib/src/rust/library/core/src/ops/function.rs:227:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` fo
r a verbose backtrace.
```

Veja em nosso backtrace, na linha 1, exatamente onde ocorreu o `panic!`. A mensagem ainda sugere utilizar `RUST_BACKTRACE=full` case quisermos ver um backtrace ainda mais verboso. Ao usar `RUST_BACKTRACE=full` temos o seguinte:

```
$ RUST_BACKTRACE=full cargo run --quiet
warning: unreachable statement
 --> src/main.rs:4:5
  |
2 |     panic!("PALMA PALMA, NÃO PRIEMOS CANICO");
  |     ----------------------------------------- any code follo
wing this expression is unreachable
3 |
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
4 |     println!("Esse código não será usado!");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ unreachable stat
ement
  |
  = note: `#[warn(unreachable_code)]` on by default
  = note: this warning originates in a macro (in Nightly builds,
run with -Z macro-backtrace for more info)

warning: 1 warning emitted

thread 'main' panicked at 'PALMA PALMA, NÃO PRIEMOS CANICO', src/
main.rs:2:5
stack backtrace:
   0:     0x55eb9c69a760 - std::backtrace_rs::backtrace::libunwin
d::trace::hb4de9797f80b7b8c
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/../../backtrace/src/backtrace/li
bunwind.rs:90:5
   1:     0x55eb9c69a760 - std::backtrace_rs::backtrace::trace_un
synchronized::h59566d0bd20efff7
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/../../backtrace/src/backtrace/mo
d.rs:66:5
   2:     0x55eb9c69a760 - std::sys_common::backtrace::_print_fmt
::hcfa5ce6be50275d4
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/sys_common/backtrace.rs:67:5
   3:     0x55eb9c69a760 - <std::sys_common::backtrace::_print::D
isplayBacktrace as core::fmt::Display>::fmt::hd8c307a38b9bab04
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/sys_common/backtrace.rs:46:22
   4:     0x55eb9c6b105c - core::fmt::write::h3868db8542c90941
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/core/src/fmt/mod.rs:1096:17
   5:     0x55eb9c698942 - std::io::Write::write_fmt::hab90295d5a
0f197d
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/io/mod.rs:1568:15
   6:     0x55eb9c69c3b5 - std::sys_common::backtrace::_print::h1
9224910b8700cf1
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/sys_common/backtrace.rs:49:5
   7:     0x55eb9c69c3b5 - std::sys_common::backtrace::print::h25
dedfa68c5fb84a
                               at /rustc/07194ffcd25b0871ce560b9f
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
702e52db27ac9f77/library/std/src/sys_common/backtrace.rs:36:9
   8:     0x55eb9c69c3b5 - std::panicking::default_hook::{{closur
e}}::h12c5765653a72a42
                                at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/panicking.rs:208:50
   9:     0x55eb9c69bf13 - std::panicking::default_hook::he65cee7
1c4209f0c
                                at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/panicking.rs:225:9
  10:     0x55eb9c69cb51 - std::panicking::rust_panic_with_hook::
h01a674d863fe0d8a
                                at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/panicking.rs:591:17
  11:     0x55eb9c68381a - std::panicking::begin_panic::{{closure
}}::h10b423b7fd0a7994
                                at /[caminho da versão do toolchai
n do Rust]/lib/rustlib/src/rust/library/std/src/panicking.rs:520:
9
  12:     0x55eb9c6841f8 - std::sys_common::backtrace::__rust_end
_short_backtrace::hc83e2c5a76db6ea3
                                at /[caminho da versão do toolchai
n do Rust]/lib/rustlib/src/rust/library/std/src/sys_common/backtr
ace.rs:141:18
  13:     0x55eb9c683757 - std::panicking::begin_panic::h44199828
911c2788
                                at /[caminho da versão do toolchai
n do Rust]/lib/rustlib/src/rust/library/std/src/panicking.rs:519:
12
  14:     0x55eb9c6831bc - macro_panic::main::hb5c03c2b8e3f950e
                                at /[caminho da versão do toolchai
n do Rust]/examples/08/macro_panic/src/main.rs:2:5
  15:     0x55eb9c68437b - core::ops::function::FnOnce::call_once
::h8b3db8c6a7559e7d
                                at /[caminho da versão do toolchai
n do Rust]/lib/rustlib/src/rust/library/core/src/ops/function.rs:
227:5
  16:     0x55eb9c68422e - std::sys_common::backtrace::__rust_beg
in_short_backtrace::hbc3c34f110665287
                                at /[caminho da versão do toolchai
n do Rust]/lib/rustlib/src/rust/library/std/src/sys_common/backtr
ace.rs:125:18
  17:     0x55eb9c683f11 - std::rt::lang_start::{{closure}}::hd76
f60a2ab2aef71
                                at /[caminho da versão do toolchai
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
n do Rust]/lib/rustlib/src/rust/library/std/src/rt.rs:66:18
  18:     0x55eb9c69cf77 - core::ops::function::impls::<impl core
::ops::function::FnOnce<A> for &F>::call_once::h2c4fd7d4128112a8
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/core/src/ops/function.rs:259:13
  19:     0x55eb9c69cf77 - std::panicking::try::do_call::h41fb7db
bbe7bece6
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/panicking.rs:379:40
  20:     0x55eb9c69cf77 - std::panicking::try::hd38008ccba72bce1
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/panicking.rs:343:19
  21:     0x55eb9c69cf77 - std::panic::catch_unwind::h057e765cd60
f2d13
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/panic.rs:431:14
  22:     0x55eb9c69cf77 - std::rt::lang_start_internal::hdb9d0b9
d2cdcda8b
                               at /rustc/07194ffcd25b0871ce560b9f
702e52db27ac9f77/library/std/src/rt.rs:51:25
  23:     0x55eb9c683ee7 - std::rt::lang_start::hfef323e1bc0adae5
                               at /[caminho da versão do toolchai
n do Rust]/lib/rustlib/src/rust/library/std/src/rt.rs:65:5
  24:     0x55eb9c6831ea - main
  25:     0x7fac9e650b25 - __libc_start_main
  26:     0x55eb9c68308e - _start
  27:                0x0 - <unknown>
```

Essa versão é muito mais verbosa e mostra muita coisa interna do Rust. Nosso erro está ali na linha 14 do backtrace.

O backtrace pode ser muito útil para identificar problemas cotidianos, por exemplo, se o seu código fez um caminho inesperado na execução.

A macro `panic!` é importada para o seu ambiente no `prelude` e está disponível sempre que você precisar.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

**Curiosidade da edição 2021**

A macro `panic!` não era totalmente compatível com a *syntax* de formatação de string que a macro `println!` usa, sendo apenas parcialmente compatível. Isso causava algumas confusões para quem desenvolve.

A correção não podia ser feita em uma atualização simples do Rust porque isso geraria quebra de compatibilidade, portanto a mudança foi feita na edição 2021 da linguagem. Se você está usando uma versão nova da linguagem (como estamos fazendo neste livro), você pode assumir que a macro `panic!` usa a mesma formatação de string da macro `println!`.

## 11.2 MACROS DE ASSERÇÃO

Rust possui duas macros de asserção que verificam se algo é o esperado, retornando `true` se tudo ocorrer bem e `panic!` se houver algum problema. A primeira delas é o `assert!`, que verifica se uma expressão informada é verdadeira ou não. Veja o exemplo:

```
fn main() {
    assert!(2 + 2 == 8 / 2);
    assert!(true);
    assert!('a'.is_alphabetic());
    assert!('1'.is_numeric());
    assert!('a' == 'b');
}
```

E o resultado da execução será:


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
$ cargo run --quiet
thread 'main' panicked at 'assertion failed: \'a\' == \'b\'', src
/main.rs:6:5
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace
```

Todas as asserções passaram, exceto a nossa tentativa de comparar `'a'` com `'b'`. Perceba que a macro `assert!` pode receber qualquer código válido Rust, desde que ele seja passível de validação como um booleano, isto é, desde que o resultado final seja `true` ou `false`.

Outra macro de asserção é a `assert_eq!`, que recebe dois parâmetros e compara se eles são iguais. Se sim, a vida segue; caso contrário, recebemos um `panic!`.

```
fn main() {
    assert_eq!(9, 3 * 3);
    assert_eq!(true, 'a' == 'a');
    assert_eq!(false, 'a' == 'b');
}
```

No código apresentado, todas as asserções são verdadeiras, e não haverá saída alguma para apresentar. Ambas as macros também são importadas no prelúdio. Se alguma asserção falhar, o erro será apresentado, como a seguir.

```
fn main() {
    assert_eq!(false, 'a' == 'a');
}
```

```
$ cargo run --quiet
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `false`,
 right: `true`', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 11.3 DESEMPACOTAMENTO E O OPERADOR ?

Ao longo do livro, já vimos exceções estourando durante a execução do nosso código. Um exemplo seria a tentativa de parsear uma string para uma variável do tipo inteiro. Veja:

```
fn main() {
    let str: &'static str = "String is not a number";
    let num: i32 = str.parse();
    println!("{}", num);
}
```

Esse código vai estourar uma exceção ao tentarmos compilá-lo:

```
$ cargo build --quiet
error[E0308]: mismatched types
 --> src/main.rs:3:20
  |
3 |     let num: i32 = str.parse();
  |              ---   ^^^^^^^^^^^ expected `i32`, found enum `Re
sult`
  |              |
  |              expected due to this
  |
  = note: expected type `i32`
             found enum `Result<_, _>`

For more information about this error, try `rustc --explain E0308
`.
error: could not compile `macro_try` due to previous error
```

Por convenção, o Rust utiliza um sistema de **empacotamento** de retornos em dois tipos, `Option` e `Result`. Eles implementam um método `unwrap()` que provê o **desempacotamento** desse retorno, no valor esperado ou em uma exceção.

No exemplo apresentado, tentamos converter uma string para um inteiro. Entretanto, no mundo real, você pode deparar-se com


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

situações nas quais o dado a ser parseado deveria ser um inteiro, mas não é, por algum motivo (como uma entrada equivocada do usuário). O desempacotamento possibilita o tratamento da exceção no momento de execução, e não na compilação pelo `unwrap()`.

```
fn main() {
    let str: &'static str = "String is not a number";
    let num: i32 = str.parse().unwrap();
    println!("{}", num);
}
```

Esse código compila sem problemas, mas a sua execução ainda vai estourar uma exceção:

```
$ cargo run --quiet
thread 'main' panicked at 'called `Result::unwrap()` on an `Err`
value: ParseIntError { kind: InvalidDigit }', src/main.rs:3:32
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace
```

Isso ocorre porque é impossível parsear uma string para um inteiro. Porém, o uso do `unwrap()` diz ao Rust para tentar executar aquela operação: "se tudo correr bem, dê-me o retorno e siga em frente; caso contrário, mostre o problema e encerre".

Como dito, o `unwrap()` é um método dos tipos `Option` e `Result`. O primeiro é um tipo que possibilita termos algum resultado, inclusive nenhum, e o segundo viabiliza uma execução ou uma falha. Vamos explorar os dois, começando com o `Option`.

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

Ao olharmos o `Option`, percebemos que ele é um `enum` com duas opções possíveis: nada, representado por `None`; ou algo do



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

tipo `<T>`, representado por `Some<T>`. No exemplo a seguir, implementamos um código que gera uma sequência de Fibonacci, até 21.

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

impl Iterator for Fibonacci {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        let new_next = self.curr + self.next;

        self.curr = self.next;
        self.next = new_next;

        // Fibonacci é infinito, mas para fins
        // didáticos o nosso irá até 21
        if self.curr > 21 {
            None
        } else {
            Some(self.curr)
        }
    }
}

fn main() {
    let fib1 = Fibonacci { curr: 1, next: 1 };
    println!("Os primeiros 4 números da sequência Fibonacci são:
");
    for i in fib1.take(4) {
        println!("> {}", i);
    }

    let fib2 = Fibonacci { curr: 1, next: 1 };
    println!("Os próximos 4 números da sequência Fibonacci são: "
);
    for i in fib2.skip(4).take(4) {
        println!("> {}", i);
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Nesse código, o `None` é usado para finalizar nosso iterador. Perceba que é aceitável termos uma sequência que tem um fim e que, em um iterador, este término indica que a iteração acabou. Contudo, em uma situação na qual um resultado é esperado, o retorno de `None` vai disparar uma exceção. Veja o exemplo a seguir, em que validamos se um número passado é par ou ímpar. Se for par, devolvemos o algarismo; caso contrário, não retornamos nada.

```
fn even_test(number: i32) -> Option<i32> {
    if number % 2 == 0 {
        Some(number)
    } else {
        None
    }
}

fn main() {
    println!("Esse número é par: {}", even_test(22).unwrap());
}
```

Esse exemplo funciona corretamente, pois passamos um par para o nosso método `even_test` .

```
$ cargo run --quiet
Esse número é par: 22
```

Caso mudemos o código e passemos um número ímpar, o resultado será bem diferente.

```
$ cargo run --quiet
thread 'main' panicked at 'called `Option::unwrap()` on a `None`
value', src/main.rs:10:53
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace
```

Isso ocorre por não existir nada para desempacotar e, como dito anteriormente, o `unwrap()` faz com que Rust tente executar


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

aquela operação.

Além do `Option`, Rust provê o `Result`, que é similar a ele. A diferença é que `Result` pode retornar algo ou um erro em vez de algo ou nada.

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

Se você já fez algum tratamento de exceções com blocos com `begin`, `rescue` e afins, sabe que a vida pode ser bem difícil quando desejamos cercar todos os problemas que possam ocorrer em nosso sistema. Mas estamos codando em Rust, e Rust foi feito para ser seguro desde o começo. E é aí que entra o operador `?`, que é um substituto para a antiga macro `try!`.

Blocos do tipo `begin|rescue` obrigam o tratamento de cada uma das exceções que podem ocorrer. Se a linguagem de programação que você está usando utiliza um modelo de herança para as exceções, você pode subir um ou vários níveis e checar a exceção pai da árvore, como no exemplo a seguir, em Ruby.

```
def do_something!
  # ... Faz algo aqui ...
  something_succeeded
rescue Exception
  something_failed
end
```

O operador `?` lhe permite tratar um retorno com um `Result` da maneira mais simples. Veja o exemplo a seguir, adaptado a partir de um disponível na documentação oficial do Rust, em: https://doc.rust-lang.org/std/result/index.html.


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Neste exemplo, criamos uma estrutura chamada `Info`, que recebe dados de pessoas. Temos um método chamado `write_info` que escreve os dados de `Info` em um arquivo de texto e verifica se eles efetivamente foram escritos. Caso não seja possível, uma exceção é disparada. Primeiro, vamos verificar com o `if` cada escrita.

```
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    name: &'static str,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = match File::create("best_friends.txt") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };
    if let Err(e) = file.write_all(format!("nome: {}\n", info.nam
e).as_bytes()) {
        return Err(e);
    }

    if let Err(e) = file.write_all(format!("idade: {}\n", info.ag
e).as_bytes()) {
        return Err(e);
    }
    if let Err(e) = file.write_all(format!("avaliação: {}\n", inf
o.rating).as_bytes()) {
        return Err(e);
    }
    Ok(())
}

fn main() {
    let friend01 = Info {
        name: "Rachel Karen Green",
        age: 30,
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
        rating: 10,
    };
    match write_info(&friend01) {
        Err(e) => println!("Ops, algo deu errado -> {}", e),
        Ok(()) => println!("Tudo em seu devido lugar"),
    };
}
```

O resultado da execução será:

```
$ cargo run --quiet
Tudo em seu devido lugar
```

Nesse exemplo, verificamos se ocorreu uma exceção para cada ação. Em caso positivo, a retornamos; do contrário, seguimos para o próximo passo. Depois, tratamos o retorno com um `match`. Se nenhuma exceção disparou, imprimimos a mensagem de que está tudo certo.

Antes de falarmos sobre como o operador `?` pode facilitar a nossa vida nesse caso, vamos dar uma olhada na construção do `match` que trata o `Result`.

```
match write_info(&friend01) {
    Err(e) => println!("Ops, algo deu errado -> {}", e),
    Ok(()) => println!("Tudo em seu devido lugar"),
};
```

Nosso primeiro padrão de busca é o `Err(e)`, que é bem autoexplicativo. Se ocorreu um erro, ele será jogado no tipo `Err`, de `Result`, dentro da variável `e`. Já o padrão `Ok(())` pode ser um pouco estranho. Ele é o tipo `Ok` com uma tupla vazia, indicando que esperamos receber um `Ok` com qualquer informação dentro. Ele poderia ser escrito como a seguir, substituindo o `()` por um *catch-all* `_`.

```
match write_info(&friend01) {
    Err(e) => println!("Ops, algo deu errado -> {}", e),
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    Ok(_) => println!("Tudo em seu devido lugar"),
};
```

Agora, vamos reescrever nosso código com o operador `?` e ver como isso pode simplificar as coisas.

```
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    name: &'static str,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("best_friends.txt")?;
    // Early return on error
    file.write_all(format!("nome: {}\n", info.name).as_bytes())?;
    file.write_all(format!("idade: {}\n", info.age).as_bytes())?;
    file.write_all(format!("avaliação: {}\n", info.rating).as_byt
es())?;
    Ok(())
}

fn main() {
    let friend01 = Info {
        name: "Rachel Karen Green",
        age: 30,
        rating: 10,
    };
    match write_info(&friend01) {
        Err(e) => println!("Ops, algo deu errado -> {}", e),
        Ok(()) => println!("Tudo em seu devido lugar"),
    };
}
```

O resultado da execução será o mesmo, mas com menos linhas. Isso porque o operador `?` insere, em cada trecho do código, um `match` que faz o retorno da exceção para nós, sem precisarmos nos preocupar com isso.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 11.4 ESCREVENDO TESTES

Veremos agora como podemos usar o `cargo` para escrever testes para os nossos projetos. Vamos criar um projeto de biblioteca simples chamado *calculator*, que implementará quatro operações simples: *sum*, *subtract*, *multiply* e *divide*. Elas receberão dois parâmetros e retornarão o resultado.

Se você já estudou outras linguagens deve estar cansado de exemplos de testes que usam uma calculadora como exemplo, nós sabemos, mas este exemplo vai ser útil nesse momento para focamos mais no ecossistema de testes da linguagem, em vez de irmos a fundo no ato de testar. Vamos exemplificar e detalhar como lidamos com testes em Rust, utilizando o `cargo` .

Para criar um projeto do tipo biblioteca, use o comando `cargo new` , como a seguir.

```
$ cargo new --lib calculator
     Created library `calculator` project
```

Acesse a pasta `calculator` e execute o `cargo build` para que as dependências do projeto sejam adicionadas e as pastas de build, criadas.

```
$ cargo build --quiet
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── target
    ├── CACHEDIR.TAG
    └── debug
        ├── build
        ├── deps
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
│   ├── calculator-79abad71ef65d7ab.d
│   ├── libcalculator-79abad71ef65d7ab.rlib
│   └── libcalculator-79abad71ef65d7ab.rmeta
├── examples
├── incremental
│   └── calculator-39wuh2xkopthp
│       ├── s-fyishw35md-iq021a-1ud28uq4r9ql4
│       │   ├── 1iuy0drtde6sjtu0.o
│       │   ├── dep-graph.bin
│       │   ├── query-cache.bin
│       │   └── work-products.bin
│       └── s-fyishw35md-iq021a.lock
├── libcalculator.d
└── libcalculator.rlib

9 directories, 14 files
```

Abra agora o arquivo `lib.rs`, dentro da pasta `src`, e adicione o código do nosso projeto no começo do arquivo como a seguir. Repare que todos os nossos métodos são públicos, ou seja, acessíveis de fora, de qualquer lugar. Isso é feito com a declaração `pub fn` em vez de apenas `fn`.

```
pub fn sum(a: i32, b:i32) -> i32 {
    a + b
}

pub fn subtract(a: i32, b:i32) -> i32 {
    a - b
}

pub fn multiply(a: i32, b:i32) -> i32 {
    a * b
}

pub fn divide(a: i32, b:i32) -> i32 {
    a / b
}

// o resto do arquivo abaixo
```

A compilação deve ocorrer sem problemas; por enquanto não



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

precisamos nos preocupar com isso. Com nossa base de código, vamos agora ao que nos interessa: escrever testes. Você pode ter notado que o arquivo gerado veio com um código de exemplo para os testes, depois de adicionar o código acima, o arquivo `lib.rs` ficou assim:

```
pub fn sum(a: i32, b: i32) -> i32 {
    a + b
}

pub fn subtract(a: i32, b: i32) -> i32 {
    a - b
}

pub fn multiply(a: i32, b: i32) -> i32 {
    a * b
}

pub fn divide(a: i32, b: i32) -> i32 {
    a / b
}

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Os testes são geralmente criados dentro de um módulo e marcados com o atributo `#[cfg(test)]` . O `cfg` define compilação condicional, para que a declaração abaixo dele só seja compilada quando a configuração entre parênteses ( `test` , nesse caso) estiver ativa. Isso quer dizer que o módulo que contém os nossos testes só será compilado quando pedirmos para executar os testes e não estará disponível no binário final.

As funções de teste são marcadas com o atributo `#[test]` e



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

devem utilizar as macros de asserção exibidas anteriormente. No código de exemplo gerado, a macro usada é a `assert_eq!`, que verifica se dois valores são iguais.

Para executar os testes do projeto, basta utilizar o comando `cargo test`. Vamos usar a flag `--quiet` assim como fazemos com a compilação, só para deixar a saída do comando mais limpa para os exemplos do livro, mas ela não é necessária no dia a dia:

```
$ cargo test --quiet

running 1 test
.
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s


running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s
```

Vamos alterar o `test` gerado para que ele não passe, e ver o resultado da execução.

```
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 42);
    }
}
```

```
$ cargo test --quiet

running 1 test
F
failures:

---- tests::it_works stdout ----
thread 'tests::it_works' panicked at 'assertion failed: `(left ==
 right)`
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
  left: `4`,
 right: `42`', src/lib.rs:21:9
note: run with `RUST_BACKTRACE=1` environment variable to display
 a backtrace


failures:
    tests::it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
 filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

A saída do teste nos mostra onde ocorreu e qual foi o problema, como podemos ver a seguir. O teste falou porque o lado esquerdo e o lado direito do `assert_eq!` não são iguais, na linha 21 do nosso arquivo `lib.rs` .

```
  left: `4`,
 right: `42`', src/lib.rs:21:9
```

Os arquivos dentro da pasta `tests` são tratados como arquivos externos ao projeto, sendo assim, você precisa referenciar o `calculator` com `extern crate calculator;` no começo do arquivo de testes. Feito isso, vamos escrever os testes.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn sum_test() {
        assert_eq!(4, sum(2, 2));
        assert_eq!(10, sum(8, 2));
    }

    #[test]
    fn subtract_test() {
        assert_eq!(0, subtract(2, 2));
        assert_eq!(6, subtract(8, 2));
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
    }

    #[test]
    fn multiply_test() {
        assert_eq!(4, multiply(2, 2));
        assert_eq!(16, multiply(8, 2));
    }

    #[test]
    fn divide_test() {
        assert_eq!(1, divide(2, 2));
        assert_eq!(4, divide(8, 2));
    }
}
```

A diretiva `use super::*` traz todas as funções implementadas no escopo acima para o módulo de testes, No nosso caso, o `use super::*` vai deixar as funções `sum`, `subtract`, `multiply` e `divide` disponíveis para serem testadas. Ao executarmos os testes acima temos o seguinte resultado:

```
$ cargo test --quiet

running 4 tests
....
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s


running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s
```

Caso você deseje marcar um teste para não executar, utilize o atributo `#[ignore]`. Dessa forma, os testes que você não deseja rodar naquele momento são ignorados pelo `cargo`. Veja a seguir.

```
pub fn sum(a: i32, b: i32) -> i32 {
    a + b
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
}

pub fn subtract(a: i32, b: i32) -> i32 {
    a - b
}

pub fn multiply(a: i32, b: i32) -> i32 {
    a * b
}

pub fn divide(a: i32, b: i32) -> i32 {
    a / b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn sum_test() {
        assert_eq!(4, sum(2, 2));
        assert_eq!(10, sum(8, 2));
    }

    #[test]
    #[ignore]
    fn subtract_test() {
        assert_eq!(0, subtract(2, 2));
        assert_eq!(6, subtract(8, 2));
    }

    #[test]
    #[ignore]
    fn multiply_test() {
        assert_eq!(4, multiply(2, 2));
        assert_eq!(16, multiply(8, 2));
    }

    #[test]
    #[ignore]
    fn divide_test() {
        assert_eq!(1, divide(2, 2));
        assert_eq!(4, divide(8, 2));
    }
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Ao usar o `--quiet` os testes que falham são mostrados com `i` em vez de `.`. Vamos executar sem o `--quiet` dessa vez para ficar mais claro:

```
$ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests (target/debug/deps/calculator_with_tests_a
nd_ignore-d95686a1bf5ca4de)

running 4 tests
test tests::divide_test ... ignored
test tests::multiply_test ... ignored
test tests::subtract_test ... ignored
test tests::sum_test ... ok

test result: ok. 1 passed; 0 failed; 3 ignored; 0 measured; 0 fil
tered out; finished in 0.00s

   Doc-tests calculator_with_tests_and_ignore

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s
```

Um ponto importante sobre a execução de testes com o `cargo` é que eles são executados cada um em sua própria thread, o que faz a execução ficar muito mais rápida. Caso os testes não possam executar em paralelo por necessitar de algum recurso em comum (alterar o mesmo arquivo, por exemplo), esse comportamento pode ser modificado.

O comando `cargo run`, que usamos para executar os códigos durante quase todo o livro, compila o código e executa o binário final. O comando `cargo test` tem a mesma premissa, mas compila o código em modo de teste e executa o binário de teste depois. Podemos passar instruções para o `cargo test` em si,



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

como fazemos com o `--quiet` ou para o binário de teste que será gerado.

Para enviar instruções para o binário de testes, podemos usar `-- <instrução>`. Um exemplo disso é conseguir uma lista de instruções aceitas pelo binário de testes:

```
$ cargo test -- --help
```

Nessa lista podemos ver a instrução que limita o número de threads em execução:

```
$ cargo test --quiet -- --test-threads=1

running 4 tests
....
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s


running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 fil
tered out; finished in 0.00s
```

Veja que no exemplo acima utilizamos `--quiet` para o `cargo test` e `--test-threads=1` para o binário de testes.

## Conclusão

Neste capítulo, vimos como tratar os retornos `Option` e `Result`, que são encontrados por praticamente todos os cantos da linguagem Rust. Também conhecemos macros que nos permitem realizar asserções em nosso código, interromper a sua execução ou validar se ele funcionou como esperado quando foi executado.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Aprendemos também como criar o esqueleto básico de uma biblioteca Rust e como escrever testes para ela, bem como ignorar testes escritos e agrupá-los em contextos.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 12

# ALGUNS PONTEIROS PARA RUST

Desde o lançamento da primeira versão deste livro, temos recebido várias perguntas sobre a linguagem e o que dá para fazer com ela. Decidimos dedicar este capítulo para trazer algumas respostas e ponteiros para que quem está lendo possa encontrar bons caminhos e tirar dúvidas comuns sobre os próximos passos para aplicar os conhecimentos adquiridos aqui.

## 12.1 RUST PARA A WEB

Depois de ler sobre Rust e ver tanto material sobre o quanto Rust se preocupa com a forma que o software vai usar a memória, muitas pessoas ficam se perguntando se Rust funciona bem para desenvolvimento web, que é algo muito mais próximo da realidade delas.

Rust não possui frameworks web completos como o Rails (do Ruby), o Django (do Python), ou o Laravel (do PHP), mas provê diversas pequenas bibliotecas para você compor o seu software da forma que melhor se adapta a sua necessidade. Digamos que Rust está muito mais perto da linguagem Go do que de Ruby, Python ou PHP no quesito desenvolvimento web. Em outras palavras, as



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

ferramentas que Rust provê estão muito mais próximas de Sinatra (Ruby) ou Flask (Python) do que de grandes frameworks (Rails, Django, etc.).

Quem já fez APIs com Sinatra ou Flask vai se sentir em casa usando o Rocket (https://rocket.rs). Apesar de Rust precisar de muito mais cerimonias (tipagem forte, controle mais próximo de memória, borrow, ownership etc.), o código final ainda é bastante simples e enxuto. Veja este exemplo adaptado do site oficial do Rocket:

```
#[macro_use]
extern crate rocket;

#[get("/<name>/<age>")]
fn hello(name: &str, age: u8) -> String {
    format!("Olá, pessoa de {} anos chamada {}!", age, name)
}

#[launch]
fn rocket() -> _ {
    rocket::build().mount("/", routes![hello])
}
```

Rocket não é o único, existem vários microframeworks web para Rust. O site "Are we web yet?" (https://www.arewewebyet.org/) lista vários desses microframeworks e ferramentas disponíveis para quem está buscando uma solução rápida e escalável para desenvolvimento de APIs.

Se sua necessidade aceitar o uso de Webassembly e você estiver com disponibilidade para desbravar tecnologias pouco provadas no mercado, Rust também provê uma biblioteca chamada **Yew** (https://yew.rs/) para desenvolvimento frontend com Webassembly.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 12.2 RUST VS. GO

Mencionamos acima que Rust está muito mais próxima de Go para desenvolvimento web do que de outras linguagens como Ruby e Python. O fato de as duas linguagens possuírem uma intersecção de uso no mercado faz com que as pessoas que desenvolvem nessas linguagens criem uma certa rivalidade.

Rust e Go foram criadas para finalidades bastante distintas. Go é uma linguagem que foi criada para resolver um problema do Google de criar serviços web com uma sintaxe que seja fácil para pessoas no começo de carreira se familiarizar facilmente, mas ainda atender demanda de times e aplicações grandes, que precisam de performance de desenvolvimento e produção. Tem uma versão antiga (em inglês) de uma palestra do criador do Go, Rob Pike, que descreve um pouco sobre isso: https://talks.golang.org/2012/splash.article).

A linguagem Rust foi criada para resolver alguns problemas complicados que apareciam dentro do Firefox, principalmente os que eram relacionados com concorrência. Diferente de Go, os autores da linguagem Rust queriam manter as mesmas premissas que eles tinham com C++ e não ter custo em runtime, por isso a implementação de co-rotinas foi removida da linguagem antes da versão 1.0. Com o tempo, Rust evoluiu para se tornar a linguagem moderna para atender o mesmo propósito de coisas que antes precisavam da linguagem C.

Coisas que se resolvem com Go podem ser resolvidas com Rust. Algumas coisas que são resolvidas com Rust não necessariamente podem ser resolvidas com Go, dado o custo de runtime que Go traz junto. Em compensação, Go não é tão estrita


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

quanto ao código e deixa a pessoa que programa mais livre para fazer o código como bem entender, o que pode acelerar o processo de desenvolvimento sem custos adicionais para coisas pequenas.

A conclusão é que ambas linguagens atendem muito bem vários problemas em comum e cabe a você decidir qual delas deve ser escolhida. Nós, autores deste livro, temos a nossa preferência, como você deve imaginar.

## 12.3 RUST E WEBASSEMBLY

É bem comum ver menções de Rust quando se fala em Webassembly (wasm) e até mencionamos o Yew na seção de *Rust para a web* deste capítulo, dada sua popularidade.

Rust tem vários "targets", que são plataformas que a linguagem suporta e pode compilar para, e wasm é um desses targets. Claro que não é apenas mudar o target que qualquer código vai funcionar em qualquer plataforma, mas há suporte e o ferramental está cada vez melhor para que as mudanças sejam mais fáceis.

> **Curiosidade sobre os targets**
>
> Existem dezenas de targets com diferentes níveis de suporte. Tudo isso está documentado aqui: https://doc.rust-lang.org/nightly/rustc/platform-support.html

Rust é uma escolha popular para fazer coisas com wasm por ser uma linguagem que não exige um runtime (Garbage collector,



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

controle de green threads etc.), tem ferramental moderno, e já possui bom suporte para integração com wasm. Ferramentas como wasm-pack (https://github.com/rustwasm/wasm-pack) fazem o trabalho de escrever Rust e integrar com JavaScript via wasm ficar muito simples.

O exemplo básico, que também é mencionado na documentação do wasm-pack, é chamar um alert do navegador dentro do Rust. Para isso, precisamos criar um `cdylib` (lib dinâmica que pode ser linkada em outras coisas) usando `cargo new --lib` e adicionando `crate-type = ["cdylib"]` na seção `[lib]` do `Cargo.toml`).

Para chamar a função `alert`, o código usa o crate `wasm-bindgen` para trazer as referências para as funções para dentro do Rust. O código fica assim:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    pub fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Fique alerta via {}!", name));
}
```

Agora basta compilar o código para a web usando o wasm-pack:

```
$ wasm-pack build --target web
[INFO]: Checking for the Wasm target...
info: downloading component 'rust-std' for 'wasm32-unknown-unknow
n'
info: installing component 'rust-std' for 'wasm32-unknown-unknown
'
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

```
[INFO]: Compiling to Wasm...
    [...] Compiling dependencies
    Finished release [optimized] target(s) in 16.28s
[INFO]: Installing wasm-bindgen...
[INFO]: Optimizing wasm binaries with `wasm-opt`...
[INFO]: Optional fields missing from Cargo.toml: 'description', '
repository', and 'license'. These are not necessary, but recommen
ded
[INFO]: :-) Done in 39.82s
[INFO]: :-) Your wasm pkg is ready to publish at examples/10/aler
t_wasm/pkg.
```

O wasm-pack já faz o trabalho de preparar o target certo para compilação e usar o máximo possível de ferramentas para deixar pronto para uso.

Depois de executar o comando acima, já temos um arquivo JavaScript e outro arquivo `wasm` prontos para serem usados. Podemos usar o seguinte HTML para ver esse código funcionando:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>hello-wasm example</title>
  </head>
  <body>
    <script type="module">
      import init, {greet} from "./pkg/alert_wasm.js";
      init()
        .then(() => {
          greet("WebAssembly")
        });
      </script>
  </body>
</html>
```

*Lembre-se de ter um servidor local servindo o HTML para que as coisas funcionem corretamente (dica: `python -m http.server` vai resolver se você tiver Python 3 instalado).*



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

É possível e bastante viável compilar para wasm vindo de Rust.

## 12.4 RUST PARA SOFTWARE EMBARCADO

Quando falamos em desenvolvimento de software atualmente é normal vir à mente o desenvolvimento de aplicações web, afinal é um mercado enorme e que cresce a cada dia. Porém, o desenvolvimento de aplicações para web não é o único existente.

Temos o desenvolvimento de aplicações desktop e também o de aplicações embarcadas, ou seja, aquelas aplicações que estão dentro de algum hardware. Veja, quando você aciona o controle remoto de sua TV, dentro daquele bastão de plástico existe um processador e uma inteligência em formato de software. Essa inteligência sabe que, ao apertar o botão "ligar" sua TV, o controle remoto deve a informar a ela para sair do estado de repouso e começar a funcionar.

Você encontra software embarcado em praticamente tudo o que é tecnológico e está ao seu redor. O micro-ondas da cozinha? Tem software lá. O painel do elevador? Tem software lá. O teclado de seu computador? O repetidor Wi-Fi? O seu carro? O relógio digital da cozinha? O controle remoto do portão? Todos eles tem algum software lá. O nome é software *embarcado*, pois está gravado dentro daquele hardware. Ele fica ali, para sempre, e raramente é atualizado.

Ora, escrever software embarcado pode ser uma grande responsabilidade. Todo equipamento médico possui um software altamente complexo rodando ali, e esse software pode inclusive matar pessoas se não estiver funcionando corretamente. Um


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

exemplo clássico é o das máquinas de raio-x Therac-25, que por conta de um erro do tipo "condição de corrida" (que exploramos um pouco em Rust no capítulo 7) causou ferimentos graves ou a morte de várias pessoas entre os anos de 1985 e 1987.

Hoje as linguagens mais populares para o desenvolvimento de software são o C e o Assembly, por produzirem código performático e facilitar a comunicação entre o processador e periféricos (como o tecladinho do controle remoto ou o emissor de raio-x da Therac-25). Rust surge como uma ótima opção neste mercado por suas características de performance, uso de memória e concorrência segura.

Já existem diversos casos de uso de Rust, bem como extensa documentação sobre o assunto. Tudo isso está compilado no site https://www.rust-lang.org/what/embedded. Nesse link, além de encontrar diversos casos de uso de Rust embarcado em produção, você encontrará três livros online:

- *Discovery book*: um livro para iniciantes em desenvolvimento embarcado, com Rust.
- *The Embedded Rust book*: a versão do Rust Book para desenvolvimento embarcado.
- *The Embedonomicon*: a versão do Rustnomicon para Rust embarcado.

Vale ressaltar que o desenvolvimento embarcado tem diversas particularidades, como a necessidade de conhecimento de eletrônica, a dificuldade de debugar o software sem o uso de ferramentas como JTag (hardware específico para debugar aplicações rodando no bare metal) e também o tempo envolvido na criação e boot do projeto (afinal seu software rodará sozinho no


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

hardware, sem sistema operacional). É um mercado já estabelecido, desafiador e interessante. Certamente, Rust chega para ocupar um espaço enorme neste segmento.

## Cross compilation

Uma das características que possibilitam o uso de Rust para o desenvolvimento embarcado é a capacidade de compilar o código-fonte para diversos tipos de destino. Quando falamos em destino pense em diversas possibilidades, desde sistemas operacionais diversos até hardwares diversos.

Você pode, por exemplo, rodar o seu código-fonte em uma placa Raspberry alterando a forma como seu programa será compilado. O Raspberry utiliza um processador da família ARM 7, dessa forma tudo o que você precisa é adicionar um linker que vai montar o código compilado em formato ARM.

Caso esteja usando um Mac basta instalar o pacote de ferramentas `arm-linux-gnueabihf-binutils` .

```
$ brew install arm-linux-gnueabihf-binutils
$ rustup target add armv7-unknown-linux-musleabihf
```

Dentro de seu projeto, você diz ao `cargo` para ele utilizar esse linker, no arquivo `.cargo/config` :

```
[build]
target = "armv7-unknown-linux-musleabihf"
[target.armv7-unknown-linux-c]
linker = "arm-linux-gnueabihf-ld"
```

E ao rodar o `cargo build` seu código será compilado para a plataforma escolhida.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Para Windows a cross compilação deve ser feita com o pacote `stable-x86_64-pc-windows-gnu`, com a target `armv7-unknown-linux-gnueabihf`, e no Linux, basta adicionar a target `armv7-unknown-linux-gnueabihf`.

## Biblioteca padrão

Vale ressaltar que o Rust, por padrão, inclui acesso a sua biblioteca padrão em todo binário ou biblioteca para fazer o desenvolvimento ficar mais fácil. Essa biblioteca padrão assume que existe por debaixo dos panos um sistema operacional que vai fornecer funcionalidades padrão.

Isso não é o caso quando falamos de desenvolvimento embarcado. Veja, você está escrevendo um código que vai falar diretamente com o processador e ele não vai nos dar uma saída padrão ou um sistema de arquivos. Nesse tipo de ambiente não faz sentido termos uma biblioteca padrão em nosso projeto, pois o código sequer compilaria.

Para isso, o Rust provê algumas o atributo `#![no_std]` que deve ser usada de forma global em nosso projeto (o sinal de `!` após o sinal `#` indica que a diretiva é válida para todo o projeto). Essa diretiva diz ao Rust *não carregue a biblioteca padrão neste projeto*.

Outro ponto interessante é que, como não temos um sistema operacional, ninguém vai chamar o método `main` quando nosso código começar a executar, portanto, você terá que definir por onde o seu código começa. Porém, o compilador exige que o método `main` seja declarado quando estamos construindo um binário. Para isso temos que usar a diretiva `#![no_main]` e



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

informar o ponto de entrada de nosso projeto com a macro `entry_point!`, que é fornecida pelo crate `bootloader`.

Um código bem básico ficaria assim:

```
#![no_std]
#![no_main]
use bootloader::{entry_point, BootInfo};

entry_point!(kernel_main)

fn kernel_main(boot_info: &'static BootInfo) -> !
{
  // loop infinito
}
```



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

Capítulo 13

# O COMEÇO DE UMA JORNADA

Você já assistiu ao filme *Dr. Estranho*, da Marvel? Se não assistiu, deveria ver. Além de ser um dos super-heróis favoritos de um dos autores (o Castellani), tem umas sacadas muito interessantes sobre aprendizado.

Em determinado momento do filme, após ter passado por uma situação que obrigou um recomeço, Stephen Strange indaga a Anciã sobre como ele poderia aprender o que ela sabe, para tornar-se um mestre da magia.

Ela olha-o nos olhos e pergunta como ele havia se tornado um grande cirurgião. Então, Strange responde que foi com anos de estudo e prática. Realmente não existe outro jeito. Se você deseja trabalhar com Rust em seu dia a dia, apenas o estudo contínuo vai capacitá-lo para essa tarefa.

Neste capítulo, falaremos sobre outros recursos relacionados à linguagem Rust que você deveria experimentar para ficar cada vez melhor e ter mais produtividade ao utilizá-la, até que ela se torne a sua linguagem principal no cotidiano.



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

## 13.1 MATERIAL ONLINE

O principal material para aprender Rust é chamado de **The Book**, e está disponível em https://doc.rust-lang.org/book/. Ele cobre praticamente todos os aspectos da linguagem, alguns não abordados aqui.

O livro está em inglês e é um material extenso e, às vezes, um tanto cansativo, mas vale o tempo se você realmente se interessou pela linguagem e deseja se aprofundar mais. O material é dividido em sete capítulos e, ao menos, dois são obrigatórios.

O primeiro é o *Getting started*, uma introdução geral a linguagem Rust. O segundo é o *Tutorial: Guessing Game*, que apresenta conceitos básicos de Rust, mas sem muitos detalhes. Dê uma rápida passada por eles, estão bem no começo do livro e dão uma ótima introdução.

Depois disso o livro começa a entrar mais a fundo nas entranhas do Rust. Tentamos trazer aqui uma visão mais superficial, mas com uma narrativa mais fluida para que você possa ver o básico para começar a fazer software com Rust, mas o livro oficial tem uma visão bem mais completa e recomendamos a leitura agora que você já tem uma visão geral do básico.

Para quem quiser acompanhar o futuro do Rust, basta verificar o capítulo *Nightly Rust* nos apêndices. Esse capítulo fala sobre as novidades que estão em desenvolvimento e podem ser testadas em versões não estáveis da linguagem.

Além do *The Book*, a comunidade Rust mantém o já citado **Rustonomicon**, em https://doc.rust-lang.org/nomicon/. Logo no


E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

começo da página, você percebe seu objetivo, *As artes ocultas de programação insegura e avançada em Rust*. Abaixo, há uma pequena piada em forma de *disclaimer*, que diz algo nessa linha: *"Este documento provê conhecimento sem garantias e pode liberar horrores indescritíveis.*

Outro trabalho interessante e gratuito, em inglês, é o **Why Rust?**, escrito por Jim Blandy para a editora O'Reilly. Você pode encontrá-lo em https://www.oreilly.com/content/why-rust/. Sua leitura é rápida e sem sobressaltos; são 62 páginas nas quais o autor dá diversos motivos sólidos para usar a linguagem.

Além desses, o repositório do GitHub *Rust RFCs* (https://github.com/rust-lang/rfcs) conta com as definições de arquitetura do que já foi ou está sendo implementado para a linguagem, além de novas propostas para o futuro. Vale acompanhar os commits.

E, claro, a documentação da biblioteca padrão da linguagem, disponível em: https://doc.rust-lang.org/std/. Referência clara e concisa de cada um dos elementos que você usará no dia a dia.

## Conclusão

Mais importante do que começar a jornada é saber que ainda falta muito para chegarmos aonde desejamos, e que o ponto de chegada é apenas um novo ponto de partida. Você precisa definir aonde quer chegar com Rust, e se quer fazer dela mais uma ferramenta em sua grande caixa ou torná-la sua primeira opção no dia a dia.

Rust veio para ficar, tem crescido e se popularizado cada vez



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com

mais. Na primeira edição deste livro, essa conclusão tentava convencer o leitor ou leitora de que essa era uma linguagem em que valia apostar, mas hoje, mais de 4 anos depois, Rust já está bem consolidada no seu nicho. Grandes empresas estão investindo na linguagem, várias ótimas ferramentas estão sendo criadas, e até mesmo o kernel do Linux está aos poucos testando Rust como uma alternativa.

Se você leu o livro até aqui, nós esperamos ter cumprido nosso papel de mostrar o básico necessário para começar a fazer software em Rust e ter motivado você o suficiente a continuar a sua busca por mais conhecimento sobre a linguagem.

Nós nos colocamos à disposição para participar da sua jornada com Rust na medida do possível. Você pode encontrar nosso email na seção *Quem escreveu este livro?* ou falar conosco via redes sociais. Além de autores deste livro, somos entusiastas de Rust e participantes da comunidade. Vamos juntos!



E-book gerado especialmente para Gerson Barboza - barson.lixo@gmail.com