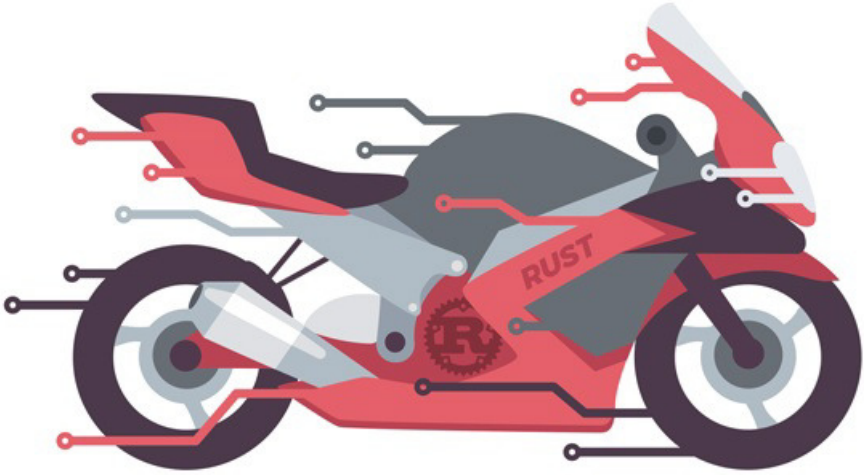
Rust

Concorrência e alta performance com segurança



Casa do Código

MARCELO CASTELLANI

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

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

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

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 faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-33-5

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.

DEDICATÓRIA

Este livro é dedicado a todos 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 todos que conseguem se divertir aprendendo algo novo, que sabem que o aprendizado de ontem pode não valer mais hoje, e que sintam 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.

AGRADECIMENTOS

Agradeço a todos da Casa do Código; aos envolvidos com a linguagem Rust, aqui e no mundo; a toda minha família, em especial a meus pais, Adelia e Nelson, e à Ana Luz, por me apoiarem e me deixarem concentrado enquanto escrevia aos finais de noite.

Agradeço também 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.

E, claro, agradeço a todos na Mozilla e ao Graydon, que trouxe essa belezinha chamada Rust ao mundo.

QUEM ESCREVEU ESTE LIVRO?

Marcelo F. Castellani é desenvolvedor de software há mais de vinte anos. Iniciou sua carreira 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. Hoje, é especialista em desenvolvimento de software na TOTVS, uma das maiores empresas de tecnologia do mundo.

Desde 2007, é desenvolvedor Ruby e Rails, tendo participado da fundação do GURU-SP e palestrado em diversos eventos relacionados, como *Ruby e Rails no mundo real*, *Sun Tech Days*, *RS on Rails* e *The Developers Conference*. 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*.

Você pode contatá-lo pelo e-mail marcelofc.rock@gmail.com para dúvidas sobre este livro ou outros assuntos relacionados.

A FUNÇÃO MAIN

Para a maioria dos programadores, a função `main` é comumente o ponto de entrada de um complexo e fascinante universo: o seu código. Seja ele 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 eu vou me apresentar e mostrar meus objetivos, dizer quem eu quero atingir e contar meus planos de dominação mundial.

Sim, dominação mundial. E não espero menos dessa nova e 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 entra no nicho de *Go*, a linguagem do Google, que visa aposentar clássicos como *C* e *C++* na construção de aplicativos complexos, como browsers ou sistemas operacionais.

De fato, duas das maiores aplicações hoje feitas puramente em Rust são um motor para browsers multiplataforma, o **Servo**, e um sistema operacional, o **Redox-OS**. Não sei qual é a sua experiência

como desenvolvedor, mas você deve ter noção de que essas aplicações são bem complexas de serem feitas. Exigem uma performance que não encontramos em queridinhos das comunidades, como linguagens interpretadas em geral ou aquelas baseadas em máquinas virtuais, que dependem de um sistema operacional ou de uma implementação do processador – como o *Java* e o seu uso em processadores com *Jazelle* e relacionados.

Rust permite a criação de programas que rodem no metal, criando código de máquina nativo. Entre as suas principais características, a mais importante é a 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).

Por si só, esse já seria um motivo enorme para termos um livro sobre Rust, mas acho que você precisa de mais para se convencer. Rust é a linguagem mais amada pelos desenvolvedores de software de todo o mundo, de acordo com uma pesquisa anual do Stack Overflow (veja o link disponível no capítulo *Primeiros passos*). Se amor não for motivo suficiente para lhe convencer, não sei 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, provavelmente a leitura deste material será complexa e desinteressante, mesmo com o autor tendo se esforçado muito para ser engraçado em algumas partes.

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-chave e tudo mais que outras linguagens também apresentam. Por ser compilada, ela possui todo o fluxo de codificar, compilar e executar, e todas as facilidades e dificuldades que esse modelo de programação suporta.

Se você vem de Java ou .NET, não apresentarei uma IDE bonitona, nem você verá exemplos envolvendo cliques em botões no Eclipse ou no Visual Studio. Este é um livro para programadores do estilo antigo, que usam **Vim**, **Emacs** ou um editor de textos desses por aí. O uso do terminal não é opcional.

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

faz uso de estruturas bem definidas de namespaces, assim como uma biblioteca padrão completa.

Diria que, como Ruby e Python, Rust já vem com as 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 escrito no **GNU Emacs 24.5.1** e compilado com o **Rustc 1.19.0**. A não ser que você já seja um usuário de Emacs, recomendo que use o editor de textos de sua preferência, com alguma extensão para destaque da sintaxe do Rust. Aprender Emacs é algo valioso, mas não é simples.

Esse editor, felizmente, já possui uma extensão desse tipo que funciona muito bem. E, repetindo um conselho que ouvi em meados dos anos 90: "aprenda Emacs e nunca mais precisará aprender outro editor".

Eu uso Linux há algum tempo, talvez por quase toda a sua existência e, para escrever este livro, usei um *laptop* bem simples com o Kubuntu instalado. Você não precisará de mais do que isto para rodar seu código: um editor de textos e as ferramentas do Rust em seu sistema. Claro, você vai precisar de um terminal como o *bash* ou o *Zsh*.

Rust é 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 citarei no momento oportuno.

Sumário

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. A linguagem me chamou atenção quando li em algum lugar que sua performance era comparável à do C++, porém ela é mais segura, concorrente e fácil de usar do que o seu predecessor. Trabalhei por muito tempo com C++ e posso 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".

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. Mas Rust foi desenvolvida pensando em evitar esse tipo de situação. Ela realmente foi feita para ser segura a ponto de possibilitar uma concorrência limpa sem a famigerada *race condition*.

Ela faz um gerenciamento de memória manual, que não usa coletor de lixo (*garbage collector*) e não deixa as amarras soltas. Assim, evita aquelas exceções de ponteiro nulo que nos fazem ligar para casa e avisar que a noite será longa.

De fato, 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. E isso, quando comparado ao desenvolvimento com C++, é como acordar em um paraíso em dia de sol.

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*.

Comecei a ler sobre Rust e seu ecossistema para ver onde a linguagem é usada, e conheci o projeto Servo (https://servo.org/), autointitulado um browser engine moderno e de alta performance para aplicações e uso em sistemas embarcados. Mantido pela Mozilla Research, o Servo é totalmente escrito em Rust e usa o

`cargo`, o gerenciador de pacotes do Rust, como ponto de partida. O Servo atualmente funciona em Linux, macOS, Windows, Android e no Firefox OS/Gonk.

Isso me 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 de estilo, imagens e outras mídias relacionadas, processá-las e desenhá-las na tela no lugar certo, e mantê-las lá quando você mover a barra de rolagem.

Enfim, é um projeto realmente complexo, todo feito em Rust. Não só complexo 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. E, novamente, todo feito em Rust. 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.

Comecei a desenvolver uma série de pequenos projetos apenas por diversão, usando Rust, e ficava mais animado a cada compilação bem-sucedida. Comecei a frequentar os fóruns e ler freneticamente sobre essa nova linguagem, com uma euforia que eu só havia sentido quando aprendi Ruby.

Neste primeiro capítulo, vou 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 de 2016, em uma pesquisa feita pelo Stack Overflow (https://stackoverflow.com/research/developer-survey-

2016#technology-most-loved-dreaded-and-wanted).

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 que escrevi apenas para entender como isso funciona. Muitas são de propósito geral e servem para você fazer praticamente qualquer coisa, e muitas são focadas, 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 possa ser usada para o desenvolvimento de ferramentas que exigem performance, 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 melhoras, desde seu surgimento em 1972 aos dias atuais. Temos Go e, claro, D, que possuem características bem parecidas com Rust. Mas todas ainda possuem algum ponto em

que são superadas pela Rust.

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 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 língua 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, quando Rust foi o queridinho da comunidade e não podia errar: qualquer problema que houvesse, Rust tinha uma solução para isso, ou terá 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 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.

Enfim, mesmo com pouco tempo de vida, em comparação com a C++ e a D, Rust tem feito muito barulho – 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, paralelo e robusto. 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 a 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".

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. Tudo na Rust 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 callbacks 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 alguns mortos pelo caminho, como Graydon Hoare, que em 30 de agosto de 2013 abandonou a liderança do projeto, com o seguinte e-mail na lista Rust-Dev:

Olá,

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

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

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

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 programadores 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 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 -sSf https://static.rust-lang.org/rustup.sh | sh
```

Sua senha de root é necessária. A saída será:

```
rustup: gpg available. signatures will be verified
rustup: downloading manifest for 'stable'
rustup: downloading toolchain for 'stable'
############################################################ 100.0%
(...)
rustup: installing toolchain for 'stable'
rustup: extracting installer
install: uninstalling component 'rustc'
install: uninstalling component 'rust-std-x86_64-apple-darwin'
install: uninstalling component 'rust-docs'
install: uninstalling component 'cargo'
install: creating uninstall script at
/usr/local/lib/rustlib/uninstall.sh
install: installing component 'rustc'
install: installing component 'rust-std-x86_64-apple-darwin'
install: installing component 'rust-docs'
install: installing component 'cargo'

    Rust is ready to roll.
```

Se tudo correr bem, basta pedir a versão do compilador:

```
$ rustc --version
```

```
rustc 1.19.0 (0ade33941 2017-07-17)

$ cargo --version
cargo 0.20.0 (a60d185c8 2017-07-13)
```

Instalando Rust no Microsoft Windows

Para Windows, existe um pacote padrão MSI que deve instalar tudo o que for necessário. Ele pode ser encontrado no endereço: https://www.rust-lang.org/pt-BR/downloads.html.

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. Tente identificar o botão `Run`, que fará seu código ser executado; o modo de compilação `Debug`, para exibir o máximo de informações possíveis sobre a compilação do código; e utilize a **versão estável** do Rust.

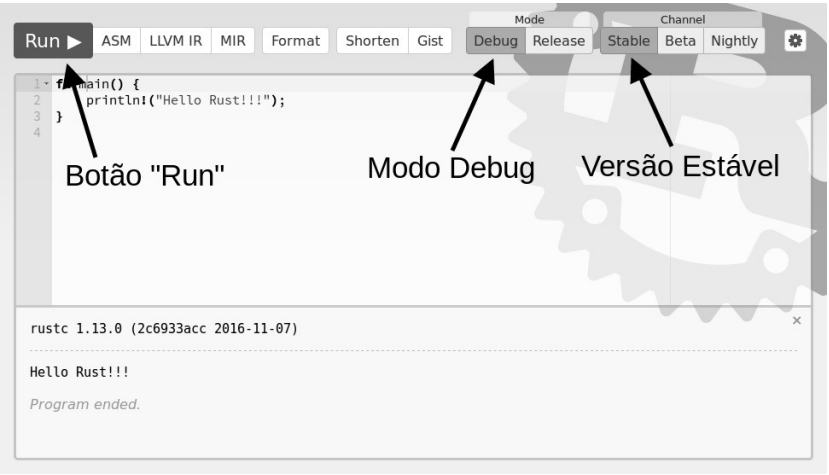


Figura 1.1: Os botões do Playground

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

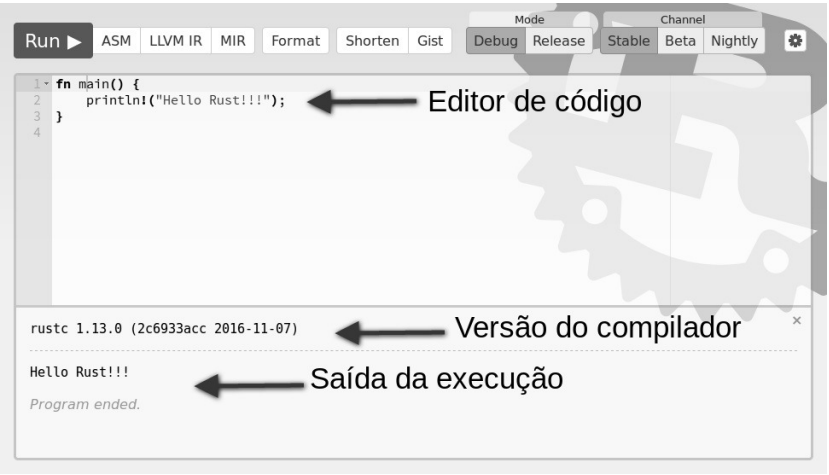


Figura 1.2: Editor, resultado da execução e mais

Apesar da facilidade do Rust Playground, recomendo que você utilize o compilador em seu console e um editor de textos como o **Atom** ou o **Sublime Text** (ou o **Emacs**, como disse anteriormente). 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, vou apresentar algumas das características que me fizeram sofrer quando comecei 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` e exibe-a 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 passível de ser executado pelo computador.

Eis o código:

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

Abra seu editor, digite o código passado e salve-o como `hello-world.rs` (`rs` é a extensão padrão de código-fonte Rust). Então, vamos compilá-lo:

```
$ rustc hello-world.rs
$ ls -al
total 644
drwxrwxr-x 2 marcelo marcelo   4096 Set 18 19:47 .
drwxrwxr-x 4 marcelo marcelo   4096 Set 18 19:44 ..
-rwxrwxr-x 1 marcelo marcelo 646656 Set 18 19:47 hello-world
-rw-rw-r-- 1 marcelo marcelo     44 Set 18 19:45 hello-world.rs
```

Tudo compilado com sucesso, é hora de executar:

```
$ ./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 --verbose
hello-world.rs:3:14: 3:22 error: expected a literal
hello-world.rs:3     println!(greeting);
                              ^~~~~~~~
<std macros>:1:33: 1:58 note: in this expansion of concat!
<std macros>:2:25: 2:56 note: in this expansion of format_args!
```

```
<std macros>:1:23: 1:60 note: in this expansion of print! (define
d in <std macros>)
hello-world.rs:3:5: 3:24 note: in this expansion of println! (def
ined in <std macros>)
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 objeto enviado para impressão seja um literal, para tornar seu código seguro. Sendo assim, 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);
}
```

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. 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 isto, seus ponteiros e referências por todos os lados. Rust torna referências algo simples e poderoso, acessível em qualquer lugar. Usaremos essa referência mutável futuramente.

Definimos nossa variável `name` como uma instância da classe `String`, criada através do `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, usamos o método `read_line` de `stdin()`, de `io`, como em:

```
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
hello-world.rs:7:5: 7:38 warning: unused result which must be
                    used, #[warn(unused_must_use)] on by
                    default
hello-world.rs:7    io::stdin().read_line(&mut name);
```

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
marcelo
Hello, marcelo
```

1.6 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 pesado, 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.

Capítulo 2

COMEÇANDO COM O CARGO

Agora que já demos uma primeira olhada em Rust, chegou a hora de entender alguns dos conceitos por trás da linguagem, 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 C.

Veja a seguir o exemplo de um programa *Hello World* feito em C pura. Ele simplesmente exibe a mensagem *Hello World* na saída padrão:

```
#include <stdio.h>

main()
{
    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 do programador, 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 da 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.

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

Empacotamento de código e reutilização

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 de código autocontido, possuem uma finalidade específica e podem ser adicionados ao seu projeto com uma declaração simples em um arquivo de manifesto.

2.2 CRATES, CARGO E OUTRAS FERRAMENTAS

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 "engradado", 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.

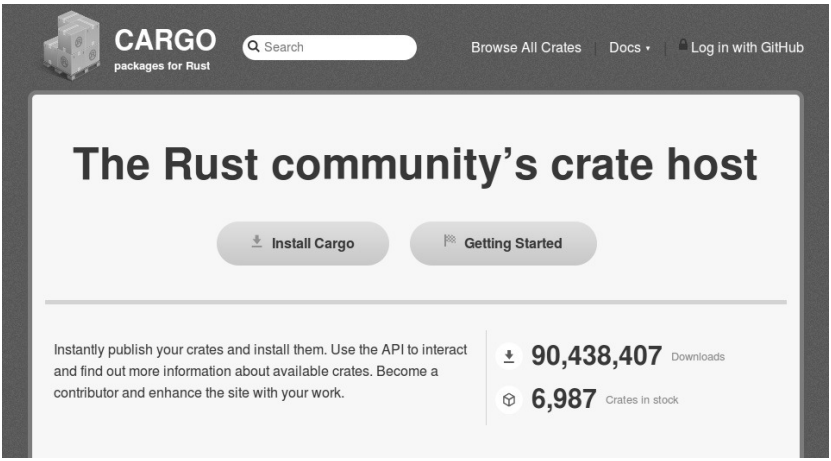


Figura 2.1: O site crates.io

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` project
```

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. Optei 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 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 = ["Marcelo Castellani <marcelofc.rock@gmail.com>"]

[dependencies]
```

Nessa versão bem básica, temos o *nome* de nosso projeto, a sua *versão* e um *array 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 encontradas em http://doc.crates.io/manifest.html.

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

```
fn main() {
    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 (file:...)
```

```
    Finished debug [unoptimized + debuginfo] target(s)
    in 1.68 secs
```

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 ou o `pom.xml`, do `Maven`.

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

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

7 directories, 4 files
```

Executando 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 isso, modifique seu código:

```
fn main() {
    println!("Hello, world from cargo run!");
```

```
}
```

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

```
$ cargo run
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/hello_world`
Hello, world from 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 from 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/
hello_world/
├── Cargo.lock
├── Cargo.toml
├── src
```

```
|   └── main.rs
└── target
    ├── debug
    |   ├── build
    |   ├── deps
    |   ├── examples
    |   ├── hello_world
    |   └── native
    └── release
        ├── build
        ├── deps
        ├── examples
        ├── hello_world
        └── native

12 directories, 5 files
```

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 from cargo run!
```

E o binário do diretório de `debug`:

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

Simples, não? Agora vamos inspecionar o conteúdo do nosso arquivo `Cargo.lock`:

```
[root]
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 = ["Marcelo Castellani - <marcelofc.rock@gmail.com>"]

[dependencies]
time = "0.1.12"
```

Adicionamos ao projeto a biblioteca `time`, com a versão 0.1.12. 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*.

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 http://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.

Por exemplo, se você especificar a versão **~1.2.3**, isso indica que qualquer versão maior ou igual a **1.2.3** e menor do que **1.3.0** é aceitável (veja que é verificado o incremento do minor). Se informar a versão **~1.2**, isso indica que qualquer versão maior ou igual a **1.2.0** e menor do que **1.3.0** é aceitável. Já ao indicar a versão **~1**, determina que qualquer versão maior ou igual a **1.0.0** e menor do que **2.0.0** é aceitável.

Ou seja, para **~1.2.3**, as versões **~1.2.3**, **~1.2.4**, **~1.2.5**, **~1.2.99**, entre outras, são aceitáveis. Quando definimos a versão do *patch* como um requisito de nosso projeto usando o ~ , será aceitável

qualquer variação dela sendo igual à mínima passada até o incremento da minor. O mesmo ocorre quando definimos a versão de minor, mas não definimos a versão de patch, ou a versão major, porém não a minor ou patch.

O acento circunflexo (`^`) ignora a minor e a patch, usando apenas o major como referência. Sendo assim, se você especificar a versão **^1.2.3**, indicará que qualquer versão maior ou igual a **1.2.3** e menor do que **2.0.0** é aceitável. Para **^1.2**, qualquer versão maior ou igual a **1.2.0** e menor do que **2.0.0** e, para **^1**, qualquer versão maior ou igual a **1.0.0** e menor do que **2.0.0**.

Voltando ao nosso projeto, 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 registry `https://github.com/rust-lang/crates.io-ind
ex`
 Downloading time v0.1.35
 Downloading libc v0.2.17
 Downloading kernel32-sys v0.2.2
 Downloading winapi v0.2.8
 Downloading winapi-build v0.1.1
   Compiling winapi v0.2.8
   Compiling libc v0.2.17
   Compiling winapi-build v0.1.1
   Compiling kernel32-sys v0.2.2
   Compiling time v0.1.35
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 2.53 secs
```

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 registry `https://github.com/rust-lang/crates.io-index`
```

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

```
Downloading time v0.1.35
Downloading libc v0.2.17
Downloading kernel32-sys v0.2.2
Downloading winapi v0.2.8
Downloading winapi-build v0.1.1
```

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

```
Compiling winapi v0.2.8
Compiling libc v0.2.17
Compiling winapi-build v0.1.1
Compiling kernel32-sys v0.2.2
Compiling time v0.1.35
```

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

```
Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 14.79 secs
```

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

```
hello_world/
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    └── debug
        ├── build
```

```
│   └── kernel32-sys-d6afa5bd3d7cfaef
│       ├── build-script-build
│       ├── out
│       └── output
├── deps
│   ├── libbuild-493a7b0628804707.rlib
│   ├── libkernel32-df86a08647459244.rlib
│   ├── liblibc-ad32fde1bd850538.rlib
│   ├── libtime-750bfdd52feafcb7.rlib
│   └── libwinapi-0889532d327ff4e2.rlib
├── examples
├── hello_world
└── native

9 directories, 11 files
```

Assim, precisamos editar nosso código para exibir a data atual, a partir do `crate time`. Para isso, vamos usar a função `now()`, que retorna uma estrutura chamada `Tm`.

Essa estrutura representa uma data, com os campos segundo, minuto, hora, dia e outros atributos separados, de forma a facilitar o tratamento. Cada um dos itens da estrutura é um inteiro de 32 bits.

```
pub struct Tm {
    pub tm_sec: i32,
    pub tm_min: i32,
    pub tm_hour: i32,
    pub tm_mday: i32,
    pub tm_mon: i32,
    pub tm_year: i32,
    pub tm_wday: i32,
    pub tm_yday: i32,
    pub tm_isdst: i32,
    pub tm_utcoff: i32,
    pub tm_nsec: i32,
}
```

Você pode encontrar mais detalhes sobre ela em https://doc.rust-lang.org/time/time/struct.Tm.html.

```
extern crate time;

fn main() {
    let d = time::now();
    println!("Today is {}/{}/{}", d.tm_mday,
             d.tm_mon, d.tm_year + 1900);
}
```

Na primeira linha de nosso código, referenciamos o uso do `crate time`:

```
extern crate time;
```

Depois, atribuímos a estrutura de data/hora a uma variável, usamos para pegar campo a campo e imprimir na tela. Repare que somamos 1900 ao ano, pois Rust traz no campo ano o número de anos decorridos a partir de 1900.

```
fn main() {
    let d = time::now();
    println!("Today is {}/{}/{}", d.tm_mday,
             d.tm_mon, d.tm_year + 1900);
}
```

O resultado pode ser visto a seguir:

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/hello_world`
Today is 24/9/2016
```

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

`time` também são adicionadas ao nosso projeto:

```
[root]
name = "hello_world"
version = "0.1.0"
dependencies = [
 "time 0.1.35 (registry+https://github.com/rust-lang/crates.io-in
dex)",
]

[[package]]
name = "kernel32-sys"
version = "0.2.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
dependencies = [
 "winapi 0.2.8 (registry+https://github.com/rust-lang/crates.io-i
ndex)",
 "winapi-build 0.1.1 (registry+https://github.com/rust-lang/crate
s.io-index)",
]

[[package]]
name = "libc"
version = "0.2.17"
source = "registry+https://github.com/rust-lang/crates.io-index"

[[package]]
name = "time"
version = "0.1.35"
source = "registry+https://github.com/rust-lang/crates.io-index"
dependencies = [
 "kernel32-sys 0.2.2 (registry+https://github.com/rust-lang/crate
s.io-index)",
 "libc 0.2.17 (registry+https://github.com/rust-lang/crates.io-in
dex)",
 "winapi 0.2.8 (registry+https://github.com/rust-lang/crates.io-i
ndex)",
]

[[package]]
name = "winapi"
version = "0.2.8"
source = "registry+https://github.com/rust-lang/crates.io-index"
```

```
[[package]]
name = "winapi-build"
version = "0.1.1"
source = "registry+https://github.com/rust-lang/crates.io-index"

[metadata] "checksum kernel32-sys 0.2.2
(registry+https://github.com/rust-lang/crates.io-index)" =
"7507624b29483431c0ba2d82aece8ca6cdba9382bff4ddd0f7490560c056098d
"

"checksum libc 0.2.17
(registry+https://github.com/rust-lang/crates.io-index)" =
"044d1360593a78f5c8e5e710beccdc24ab71d1f01bc19a29bcacdba22e8475d8
"

"checksum time 0.1.35
(registry+https://github.com/rust-lang/crates.io-index)" =
"3c7ec6d62a20df54e07ab3b78b9a3932972f4b7981de295563686849eb3989af
"

"checksum winapi 0.2.8
(registry+https://github.com/rust-lang/crates.io-index)" =
"167dc9d6949a9b857f3451275e911c3f44255842c1f7a76f33c55103a909087a
"

"checksum winapi-build 0.1.1
(registry+https://github.com/rust-lang/crates.io-index)" =
"2d315eee3b34aca4797b2da6b13ed88266e6d612562a0c46390af8299fc699bc
"
```

2.5 CONCLUSÃO

Neste capítulo, vimos como usar uma biblioteca `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.

Capítulo 3

MERGULHANDO NO OCEANO RUST

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

Antes, vamos falar rapidamente de um conceito importante que está disponível em toda a Rust: 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 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. Ele é carregado no início do `prelude`, que comentei no capítulo *Começando com o cargo*, e é formado por:

- Um conjunto para alocação de memória e gestão de ponteiros chamado `std::alloc`, que é onde está o `Box`.
- Um conjunto que provê diversas coleções otimizadas para uso, como árvores binárias, sequências de bytes (strings) e vetores, chamado `std::collections`.
- O `std::core`, no qual temos os tipos inteiros (`i8`, `u16` e por aí vai), os tipos de ponto flutuante, os tipos `Option` e `Result`, os traits `Default` e `Copy`, entre outros.
- O `std::libc`, que contém diversos *bindings* para a biblioteca padrão C, organizados por plataforma.
- O `std::unicode`, que contém diversas funções para a gestão de caracteres unicode, e estende o tipo básico `char`, disponível no `std::core`.

Uma observação, existe uma discussão na comunidade Rust sobre a existência do `std::alloc` como uma entidade separada. No momento em que escrevo este livro ele ainda é uma entidade

única, mas possivelmente ele passará a ser parte do `std::collections`, como pode ser visto em https://github.com/rust-lang/rust/issues/27783.

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 retorno da função `time::now()` a uma variável chamada `d`.

```
let d = time::now();
```

Rust possui um conceito chamado **inferência de tipos**, em que ele, no momento da atribuição, identifica que o retorno da função `time::now()` é uma instância de `Tm` e atribui o tipo correto (instância de `Tm`) à nossa variável `x`. Isso é importante, pois Rust é uma linguagem de tipos estáticos, ou seja, 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 `d`.

```
fn main() {
    let d = time::now();
    println!("Today is {}/{}/{}", d.tm_mday,
             d.tm_mon, d.tm_year + 1900);
```

```
    d = 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 `d` é do tipo `time::Tm`, e a atribuição do inteiro `2` espera que `d` seja do tipo `{integer}`.

```
Compiling hello_world v0.1.0 (file://...)
error[E0308]: mismatched types
 --> src/main.rs:8:9
  |
8 |     d = 2;
  |         ^ expected struct `time::Tm`,
  |           found integral variable
  |
  = note: expected type `time::Tm`
             found type `{integer}`

error: aborting due to previous error(s)
```

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 dia, uma com o mês e outra com o ano.

```
extern crate time;

fn main() {
    let d = time::now();
    let (day, month, year) = (d.tm_mday, d.tm_mon,
                              d.tm_year + 1900);
    println!("Today is {}/{}/{}", day,
             month, year);
}
```

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;
```

No caso anterior, 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 algum lugar, vai impedir a compilação com um erro.

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

```
fn main() {
    let d = time::now();
    let day:   i32 = d.tm_mday;
    let month: i32 = d.tm_mon;
    let year:  i32 = d.tm_year + 1900;

    println!("Today is {}/{}/{}", day,
             month, year);
}
```

Em Rust, isso é chamado **anotação de tipo**, e é particularmente interessante quando falamos de inteiros devido aos diversos formatos que eles podem assumir.

Declarando variáveis mutáveis

No capítulo *Primeiros passos*, vimos rapidamente como criar 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 = 20;
    a = 22;
}
```

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

```
$ cargo run
   Compiling hello_world v0.1.0 (file:...)
error[E0384]: re-assignment of immutable variable `a`
 --> src/main.rs:3:5
  |
2 |     let a = 20;
  |         - first assignment to `a`
3 |     a = 22;
  |     ^^^^^^ re-assignment of immutable variable

error: aborting due to previous error
```

Você pode dizer: *mas isso é uma péssima ideia, pois vou ter de 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`. Explícito sempre é melhor e mais seguro.

```
fn main() {
    let mut a = 20;
    a = 22;
}
```

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;

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

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

```
$ cargo build
   Compiling hello_world v0.1.0 (file:...)
warning: constant `y` should have an upper case name such as
`Y`, #[warn(non_upper_case_globals)] on by default
 --> src/main.rs:2:5
  |
2 |     const y: i32 = 3;
  |     ^^^^^^^^^^^^^^^^^

    Finished debug [unoptimized + debuginfo] target(s) in
    0.26 secs
```

Em nosso código que imprime o dia de hoje, temos de somar `1900` no ano gerado como explicado anteriormente. Vamos modificar nosso código para que este valor esteja dentro de uma constante.

```
extern crate time;

fn main() {
    const THE_1900: i32 = 1900;

    let d = time::now();
    let day:   i32 = d.tm_mday;
    let month: i32 = d.tm_mon;
    let year:  i32 = d.tm_year + THE_1900;

    println!("Today is {}/{}/{}", day,
```

```
            month, year);
}
```

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 dele basicamente é: chamar funções, pegar o resultado e utilizá-lo em novas chamadas de funções.

Declarando funções

Vamos modificar nosso código para que a impressão do dia de hoje ocorra dentro de uma função `print_today`, chamada em `main`.

```
extern crate time;

fn print_today() {
    const THE_1900: i32 = 1900;

    let d = time::now();
    let day:   i32 = d.tm_mday;
    let month: i32 = d.tm_mon;
    let year:  i32 = d.tm_year + THE_1900;

    println!("Today is {}/{}/{}", day,
             month, year);
}

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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/hello_world`
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 }
```

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

```
error[E0269]: not all control paths return a value
 --> src/main.rs:3:1
  |
3 | fn sum(a: i32, b: i32) -> i32 {
  | ^
  |
help: consider removing this semicolon:
```

```
 --> src/main.rs:4:10
  |
4 |     a + b;
  |          ^
```

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. Neste caso, faz-se necessário a remoção do `;` (ponto e vírgula), ou o compilador retorna o erro *not all control paths return a value*.

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;
    a + b
}

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

Execute o código e veja que a 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
```

```
   Compiling hello_world v0.1.0 (file:///...)
warning: unused variable: `a`,
#[warn(unused_variables)] on by default
 --> src/main.rs:3:8
  |
3 | fn sum(a: i32, b: i32) -> i32 {
  |        ^

warning: unused variable: `b`,
#[warn(unused_variables)] on by default
 --> src/main.rs:3:16
  |
3 | fn sum(a: i32, b: i32) -> i32 {
  |                ^

warning: unreachable expression,
#[warn(unreachable_code)] on by default
 --> src/main.rs:5:5
  |
5 |     a + b
  |     ^^^^^

    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running `target/debug/hello_world`
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. Isto 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:

```
extern crate time;

fn print_today() {
    const THE_1900: i32 = 1900;
```

```
    let d = time::now();
    let day:   i32 = d.tm_mday;
    let month: i32 = d.tm_mon;
    let year:  i32 = d.tm_year + THE_1900;

    println!("Today is {}/{}/{}", day,
             month, year);
}

fn do_the_things(function: fn()) {
    function()
}

fn main() {
    let pointer_to_function: fn() = print_today;
    do_the_things(pointer_to_function);
}
```

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

Em nossa função `main`, criamos uma variável chamada `pointer_to_function` que aponta para `print_today`. Na sequência, chamamos a função `do_the_things` 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 `do_the_things`, 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 também funciona:

```
extern crate time;

fn print_today() {
    const THE_1900: i32 = 1900;

    let d = time::now();
    let day:   i32 = d.tm_mday;
    let month: i32 = d.tm_mon;
    let year:  i32 = d.tm_year + THE_1900;

    println!("Today is {}/{}/{}", day,
             month, year);
}

fn do_the_things(function: fn()) {
    function()
}

fn main() {
    let pointer_to_function = print_today;
    do_the_things(pointer_to_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.

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!("{} is a digit? {}", a, a.is_digit(10));
    println!("{} is a binary? {}", a, a.is_digit(2));

    println!("{} is a digit? {}", b, b.is_digit(10));
    println!("{} is a binary? {}", b, b.is_digit(2));

    println!("{} is a digit? {}", c, c.is_digit(10));
    println!("{} is a binary? {}", 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
❤ is a digit? false
❤ is a binary? false
9 is a digit? true
9 is a binary? false
0 is a digit? true
0 is a binary? 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 uma instância de `EscapeUnicode`, que pode ser colocado em uma instância de `String` através do método `collect()`. Isso possibilita a interoperabilidade com sistemas que não suportam unicode, ou em protocolos de comunicação.

Vamos definir uma varável 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.46 secs
     Running `target/debug/rust_hello_world`
\u{2192}
```

Caso você não saiba o que é unicode ou como ele funciona por debaixo dos panos, este artigo da **Better Explained** 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.21 secs
     Running `target/debug/rust_hello_world`
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!("Uppercase -> {}", 'a'.is_uppercase());

    println!("Lowercase -> {}", 'a'.is_lowercase());

    println!("Whitespace -> {}", 'a'.is_whitespace());

    println!("Alphanumeric -> {}", 'a'.is_alphanumeric());

    println!("Numeric -> {}", 'a'.is_numeric());
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
```

```
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.21 secs
     Running `target/debug/rust_hello_world`
Uppercase -> false
Lowercase -> true
Whitespace -> false
Alphanumeric -> true
Numeric -> false
```

> A documentação completa do tipo `char` pode ser encontrada em 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 is true!");
    }
    if y {
        println!("Y is 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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/hello_world`
Y is true!
```

O `if` espera um booleano, diferente de linguagens como C, em que **0** corresponde a `false`, e o que for diferente de zero corresponde a `true`. O código a seguir não vai compilar, apesar de parecer válido para programadores 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
   Compiling hello_world v0.1.0 (file:...)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if a {
  |        ^ expected bool, found
  |          integral variable
  |
  = note: expected type `bool`
  = note:    found type `{integer}`

error: aborting due to previous error
```

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 (`!`).

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

> A documentação completa do tipo `bool` pode ser encontrada em 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 warning **warn(overflowing_literals)**, dizendo que nosso literal estoura o tamanho máximo que a variável suporta:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
warning: literal out of range for i32,
#[warn(overflowing_literals)] on by default
 --> src/main.rs:4:18
  |
4 |     let a: i32 = 4294967295;
  |                  ^^^^^^^^^^

    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/hello_world`
-1
```

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

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

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/hello_world`
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());
    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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/hello_world`
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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
Uns: 1
Zeros: 7
```

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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
>>: -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)
}
```

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. Como a definimos com 64 bits, ela exibe todos os dígitos que definimos.

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!("Floor: {}", a.floor());
    println!("Ceil: {}", a.ceil());
    println!("Round: {}", a.round());
    println!("Truncate: {}", a.trunc());
    println!("Fractional: {}", a.fract());

    println!("Is Finite?: {}", a.is_finite());
    println!("Is Infinite?: {}", a.is_infinite());
    println!("Is NaN?: {}", a.is_nan());
}
```

Esse código traz o seguinte resultado:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/rust_hello_world`
Floor: 3
Ceil: 4
Round: 4
Truncate: 3
Fractional: 0.54923606
Is Finite?: true
Is Infinite?: false
Is 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 = ["Marcelo", "Rust", "Castellani"];
    println!("Programando em {}, por {} {}",
             a[1], a[0], a[2]);
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running `target/debug/hello_world`
Programando em Rust, por Marcelo Castellani
```

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

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

Teremos um array imutável com cinco elementos, todos inicializados com o valor 0. Ele não difere em nada do inicializado item a item e 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 {}, por {} {}",
             a[1], a[0], a[2]);
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/hello_world`
Programando em 0, por 0 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
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/hello_world`
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.

```
fn main() {
    let mut a = [0; 5];
    a[1] = "Rust";

    println!("Programando em {}, por {} {}",
             a[1], a[0], a[2]);
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:...)
error[E0308]: mismatched types
 --> src/main.rs:3:12
  |
3 |     a[1] = "Rust";
  |            ^^^^^^ expected integral variable,
  |                    found reference
  |
  = note: expected type `{integer}`
  = note:    found type `&'static str`

error: aborting due to previous error

error: Could not compile `hello_world`.
```

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 `string`. Usaremos uma string vazia para isso, `""`.

```
fn main() {
    let mut a = [""; 5];
    a[0] = "Marcelo";
    a[1] = "Rust";
    a[2] = "Castellani";

    println!("Programando em {}, por {} {}",
             a[1], a[0], a[2]);
}
```

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`, uma cópia 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
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.29 secs
     Running `target/debug/hello_world`
a tem 10 elementos
b tem 10 elementos
c tem 2 elementos
```

Para percorrer os elementos de um array, Rust provê um método chamado `iter()`, que possibilita iterar por seus elementos dentro de um loop, pegando um a um. Para isso, usamos a construção `for x in array.iter()`, que vai executar o código entre chaves para cada elemento do array, colocando-o dentro da variável `x`.

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

    for x in a.iter() {
        println!("{}", x);
    }
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.41 secs
     Running `target/debug/rust_hello_world`
a
b
c
d
```

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 = (32, "integer");
let c = ("Nome", "Marcelo");
let d = (1, "Teste", 'c');
```

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

```
let a = (1, "Teste", 'c');
let a0 = a.0;
let a1 = a.1;
let a2 = a.2;
```

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

```
let a = (1, "Teste", 'c');
let (a0, a1, a2) = a;
```

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` e, ao colocarmo-las em um array, tudo funciona corretamente:

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

```
let c = (3, "tres");

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

Mas no próximo exemplo, temos três tuplas totalmente diferentes entre si: a primeira tem um inteiro e um ponto flutuante; a segunda, duas `strings` ; e a terceira, uma `string` 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
   Compiling hello_world v0.1.0 (file:...)
error[E0308]: mismatched types
 --> src/main.rs:6:17
  |
6 |     let d = [a, b, c];
  |                 ^ expected integral variable,
  |                   found &str
  |
  = note: expected type `({integer}, {float})`
  = note:    found type `(&str, &str)`

error[E0308]: mismatched types
 --> src/main.rs:6:20
  |
6 |     let d = [a, b, c];
  |                    ^ expected integral variable,
  |                      found &str
  |
  = note: expected type `({integer}, {float})`
  = note:    found type `(&str, char)`
```

```
error: aborting due to 2 previous errors
```

Temos dois erros do tipo **error[E0308]: mismatched types**, que indicam 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:

- A tupla `a`, cujo tipo é `(i32, f32)`;
- A tupla `b`, cujo tipo é `(str, str)`;
- A tupla `c`, cujo tipo é `(str, char)`.

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

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 criar um `enum` chamado `Gender`, no qual teremos uma lista de valores representando o sexo das pessoas. A seguir, criaremos uma estrutura chamada `Person`, que possui os atributos `name` e `gender`, em que `name` é uma string e `gender`, um item de nosso `enum Gender`.

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

```
enum Gender {
    Female,
    Male,
```

```
    Other
}

struct Person {
    name: &'static str,
    gender: Gender
}

fn main() {
    let nelson = Person {
        name: "Nelson Castellani",
        gender: Gender::Male
    };
    let adelia = Person {
        name: "Adelia Maria Fontes",
        gender: Gender::Female
    };
}
```

Ao ser compilado, esse código gerará diversos alertas, pois não estamos usando as variáveis, como pode ser visto no exemplo a seguir. Esses alertas podem ser ignorados por enquanto, já que o objetivo é apenas demonstrar como declaramos e usamos um `enum`.

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
warning: variant is never used: `Other`,
#[warn(dead_code)] on by default
 --> src/main.rs:4:5
  |
4 |     Other
  |     ^^^^^

warning: struct field is never used: `name`,
#[warn(dead_code)] on by default
 --> src/main.rs:8:5
  |
8 |     name: &'static str,
  |     ^^^^^^^^^^^^^^^^^^

warning: struct field is never used: `gender`,
```

```
#[warn(dead_code)] on by default
 --> src/main.rs:9:5
  |
9 |     gender: Gender
  |     ^^^^^^^^^^^^^^

warning: unused variable: `nelson`,
#[warn(unused_variables)] on by default
  --> src/main.rs:13:9
   |
13 |     let nelson = Person {
   |         ^^^^^^

warning: unused variable: `adelia`,
#[warn(unused_variables)] on by default
  --> src/main.rs:17:9
   |
17 |     let adelia = Person {
   |         ^^^^^^

    Finished debug [unoptimized + debuginfo] target(s)
    in 0.32 secs
     Running `target/debug/hello_world`
```

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
    }

    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
   Compiling rust_hello_world v0.1.0 (file:///...
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.23 secs
     Running `target/debug/rust_hello_world`
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
    }
}

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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
```

```
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 o próximo programador 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:

```
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 `ifs` 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 is 15"
    } else {
        "10 + 5 is not 15"
    };
    println!("{}", x);
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/hello_world`
10 + 5 is 15
```

No próximo exemplo, temos uma construção usando vários `ifs` 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!("Disapproved");
    } else if grade >= 4.9 && grade < 6.0 {
        println!("Exam");
    } else if grade >= 6.0 {
        println!("Approved");
    } else {
        println!("Invalid grade!!!!");
    }
}

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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/hello_world`
Disapproved
Disapproved
Exam
Approved
```

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 da 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!("Disapproved"),
        4.9...5.9 => println!("Exam"),
        6.0...10.0 => println!("Approved"),
```

```
        _ => println!("Invalid grade!!!!"),
    }
}

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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/hello_world`
Disapproved
Disapproved
Exam
Approved
```

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"),
        2 | 3 | 5 | 7 | 11 => println!("Prime"),
        _ => println!("Any number"),
    }
}

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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.29 secs
     Running `target/debug/hello_world`
Zero
Prime
Any number
```

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 sexo 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.

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 sexo *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!("Other"),
        Gender::Male => println!("Male"),
        Gender::Female => println!("Female")
    }
}
```

E o resultado da execução:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/hello_world`
Male
```

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!("Second is zero"),
        (0, x) => println!("First is zero"),
        _ => println!("No zeroes"),
    }
}

fn main() {
    check_tuple((0, 10));
```

```
    check_tuple((33, 0));
    check_tuple((8, 12));
}
```

E o resultado será:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/hello_world`
First is zero
Second is zero
No zeroes
```

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 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!("Second is zero"),
        (0, _x) => println!("First is zero"),
        _ => println!("No zeroes"),
    }
}

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() {
    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!("{} has {} vowel(s) and {} consonant(s)",
             name, vowel_count, consonant_count);
}
```

E o resultado de nossa execução será:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
```

```
    in 0.27 secs
     Running `target/debug/hello_world`
Marcelo has 3 vowel(s) and 4 consonant(s)
```

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 {
    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!("Now a is {}", a);
    }
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/hello_world`
Now a is 1
Now a is 2
Now a is 3
Now a is 4
Now a is 5
Now a is 6
Now a is 7
Now a is 8
Now a is 9
```

```
Now a is 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 for um programador 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 programadores que não são habituados 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 true`, que executará para sempre, e é muito empregado em sistemas embarcados. Veja o código:

```
fn main() {
    loop {
        println!("Running forever...");
    }
}
```

Ao executá-lo, você verá a mensagem *Running forever...* 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!("Running forever...");
        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!("Running forever...");
        a += 1;
    }
}
```

3.10 FOR

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 de dados separados por pontos. Em

Rust, temos dois tipos de `range`, os inclusivos e os exclusivos.

Os `ranges` exclusivos diferem-se de 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.

Vamos escrever o código que imprime os números de 1 a 10, um por vez. Para isso, utilizaremos o `for`.

```
fn main() {
    for a in 1..11 {
        println!("Now a is {}", a);
    }
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/hello_world`
Now a is 1
Now a is 2
Now a is 3
Now a is 4
Now a is 5
Now a is 6
Now a is 7
Now a is 8
Now a is 9
Now a is 10
```

É importante ressaltar que, atualmente, o `for` não suporta `ranges` inclusivos. Isso quer dizer que o código a seguir não compila.

```
fn main() {
    for a in 1...11 {
        println!("Now a is {}", a);
    }
}
```

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
error: inclusive range syntax is experimental
(see issue #28237)
 --> src/main.rs:2:14
  |
2 |     for a in 1...11 {
  |              ^^^^^^

error: aborting due to previous error
```

3.11 CONCLUSÃO

Neste capítulo, abordamos mais conceitualmente os tipos de dados básicos disponíveis no `crate std` da Rust, bem como outras características da linguagem, como comentários e módulos. Além disso, vimos os operadores condicionais e de repetição da 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.

Capítulo 4

TRAITS E ESTRUTURAS

Interfaces são um dos conceitos da linguagem de programação Java que mais gosto. 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;
```

```
    // 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 simularão 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;
```

```
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 salvará 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
    }
```

```
    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
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
Bem-vindo usuário Corleone
Usuário do tipo ADMIN entrou no sistema
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 no 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"
    }

    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");

    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
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
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

É possível utilizar traits do Rust pela diretiva `derive`, que funciona como uma espécie de implementação do trait em seu código atual. Isso possibilita adicionar mecanismos poderosos e funcionais a suas classes, como ordenação, inspeção de estruturas, igualdade e outros, sem muito esforço.

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 numéricos.

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 no sistema
    fn logout(&self) -> &'static str;

    // verificar se está logado
    fn is_logged_in(&self) -> bool {
        false
    }
}

#[derive(Debug)]
struct Admin { 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"
    }

    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
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
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.

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**.

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, 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**.

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 pode ser pelo `!=` ou o `ne` (*not equal*).

Cada tipo possui a 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));

    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 = "Castellani";
    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
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
```

```
s
     Running `target/debug/rust_hello_world`
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

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 este 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 meus, ela ainda não será o autor 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() {
    let a = MyStruct { member: 1 };
    let b = MyStruct { member: 1 };

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

Veja a execução:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.30 secs
     Running `target/debug/hello_world`
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,
    Hardback,
    Ebook
}

struct Book {
    isbn: i32,
    title: &'static str,
    format: BookFormat
}

impl PartialEq for Book {
    fn eq(&self, other: &Book) -> bool {
        self.isbn == other.isbn
    }
}

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::Hardback
    };
```

```
    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 o resultado da execução:

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/hello_world`
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. Sendo assim, dois valores `NaN` nunca serão iguais (`NaN != NaN`).

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
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
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 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() {
    let p1 = Person {
        age: 6,
        name: "Nicolas"
    };
    let p2 = Person {
        age: 31,
        name: "Ana"
    };
    let p3 = Person {
        age: 40,
        name: "Marcelo"
    };

    older(p1, p2);
```

```
    older(p2, p3);
    older(p1, p3);
}
```

Veja o resultado dessa execução:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.34 secs
     Running `target/debug/hello_world`
Ana tem mais anos de vida do que Nicolas
Marcelo tem mais anos de vida do que Ana
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. 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.

Gerenciamento de memória

O gerenciamento de memória da 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 termos 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.

Se você remover os traits `Copy` e `Clone` da linha `#[derive` e compilar o código, verá uma explicação detalhada disso:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
error[E0382]: use of moved value: `p2`
  --> src/main.rs:40:11
   |
39 |     older(p1, p2);
   |               -- value moved here
40 |     older(p2, p3);
   |           ^^ value used here after move
   |
   = note: move occurs because `p2` has type `Person`,
     which does not implement the `Copy` trait
```

```
error[E0382]: use of moved value: `p1`
  --> src/main.rs:41:11
   |
39 |     older(p1, p2);
   |           -- value moved here
40 |     older(p2, p3);
41 |     older(p1, p3);
   |           ^^ value used here after move
   |
   = note: move occurs because `p1` has type `Person`,
     which does not implement the `Copy` trait

error[E0382]: use of moved value: `p3`
  --> src/main.rs:41:15
   |
40 |     older(p2, p3);
   |               -- value moved here
41 |     older(p1, p3);
   |               ^^ value used here after move
   |
   = note: move occurs because `p3` has type `Person`,
     which does not implement the `Copy` trait
```

Dessa forma, precisamos do trait `Copy`, que manterá nossos dados a salvo em `p1`, utilizando a cópia dos dados nas operações em vez de movê-los. E o trait `Copy` depende do trait `Clone`, pois todo objeto que pode ser copiado também pode ser clonado, por definição.

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	Add
a - b	Sub

Operação	Trait
-a	Neg
a * b	Mul
a / b	Div
a % b	Rem
!a	Not
a & b	BitAnd
a ^ b	BitXor
a << b	Shl
a >> b	Shr

Com exceção do trait `Rem`, todos os outros são facilmente 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);
}
```

Veja a execução:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.97 secs
     Running `target/debug/hello_world`
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 com 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
    }
}

fn main() {
    let p1 = Person { name: "Marcelo", age: 39 };
    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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running `target/debug/hello_world`
A nova idade de Marcelo é 49
```

4.5 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.

CAPÍTULO 5

VETORES, STRINGS E TIPOS GENÉRICOS

Este capítulo será dedicado a estruturas mais complexas, como vetores, strings e tipos genéricos em Rust. Tais estruturas facilitam o desenvolvimento de projetos, pois possuem diversos e poderosos métodos para manipulação de dados.

5.1 VETORES

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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
```

```
     Running `target/debug/rust_hello_world`
[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 vetor com cinco zeros. Veja:

```
fn main() {
    let vector = vec![0; 5];

    println!("{:?}", vector);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.29 secs
     Running `target/debug/rust_hello_world`
[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 de 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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];
    println!("{:?}", agenda);
}
```

Ao compilarmos, teremos:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.27 secs
     Running `target/debug/rust_hello_world`
[Contact { name: "Somebody", phone_number: "+55(11) 9.1234.5678"
},
 Contact { name: "Someone", 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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
```

```
    };

    let contact_2 = Contact {
        name: "Someone",
        phone_number: "+55(11) 9.8765.4321"
    };

    let mut agenda = vec![contact_1, contact_2];
    println!("Agenda with 2 contacts \n {:?}", agenda);

    let contact_3 = Contact {
        name: "Nobody",
        phone_number: "+55(11) 9.9999.0000"
    };

    agenda.push(contact_3);
    println!("\nAgenda with 3 contacts \n {:?}", agenda);
}
```

O resultado é:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/rust_hello_world`
Agenda with 2 contacts
 [Contact { name: "Somebody", phone_number: "+55(11) 9.1234.5678"
 },
  Contact { name: "Someone", phone_number: "+55(11) 9.8765.4321"
}]

Agenda with 3 contacts
 [Contact { name: "Somebody", phone_number: "+55(11) 9.1234.5678"
 },
  Contact { name: "Someone", 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 é retornado e, como em Rust você não pode simplesmente descartar o retorno, 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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        phone_number: "+55(11) 9.8765.4321"
    };

    let mut agenda = vec![contact_1, contact_2];
    println!("Agenda with 2 contacts \n {:?}", agenda);

    let _ = agenda.pop();
    println!("\nAgenda with 1 contact \n {:?}", agenda);
}
```

O resultado é:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.29 secs
     Running `target/debug/rust_hello_world`
Agenda with 2 contacts
 [Contact { name: "Somebody", phone_number: "+55(11) 9.1234.5678"
 },
  Contact { name: "Someone", phone_number: "+55(11) 9.8765.4321"
}]

Agenda with 1 contact
 [Contact { name: "Somebody", 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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    println!("Contact 1: {:?}", agenda[0]);
    println!("Contact 2: {:?}", agenda[1]);
}
```

Com isso, temos:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running `target/debug/rust_hello_world`
Contact 1: Contact { name: "Somebody",
phone_number: "+55(11) 9.1234.5678" }
Contact 2: Contact { name: "Someone",
phone_number: "+55(11) 9.8765.4321" }
```

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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    println!("Contact 50: {:?}", agenda[50]);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.27 secs
     Running `target/debug/rust_hello_world`
thread 'main' panicked at 'index out of bounds: the len
is 2 but the index is 50', ../src/libcollections/vec.rs:1307
```

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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        phone_number: "+55(11) 9.8765.4321"
    };

    let agenda = vec![contact_1, contact_2];

    for contact in agenda {
        println!("{:?}", contact);
    }
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.27 secs
     Running `target/debug/rust_hello_world`
Contact: Contact { name: "Somebody", phone_number:
```

```
"+55(11) 9.1234.5678" }
Contact: Contact { name: "Someone", 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: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        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());
}
```

Map, blocos e collect

Um dos recursos dos iteradores de vetores 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 {
        name: "Somebody",
        phone_number: "+55(11) 9.1234.5678"
    };

    let contact_2 = Contact {
        name: "Someone",
        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!("Names: {:?}", names);

    let phone_numbers = agenda.iter()
        .map(|contact| { contact.phone_number })
        .collect::<Vec<_>>();

    println!("Phones: {:?}", phone_numbers);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.39 secs
     Running `target/debug/rust_hello_world`
Names: ["Somebody", "Someone"]
Phones: ["+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:

```
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,
    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: "Zenaide",
        phone_number: "+55(11) 9.1234.5678"
    };

    let c2 = Contact {
        name: "Alzira",
        phone_number: "+55(11) 9.8765.4321"
    };

    let c3 = Contact {
        name: "Marcelo",
        phone_number: "+55(11) 9.9999.8888"
    };

    let c4 = Contact {
        name: "Ana",
        phone_number: "+55(11) 9.4567.7654"
    };

    let mut agenda = vec![c1, c2, c3, c4];

    agenda.sort();

    println!("{:?}", agenda);
}
```

O resultado da execução será o vetor ordenado por nome:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.43 secs
     Running `target/debug/rust_hello_world`
Names:
[Contact { name: "Alzira", phone_number: "+55(11) 9.8765.4321" },
 Contact { name: "Ana", phone_number: "+55(11) 9.4567.7654" },
 Contact { name: "Marcelo", phone_number: "+55(11) 9.9999.8888" }
```

```
,
 Contact { name: "Zenaide", phone_number: "+55(11) 9.1234.5678" }
]
```

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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.31 secs
     Running `target/debug/rust_hello_world`
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, esta 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 uma parte do poder disponibilizado por essas estruturas. Para mais detalhes, consulte a documentação, em: https://doc.rust-lang.org/std/vec/.

5.2 STRINGS

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.

O crate `std::collection` provê um mecanismo poderoso para a manipulação de sequências de caracteres. Além do tipo `String`, ainda existe o trait `ToString`, que possibilita converter um tipo qualquer para um objeto `String`.

Você pode criar uma `String` utilizando o método `to_string()`, disponível no tipo básico `str`. Você também pode explicitar o tipo `String` no seu `let`, junto ao método `into()`, ou parsear uma constante com o construtor `from()`.

```
fn main() {
    let editora = "Casa do Código".to_string();
    let autor = String::from("Marcelo Castellani");
    let livro: String = "A linguagem Rust".into();

    println!("Este é o livro {} da {},\nescrito por {}",
             livro, editora, autor);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 3.27 secs
     Running `target/debug/rust_hello_world`
Este é o livro A linguagem Rust da Casa do Código,
escrito por Marcelo Castellani
```

Também é possível criar uma instância de `String` a partir de

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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/rust_hello_world`
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, obtido a partir de uma instância de `String` usando o sinal `&` (E comercial). Isso se chama coerção *Deref*. Veja-a a seguir.

```
fn main() {
    let editora = "Casa do Código".to_string();
    let autor = String::from("Marcelo Castellani");
    let livro: String = "A linguagem Rust".into();

    let mut sentence = String::from("Este é o livro ");
    sentence += &livro;
    sentence += " da ";
    sentence += &editora;
    sentence += ",\nescrito por ";
    sentence += &autor;

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

```
$ cargo run
```

```
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.31 secs
     Running `target/debug/rust_hello_world`
Este é o livro A linguagem Rust da Casa do Código,
escrito por Marcelo Castellani
```

Podemos ainda utilizar o `push_str()`, de funcionamento semelhante ao `+`. Veja o mesmo exemplo, agora apresentado com o `push_str()` no lugar do `+`.

```
fn main() {
    let editora = "Casa do Código".to_string();
    let autor = String::from("Marcelo Castellani");
    let livro: String = "A linguagem Rust".into();

    let mut sentence = String::from("Este é o livro ");
    sentence.push_str(&livro);
    sentence.push_str(" da ");
    sentence.push_str(&editora);
    sentence.push_str(",\nescrito por ");
    sentence.push_str(&autor);

    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("Hey ");
    sentence.push('Y');
    sentence.push('a');
    sentence.push('a');
    sentence.push('h');
    sentence.push('!');

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

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
```

```
    in 0.29 secs
     Running `target/debug/rust_hello_world`
Hey Yaah!
```

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 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/rust_hello_world`
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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/rust_hello_world`
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() {
    let mut a = String::from("Marcelo");
    a.reserve(20);

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

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running `target/debug/rust_hello_world`
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!("a capacity sem shrink: {}", a.capacity());
```

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

    a.shrink_to_fit();

    println!("a capacity com shrink: {}", a.capacity());
    println!("a len com shrink: {}", a.len());
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/rust_hello_world`
a capacity sem shrink: 14
a len sem shrink: 4
a capacity com shrink: 4
a 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 alocamos um espaço de armazenamento de mais cinquenta caracteres. 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.31 secs
     Running `target/debug/rust_hello_world`
a: 54 -> Rust rules
a: 11 -> Rust rules
a: 11 -> 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!("Before remove: {}", a);

    a.remove(2);
    println!("After remove: {}", a);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.26 secs
     Running `target/debug/rust_hello_world`
Before remove: Rust
After remove: 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!("No more chars..."),
        }
    }
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/rust_hello_world`
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)
    }

    for bte in a.bytes() {
        println!("Byte: {}", bte)
    }
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.27 secs
     Running `target/debug/rust_hello_world`
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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/rust_hello_world`
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.

Casting de String

Existem situações em que precisamos converter uma `String` para outros tipos de dados – transformar a `String` "14" no inteiro 14, 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, converto uma instância de `String` com o número `8` 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("8");

    let b = match a.parse::<i8>() {
        Ok(c) => c + 1,
        Err(_d) => 0,
    };
    println!("8 + 1 = {}", b);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.23 secs
     Running `target/debug/rust_hello_world`
8 + 1 = 9
```

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,
    };
    println!("8 + 1 = {}", b);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
8 + 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 emprego requer muita atenção. Veja o exemplo a seguir, que parece ser válido, mas não compila.

```
fn say_my_name(name: String) {
    println!("Your name is {}", name);
}

fn main() {
    let name = "Marcelo";
    say_my_name(name);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
error[E0308]: mismatched types
 --> src/main.rs:7:17
  |
7 |     say_my_name(name);
  |                 ^^^^ expected struct
```

```
  |                      `std::string::String`,
  |                      found &str
  |
  = note: expected type `std::string::String`
  = note:    found type `&str`

error: aborting due to previous error
```

Isso ocorre porque usamos o tipo literal `str`, e não uma instância de `String` – que era o esperado pelo método. Pode parecer que `"Marcelo"` 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!("Your name is {}", name);
}

fn main() {
    let name = "Marcelo".to_string();
    say_my_name(name);
}
```

Outra forma é trabalhar diretamente com uma referência à nossa string estática. 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!("Your name is {}", name);
}

fn main() {
    let name = "Marcelo";
    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 à Rust que devemos criar um **lifetime**, ou seja, que a variável emprestada deve existir dentro do nosso escopo atual, vindo de um escopo externo.

No desenvolvimento de sistemas embarcados, em que memória é um luxo, o uso de lifetimes é bastante útil. Eles evitam a cópia desnecessária do conteúdo de uma variável para uma nova, com o mesmo conteúdo da anterior. Em vez de ocorrer a sua duplicação em memória, temos uma nova referência ao conteúdo da variável passada como parâmetro.

Isso quer dizer que, durante toda a execução do código anterior, existe em memória apenas uma cópia e duas referências a `Marcelo`. Poderíamos modificar nosso código para que ele não criasse um lifetime, mas sim uma cópia duplicando o uso de memória, como a seguir.

```
fn say_my_name(name_inside_method: &str) {
    println!("Your name is {}",
             name_inside_method);
}

fn main() {
    let name = "Marcelo";
    say_my_name(name);
}
```

Contudo, em ambos os casos, estamos trabalhando com uma instância de `str`, que não apresenta todos os métodos e todo o poder que temos em `String`. Por isso, recomendo o uso de `to_string()` e de `String` sempre que possível.

Claro, se você estiver utilizando Rust para sistemas embarcados com pouca memória, 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, `0`.

```
fn main() {
    let name = String::from("Marcelo Castellani");
    let space = name.find(" ").unwrap_or(0);

    println!("Space at: {}", space);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.33 secs
     Running `target/debug/rust_hello_world`
Space at: 7
```

Você pode pensar: "mas o sétimo caractere é um "o", não um espaço em branco". Na verdade, strings em Rust iniciam como o índice `0`, sendo assim:

```
Marcelo Castellani
01234567
```

O espaço está na posição `7`. 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 *Marcelo*, 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 variável, a `first_name`. Para fazê-lo, usamos o `collect`, responsável por pegar o resultado de uma execução e retorná-lo. Nosso código ficará assim:

```
fn main() {
    let mut name = String::from("Marcelo Castellani");
    let space = name.find(" ").unwrap_or(0);

    let first_name: String = name.drain(..space).collect();

    println!("First name: {}", first_name);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 2.99 secs
```

```
     Running `target/debug/rust_hello_world`
First name: Marcelo
```

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("Marcelo Castellani");
    let space = name.find(" ").unwrap_or(0);

    let first_name: String = name.drain(..space).collect();

    println!("First name: {}", first_name);
    println!("Original name: {}", name);
}
```

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 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("Marcelo Castellani");
    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!("First name: {}", first_name);
    println!("Last name: {}", last_name);

    println!("Original string is empty? {}", name.is_empty());
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.45 secs
     Running `target/debug/rust_hello_world`
First name: Marcelo
Last name:  Castellani
Original string is empty? true
```

5.3 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.

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
}

fn main() {
    println!("{}", sum_i32(1, 2));
    println!("{}", sum_f32(1.3, 2.24));
```

```
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.21 secs
     Running `target/debug/rust_hello_world`
3
3.54
```

Funções genéricas

Vamos reescrever nosso exemplo de soma para uma forma mais genérica, na qual podemos reutilizar o código em qualquer situação. Para isso, 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 números, bem como suporte a métodos genéricos que não estão disponíveis nos tipos padrão do Rust.

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 <marcelofc.rock@gmail.com>"]

[dependencies]
num = "0.1"
```

Com o `num`, podemos criar um método `generic_sum` que possibilita somar um par de qualquer tipo numérico. Essa definição de tipo genérico é feita com o uso de `<T>`, em que `T` indica um tipo qualquer.

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 do 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.

```
extern crate num;

fn generic_sum<T: num::Num>(x: T, y: T) -> T {
    x.add(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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running `target/debug/rust_hello_world`
```

```
3
7
11
3.54
```

5.4 CONCLUSÃO

O uso de vetores e strings facilita muito o dia a dia de um desenvolvedor, pois ambos possuem métodos poderosos para a manipulação dos seus dados. Já o uso de tipos genéricos possibilita a criação de menos código para realizar mais, sendo outra forma de obter produtividade no desenvolvimento dos seus projetos.

No próximo capítulo, falaremos sobre macros, uma outra maneira de se escrever código genérico. Elas podem ser adaptadas para resolver duplicação de código, mas de forma mais poderosa do que usando tipos genéricos. Porém, antes, vamos falar sobre programação concorrente.

Capítulo 6

ALOCAÇÃO E GERENCIAMENTO DE MEMÓRIA

Neste capítulo, vamos falar sobre como a Rust gerencia memória. Esse é um ponto importante, pois ela faz um misto de coletor de lixo e gerenciamento manual de memória, que pode confundir programadores de outras linguagens.

Quem veio de linguagens como Ruby, Python ou Java está acostumado a usar instâncias de classes e não se preocupar em liberar espaço quando elas deixam de ser necessárias. Já programadores C e C++ sofrem com o uso de alocação e desalocação de memória. Aqui também aprenderemos sobre processamento paralelo e o uso de threads.

6.1 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 gerenciamento é feito de maneira eficiente por meio de uma predição de desalocação de recurso, que sabe o momento em que um objeto pode ser removido da memória.

Coletores de lixo trabalham geralmente com um 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 temos um em que o uso de memória é um fator crucial (como serviços rodando em servidores ou software embarcado), uma gestão eficiente de memória torna-se um fator crucial.

Rust tem um modelo bem definido de alocação e escopo de variáveis, ou seja, quando elas devem existir e quando devem deixar de existir. Isso possibilita que os objetos referenciados por elas vivam apenas o tempo necessário, sem fazer hora extra na RAM de sua máquina.

6.2 ESCOPO DE VARIÁVEIS

Quando falamos em escopo, falamos sobre o espaço onde um objeto 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 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
   Compiling rust_hello_world v0.1.0 (file:///...)
error[E0425]: unresolved name `c`
  --> src/main.rs:10:20
   |
10 |     println!("{}", c);
   |                    ^ did you mean `a`?

error: aborting due to previous error
```

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 método. Um objeto criado 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
   Compiling rust_hello_world v0.1.0 (file:///...)
error[E0425]: unresolved name `b`
 --> src/main.rs:7:20
  |
7 |     println!("{}", b);
  |                    ^ did you mean `a`?

error: aborting due to previous error
```

6.3 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 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 um *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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.21 secs
     Running `target/debug/rust_hello_world`
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);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.21 secs
     Running `target/debug/rust_hello_world`
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
```

6.4 PONTEIROS, BOX E DROP

Por padrão, qualquer instância de uma classe em Rust é alocada no bloco de memória conhecido como **stack** ou **pilha**. Esse bloco "empilha" os objetos na sequência em que eles são alocados, possibilitando que sejam desalocados quando a pilha estiver cheia, para dar lugar a outros objetos.

Uma pilha usa o conceito de **LIFO**, ou seja, *Last In, First Out* (o último a chegar é o primeiro a sair). Objetos são empilhados conforme instanciados, e desalocados 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:

```
| Pilha |
|-------|
| car_a |
```

Vamos adicionar mais um método em nosso código, que instancia duas variáveis:

```
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();
}
```

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() {
      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:

```
| Pilha |
|-------|
| car_a |
```

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 };
      let car_c = RaceCar { number: 8 };
  }

  fn main() {
      let car_a = RaceCar { number: 3 };
      two_cars();
  }
```

```
| Pilha |
|-------|
| car_b |
| car_a |
```

```
  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();
  }
```

```
| Pilha |
|-------|
| car_c |
| car_b |
| car_a |
```

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.

```
  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();
> }
```

```
| Pilha |
|-------|
| car_b |
| car_a |
```

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();
> }
```

```
| Pilha |
|-------|
| car_a |
```

Como você deve imaginar, após o fim da execução do programa, `car_a` também deixa de existir.

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 um objeto 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. Para alocar um objeto no heap em Rust, usamos o tipo `Box<T>`. Além disso, a alocação de um objeto no heap também instancia um objeto na pilha, o que chamamos de ponteiro.

Veja o exemplo a seguir:

```
fn main() {
    let a = 10;
    let b = Box::new(10);
}
```

Nossa pilha ficará assim:

```
| Pilha | Valor |
|-------|-------|
|   b   | xxxxx |
```

```
|   a   | 10   |
```

O valor `xxxxx` é um endereço de memória que aponta para onde nosso objeto, apontado por `b`, se encontra. Isso quer dizer que, dentro da nossa memória, existem dois objetos `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 objetos do heap é feita por meio de chamadas de métodos para sua alocação e desalocação.

Isso permite que você mate um objeto 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 na limpeza do objeto da memória. Esse trait possui um método denominado `drop()`, chamado quando um objeto é retirado da memória.

```
struct RaceCar {
    number: i32,
}

impl Drop for RaceCar {
    fn drop(&mut self) {
        println!("Car {} finished the run", 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, perceba que os objetos são desalocados do heap no esquema LIFO, pois eles estão referenciados em nossa pilha a partir de ponteiros.

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.21 secs
Car 8 finished the run
Car 5 finished the run
Car 3 finished the run
```

6.5 OUTROS TIPOS DE PONTEIROS

Um ponto importante é que Rust, em suas primeiras versões, possuía diversos tipos de ponteiros como os *smart pointers*. Eles foram removidos com a introdução do `Box`. É possível que você encontre em blogs e sites pela internet exemplos de código como:

```
fn f() {
    let x: ~int = ~1024;
    println(fmt!("%d", *x));
}
```

Declarações de variáveis como o sinal de `~` (til) indicavam o uso de *smart pointers*, que não estão mais disponíveis. Se tiver curiosidade sobre como isso funcionava antes do `Box`, leia o artigo de Patrick Walton, de março de 2013, no link: http://pcwalton.github.io/blog/2013/03/18/an-overview-of-memory-management-in-rust/.

Ele lhe dará uma dimensão de como o `Box` facilitou a vida do desenvolvedor, deixando o código mais simples de ler e mais

intuitivo.

6.6 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 objetos e variáveis.

Como diz a documentação oficial da Rust: "*Compartilhamento de um estado mutável é a raiz de todo o mal. A maior parte das linguagens tenta resolver esse problema a partir da parte mutável, mas Rust resolve isso tratando o lado do compartilhamento.*"

Esse modo particular de lidar com tal situação é relacionado à maneira como Rust gerencia a propriedade dos objetos. 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**.

RACE CONDITION

Uma race condition (ou condição de corrida) ocorre quando nosso código possui processamento paralelo dependente de threads, mas a continuidade de sua execução depende da sincronia dessas threads, algo que nunca é garantido.

6.7 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!("Hello thread a!");
    });

    let _b = thread::spawn(|| {
        println!("Hello thread B!");
    });
}
```

Esse exemplo funciona, mas não veremos nenhuma mensagem impressa na saída padrão.

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
```

```
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.35 secs
     Running `target/debug/rust_hello_world`
```

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!("Hello thread a!");
    });

    let b = thread::spawn(|| {
        println!("Hello 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
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
Hello thread B!
Hello thread a!

$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
Hello thread B!
Hello thread a!
```

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running `target/debug/rust_hello_world`
Hello thread a!
Hello thread B!
```

6.8 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();
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
error[E0373]: closure may outlive the current function,
but it borrows `value`, which is owned by the current
function

--> src/main.rs:6:27
  |
6 |     let a = thread::spawn(|| {
  |                           ^^ may outlive borrowed value
  |                              `value`
7 |         println!("{}", value);
  |                        ----- `value` is borrowed here
  |

help: to force the closure to take ownership of `value` (and
```

```
any other referenced variables), use the `move` keyword, as shown
:
  |     let a = thread::spawn(move || {

error: aborting due to previous error

error: Could not compile `rust_hello_world`.

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);
    });

    let _ = a.join();
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.35 secs
     Running `target/debug/rust_hello_world`
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!("Main thread {}", value);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.40 secs
     Running `target/debug/rust_hello_world`
Thread A 133
Thread B 11
Main thread 10

$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/rust_hello_world`
Thread B 11
Thread A 133
Main thread 10
```

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.

6.9 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 **Communicating Sequential Processes**, ou simplesmente **CSP**. Inicialmente elaborado por Tony Hoare, ele define boas práticas para especificação e verificação de aspectos de concorrência em diversos sistemas.

Um dos conceitos-chave 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`. Este 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`; esta apresentará uma quantidade predefinida de carros que participarão de uma corrida imaginária. Nosso programa vai imprimir na tela a mensagem *Car xx finished the race* – 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 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 {
    let thread_sender = sender.clone();

    thread::spawn(move || {
        let _ = thread_sender.send(id);

        println!("Car {} finished the race", 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!("Car {} finished the race", id);
        });
    }

    let mut ids = Vec::with_capacity(CARS as usize);
    for _ in 0..CARS {
        ids.push(receiver.recv());
    }

    println!("Final order -> {:?}", ids);
}
```

Algumas execuções do código mostram que nem todas as corridas são iguais.

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/rust_hello_world`
Car 0 finished the race
Car 2 finished the race
Car 1 finished the race
Car 4 finished the race
Car 3 finished the race
Final order -> [Ok(0), Ok(2), Ok(1), Ok(4), Ok(3)]

$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/rust_hello_world`
Car 2 finished the race
Car 1 finished the race
Car 4 finished the race
Car 3 finished the race
Car 0 finished the race
Final order -> [Ok(2), Ok(1), Ok(4), Ok(3), Ok(0)]

$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/rust_hello_world`
Car 2 finished the race
Car 1 finished the race
Car 0 finished the race
Car 4 finished the race
Car 3 finished the race
Final order -> [Ok(2), Ok(1), Ok(0), Ok(4), Ok(3)]
```

Este exemplo foi livremente adaptado do disponível em: http://rustbyexample.com/std_misc/channels.html.

6.10 CONCLUSÃO

Neste capítulo, falamos sobre como Rust gerencia a memória, como é feita a sua 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 `~`.

Além disso, 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. No próximo capítulo, discutiremos sobre macros.

Capítulo 7

MACROS

Neste capítulo, daremos uma olhada no poderoso mecanismo de macros da linguagem Rust, um dos seus pontos mais importantes. Seu uso é fundamental para estender a linguagem.

7.1 POR QUE MACROS?

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.

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.

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. Repare que, na definição dessa macro, 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 {
    () => { ... };
    ($fmt:expr) => { ... };
    ($fmt:expr, $($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, 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"); }
}

fn main() {
    hello!();
}
```

A execução desse código vai imprimir a mensagem "hello world" 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`.

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. Como parâmetro do `grep`, passarei o nome de nosso binário (no caso, `hello_world`), para que ele mostre apenas os símbolos dele.

Veja a compilação e a execução:

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.24 secs
     Running `target/debug/hello_world`
hello world
```

Veja os símbolos impressos com o `nm`:

```
$ nm target/debug/hello_world | grep hello_world
0000000000005750 t _ZN11hello_world4main17h77d9d462919d3604E
```

Vamos agora reescrever nosso código utilizando um método, como já vimos antes:

```
fn hello() {
    println!("hello world");
}

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

Não repetirei a execução aqui, mas veja o resultado do comando `nm`:

```
$ 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 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");
}
```

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 atendê-la. Em nosso caso, o padrão era `()`, ou seja, nenhum argumento.

7.2 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.38 secs
     Running `target/debug/rust_hello_world`
2, 2
```

Agora, recebemos três argumentos em nossa macro. Eles são atribuídos a metavariáveis, e 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`).

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:

```
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
   Compiling rust_hello_world v0.1.0 (file:...)
```

```
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.23 secs
     Running `target/debug/rust_hello_world`
49, 938
```

Poderíamos ainda modificar nossa macro e nomear as 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.23 secs
     Running `target/debug/rust_hello_world`
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;
            $(
                temp = temp + $x;
            )*
            temp
        };
    }
}

fn main() {
    sum!(x, 1, 2, 3, 4, 5, 6, 7);
    println!("{}", x);
}
```

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.31 secs
     Running `target/debug/rust_hello_world`
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.

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), com isso, 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
   Compiling hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.28 secs
     Running `target/debug/hello_world`
28
11.780000000000001
```

7.3 ÁRVORES DE TOKENS

Nossa macro de soma mostrou como iterar em uma coleção de parâmetros. Nesta seção, veremos como iterar sobre um tipo de dados conhecido como *token tree* (ou `tt`).

Esse exemplo foi extraído da versão beta do *The Book*, o livro de Rust mantido pela comunidade, em https://doc.rust-lang.org/beta/book/macros.html. 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["My page"]]
        body[
            h1["Welcome to my page"]
             p["This is awesome"]
             p["Do you agree?"]]
]);
```

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 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["My page"]]
        body[
            h1["Welcome to my page"]
             p["This is awesome"]
             p["Do you agree?"]]
]);

head[title["My page"]]
    body[
        h1["Welcome to my page"]
         p["This is awesome"]
         p["Do you agree?"]]

body[
    h1["Welcome to my page"]
     p["This is awesome"]
     p["Do you agree?"]]

h1["Welcome to my page"]
 p["This is awesome"]
```

```
 p["Do you agree?"]]

p["This is awesome"]

p["Do you agree?"]
```

Precisamos chamar nossa macro recursivamente para que o código HTML seja gerado da maneira correta 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. Esta última vai efetivamente criar a tag HTML de abertura (com o `<p>`), adicionar o seu conteúdo através de outra chamada à nossa macro e fechá-la (com o `</p>`).

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. Vejamos como fica nosso código:

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
```

```
        html[
            head[title["My page"]]
                body[
                    h1["Welcome to my page"]
                     p["This is awesome"]
                     p["Do you agree?"]]
        ]);

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

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.36 secs
     Running `target/debug/rust_hello_world`
<html><head><title>My page</title></head><body><h1>
Welcome to my page</h1><p>This is awesome</p><p>Do
you agree?</p></body></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, instale também a extensão `rustfmt`, com o comando `cargo install rustfmt`. Uma vez instaladas, basta executar o comando `cargo expand` para gerar o código da macro expandida.

Como resultado, temos cerca de trezentas linhas, porém reproduzirei 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(
    ::std::fmt::Arguments::new_v1({
        static __STATIC_FMTSTR:
        &'static [&'static str]
            =
            &["<", ">"];
        __STATIC_FMTSTR
    },
    &match (&"html",) {
        (__arg0,) =>
            [::std::fmt::ArgumentV1::new(__arg0,
             ::std::fmt::Display::fmt)],
    }));
```

7.4 POR QUE USAMOS MACROS?

O uso de macros nos possibilita olhar para dentro do nosso código e gerar um novo a partir das 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.

Macros permitem que o código compilado seja mais rápido, pois as decisões a serem tomadas em sua execução foram feitas em tempo de compilação.

7.5 AS DUAS RUSTS

Neste capítulo, falamos sobre o poderoso sistema de macros de Rust, que nos permite escrever código que modifica o nosso próprio código. Esse poder, aliado a algumas features não disponíveis na versão oficial, gerou o que alguns chamam de **a segunda Rust**.

A primeira Rust seria o conjunto da versão estável, liberada para instalação, e da *beta*, que mostra as novidades da próxima versão da linguagem. A segunda seria a chamada de **nightly**, que possui diversas features ainda não disponíveis na versão oficial e, por estar em estágio de desenvolvimento experimental, possibilita uma maior abertura a funcionalidades em teste, que talvez nunca sejam incorporadas nas versões finais.

É na **nightly** que encontramos o poder necessário para tornar Rust uma linguagem interessante para uma gama maior de aplicações do que apenas sistemas para servidores ou software embarcado. Tal poder pode ser interpretado como um pouco de *rédeas soltas* quando falamos sobre segurança e estabilidade, principalmente por conta das funcionalidades como `#![feature(plugin)]`, que nunca farão parte da versão final da linguagem.

A **nightly** também conta com inúmeras macros que não estão disponíveis na versão estável. Por ser uma versão de testes, ela dá ampla liberdade aos desenvolvedores para escreverem códigos que as regras rígidas da Rust oficial considerariam inseguros, mas que são necessários quando falamos de projetos de ORMs ou frameworks web.

No fundo, tudo isso tem a ver com as macros. O mecanismo das macros permite a capacidade de uma espécie de "metaprogramação", aliado a diversas features consideradas inseguras que, por isso, estão fora da versão estável (como os plugins). É isto que torna essa versão não oficial tão atrativa para determinados projetos.

Você pode ler mais sobre essas diferenças no artigo *A tale of two Rusts*, escrito por Karol Kuczmarski, disponível em: http://xion.io/post/programming/rust-nightly-vs-stable.html.
Também pode dar uma olhada na documentação do **Rocket**, o framework web escrito em Rust que está ganhando espaço rapidamente no mercado de APIs, em: https://rocket.rs/guide/getting-started/#nightly-version.

7.6 CONCLUSÃO

Macros são um dos pontos mais *sexys* da linguagem Rust. Poderosas e robustas, possibilitam escrever código que automaticamente gera outro código para você, evitando repetições.

CAPÍTULO 8

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 um 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 da 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, vou apresentar macros que possibilitam escrever testes em código Rust, e falar sobre o trait `Error` e sobre como utilizar o `Debug` para inspecionar suas instâncias durante a execução.

8.1 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!("THIS ENDS HERE");
```

```
    println!("This code will never be used!");
}
```

No exemplo, a linha com o `println!` nunca será executada. Veja:

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/rust_hello_world`
thread 'main' panicked at 'THIS ENDS HERE',
src/main.rs:2
note: Run with `RUST_BACKTRACE=1` for 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
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.0 secs
     Running `target/debug/rust_hello_world`
thread 'main' panicked at 'THIS ENDS HERE', src/main
.rs:2
stack backtrace:
   1:     0x55bf4ffe4d5f - std::sys::backtrace::
          tracing::imp::write::h6f1d53a70916b90d

   2:     0x55bf4ffe780d - std::panicking::default
          _hook::{{closure}}::h137e876f7d3b5850

   3:     0x55bf4ffe6d6a - std::panicking::default
          _hook::h0ac3811ec7cee78c
```

```
  4:     0x55bf4ffe72b8 - std::panicking::rust_
         panic_with_hook::hc303199e04562edf

  5:     0x55bf4ffe07c3 - std::panicking::begin
         _panic::h341b039f84d0b176
         at /buildslave/rust-buildbot/slave/stable
     -dist-rustc-linux/build/obj/../src/libstd
     /panicking.rs:413

  6:     0x55bf4ffe0982 - rust_hello_world::main::
         h4da8d2ac76b3ea3a
         at /home/marcelo/Workspace/rust-book/rust
     _hello_world/src/main.rs:2

  7:     0x55bf4ffef2d6 - __rust_maybe_catch_panic

  8:     0x55bf4ffe65e1 - std::rt::lang_start
         ::h538f8960e7644c80

  9:     0x55bf4ffe09c3 - main

 10:     0x7f14e665e82f - __libc_start_main

 11:     0x55bf4ffe05e8 - _start

 12:                0x0 - <unknown>
```

Veja em nosso backtrace, na linha 6, exatamente onde ocorreu o `panic!`. O backtrace pode ser muito útil para identificar problemas cotidianos como, 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.

8.2 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:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.35 secs
     Running `target/debug/rust_hello_world`
thread 'main' panicked at 'assertion failed:
'a' == 'b'', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for 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
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
thread 'main' panicked at 'assertion failed:
`(left == right)` (left: `false`, right: `true`)',
src/main.rs:2
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

8.3 DESEMPACOTAMENTO E A MACRO TRY!

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
   Compiling rust_hello_world v0.1.0 (file:///...)
error[E0308]: mismatched types
 --> src/main.rs:3:20
  |
3 |     let num: i32 = str.parse();
  |                    ^^^^^^^^^^^ expected i32,
```

```
  |               found enum `std::result::Result`
  |
  = note: expected type `i32`
  = note:    found type `std::result::Result<_, _>`

error: aborting due to previous error
```

Por convenção, a 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, tentei converter uma string para um inteiro. Entretanto, no mundo real, você pode deparar-se 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. Isso ocorre porque é impossível parsear uma string para um inteiro. Porém, o uso do `unwrap()` diz à 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 disse, 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 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!("The first 4 terms of the Fibonacci sequence are: ")
;
    for i in fib1.take(4) {
```

```
        println!("> {}", i);
    }

    let fib2 = Fibonacci { curr: 1, next: 1 };
    println!("The next 4 terms of the Fibonacci sequence are: ");
    for i in fib2.skip(4).take(4) {
        println!("> {}", i);
    }
}
```

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!("This number is even: {}", even_test(22).unwrap());
}
```

Esse exemplo funciona corretamente, pois passamos um par para o nosso método `even_test` .

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
This number is even: 22
```

Caso mudemos o código e passemos um número ímpar, o resultado será bem diferente.

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.22 secs
     Running `target/debug/rust_hello_world`
thread 'main' panicked at 'called `Option::unwrap()`
on a `None` value', ../src/libcore/option.rs:323
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Isso ocorre por não existir nada para desempacotar e, como dito anteriormente, o `unwrap()` faz com que Rust tente executar 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 feita para ser segura desde o começo. E é aí que entra a 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!
  # ... do something ...
  something_succeeded
rescue Exception
  something_failed
end
```

O `try!` 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 da Rust, em: https://doc.rust-lang.org/std/result/index.html.

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::prelude::*;
use std::io;

struct Info {
    name: &'static str,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    // Early return on error
    let mut file = match File::create("my_best_friends.txt") {
           Err(e) => return Err(e),
           Ok(f) => f,
    };
    if let Err(e) = file.write_all(
                     format!("name: {}\n", info.name).as_bytes()
) {
        return Err(e)
    }
```

```
    if let Err(e) = file.write_all(
                        format!("age: {}\n", info.age).as_bytes())
{
        return Err(e)
    }
    if let Err(e) = file.write_all(
                        format!("rating: {}\n", info.rating).as_byt
es()) {
        return Err(e)
    }
    Ok(())
}

fn main() {
    let friend01 = Info {
        name: "Nelson Castellani",
        age: 72,
        rating: 10
    };
    match write_info(&friend01){
        Err(e) => println!("Ops, something wrong -> {}", e),
        Ok(())  => println!("Everything is in the right place"),
    };
}
```

O resultado da execução será:

```
$ cargo run
   Compiling rust_hello_world v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.29 secs
     Running `target/debug/rust_hello_world`
Everything is in the right place
```

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 `try!` 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, something wrong -> {}", e),
    Ok(()) => println!("Everything is in the right place"),
};
```

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, something wrong -> {}", e),
    Ok(_)  => println!("Everything is in the right place"),
};
```

Agora, vamos reescrever nosso código com o `try!` e ver como isso pode simplificar as coisas.

```
use std::fs::File;
use std::io::prelude::*;
use std::io;

struct Info {
    name: &'static str,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = try!(File::create("my_best_friends.txt"));
    // Early return on error
    try!(file.write_all(format!("name: {}\n", info.name).as_bytes
()));
    try!(file.write_all(format!("age: {}\n", info.age).as_bytes()
```

```
));
    try!(file.write_all(format!("rating: {}\n", info.rating).as_b
ytes()));
    Ok(())
}

fn main() {
    let friend01 = Info {
        name: "Nelson Castellani",
        age: 72,
        rating: 10
    };
    match write_info(&friend01){
        Err(e) => println!("Ops, something wrong -> {}", e),
        Ok(())  => println!("Everything is in the right place"),
    };
}
```

O resultado da execução será o mesmo, mas com menos linhas. Isso porque o `try!` insere, em cada trecho do código, um `match` que faz o retorno da exceção para nós, sem precisarmos nos preocupar com isso.

```
macro_rules! try {
    ($e:expr) => (match $e { Ok(e) => e, Err(e) => return Err(e)
})
}
```

Como o `try!` é uma macro, ele vai inserir o trecho de código do `match`, que faz a validação da execução do bloco passado em `$e` e recebe um `Result`. Ele continuará a execução em caso de sucesso (`Ok`), ou retornará a exceção por meio de `Err`.

8.4 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.

Isso pode parecer simplório, mas serve para 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 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
   Compiling calculator v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.20 secs

$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        │   └── libcalculator.rlib
        ├── examples
        ├── libcalculator.rlib
        └── native

7 directories, 5 files
```

Abra agora o arquivo `lib.rs`, dentro da pasta `src`, e adicione o código do nosso projeto 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
}
```

A compilação deve ocorrer sem problemas; por enquanto não precisamos nos preocupar com isso. Com nossa base de código, vamos agora ao que nos interessa: escrever testes. Para isso, crie uma pasta `tests` na estrutura de seu projeto e adicione um arquivo chamado `methods_test.rs`. A estrutura do projeto ficará assim:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
├── target
│   └── debug
│       ├── build
│       ├── deps
│       │   └── libcalculator.rlib
│       ├── examples
│       ├── libcalculator.rlib
│       └── native
└── tests
```

```
    └── methods_test.rs

8 directories, 6 files
```

Vamos escrever um primeiro teste para conhecer seu formato. Um teste é definido pela marcação `#[test]` e deve utilizar as macros de asserção exibidas anteriormente. Abra o arquivo `methods_test.rs` e crie um teste que passa, como a seguir.

```
#[test]
fn pass_test() {
    assert!(true);
}
```

Para executar os testes do projeto, basta utilizar o comando `cargo test`. Veja:

```
$ cargo test
   Compiling calculator v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.41 secs
     Running target/debug/deps/calculator-6d356ac71b001406

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/methods_test-29195111853d4d05

running 1 test
test pass_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Vamos alterar nosso teste para que ele não passe, e ver o resultado da execução.

```
#[test]
fn pass_test() {
    assert!(false);
}
```

```
$ cargo test
   Compiling calculator v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.25 secs
     Running target/debug/deps/calculator-6ee1934d6ae90e97

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/deps/methods_test-29195111853d4d05

running 1 test
test pass_test ... FAILED

failures:

---- pass_test stdout ----
    thread 'pass_test' panicked at 'assertion failed:
    false', tests/methods_test.rs:3
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    pass_test

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

error: test failed
```

A saída do teste nos mostra onde ocorreu e qual foi o problema, como podemos ver a seguir. Nosso problema foi um `false` em uma asserção, na linha 3 do arquivo `methods_test.rs`.

```
    thread 'pass_test' panicked at 'assertion failed:
    false', tests/methods_test.rs:3
```

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.

```
extern crate calculator;

#[test]
fn sum_test() {
    assert_eq!(4, calculator::sum(2, 2));
    assert_eq!(10, calculator::sum(8, 2));
}

#[test]
fn subtract_test() {
    assert_eq!(0, calculator::subtract(2, 2));
    assert_eq!(6, calculator::subtract(8, 2));
}

#[test]
fn multiply_test() {
    assert_eq!(4, calculator::multiply(2, 2));
    assert_eq!(16, calculator::multiply(8, 2));
}

#[test]
fn divide_test() {
    assert_eq!(1, calculator::divide(2, 2));
    assert_eq!(4, calculator::divide(8, 2));
}
```

```
$ cargo test
   Compiling calculator v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.35 secs
     Running target/debug/deps/calculator-6d356ac71b001406

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/methods_test-29195111853d4d05

running 4 tests
test subtract_test ... ok
test divide_test ... ok
test sum_test ... ok
```

```
test multiply_test ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured
```

Os testes não precisam necessariamente estar dentro de um arquivo separado, em outra pasta. É possível mantê-los junto ao código, como alguns desenvolvedores Rust fazem. O `cargo`, inclusive, primeiro tenta rodar os testes nos arquivos do projeto para depois buscar outras pastas. A sintaxe é a mesma.

Você pode agrupar os testes em um módulo, como a seguir, e criar contextos de testes dentro dele, como funções encapsuladas. O código de um módulo marcado com `#[cfg(test)]` é compilado apenas quando dissermos ao `cargo` para rodar os testes; do contrário, é ignorado. Veja um exemplo de teste com uma função auxiliar de contexto.

No código a seguir, vamos avaliar se o resultado do nosso apelido para o método definido em `calculator` é o mesmo do método original.

```
#[cfg(test)]
mod tests {
    extern crate calculator;

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

    #[test]
    fn sum_test() {
        assert_eq!(local_sum(2, 2),
                   calculator::sum(2, 2));
        assert_eq!(local_sum(8, 2),
                   calculator::sum(8, 2));
    }

    #[test]
    fn subtract_test() {
```

```
        assert_eq!(0, calculator::subtract(2, 2));
        assert_eq!(6, calculator::subtract(8, 2));
    }

    #[test]
    fn multiply_test() {
        assert_eq!(4, calculator::multiply(2, 2));
        assert_eq!(16, calculator::multiply(8, 2));
    }

    #[test]
    fn divide_test() {
        assert_eq!(1, calculator::divide(2, 2));
        assert_eq!(4, calculator::divide(8, 2));
    }
}
```

```
$ cargo test
   Compiling calculator v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.30 secs
     Running target/debug/deps/calculator-6d356ac71b001406

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/methods_test-29195111853d4d05

running 4 tests
test tests::divide_test ... ok
test tests::multiply_test ... ok
test tests::subtract_test ... ok
test tests::sum_test ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured
```

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.

```
#[cfg(test)]
mod tests {
    extern crate calculator;
```

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

    #[test]
    fn sum_test() {
        assert_eq!(local_sum(2, 2),
                   calculator::sum(2, 2));
        assert_eq!(local_sum(8, 2),
                   calculator::sum(8, 2));
    }

    #[test]
    #[ignore]
    fn subtract_test() {
        assert_eq!(0, calculator::subtract(2, 2));
        assert_eq!(6, calculator::subtract(8, 2));
    }

    #[test]
    #[ignore]
    fn multiply_test() {
        assert_eq!(4, calculator::multiply(2, 2));
        assert_eq!(16, calculator::multiply(8, 2));
    }

    #[test]
    #[ignore]
    fn divide_test() {
        assert_eq!(1, calculator::divide(2, 2));
        assert_eq!(4, calculator::divide(8, 2));
    }
}
```

```
$ cargo test
   Compiling calculator v0.1.0 (file:///...)
    Finished debug [unoptimized + debuginfo] target(s)
    in 0.29 secs
     Running target/debug/deps/calculator-6d356ac71b001406

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

```
     Running target/debug/methods_test-29195111853d4d05

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
```

Um ponto importante sobre a execução de testes com o `cargo` é que eles são executados cada um em sua própria thread. Isso poderá gerar problemas quando precisarmos manter um contexto para outras execuções.

Para isso, o `cargo` possui a opção de limitar o número de threads de execução, deixando o teste mais lento, mas linear. Para tanto, utilize o comando a seguir:

```
$ cargo test -- --test-threads=1
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 sec
s
     Running target/debug/deps/calculator-6d356ac71b001406

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/methods_test-29195111853d4d05

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
```

8.5 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.

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.

Capítulo 9

COMO RUST COMPILA O SEU CÓDIGO

Antes de finalizarmos, não poderia deixar de tratar da magia que ocorre quando pedimos gentilmente ao nosso compilador que transforme nosso código Rust em um código executável.

Talvez este capítulo soe mais complexo do que os anteriores, pois nele trataremos de alguns conceitos não muito familiares à maioria dos desenvolvedores, como o processamento léxico e a "desaçucarização do código".

9.1 O PASSO A PASSO DA COMPILAÇÃO

Análise e desaçucarização do código

O primeiro passo pelo qual seu código passa é a desaçucarização de código. Falamos no capítulo *Traits e estruturas* que itens como `==` ou `!=` são, na verdade, açúcar sintático para métodos como `eq` e `ne`. Neste ponto da compilação, todos os itens açúcares são removidos em prol dos métodos verdadeiros a serem executados.

Na sequência, ocorre a análise léxica, que verifica se o que está

escrito é um código Rust válido, e se os métodos nomeados não sobrepõem equivocadamente métodos da linguagem. E, como falamos no capítulo *Testar, o tempo todo*, as macros são substituídas por código real, para cada situação em que são chamadas.

Verificação de tipos

O próximo passo é a verificação dos tipos. Nesse ponto, Rust identifica se o valor passado ou recebido está aderente ao valor declarado. Isso quer dizer que a linguagem verifica cada atribuição ou definição de variável e cada chamada de método, inferindo se o valor que pode ser passado ali corresponde ao contexto em que será executado.

Verificação de empréstimos

Falamos sobre empréstimos no capítulo *Vetores, strings e tipos genéricos*, quando verificamos se uma sequência de caracteres era uma `String` ou um `str` . Nesse ponto da compilação, a Rust vai verificar se o uso das variáveis e suas referências obedece ao escopo que pertencem, ou se são passadas por empréstimo para outro escopo. Essa validação busca eventuais falhas de uso de referências, como foi apresentado no capítulo *Macros*, quando falamos sobre o tempo de vida de uma variável na pilha.

Tradução para LLVM

Rust usa o LLVM, um projeto *open source* que provê um conjunto de componentes reutilizáveis para a compilação do código. Nesse momento, o código Rust é traduzido para código

LLVM, o que possibilita toda a interoperabilidade de plataformas suportadas pela linguagem.

É o LLVM que permite que um código Rust seja compilado para um processador ARM ou para o processador Intel de seu notebook, respeitando o sistema operacional que roda nele. Ele é a base de diversas linguagens, como **Crystal**, **Haskell**, **Lua**, **Bytecode Java**, **Objective-C**, **Swift**, **C#**, **R** e outras.

É correto dizer que, neste momento, o código Rust deixa de existir, passando a existir código LLVM. Apenas para ilustrar, segue um código C e sua representação em LLVM.

```
int mul_add(int x, int y, int z) {
  return x * y + z;
}
```

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

Se desejar conhecer mais sobre o LLVM, recomendo visitar o site oficial do projeto, em http://llvm.org/. Você também pode dar uma lida no tutorial disponível em: http://releases.llvm.org/2.6/docs/tutorial/.

Otimização e geração de executável

Com o código traduzido para o LLVM, começa o processo de otimização. Esse processo inclui diversas fases, como a conversão de loops em saltos ou o uso de ponteiros.

É nesse momento que seu código Rust passa a ser tão rápido

quanto um código escrito em *C*. Muito do mérito disso se deve ao LLVM, mas também ao excelente tradutor Rust para LLVM, escrito pelos mantenedores da linguagem. Sem esse trabalho, de nada adiantaria o LLVM.

9.2 O MIR E A MELHORIA NO PROCESSO DE COMPILAÇÃO

Recentemente, o processo de compilação da Rust passou a ter mais uma etapa: a geração de um código chamado MIR. Antes do código Rust se transformar em código LLVM (conhecido como IR, ou *Intermediate Representation*), ele passará a ser convertido para um formato criado pela equipe da Rust chamado **mid-level intermediate representation**, que é uma representação do código pré-LLVM.

O objetivo do MIR é deixar a compilação da Rust mais rápida. Se você começar a escrever código Rust complexo, verá que o processo de compilação pode facilmente chegar à casa de alguns minutos para finalizar. Já com o uso de MIR, o processo será mais inteligente, possibilitando uma compilação apenas do novo material que entrou em seu projeto em vez de uma completa a cada execução do `cargo` .

Para mais detalhes sobre o MIR, recomendo a leitura do artigo publicado no blog oficial da linguagem Rust, em: https://blog.rust-lang.org/2016/04/19/MIR.html.

Capítulo 10

O COMEÇO DE UMA JORNADA

Você já assistiu ao filme *Dr. Estranho*, da Marvel? Se não assistiu, deveria ver. Além de ser um dos meus super-heróis favoritos, tem uma das sacadas mais interessantes sobre aprendizado que vi em muito tempo.

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, falarei sobre outros recursos relacionados à linguagem Rust que você deveria experimentar para ficar cada vez melhor e mais produtivo ao utilizá-la, até que ela se torne a sua linguagem principal no cotidiano.

10.1 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.

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 Rust. O segundo é o *Tutorial: Guessing Game*, que apresenta conceitos básicos de Rust, mas sem muitos detalhes. Dê uma rápida passada por eles.

A parte que mais vale a pena é a *Syntax and Semantics*, que mostra de forma bem detalhada cada bit de Rust. A maioria das informações desse capítulo foi apresentada neste livro, mas não trouxe para cá algumas peculiaridades não tão usadas, que talvez sejam interessantes de conhecer.

Outro capítulo interessante é o seguinte, o *Effective Rust*, mostrando conceitos para escrever código Rust de alta performance. Parte das informações está aqui nesta obra, com alguns toques pessoais e comentários que você não encontra no *The Book*, mas vale dar uma olhada em como o pessoal que criou a linguagem pensou em usá-la.

Para quem quiser acompanhar o futuro da Rust, basta verificar o capítulo *Nightly Rust*, que fala sobre as novidades que estão em

desenvolvimento e podem ser testadas em versões não estáveis da linguagem. Por fim, ao fechar o livro, temos um *Glossary* e a tradicional *Bibliography*.

Além do *The Book*, a comunidade Rust mantém o já citado **Rustonomicon**, em https://doc.rust-lang.org/nomicon/. Logo no 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*: *"Este documento é um rascunho e pode conter sérios erros"*.

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 http://www.oreilly.com/programming/free/files/why-rust.pdf. 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 da linguagem e afins, e é periodicamente atualizado pela comunidade. Vale acompanhar os commits.

E, claro, a documentação da **Standard Library** da Rust, disponível em: https://doc.rust-lang.org/std/. Referência clara e concisa de cada um dos elementos que você usará no dia a dia.

10.2 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 mais, e espero que ela amadureça e se popularize no mercado brasileiro, assim como ocorreu com Ruby.

Falando por mim, quando comecei com Ruby em 2006, escrevendo pequenos scripts para automação de servidor, não imaginava que ela um dia se tornaria o meu ganha-pão nem que eu participaria da fundação de uma das comunidades de desenvolvedores mais legais que conheço, o Guru-SP. Espero que, com Rust, as coisas ganhem o mesmo andamento, e sem demora.

Se você leu o livro até aqui, coloco-me à disposição para lhe ajudar. Basta entrar em contato pelo e-mail marcelofc.rock@gmail.com para papearmos. Mais do que autor deste livro, sou um entusiasta de Rust e sei que ela somente ganhará mercado se existirem pessoas utilizando-a. Vamos juntos!