



# Introdução à Linguagem Rust

Paradigmas de Linguagens de Programação

**Eric Hoffmann Fernandes Braga**  
**Ausberto S. Castro Vera**

21 de junho de 2024



# Rust

---

**Disciplina:** Paradigmas de Linguagens de Programação 2023

**Linguagem:** Scala

**Aluno:** *Nome Completo do aluno*

### Ficha de avaliação:

Aspectos de avaliação (requisitos mínimos)	Pontos
<b>Introdução (Máximo: 01 pontos)</b> <ul style="list-style-type: none"> <li>• Aspectos históricos</li> <li>• Áreas de Aplicação da linguagem</li> </ul>	
<b>Elementos básicos da linguagem (Máximo: 01 pontos)</b> <ul style="list-style-type: none"> <li>• Sintaxe (variáveis, constantes, comandos, operações, etc.)</li> <li>• Cada elemento com exemplos (código e execução)</li> </ul>	
<b>Aspectos Avançados da linguagem (Máximo: 2,0 pontos)</b> <ul style="list-style-type: none"> <li>• Sintaxe (variáveis, constantes, comandos, operações, etc.)</li> <li>• Cada elemento com exemplos (código e execução)</li> <li>• Exemplos com fonte diferenciada (listing)</li> </ul>	
<b>Mínimo 5 Aplicações completas - Aplicações (Máximo : 2,0 pontos)</b> <ul style="list-style-type: none"> <li>• Uso de rotinas-funções-procedimentos, E/S formatadas</li> <li>• Uma Calculadora</li> <li>• Gráficos</li> <li>• Algoritmo QuickSort</li> <li>• Outra aplicação</li> <li>• Outras aplicações ...</li> </ul>	
<b>Ferramentas (compiladores, interpretadores, etc.) (Máximo : 1,0 pontos)</b> <ul style="list-style-type: none"> <li>• Ferramentas utilizadas nos exemplos: pelo menos DUAS</li> <li>• Descrição de Ferramentas existentes: máximo 5</li> <li>• Mostrar as telas dos exemplos junto ao compilador-interpretador</li> <li>• Mostrar as telas dos resultados com o uso das ferramentas</li> <li>• Descrição das ferramentas (autor, versão, homepage, tipo, etc.)</li> </ul>	
<b>Organização do trabalho (Máximo: 01 ponto)</b> <ul style="list-style-type: none"> <li>• Conteúdo, Historia, Seções, gráficos, exemplos, conclusões, bibliografia</li> <li>• Cada elemento com exemplos (código e execução, ferramenta, nome do aluno)</li> </ul>	
<b>Uso de Bibliografia (Máximo: 01 ponto)</b> <ul style="list-style-type: none"> <li>• Livros: pelo menos 3</li> <li>• Artigos científicos: pelo menos 3 (IEEE Xplore, ACM Library)</li> <li>• Todas as Referências dentro do texto, tipo [ABC 04]</li> <li>• Evite Referências da Internet</li> </ul>	
<b>Conceito do Professor (Opcional: 01 ponto)</b>	
	<b>Nota Final do trabalho:</b>

*Observação:* Requisitos mínimos significa a *metade* dos pontos

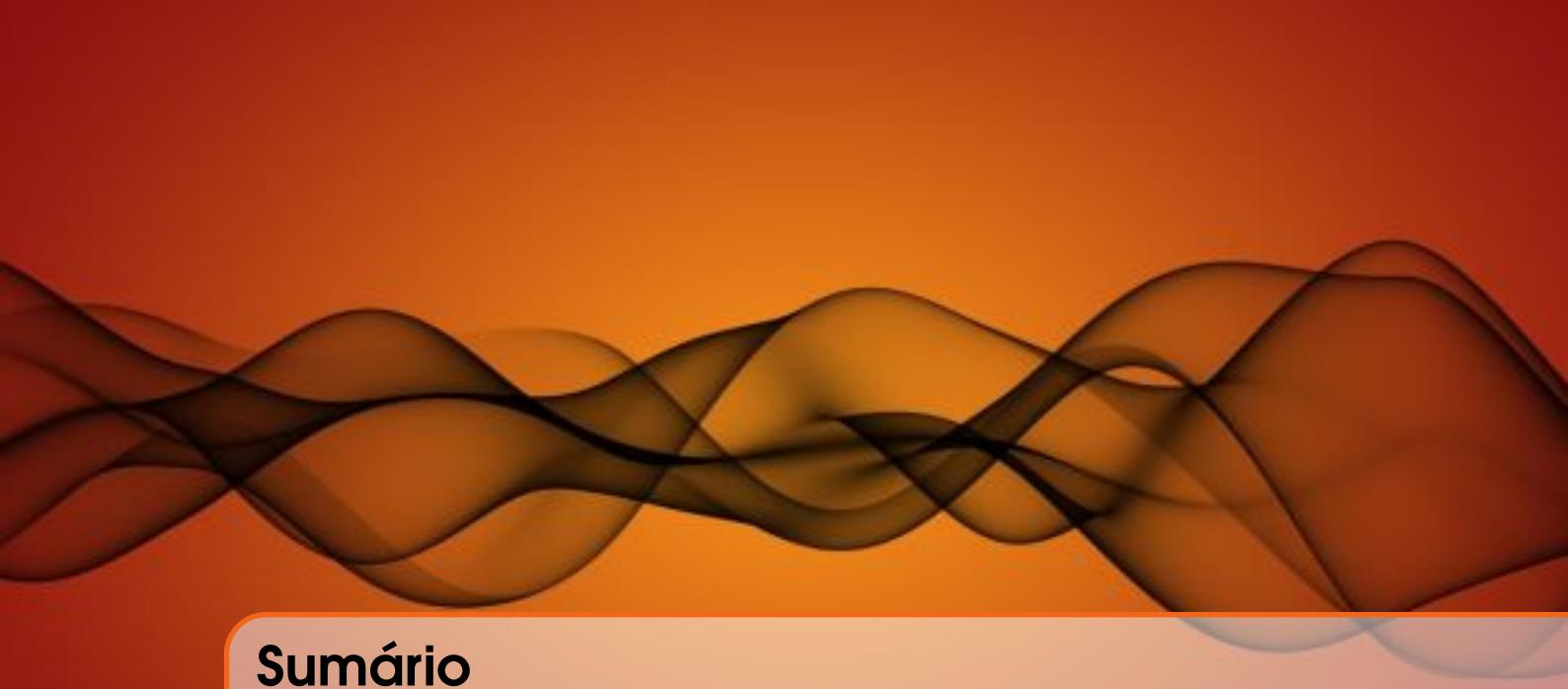
Copyright © 2024 Eric Hoffmann Fernandes Braga e Ausberto S. Castro Vera

UENF - UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO

CCT - CENTRO DE CIÊNCIA E TECNOLOGIA  
LCMAT - LABORATÓRIO DE MATEMÁTICAS  
CC - CURSO DE CIÊNCIA DA COMPUTAÇÃO

*Primeira edição, Abril 2024*





# Sumário

<b>1</b>	<b>Introdução</b>	<b>7</b>
1.1	Aspectos históricos da linguagem Rust	7
1.2	Áreas de Aplicação da Linguagem	8
1.2.1	Backend	8
1.2.2	Sistemas Operacionais e Hardware	8
<b>2</b>	<b>Conceitos básicos da Linguagem Rust</b>	<b>9</b>
2.1	Variáveis e constantes	9
2.1.1	Constantes	11
2.1.2	Shadowing	11
2.2	Tipos de Dados Básicos	12
2.2.1	Inteiros	12
2.2.2	Floats	13
2.2.3	Booleano	14
2.2.4	Caracteres	14
2.2.5	Strings	14
2.3	Tipos de Dados Compostos	14
2.3.1	Tuplas	14
2.3.2	Arrays	16
<b>2.4</b>	<b>Operadores e Expressões em Rust</b>	<b>17</b>
2.4.1	Operadores	17
2.4.2	Expressões	17

<b>3</b>	<b>Programação em Rust</b>	<b>19</b>
<b>3.1</b>	<b>Entradas e saídas</b>	<b>19</b>
3.1.1	Interpolação	20
3.1.2	Format!	21
3.1.3	Print!	21
3.1.4	Println!	21
3.1.5	Saidas de Erros	21
<b>3.2</b>	<b>Controle de Fluxo</b>	<b>22</b>
3.2.1	If e If else	22
3.2.2	Match	22
3.2.3	Laços de Repetição	23
<b>3.3</b>	<b>Funções</b>	<b>24</b>
<b>3.4</b>	<b>Módulos</b>	<b>25</b>
<b>4</b>	<b>Aplicações da Linguagem Rust</b>	<b>27</b>
<b>4.1</b>	<b>Insertion Sort</b>	<b>27</b>
<b>4.2</b>	<b>Linked List</b>	<b>27</b>
<b>4.3</b>	<b>Quick Sort</b>	<b>32</b>
<b>4.4</b>	<b>IsPrime</b>	<b>35</b>
<b>4.5</b>	<b>Binary Search</b>	<b>36</b>
<b>5</b>	<b>Ferramentas</b>	<b>37</b>
<b>5.1</b>	<b>RustUp</b>	<b>37</b>
<b>5.2</b>	<b>Cargo</b>	<b>37</b>
<b>6</b>	<b>Considerações Finais</b>	<b>39</b>
	<b>Bibliografia</b>	<b>41</b>

# 1. Introdução

Rust é uma linguagem de programação de alto nível multi-paradigma e compilada, Rust nasceu originalmente como projeto pessoal de Graydon Hoare em 2006 enquanto trabalhava na Mozilla Research e começou a ser patrocinada pela mesma em 2009 como parte de um navegador experimental chamado Servo. O sistema de posse e memória pelo qual Rust é famosa vem de influência das linguagens Cyclone e ML Kit.

Em fevereiro de 2024 a Casa Branca fez um anúncio público com o título *BACK TO THE BUILDING BLOCKS: A Path Toward Secure And Measurable Software* descrevendo em 16 páginas como deveríamos usar linguagens "inseguras" em termos de memória e começar a usar linguagens seguras para nossos softwares. No documento a Casa Branca recomenda o uso de linguagens seguras como Rust.

## 1.1 Aspectos históricos da linguagem Rust

Rust foi anunciada publicamente em 2010 pela Mozilla e em torno da mesma época mudou-se o foco do compilador original escrito em O-Caml para um compilador auto-hospedado usando LLVM. Em 2011 o compilador escrito em Rust se compilou com sucesso pela primeira vez. Mais tarde no mesmo ano a logo foi decidida inspirada na coroa de uma bicicleta.

Em março de 2012 foi lançada a versão 0.2, nela foram introduzidas classes pela primeira vez. Quatro meses depois destrutores e polimorfismo foram adicionados na versão 0.3 através do uso de interfaces. Em outubro de 2012 *traits* foram adicionados como meio de se implementar herança na linguagem, interfaces também foram combinadas com *traits* e removidas como funcionalidade independente. Em 2013 o coletor de lixo da linguagem foi removido em favor das regras de posse que foi desenvolvida ao longo de 2010 onde o gerenciamento de memória através do sistema de posse foi consolidado para prevenir *bugs* de memória.

Em janeiro de 2014 o editor chefe do jornal Dr. Dobb, Andrew Binstock [Bin14], comentou sobre as chances de Rust se tornar um competidor legítimo para C++, junto com outras linguagens como: D, Go, Nim. De acordo com Binstock enquanto Rust era "Uma linguagem amplamente vista como notavelmente elegante", sua adoção era devagar pois mudava radicalmente de versão para

versão. A primeira versão estável de Rust, a versão 1.0 foi anunciada no dia 15 de maio de 2015. O desenvolvimento do navegador Servo continuou lado a lado com o crescimento do Rust e em setembro de 2017, Firefox 57 foi lançado como a primeira versão do navegador que incorporava componentes do navegador Servo em um projeto chamado "Firefox Quantum"

Escrever de 2020-Presente encapsulando as demissões da Mozilla, a Rust Foundation, a inclusão de Rust no kernel do Linux e o pedido da casa branca.

## 1.2 Áreas de Aplicação da Linguagem

Como Rust é uma linguagem de propósito geral ela possui várias áreas onde é usada as quais são:

### 1.2.1 Backend

Devido a sua velocidade e segurança Rust é bastante utilizado em projetos de *backend* para poder se certificar de que o sistema seja rápido e aguente um alto volume de usuários concorrentes sem que o sistema saia do ar, uma empresa que usa Rust em seus projetos *backend* é a Amazon nos seus produtos AWS como:

- Firecracker uma solução para virtualização.
- Bottlerocket, uma distribuição Linux e sistema de conteinerização.
- Tokio, uma solução de rede assíncrona.

Outro local onde Rust pode ajudar muito com seus pontos fortes e no próprio sistema da rotas da internet onde milhões de usuários devem ser redirecionados em milissegundos para o conteúdo correto para que sua experiência seja fluída e ininterrupta, para isso a Cloudflare criou sua rede de entrega de conteúdo:

- Pingora [Wu24], um firewall de casamento de padrões e um *framework* para construir redes programáveis que servem mais de 40 milhões de requisições por segundo

Com o movimento da internet das coisas crescendo várias empresas como a Microsoft vem tentando integrar seus sistemas de nuvem com dispositivos da internet das coisas para que você possa controlar toda sua casa através de um app para que o sistema de nuvem seja capaz de interagir com múltiplos aparelhos de forma assíncrona e com agilidade a Microsoft optou por Rust:

- Azure-IoT é um sistema de servidores *Edge* usado para rodar serviços Azure em aparelhos que sejam "Internet das Coisas"
- A Microsoft também usa Rust para criar módulos conteinerizados usando WebAssembly e Kubernetes para fácil interação com os aparelhos usando interfaces web.

### 1.2.2 Sistemas Operacionais e Hardware

Rust sendo uma linguagem focada em performance e segurança de memória não demorou muito para que fosse feita a proposta de que ela fosse usada no desenvolvimento do kernel do Linux e aplicações que lidassem diretamente com hardware. Em 2021 um pedido para comentário foi feito por Ojeda e então começou o trabalho para incluir Rust no kernel, agora em 2024 suporte experimental para Rust no kernel está começando. Mesmo Rust ainda estando no caminho de ser implementado no kernel do Linux já temos exemplos da linguagem sendo usada em outros sistemas operacionais, a Microsoft já começou a implementar Rust no kernel do Windows em bibliotecas contidas [Wes23].

## 2. Conceitos básicos da Linguagem Rust

Neste capítulo será apresentado os conceitos basicos da linguagem Rust como vistos nos livros [Kla14], [Gou15] e [Gou16]. Estes livros sao os livros basicos para o estudo linguagem Rust.

### 2.1 Variáveis e constantes

Diferente de muitas linguagens em Rust o padrão para variaveis é ser imutavel/constante, isso significa que uma variavel não pode receber outro valor depois que inicialmente atribuida, essa decisão feita pela fundação Rust garante que você possa aproveitar da segurança e concorrência que Rust oferece mais facilmente, mas você ainda tem a opção de transformar uma variavel em mutavel caso deseje.

```
fn main() {  
    // Cria uma variavel imutavel  
    let a = 2;  
  
    // Cria uma variavel mutavel  
    let mut b = 3  
}
```

Vamos ver exemplos de como isso difere de uma linguagem como C:

```
// variables.rs
fn main() {
    // Cria uma variavel imutavel
    let x = 2;

    // O esperado aqui em C e que se escreva o numero 2 no console
    println!("O valor de x e: {}");

    x = 3;

    // O esperado aqui em C e que se escreva o numero 3 no console
    println!("O valor de x e: {}");
}
```

```
$ rustc variables.rs
$ ./variables
> 2
> 2
```

Aqui podemos ver que apesar de termos feito `x = 3` no código a variável continuou com o valor de '2' isso acontece pois as variáveis são imutáveis por padrão, logo, se quisermos mudar o valor de uma variável em Rust temos que explicitamente declará-la como mutável usando 'mut'.

```
// variables.rs
fn main() {
    // Cria uma variavel imutavel
    let mut x = 2;

    println!("O valor de x e: {}");

    x = 3;

    println!("O valor de x e: {}");
}
```

```
$ rustc variables.rs
$ ./variables
> 2
> 3
```

Agora podemos ver que nosso código está funcionando como esperavamo, e o valor de `x` muda quando nos atribuímos o valor 3 a ele no código, isso ocorre pois declararmos a variável como 'mut'.

### 2.1.1 Constantes

Como variaveis imutaveis, constantes são valores que se ligam a um nome e não podem mudar, mas existem algumas diferenças entre constantes e variaveis. Primeiro, voce não pode usar 'mut' em uma constante, pois elas não são só imutaveis por padrão, elas são imutaveis sempre. Segundo, constantes se declaram com a palavra chave 'const' ao inves de 'let' e o seu tipo deve obrigatoriamente ser definido usando *type-annotations*, não se preocupe, logo em seguida voce descobrirá o que são *type annotations*.

Constantes podem ser declaradas em qualquer escopo, incluindo o global, e a última diferença é que constantes só podem ser atribuidas valores de expressões constantes, significando que só não podem ser valores que só são possíveis de se computar na execução do programa.

Aqui está um exemplo da declaração de uma constante:

```
// constant.rs
fn main() {
    // Cria uma constante
    const MyConst: u32 = 2;
}
```

### 2.1.2 Shadowing

*Shadowing* é o nome que programadores de Rust deram pra quando você re-declara uma variável que já existia antes, assim você sobrepõe o valor antigo da variavel com o valor novo, essa tecnica é comumente usada com variaveis imutáveis. Uma variável que sofreu *shadowing* continua existindo como seu novo valor até que ela mesma sofra *shadowing* ou que acabe o escopo em que ela exista, isso permite padrões avancados de código como:

```
// shadowing.rs
fn main() {
    // Cria uma variavel imutavel x
    let x: u32 = 5;

    // Aplica o shadowing dentro deste escopo
    // transformando a variavel em 6
    let x = x + 1;
    {

        // Aplica shadowing dentro deste escopo
        // transformando a variavel em 12
        let x = x * 2
        println!("Valor de X no escopo interior e: {x}");
    }
    // O escopo finaliza logo a variavel volta a ser 6
    println!("Valor de X no escopo exterior e: {x}");
}
```

```
$ rustc shadowing.rs
$ ./shadowing
> Valor de X no escopo interior e: 12
> Valor de X no escopo exterior e: 6
```

*Shadowing* se difere de declarar uma variável como 'mut' pois teremos um erro de compilação se tentarmos re-atribuir a variável sem usar a palavra chave 'let'. Usando 'let' nós podemos realizar algumas transformações em um valor mas ter o valor imutável depois que estas transformações acabarem. Outra vantagem de *shadowing* é, como estamos basicamente criando outra variável quando usamos 'let', nós podemos mudar o valor da variável, mas reusar o mesmo nome.

## 2.2 Tipos de Dados Básicos

Assim como outras linguagens Rust possui diversos tipos de dados que permitem que o compilador ache erros no código e otimize onde possível, mas a linguagem também possui tipos dinâmicos permitindo que os tipos que devem ser usados sejam inferidos pelo compilador durante a etapa de compilação sem que o usuário especifique explicitamente.

Então se apenas declararmos a variável, o compilador é encarregado de decidir o tipo apropriado da variável em questão, porém se quisermos ter certeza de que estamos trabalhando com um tipo de dado específico podemos definí-lo usando *type-annotations* onde entre a variável e seu valor decidimos seu tipo.

```
fn main() {
    // O compilador infere o tipo deste dado
    let a = 2;

    // Nos informamos o tipo u32 (Inteiro de 32 bits sem sinal)
    let b: u32 = 3
}
```

### 2.2.1 Inteiros

Em Rust se quisermos trabalhar com inteiros temos opções desde i8 até i128 onde o número decide quantos bits de informação aquela variável pode guardar, então, um i8 pode guardar desde os números -127 até o número 128. Se não quisermos números negativos podemos especificar trocando a letra de 'i' para 'u' que significa *unsigned* (sem sinal) assim um u8 pode guardar desde 0 até 255 e ainda temos um tipo especial 'isize' que se define de acordo com a arquitetura de seu computador, sendo um i32 se o computador for 32 bits ou um i64 se o computador for 64 bits. Aqui estão todos os inteiros:

```
//types.rs
fn main() {
    // O compilador infere o tipo deste dado
    let a = 0;

    let b: i8 = 1;
    let c: i16 = -2;
    let d: i32 = 3;
    let e: i64 = -4;
    let f: i128 = 5;
    let g: isize = -6;
    ...
}
```

```
//types.rs
...
let A: u8 = 1;
let B: u16 = 2;
let C: u32 = 3;
let D: u64 = 4;
let E: u128 = 5;
let F: usize = 6;
...
```

Inteiros tambem pode ser escritos em outras bases numerais que nao sejam base decimal

Literais	Exemplo
Decimal	1_580_000
Hex	0xff
Octal	0o77
Binário	0b1111_0000
Byte	b'a'

```
//types.rs
...
// 'Underlines' podem ser usados para melhor legibilidade
let Decimal = 1_580_00;
let Hex = 0xff;
let Octal = 0o77;
let Binario = 0b1111_0000;
let Byte = b'a';
...
```

## 2.2.2 Floats

Rust possui dois tipos primitivos para *floats*, os quais são números com casas decimais. Os tipos de *float* do Rust sao 'f32' e 'f64' que são de tamanhos 32 bit e 64 bit respectivamente. O valor padrão de um *float* é 'f64' pois são capazes de mais precisão. Todos os *floats* possuem sinal.

```
//types.rs
...
let fa = 1.2; // f64

let fb: f32 = 3.1415 // f32
...
```

*Floats* em Rust seguem o padrao IEEE-754. O tipo 'f32' é de precisão única e o tipo 'f64' é de precisão dupla.

### 2.2.3 Booleano

O tipo booleano em Rust como em qualquer outra linguagem tem dois possíveis valores, *true* e *false* e são criadas usando a palavra chave 'bool'. Como qualquer outra tipo ele pode ser inferido pelo compilador ou ser *type-annotated* pelo programador

```
//types.rs
...
let boolTrue = true;

let boolFalse: bool = false // type-annotated
...
```

O uso de booleans veremos mais tarde na seção de controle de fluxo

### 2.2.4 Caracteres

```
//types.rs
...
let w = 'w';
let z: char = 'z' \\ type-annotated
}
```

Cáracteres são o tipo alfabético mais simples da linguagem, diferentemente de strings cáracteres são envolvidos em aspas simples ao invés de aspas duplas. O tipo char tem 4 bytes em tamanho e representa valores escalares de Unicode, o que significa que os caracteres no Rust podem representar desde caracteres ASCII, à caracteres Chineses à até emojis.

### 2.2.5 Strings

A fazer:

- String
- &str
- &'static str
- Box str
- Rc str
- Arc str
- byte str
- String literals
- Specialized String
- Interoperable String

## 2.3 Tipos de Dados Compostos

### 2.3.1 Tuplas

As tuplas são uma maneira de agrupar múltiplos valores de diferentes tipos em um único tipo composto, tuplas possuem tamanho fixo e não podem diminuir ou aumentar depois de criadas. Nós escrevemos uma tupla criando uma lista separada por vírgulas, tuplas podem ser tanto inferidas quanto *type-annotated*.

```
//compounds.rs
fn main() {
    let tupa = (500, 6.4, 1);

    // Type-annotated
    let tupb: (i32, f64, u8) = (500, 6.4, 1);
    ...
}
```

Nós então com uma tupla podemos usar casamento de padrão para desconstruir a tupla e atribuir cada elemento a uma variável.

```
//compounds.rs
...
let tup = (500, 6.4, 1);
let (x, y, z) = tup;
println!("O valor de x é {} e z {}", x, z)
...
```

Também podemos acessar um elemento da tupla diretamente usando a notação de ponto:

```
//compounds.rs
...
let quinhentos = tup.0;
let seis_ponto_quatro = tup.1;
let um = tup.2;
...
```

```
$ rustc compounds.rs
$ ./compounds
> Valor de X é 500 e z 1
```

Uma tupla sem nenhum valor tem um nome especial, *unit*. Esse valor e seu tipo ambos são escritos '`()`' e representam um valor vazio ou tipo de retorno vazio. Expressões implicitamente retornam o valor *unit* se elas não retornam nenhum outro valor.

### 2.3.2 Arrays

Outra maneira de ter uma coleção de valores é um *array*, porém diferente de uma tupla, todos os elementos de um *array* devem ser do mesmo tipo. Diferente de outras linguagens *arrays* em Rust tem tamanho fixo. Valores em um array são escritos em uma lista separada por vírgulas dentro de colchetes.

```
//compounds.rs
...
// O tamanho e o tipo são inferidos
let arr = [1, 2, 3, 4, 5];
...
```

*Arrays* são úteis quando você sabe que o número de elementos não muda. Por exemplo um *array* com os 12 meses do ano. Um array também como outras variáveis pode ser inicializado usando *type-annotations*.

```
//compounds.rs
...
// O tamanho e o tipo são descritos
let arrAnnotated: [i32, 5] = [1, 2, 3, 4, 5];

// Também é possível inicializar um array com o mesmo valor
// em todas as suas posições
let same = [3; 5];
// Resulta em = [3, 3, 3, 3, 3]
```

Podemos acessar os elementos de um array usando colchetes e a posição após seu nome

```
//compounds.rs
...
let arr = [1, 2, 3, 4, 5];

let first = arr[0];
let middle = arr[2];
let last = arr[4];
}
```

Se um elemento fora do array tentar ser acessado no código, a *engine* do Rust avaliará a expressão para um pânico e terminará a execução do código, resultando no seguinte resultado no terminal:

```
thread 'main' panicked at 'index out of bounds:
the len is 5 but the index is 10', src/main.rs:19:19
note: run with 'RUST_BACKTRACE=1'
environment variable to display a backtrace
```

## 2.4 Operadores e Expressões em Rust

### 2.4.1 Operadores

Operadores disponíveis em Rust se assimilam com os operadores disponíveis em C com algumas exceções específicas que veremos agora.

```
//operators.rs
fn main() {
    // Operacoes matematicas
    1 + 2; // Soma
    2 - 1; // Subtracao
    2 * 4; // Multiplicacao
    4 / 2; // Divisao

    true && false; // Operacao logica AND
    true || false; // Operacao logica OR
    !true; // Operacao logica NOT

    0b1111 & 0b11111; // Operacao Bitwise AND
    0b0110 | 0b0101; // Operacao Bitwise OR
    0b0110 ^ 0b1001; // Operacao Bitwise XOR

    1u32 << 5; // Operacao Bitshift Left
    32u32 >> 5; // Operacao Bitshift Right

    // Excessao nao presente em linguagens tipo C
    -2.5e-7; // Notacao cientifica
}
```

### 2.4.2 Expressões

Um programa em Rust é majoritariamente composto de declarações

```
//expressions.rs
fn main() {
    // declaracao
    // declaracao
    // declaracao
    // declaracao
}
```

Existem alguns tipos de declarações em Rust, as duas mais comuns são a atribuição de uma variável e usar ';' com uma expressão.

```
//expressions.rs
...
    // atribuicao de variavel
let x = 5;

    // expressoes;
x;
x + 1;
15;
...
```

Blocos também são expressões, logo eles podem ser usados como valores em atribuições

```
//expressions.rs
...
let x = 5u32;
let y = {
    let x_quadrado = x * x;
    let x_cubo = x_squared * x;

    // esta expressao vai ser atribuida a 'y'
    x_cube + x_squared + x // a falta do ';' define o retorno
};

// O ';' suprime a expressao dentro do {}
let z = {
    // logo a expressao {} não tem valor, retornando '()' 
    2 * x;
};

println!("x é {:?}", x, y, z);
}
```

```
$ rustc expressions.rs
$ ./expressions
> x é 5, y é 155 e z é ()
```

## 3. Programação em Rust

Nesse capítulo veremos como programar em Rust usando seu conjunto básico de funcionalidade.

- Entrada e Saida
- Controle de Fluxo
- Laços
- Funções
- Módulos

Ao final entenderemos sobre o processo de entrada e saída de rust, como controlar o fluxo de código com condicionais e switchs e como criar repetições como abstrair código com funções e módulos.

### 3.1 Entradas e saídas

Entrada e saída em Rust são gerenciadas por uma série de Macros definidas em `std::fmt` algumas dessas macros incluem:

- `format!` escreve texto formatado para `String` resultado
- `print!` faz o mesmo que `format!` só que escreve no console
- `println!` faz o mesmo que `print!` só que com uma nova linha ao final
- `eprint!` faz o mesmo que `print` mas escreve em erro padrão (`io::stderr`)
- `eprintln!` faz o mesmo que `eprint!` mas com uma nova linha ao final

Para podermos usar esses métodos de entrada e saída precisamos primeiro entender como a interpolação de string funciona:

### 3.1.1 Interpolação

Para interpolar em uma *String* usamos chaves '' dentro da string seguido do parametro que queremos substituir fora da *String* por exemplo:

```
interpolation.rs
fn main() {
    print("Ola {} numero {}", "terra", 4);
}
```

Neste exemplo podemos ver que as chaves pegam em ordem os argumentos, assim o primeiro par de chaves pega "terra" e o segundo pega "4".

Caso voce deseje tambem e possivel nomear as variaveis que voce deseja dentro das chaves para pega-las em quaisquer ordem atraves de seu nome.

```
interpolation.rs
fn main() {
    print("Ola {planeta} numero {numero}", planeta = "terra", numero = 4);
}
```

Outra maneira de interpolar em strings e numerar as chaves com qual variavel voce deseja usar:

```
interpolation.rs
fn main() {
    print("{0} gosta de {1} mas {1} nao gosta de {0}", "Joao", "Alice");
}
```

Dentro da chaves tambem e possivel formatar os dados sendo interpolados:

```
interpolation.rs
fn main() {
    println!("Base 10: {}", 69420); // 69420
    println!("Base 2 (binary): {:b}", 69420); // 10000111100101100
    println!("Base 8 (octal): {:o}", 69420); // 207454
    println!("Base 16 (hexadecimal): {:x}", 69420); // 10f2c
}
```

Tambem e possivel justificar texto:

```
interpolation.rs
fn main() {
    println!("{number:>5}", number=1); //      1"
    println!("{number:<5}", number=1); // "1      "
    // Voce pode substituir o caractere do padding pelo caracter que quiser:
    println!("{number:0>5}", number=1); // "00001"
    println!("{number:0<5}", number=1); // "10000"
    // e possivel usar argumentos nomeados com "$"
    println!("{number:0$width}", number = 1, width = 5); // "00001"
}
```

### 3.1.2 Format!

*Format!* recebe um string e variaveis interpola as variaveis na string e retorna como outra string para ser usada no codigo.

```
format.rs
fn main() {
    let x = format!("Olá {}!", "Mundo");
    print!(x); // Resultado: "Olá Mundo!"
}
```

### 3.1.3 Print!

*Print!* faz o mesmo que *Format!* so que escreve o resultado em std::out (console/terminal)

```
print.rs
fn main() {
    print!("Olá {}!", "Mundo");
    print!("Olá {}!", "Mundo");
    // Resultado: "Olá Mundo!Olá Mundo!"
}
```

### 3.1.4 Println!

*Println!* faz o mesmo que *Print!* so que acrescenta uma nova linha ao final.

```
println.rs
fn main() {
    println!("Olá {}!", "Mundo");
    println!("Olá {}!", "Mundo");
    // Resultado: "Olá Mundo!
    //             Olá Mundo!"
```

### 3.1.5 Saídas de Erros

*Eprint!* e *Eprintln!* funcionam igual a *Print!* e *Println!* so que sua saida de texto e em io:stderr, isso significa que essas mensagens ao ocorrer fecharao seu programa com base em uma condicao e mostraraoo essa mensagem no cosole apos o programa ter fechado.

```
errors.rs
fn main() {
    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });
}
```

Se tentarmos rodar esse programa com argumentos insuficientes teremos:

```
$ cargo run > out.txt
> Problem parsing arguments: not enough arguments
```

## 3.2 Controle de Fluxo

Temos multiplas maneiras de controlar quais pedacos de codigo queremos executar e quantas vezes queremos executar, pra isso se da o nome de controle de fluxo, pois estamos controlando por onde e como nosso programa passa e executa, algumas estruturas que usamos para isso sao:

### 3.2.1 If e If else

O *if* deixa voce avaliar expressoes logicas e com base em seu resultado executar trechos especificos de codigo:

```
if.rs
fn main() {
    let x = 5;
    if x < 6 {
        println!("X e menor que 6");
    } else if x == 6{
        println!("X e 6");
    } else {
        println!("X e maior que 6");
    }
}
```

Aqui criamos uma condicao logica onde se x for menor que 5 temos o resultado: "X e menor que 6" que e a primeira condicao do if e so acontece caso a expressao logica seja verdadeira, ao mudar x para 6 caimos no segundo caso onde olhamos outras possibilidades e avaliamos elas novamente com outro if atraves da concatenacao de um "else" e um "if", entao conseguimos o resultado "X e 6", e por ultimo caso mudemos x para 7 cairemos no caso else que so acontece caso todas as expressoes logicas anteriores sejam falsas, e teremos o resultado: "X e maior q 6".

### 3.2.2 Match

A estrutura match e extramente semelhante a estrutura de *Switch Case* em C, onde podemos especificar varios casos e usar casamento de padroes em cima de uma variavel alvo para decidir qual caso executar:

```
match.rs
fn main() {
    let x = 12;
    match number {
        // Podemos casar o padrao com um unico valor
        1 => println!("One!");
        // Com multiplos valores
        2 | 3 | 5 | 7 | 11 => println!("Numero primo!");
        // Com intervalos inclusivos
        13..=19 => println!("Um Adolescente!");
        // Lidar com quaisquer outro caso possivel
        // que nao tenha sido especificado
        _ => println!("Nada de especial");
    }
}
```

### 3.2.3 Laços de Repetição

Estruturas de laço de repetição nos permitem repetir trechos de código de diversas maneiras, desde enquanto uma expressão lógica for verdade até infinitamente:

O primeiro e mais simples laço que existe é o *loop*, uma estrutura que executa seu escopo infinitamente a não ser que encontre um *break*.

```
loops.rs
fn main() {
    loop {
        println!("Olá Mundo!");
    }
    \\ Resultado:
    \\ "Olá Mundo!"
    \\ "Olá Mundo!"
    \\ "..."
    \\ Executara infinitamente
}
```

O laço *while* pode ser usado para executar um trecho de código até que uma condição se torne falsa

```
loops.rs
fn main() {
    let mut x = 1;
    while x < 10 {
        println!("Olá Mundo!");
        x += 1;
    }
    \\ Resultado: Escreverá "Olá Mundo!" dez vezes
}
```

O último laço o *for range* permite que seja extraído um dado de uma estrutura e o use durante o laço que está sendo executado:

```
loops.rs
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];
    for x in 1..=100 { println!("{}", x); }
    \\ Resulta em todos os números de 1 a 100 sendo escritos
    \\ Aqui vemos duas estruturas de controle de fluxo
    \\ sendo usadas em paralelo
    for name in names.iter() {
        match name {
            &"Ferris" => println!("Tem um Rustaceano entre nós"),
            _ => println!("Olá {}", name),
        }
    }
    \\ Resultado: caso ele ache o nome ferris no vetor ele escreverá
    \\ "Tem um Rustaceano entre nós" e "Olá Ferris"
    \\ Caso o nome não esteja no vetor nada será escrito
}
```

### 3.3 Funções

Funções em Rust são declaradas usando a palavra chave "fn", seus valores de entrada devem ser *type-annotated* e se a função possui valor de retorno ele deve ser indicado usando "->"

A última expressão dentro de uma função será usada como valor de retorno, alternativamente a palavra chave *return* pode ser usada para retornar um valor antes da última expressão, mesmo dentro de laços de repetição ou estruturas lógicas.

Diferente de C não há restrição na ordem da definição de funções, lhe permitindo chamar uma função no código, em linhas anteriores à sua definição.

### Exemplo:

```
functions.rs
fn main() {
    // Chamada antes da definição é possível
    if is_odd(3) {
        println!("Impar");
    } else {
        println!("Par");
    }
    hello("Sophia");
}

fn is_odd(n: i32) -> bool {
    if n % 2 == 0 { return false; }
    // Para explicitar última expressão não se inclui ";"
    n % 2 == 0
}

// Quando uma função não retorna nada ela na verdade retorna "()"
// Quando a função retorna o tipo unitário "()"
// O retorno pode ser omitido
fn hello(x: str) { println!("Hello {}!", x); }
```

### 3.4 Módulos

Modulos sao uma colecao de itens como funcoes, structs, *traits*, *impl* onde voce pode acessar todos seus elementos.

```
functions.rs
mod meu_modulo{
    // Todas as funcoes em modulos sao privadas por padrao
    fn helloMod() {
        println!("Hello Module!");
    }
    // A funcao pode ser tornada publica pela palavra chave 'pub'
    pub fn helloWorld() {
        println!("Hello World");
    }
    // Itens dentro de um modulo podem ser acessados por outros itens
    // Modulos podem ser aninhados
    //
}

fn main() {
    // Se acessa itens dentro de modulos usando '::'
    meu_modulo::helloWorld();
}
```



## 4. Aplicações da Linguagem Rust

Neste capítulo veremos diversos exemplos de algoritmos escritos em Rust

### 4.1 Insertion Sort

Organiza um vetor de números usando o método de inserção

```
insertion.rs
pub fn sort(mut v: Vec<i32>) -> Vec<i32> {
    let n = v.len();
    for i in 1..n {
        let mut j = i;
        while j > 0 && v[j] < v[j - 1] {
            v.swap(j, j - 1);
            j -= 1;
        }
    }
    v
}
```

Este código é original deste livro

### 4.2 Linked List

Algoritmo para criação de um estrutura de lista encadeada

```
list.rs
pub struct List<T> {
    head: Link<T>,
}
```

```
type Link<T> = Option<Box<Node<T>>>;  
  
struct Node<T> {  
    elem: T,  
    next: Link<T>,  
}
```

```
impl<T> List<T> {  
    pub fn new() -> Self {  
        List { head: None }  
    }  
  
    pub fn push(&mut self, elem: T) {  
        let new_node = Box::new(Node {  
            elem: elem,  
            next: self.head.take(),  
        });  
  
        self.head = Some(new_node);  
    }  
  
    pub fn pop(&mut self) -> Option<T> {  
        self.head.take().map(|node| {  
            self.head = node.next;  
            node.elem  
        })  
    }  
  
    pub fn peek(&self) -> Option<&T> {  
        self.head.as_ref().map(|node| {  
            &node.elem  
        })  
    }  
}
```

```
pub fn peek_mut(&mut self) -> Option<&mut T> {
    self.head.as_mut().map(|node| {
        &mut node.elem
    })
}

pub fn into_iter(self) -> IntoIter<T> {
    IntoIter(self)
}

pub fn iter(&self) -> Iter<'_, T> {
    Iter { next: self.head.as_deref() }
}

pub fn iter_mut(&mut self) -> IterMut<'_, T> {
    IterMut { next: self.head.as_deref_mut() }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}

pub struct IntoIter<T>(List<T>);

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        // access fields of a tuple struct numerically
        self.0.pop()
    }
}

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}
```

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_deref();
            &node.elem
        })
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.take().map(|node| {
            self.next = node.next.as_deref_mut();
            &mut node.elem
        })
    }
}
```

```

main.rs
fn main() {
    let mut list = List::new();

    // Check empty list behaves right
    println!("Pop: {:?}", list.pop()); // Should print "Pop: None"
    println!("Peek: {:?}", list.peek()); // Should print "Peek: None"
    println!("Peek Mut: {:?}", list.peek_mut()); // Should print "Peek Mut: None"

    // Populate list
    list.push(1);
    list.push(2);
    list.push(3);

    // Check normal removal
    println!("Pop: {:?}", list.pop()); // Should print "Pop: Some(3)"
    println!("Pop: {:?}", list.pop()); // Should print "Pop: Some(2)"

    // Push some more just to make sure nothing's corrupted
    list.push(4);
    list.push(5);

    // Check normal removal
    println!("Pop: {:?}", list.pop()); // Should print "Pop: Some(5)"
    println!("Pop: {:?}", list.pop()); // Should print "Pop: Some(4)"

    // Check exhaustion
    println!("Pop: {:?}", list.pop()); // Should print "Pop: Some(1)"
    println!("Pop: {:?}", list.pop()); // Should print "Pop: None"

    // Peek
    list.push(6);
    println!("Peek: {:?}", list.peek()); // Should print "Peek: Some(&6)"
    println!("Peek Mut: {:?}", list.peek_mut()); // Should print "Peek Mut: Some(&mut 6)"
}

```

```

sophia ➜ .../Rust/Programas/src [master ✘ v1.78.0 ✘ 13:42 ✘ cargo run
Compiling Programas v0.1.0 (/home/sophia/dev/MyMonoRepo/Univeristy/Paradigms/Rust/Programas)
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.14s
    Running `/home/sophia/dev/MyMonoRepo/Univeristy/Paradigms/Rust/Programas/target/debug/Programas`  

Pop: None
Peek: None
Peek Mut: None
Pop: Some(3)
Pop: Some(2)
Pop: Some(5)
Pop: Some(4)
Pop: Some(1)
Pop: None
Peek: Some(6)
Peek Mut: Some(6)

```

Fonte: <https://rust-unofficial.github.io/too-many-lists/>

### 4.3 Quick Sort

Organiza um vetor de números usando o método de partição conhecido como quicksort

```
quick.rs
extern crate std as core;

use core::cmp::Ordering;

fn quicksort_helper<T, F>
    (arr: &mut [T], left: isize, right: isize, compare: &F)
where F: Fn(&T, &T) -> Ordering {
    if right <= left {
        return
    }

    let mut i: isize = left - 1;
    let mut j: isize = right;
    let mut p: isize = i;
    let mut q: isize = j;

    unsafe {
        let v: *mut T = &mut arr[right as usize];
        loop {
            i += 1;
            while compare(&arr[i as usize], &*v) == Ordering::Less {
                i += 1
            }
            j -= 1;
            while compare(&*v, &arr[j as usize]) == Ordering::Less {
                if j == left {
                    break
                }
                j -= 1;
            }
            if i >= j {
                break
            }
            arr.swap(i as usize, j as usize);
            if compare(&arr[i as usize], &*v) == Ordering::Equal {
                p += 1;
                arr.swap(p as usize, i as usize)
            }
            if compare(&*v, &arr[j as usize]) == Ordering::Equal {
                q -= 1;
                arr.swap(j as usize, q as usize)
            }
        }
    }
}
```

```
arr.swap(i as usize, right as usize);
j = i - 1;
i += 1;
let mut k: isize = left;
while k < p {
    arr.swap(k as usize, j as usize);
    k += 1;
    j -= 1;
    assert!(k < arr.len() as isize);
}
k = right - 1;
while k > q {
    arr.swap(i as usize, k as usize);
    k -= 1;
    i += 1;
    assert!(k != 0);
}

quicksort_helper(arr, left, j, compare);
quicksort_helper(arr, i, right, compare);
}
```

```
pub fn quicksort_by<T, F>
    (arr: &mut [T], compare: F) where F: Fn(&T, &T) -> Ordering {
    if arr.len() <= 1 {
        return
    }

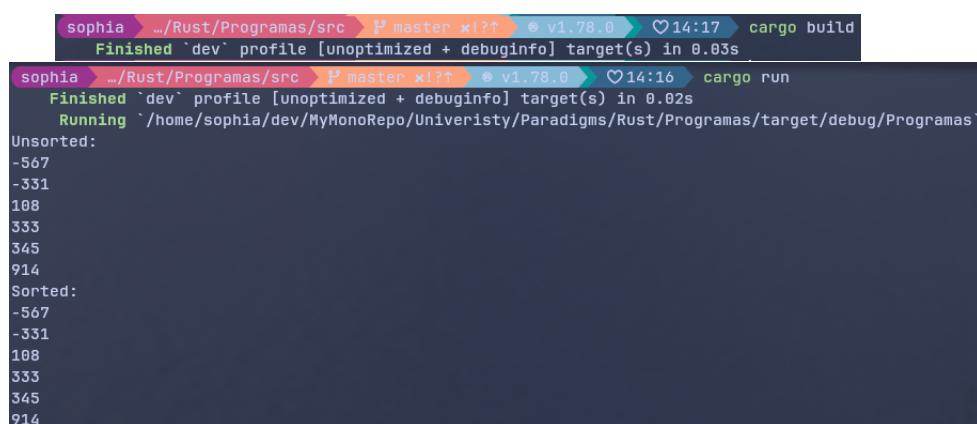
    let len = arr.len();
    quicksort_helper(arr, 0, (len - 1) as isize, &compare);
}

/// An in-place quicksort for ordered items.
#[inline]
pub fn quicksort<T>(arr: &mut [T]) where T: Ord {
    quicksort_by(arr, |a, b| a.cmp(b))
}
```

```

fn main() {
    let mut rng = rand::thread_rng();
    let len: usize = rng.gen();
    let mut v: Vec<isize> = rng.gen_iter::<isize>().take((len % 32) + 1).collect();
    for i in 0 .. v.len() - 1 {
        v[i] = v[i] % 1000;
    }
    quicksort(&mut v);
    println!("Unsorted:");
    for i in 0 .. v.len() - 1 {
        println!("{}", v[i])
    }
    println!("Sorted:");
    for i in 0 .. v.len() - 1 {
        println!("{}", v[i])
    }
}

```



```

sophia ~/Rust/Programas/src P master x!?@ v1.78.0 14:17 cargo build
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
sophia ~/Rust/Programas/src P master x!?@ v1.78.0 14:16 cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
    Running `/home/sophia/dev/MyMonoRepo/Univeristy/Paradigms/Rust/Programas/target/debug/Programas`
Unsorted:
-567
-331
108
333
345
914
Sorted:
-567
-331
108
333
345
914

```

Fonte: <http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf>

#### 4.4 IsPrime

Verifica se um numero é primo

```
prime.rs
fn main() {
    let num = 17;
    if is_prime(num) {
        println!("{} is prime", num);
    } else {
        println!("{} is not prime", num);
    }
}

fn is_prime(n: u32) -> bool {
    if n <= 1 {
        return false;
    }
    for i in 2..=n / 2 {
        if n % i == 0 {
            return false;
        }
    }
    true
}
```

```
sophia ~/Rust/Programas/src 🐾 master ✘ v1.78.0 🌟 15:01 cargo build
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
sophia ~/Rust/Programas/src 🐾 master ✘ v1.78.0 🌟 15:01 cargo run
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
  Running `/home/sophia/dev/MyMonoRepo/Univeristy/Paradigms/Rust/Programas/target/debug/Programas`
17 is prime
```

Este código é original deste livro

## 4.5 Binary Search

Verifica se um numero é primo

```
binary.rs

fn main() {
    let arr = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19];
    let target = 1;

    if let Some(index) = binary_search(&arr, target) {
        println!("Element {} found at index {}", target, index);
    } else {
        println!("Element {} not found in the array", target);
    }
}

fn binary_search(arr: &[i32], target: i32) -> Option<usize> {
    let mut left = 0;
    let mut right = arr.len() - 1;

    while left <= right {
        let mid = left + (right - left) / 2;
        if arr[mid] == target {
            return Some(mid);
        } else if arr[mid] < target {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    None
}
```

```
sophia ~/Rust/Programas/src ▶ P master x!?↑ v1.78.0 15:09 cargo build
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
sophia ~/Rust/Programas/src ▶ P master x!?↑ v1.78.0 15:09 cargo run
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
Running `/home/sophia/dev/MyMonoRepo/Univeristy/Paradigms/Rust/Programas/target/debug/Programas`
```

Este código é original deste livro

## 5. Ferramentas

Aqui exploraremos as ferramentas que permitem que a promoção em Rust seja facil e rapido, e disponibilize para nos o que precisamos para achar erros nos nossos codigos compila-los e publicalos

### 5.1 RustUp

Download: <https://rustup.rs/>

Rustup é uma ferramenta que baixa configura e instala todo o ecossistema Rust para você sem que você tenha que se preocupar com sistema, versões, etc.

Atualmente se encontra na versão 1.27.1

Para usar rustup basta que você apenas rode o seguinte comando no seu terminal:

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

### 5.2 Cargo

Cargo é o package manager e project manager do Rust, isso significa que ele é responsável por pegar as dependências dos seus projetos e gerenciar seus projetos.

Cargo é instalado pelo programa Rustup.

Para usá-lo basta apenas iniciar um projeto com "cargo init" isso criará uma pasta com o arquivo "Cargo.toml" e uma pasta src com um arquivo "main.rs", no arquivo Cargo.toml basta apenas escrever os nomes das dependências que você deseja incluir em seu projeto. Para testar o seu projeto use "Cargo run", para criar um novo módulo basta usar "Cargo new" e para compilar e exportar o seu projeto basta usar "Cargo build".



## **6. Considerações Finais**

O trabalho que foi desenvolvido em forma resumida explica as origens e funcionamento da linguagem Rust, dando ao leitor conhecimento suficiente da linguagem e seu ecossistema para que continue sozinho e desenvolva incríveis aplicações.

Os livros usados de referência foram:

<https://doc.rust-lang.org/stable/book/>

<https://doc.rust-lang.org/stable/rust-by-example/>





## Referências Bibliográficas

- [Bin14] Andrew Binstock. The rise and fall of languages in 2013, 2014. <https://drdobbs.com/jvm/the-rise-and-fall-of-languages-in-2013/240165192> [Accessed: (04.04.2024)]. Citado na página 7.
- [Gou15] Carol Nichols Goulding. Rustlings, 2015. <https://rustlings.cool/> [Accessed: (22.04.2024)]. Citado na página 9.
- [Gou16] Carol Nichols Goulding. The rust book, 2016. <https://doc.rust-lang.org/book/> [Accessed: (22.04.2024)]. Citado na página 9.
- [Kla14] Steve Klabnik. Rust by example, 2014. <https://doc.rust-lang.org/rust-by-example/> [Accessed: (22.04.2024)]. Citado na página 9.
- [Wes23] David Weston. Bluehat il 2023 - david weston - default security, 2023. <https://www.youtube.com/watch?v=8T6C1X-y2AE> [Accessed: (04.04.2024)]. Citado na página 8.
- [Wu24] Yuchen Wu. Open sourcing pingora: our rust framework for building programmable network services, 2024. <https://blog.cloudflare.com/pingora-open-source> [Accessed: (04.04.2024)]. Citado na página 8.

