Uma Introdução à Computação com Julia

Alfredo Goldman Lucas de Sousa Rosa

2025-04-05

Índice

Página Inicial **5**

Agradecimentos **6**

I Parte I: Conceitos Básicos **7**

1 Uma Breve Apresentação da História dos Computadores e Linguagens de Programação **8**
1.1 Motivação 8
1.2 Computadores Precisam Existir? 8
1.3 Ábacos, Máquinas de Calcular e Quase-Computadores 9
1.4 O Nascimento da Computação Moderna 10

2 Usando o Interpretador (REPL) como Calculadora **13**
2.1 Explorando a Sessão Interativa de Julia 13
2.2 Variáveis e Tipos de Dados 19
2.3 Saída de Dados 22
2.4 Arquivos Externos e Módulos 24
2.4.1 O que é um arquivo `.jl`? 26
2.5 Verifique seu Aprendizado 26
2.6 Explore por Conta Própria 26

3 Estruturas de Controle e Tomada de Decisões **28**
3.1 Operadores de Comparação e o Tipo Booleano 28
3.1.1 Operadores Lógicos 29
3.2 Alterando o Fluxo de Execução com if-else 30
3.2.1 A Estrutura if 31
3.2.2 Adicionando Alternativas com else 31
3.2.3 Múltiplas Condições com elseif 32
3.3 Verifique seu Aprendizado 32
3.4 Explore por Conta Própria 33

4 Introdução às Funções **34**
4.1 Funções como Abstrações Naturais 34
4.1.1 Funções Chamando Outras Funções 35

4.1.2 Acessando a Documentação de Funções . . . 36
4.1.3 Funções de Conversão . . . 36
4.1.4 Funções Matemáticas . . . 37
4.1.5 Sobrecarga de Funções . . . 37
4.2 Funções que Retornam Valores . . . 38
4.3 Introdução à Recursão . . . 40
4.3.1 Mais Exemplos de Recursão . . . 41
4.4 Verifique seu Aprendizado . . . 43
4.5 Explore por Conta Própria . . . 43

5 Mais Problemas Envolvendo Recursão 44
5.1 Pensando Recursivamente . . . 44
5.2 Problema das Escadas . . . 44
5.3 Potenciação por Quadrados . . . 46
5.4 Cálculo da Raiz Quadrada (Método de Heron) . . . 48
5.5 Coeficiente Binomial . . . 50

II Parte II: Estruturas Fundamentais e Boas Práticas 52

6 Mais algoritmos e introdução aos testes 53

7 Testes automatizados e um pouco mais de código 59
7.1 Funções caóticas . . . 62

8 Uma outra forma de se fazer laços 63

9 Aula de exercícios 66
9.1 Revisitando o cálculo do fatorial, recursivo e interativo . . . 66
9.2 Aproximação da raiz quadrada . . . 67
9.3 Verificar se um número é primo . . . 68
9.4 Verificar se um número é palíndromo . . . 69

10 Revisitando a aula passada 70
10.1 Aleatoreidade . . . 71

11 Entrada de dados e o começo de listas 75
11.1 O comando input . . . 75
11.2 Lendo através da linha de comando . . . 77
11.3 Listas . . . 78
11.3.1 Álgebra linear e Listas . . . 80
11.3.2 Exercício de permutação . . . 81

12 Exercícios com vetores **83**
12.1 Permutação 83
12.2 Histograma 85
12.3 Modelando problemas com o computador 86

13 Modelando um problema maior **89**

14 Continuando a modelagem **93**

15 Boas práticas **97**
15.1 Uso de contratos 97
15.1.1 Boa prática 1: Use tipos 97
15.2 Testes automatizados 97
15.2.1 Boa prática 2: Sempre que possível faça testes 98
15.3 Escreva código para humanos, não para computadores 98
15.3.1 Boa prática 3: Escreva código para que outros leiam 98
15.4 Aplicando as boas práticas 98

III Parte III: Conceitos Avançados **101**

16 Indo além de uma dimensão (Matrizes) **102**

17 Aula de exercícios sobre Strings **107**
17.1 Concatenação de letras 107
17.2 Inversão de String 108
17.2.1 Função reverse 109
17.3 Modificação de String 109
17.4 Rearranjo de letras 111
17.5 Encontrar a maior palavra 112
17.6 Retorno de múltiplos valores 114

IV Em andamento... **116**

Apêndices **117**

A Respostas para a pergunta "O que é um computador?" **117**

Página Inicial

Seja bem-vindo(a) ao livro "Uma Introdução à Computação com Julia". Este livro reúne as notas de aula da disciplina MAC0115 - Introdução à Computação para Ciências Exatas e Tecnologia. Com linguagem simples e objetiva, o livro apresenta os principais conceitos de programação, complementados por exemplos práticos de código.

Esta disciplina visa introduzir os fundamentos da computação através de um percurso histórico até conceitos práticos de programação utilizando a linguagem Julia. Cobre algoritmos, arquitetura de computadores, linguagens algorítmicas (com expressões, comandos, estruturas de dados e funções), metodologias de desenvolvimento e boas práticas de programação, além de proporcionar uma extensa prática de programação e depuração, formando uma base técnica completa para iniciantes.

O livro está sendo atualizado juntamente com a disciplina. Podem ocorrer mudanças na ordem e nos exemplos, mas nada que afete o conteúdo ministrado. Não se preocupe se algo não estiver exatamente onde você esperava encontrar.

Agradecimentos

Diversas pessoas contribuíram, direta ou indiretamente, para a realização deste livro.

O autor principal é o Professor Alfredo Goldman. A revisão do texto e o suporte técnico ficaram a cargo de Lucas de Sousa Rosa, seu orientando de doutorado. Agradecemos, em especial, a Beatriz Viana Costa, que colaborou na redação de alguns capítulos. Agradecemos também os alunos da disciplina MAC0115 (Introdução à Computação para Ciências Exatas e Tecnologia), turma de 2024, por suas contribuições na revisão do conteúdo.

Parte I

Parte I: Conceitos Básicos

1 Uma Breve Apresentação da História dos Computadores e Linguagens de Programação

1.1 Motivação

Mas, afinal, o que é um computador? Precisamos mesmo saber a resposta para começar a programar? Na verdade, não. Porém, entender como um computador funciona nos torna bons programadores. Definir o que é um computador pode ser mais complicado do que parece. Veremos que a ideia de "computador" mudou muito ao longo da história, sempre ligada à necessidade de contar. Antes de mergulhar na história, vamos explorar diferentes respostas para essa pergunta.

Logo no começo das aulas de 2025, na matéria MAC115 - Introdução à Computação para Ciências Exatas e Tecnologia, fizemos uma pergunta simples aos alunos: "O que é um computador?". Anotamos as respostas (ver Apêndice A) e apesar de algumas respostas engraçadas, como "um robô que vai controlar os humanos um dia", notamos que a maioria dos alunos pensava em coisas parecidas:

1. Um computador é um objeto físico; uma máquina ou dispositivo.
2. Ele é capaz de processar informações ou dados.
3. É como uma calculadora, só que mais moderna.

Essas ideias mostram um pouco do que um computador faz, mas não explicam tudo o que ele é.

1.2 Computadores Precisam Existir?

O Professor Douglas Hartree (1897–1958), no seu livro "Calculating instruments and machines", diz que computadores podem ser vistos de dois ângulos: anatômico (do que são feitos) e fisiológico (como funcionam). Tentar definir um computador pelas suas partes não seria suficiente, já que as peças dos computadores mudaram muito ao longo dos anos, então usar a anatomia para defini-los não funcionaria.

Na mesma época, o Professor Arthur L. Samuel propõe uma definição funcional do que é um computador. Segundo Samuel, um computador é "um dispositivo de processamento de informações ou dados que aceita dados em uma forma e os entrega em uma forma alterada". Esta concepção técnica concorda com o que os alunos pensam. No entanto, essa definição nos leva a um questionamento mais profundo: será que computadores precisam necessariamente existir como dispositivos ou máquinas físicas, ou podemos conceber computação para além do aspecto material?

Se perguntássemos para uma pessoa na rua o que ela acha que é um computador, é bem provável que ela diria que é uma "máquina" ou um "dispositivo". De fato, nas respostas que coletamos, essa ideia aparece com frequência. No entanto, os computadores modernos foram criados a partir de um modelo matemático abstrato, chamado Máquina de Turing.

Quando criança, aos 11 anos, Alan Mathison Turing escreveu uma carta para seus pais, Julius Mathison Turing e Ethel Sara Stoney Turing, contando sua ideia de como fazer uma máquina de escrever. Alan Turing não inventou a máquina de escrever, mas usou essa ideia para criar a Máquina de Turing: um modelo matemático de computação que manipula símbolos numa fita infinita, seguindo regras bem definidas. Esse modelo deu origem aos computadores modernos que, segundo John A. Robinson, nada mais são do que "manifestações físicas de uma mesma abstração lógica: a máquina universal de Turing." Então, não. Computadores não precisam existir.

1.3 Ábacos, Máquinas de Calcular e Quase-Computadores

É aqui que a gente começa a falar da história da computação. Nós, humanos, somos muito criativos. Estamos sempre criando ferramentas para nos ajudar. Existem evidências arqueológicas que sugerem que os humanos praticam a contagem há pelo menos 50.000 anos. A contagem era usada por povos antigos para controlar dados sociais e econômicos, como o número de pessoas no grupo, animais, propriedades ou dívidas. Então, não é surpresa que os humanos tenham inventado ferramentas para ajudar nesse processo. Em 2700–2300 AC, os humanos criaram o ábaco: uma ferramenta para fazer contas mais rápido.

Ao longo dos séculos, outras ferramentas foram criadas para ajudar nisso. Um exemplo é a régua de cálculo, do século XVII, que fazia a mesma coisa que o ábaco, mas usando propriedades da funções logarítmicas. Os primeiros computadores eram como versões mais modernas dessas ferramentas, ou seja, máquinas de calcular. O engenho diferencial (1820) de Charles Babbage, uma máquina idealizada, mas não construída, é um exemplo de máquina de calcular feita para calcular funções polinomiais.

O inglês Charles Babbage era matemático, filósofo, inventor e engenheiro mecânico, e é considerado por muitos como o "pai do computador". Esse título é por causa de outra máquina que ele idealizou. O engenho analítico (1840), diferente de todas as máquinas da época, marcou a transição da aritmética mecanizada para a computação de propósito geral. Essa máquina seria

programada por meio de cartões perfurados e incorporaria diversos recursos posteriormente adotados em computadores modernos, como controle sequencial, estruturas de ramificação e mecanismos de repetição (laços).

Durante a criação do engenho analítico, a condessa de Lovelace, Augusta Ada Byron King (filha do poeta Lord Byron), criou um algoritmo para calcular a sequência dos números de Bernoulli, e por isso é considerada a primeira programadora. Hoje em dia, a figura de Ada é usada como inspiração para aumentar a presença de mulheres na computação, que era bem maior algumas décadas atrás.

Até então todas as máquinas desenvolvidas eram estritamente mecânicas. O Z2 foi um dos primeiros exemplos de um computador digital operado eletricamente, construído com relés eletromecânicos, e foi criado pelo engenheiro civil Konrad Zuse em 1940 na Alemanha. Mas, foi a partir da invenção das válvulas termiônicas (tubos de vácuo) que damos o primeiro salto na era dos computadores digitais. As válvulas eram dispositivos totalmente elétricos que geravam corrente elétrica a partir do fenômeno de emissão termiônica e foram inventadas pelo físico John Ambrose Fleming em 1904.

O Z3, sucessor do Z2, também foi desenvolvido por Zuse e é considerado o primeiro computador digital programável e totalmente automático do mundo. Konrad Zuse também foi responsável pelo design do Plankalkül, a primeira linguagem de programação de alto nível, ou seja, mais próxima da linguagem humana e mais distante da linguagem de máquina. Embora esta linguagem nunca tenha sido implementada na época, ela introduziu conceitos fundamentais da programação moderna, como tipos de dados e estruturas de controle.

Máquinas como o Z3, Colossus e o ENIAC foram construídas manualmente, usando circuitos contendo relés ou válvulas e frequentemente usavam cartões perfurados ou fita de papel perfurada para entrada e como principal meio de armazenamento. A maioria das máquinas desse período não possuía a capacidade de armazenar e modificar programas (com exceção do Z3). No ENIAC, um "programa" era definido pela configuração de seus cabos de conexão e interruptores, característica que distingue essas máquinas dos computadores modernos. A programação nessa época era exercida majoritariamente por mulheres, porém essa predominância foi gradualmente diminuindo com o passar dos anos.

1.4 O Nascimento da Computação Moderna

Até o momento, todas as máquinas (Babbage, Zuse, Colossus) seguiam o mesmo design proposto por Babbage: construíam-se máquinas para realizar cálculos e, então, organizavam-se instruções codificadas em alguma outra forma, armazenadas separadamente, para fazê-las funcionar. A grande mudança de paradigma ocorreu em 1945 com as ideias de Alan Turing e John von Neumann. Ambos perceberam que os programas deveriam ser armazenados da mesma forma que os dados. O Manchester Baby foi o primeiro computador eletrônico de programa armazenado do mundo e executou seu primeiro programa em 21 de junho de 1948. A arquitetura

de von Neumann marca o início da era dos computadores modernos. Desde então, avanços foram feitos para torná-los mais rápidos, menores e fáceis de usar, porém seu cerne permanece o mesmo: a universalidade inerente ao computador de programa armazenado.

Na mesma década, Kathleen Booth, trabalhando no mesmo lugar que von Neumann, desenvolveu a primeira linguagem assembly—uma linguagem de programação muito próximo da linguagem de máquina. Porém, programar em assembly demandava um esforço intelectual considerável. As linguagens que surgiram posteriormente representaram tentativas de abstrair a linguagem de máquina para uma linguagem de alto nível, mais próxima da linguagem natural, humana. Embora diversas linguagens tenham sido criadas nos anos seguintes, foi apenas em 1954 que surgiu a primeira linguagem amplamente adotada: FORTRAN, desenvolvida na IBM por uma equipe liderada por John Backus. Atualmente, mais de 70 anos depois, FORTRAN ainda é utilizada para classificar a lista dos TOP500 supercomputadores mais rápidos do mundo.

A partir de 1955, a tecnologia dos transistores revolucionou a computação ao substituir os tubos de vácuo no design de computadores. Os transistores apresentavam vantagens significativas: eram menores e consumiam menos energia, consequentemente gerando menos calor que seus predecessores. O marco dessa transição foi o TRADIC Phase One, concluído em 1954, considerado o primeiro computador totalmente transistorizado.

A evolução tecnológica prosseguiu com a invenção dos circuitos integrados em 1958 por Jack Kilby, que posteriormente foi reconhecido com o Prêmio Nobel nos anos 2000 por essa contribuição. Um circuito integrado consiste em um conjunto de circuitos eletrônicos compostos por diversos componentes (transistores, resistores e capacitores) e suas interconexões. Esses componentes são minuciosamente gravados em uma pequena peça plana, conhecida como “chip”, fabricada com material semicondutor—inicialmente germânio e, atualmente, silício.

O período compreendido entre o final da década de 1960 e o final dos anos 1970 foi marcado pelo surgimento de diversas linguagens de programação, bem como pela consolidação dos principais paradigmas que conhecemos hoje. Simula, criada pelos cientistas da computação noruegueses Ole-Johan Dahl e Kristen Nygaard, destacou-se como a primeira linguagem projetada especificamente para suportar a programação orientada a objetos. Simultaneamente, Dennis Ritchie e Ken Thompson desenvolviam nos Bell Labs, entre 1969 e 1973, a linguagem C, voltada para programação de sistemas. É importante ressaltar que o desenvolvimento da linguagem C esteve intrinsecamente ligado ao sistema operacional UNIX, já que C foi criada justamente para facilitar a portabilidade deste sistema entre diferentes plataformas de hardware. Nesse mesmo contexto de inovação, a linguagem ML, concebida pelo cientista britânico Robin Milner, emergiu como pioneira entre as linguagens de programação funcional com tipagem estática.

Em termos de hardware, o microprocessador permitiu o último grande salto na história da computação. Sua evolução só foi possível graças aos circuitos integrados MOS (CMOS), que permitiram a progressiva miniaturização dos transistores. Atualmente, já somos capazes de produzir transistores com dimensões da ordem de 50 nanômetros, o que é surpreendente quando consideramos que o raio de um átomo de silício é de aproximadamente 0,13 nanômetros.

Por outro lado, a evolução das linguagens de programação foi particularmente marcante nos anos 1990, impulsionada pelo rápido crescimento da Internet. Nesse período, a produtividade dos programadores tornou-se algo importante, o que levou ao surgimento de diversas linguagens de desenvolvimento rápido de aplicativos (RAD). Essas linguagens eram acompanhadas por ambientes de desenvolvimento integrados (IDEs) e recursos de coleta de lixo. Como resultado dessa tendência, surgiram linguagens como Python, Lua, R, Ruby, Java, JavaScript e PHP.

A crescente popularização de novas linguagens de programação foi impulsionada pelos aplicativos mobile. Celulares e dispositivos móveis tornaram-se cada vez mais presentes no cotidiano das pessoas, aumentando significativamente a demanda por aplicativos. Alguns exemplos dessas linguagens incluem Dart, Kotlin, TypeScript e Swift.

Em 2012, surge a linguagem que será estudada nesta disciplina: Julia. Propondo-se a ser rápida e produtiva, Julia une o desempenho de linguagens antigas, com a produtividade de linguagens recentes. Foi desenvolvida por Jeff Bezanson, Stefan Karpinski, Viral B. Shah e Alan Edelman. O site https://julialang.org/benchmarks/ apresenta um benchmark comparativo de diferentes linguagens na resolução de alguns algoritmos. Pode-se observar que Julia demonstra desempenho equivalente ao da linguagem C.

2 Usando o Interpretador (REPL) como Calculadora

O objetivo deste capítulo é apresentar o interpretador de Julia como uma calculadora poderosa e introduzir os primeiros conceitos de programação: variáveis e funções. Mas primeiro é preciso instalar a linguagem Julia em seu computador. Mais detalhes sobre o processo de instalação podem ser encontrados neste link.

Muito provavelmente seu sistema é Windows (10 ou 11) e sua arquitetura é de 64-bits. Há algumas formas de instalar Julia no Windows:

1. Através de um arquivo executável (`.exe`).
2. Através de comandos pelo terminal (`winget`).

A princípio qualquer uma das opções é adequada. A primeira opção não requer nenhum programa adicional, enquanto que a segunda requer um terminal. Um terminal é um aplicativo que permite a comunicação com o sistema operacional por meio de uma interface de linha de comando (CLI). O terminal padrão do Windows é o Windows Terminal. É fortemente recomendado que você o tenha instalado e isso pode ser feito através da Microsoft Store.

Uma vez que você tenha acesso a um terminal há dois comandos possíveis para instalar Julia: `winget install julia -s msstore` ou `winget install -e --id Julialang.Julia`. Mais uma vez, qualquer uma das opções deve funcionar.

2.1 Explorando a Sessão Interativa de Julia

Você pode abrir uma sessão interativa (também conhecido como um *read-eval-print loop* ou REPL) de Julia digitando o comando `julia` na linha de comando do seu terminal. No Windows, após instalação da linguagem, é possível abrir uma sessão interativa clicando duas vezes no executável Julia. A sua janela deve ser parecida com

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
```

```
  | | |_| | | | (_| |  |  Version 1.11.3 (2025-01-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia>
```

Dentro da sessão podemos inserir comandos que serão lidos, avaliados e impressos na tela. Um comando só é avaliado quando teclamos **Enter**. Vamos começar com operações com números inteiros. Para somar dois números podemos digitar:

```
1 + 2
```

```
3
```

Para multiplicar outros dois número:

```
40 * 4
```

```
160
```

Como esperado, podemos utilizar as operações básicas de soma (`+`), subtração (`-`) e multiplicação (`*`), e os resultados ocorrem como previsto. No entanto, observaremos a seguir que o comportamento da divisão apresenta algumas particularidades:

```
a = 84
b = 2

# As variáveis a e b são do tipo Int64

resultado = a / b
println(resultado)
```

```
42.0
```

Notem que, neste exemplo, ocorreu uma conversão de tipo, pois 84 e 2 são números inteiros, enquanto o resultado é um número em ponto flutuante (float). Os pontos flutuantes são representações binárias de números reais, tema que exploraremos com mais detalhes em breve. Esta conversão fica evidente pela representação do resultado como `42.0`, em vez de simplesmente `42`. Caso deseje obter o resultado como um número inteiro, é possível utilizar o operador `div`:

```
div(84,2)
```

```
42
```

Ou de forma equivalente usando o operador `\div` (para conseguir ver o símbolo da divisão é necessário digitar `\div` seguido da tecla `<tab>`).

Além das operações básicas, é possível fazer exponenciação:

```
2^31
```

```
2147483648
```

Expressões mais complexas também podem ser calculadas:

```
23 + 2 * 2 + 3 * 4
```

```
39
```

Sim, a precedência de operadores usual também é válida em Julia. Entretanto, lembre-se da primeira lição de programação: *Escreva para humanos, não para máquinas*. Podemos usar parênteses para separar as operações:

```
23 + (2 * 2) + (3 * 4)
```

```
39
```

Lembra dos pontos flutuantes? Todas as operações vistas podem ser aplicadas em pontos flutuantes:

```
23.5 * 3.14
```

```
73.79
```

Ou:

```
12.5 / 2.0
```

```
6.25
```

O exemplo acima demonstra mais um código escrito de forma clara para pessoas, onde ao utilizarmos `2.0` deixamos explícito que o segundo parâmetro é um número de ponto flutuante (float). É fundamental compreender que números de ponto flutuante possuem precisão **limitada**, portanto não se surpreenda ao encontrar resultados inesperados como os demonstrados abaixo:

```
1.2 - 1.0
```

```
0.19999999999999996
```

Erros como esse são bastante raros, tanto que normalmente depositamos total confiança nas contas realizadas por computadores e calculadoras. No entanto, é importante reconhecer que existem limitações (veja os exemplos abaixo).

```
2.6 - 0.7 - 1.9
```

```
2.220446049250313e-16
```

```
0.1 + 0.2
```

```
0.30000000000000004
```

```
10e15 + 1 - 10e15
```

```
0.0
```

Esses problemas de precisão estão ligados à limitação de como os números são representados no computador. De maneira simplificada, os valores no computador são codificados em palavras, formadas por bits. Nos computadores modernos, as palavras têm 64 bits, ou 8 bytes. Logo, uma outra limitação está relacionada aos números inteiros muito grandes.

```
2^63
```

```
-9223372036854775808
```

No entanto, para um curso introdutório, é suficiente estar ciente dessas limitações. O tratamento dessas questões faz parte de disciplinas mais avançadas. Vale ressaltar que o erro mencionado anteriormente é um *erro silencioso*, ou seja, ao trabalharmos com números inteiros, pode acontecer que o valor a ser representado exceda a capacidade do número de bits disponível, resultando em uma falha que ocorre sem notificação explícita.

Voltando às contas. Um outro operador interessante é o `%` que calcula o resto da divisão

```
4 % 3
```

```
1
```

Até agora vimos como trabalhar com um único valor, como se estivéssemos usando o visor de uma calculadora. Mas podemos ir além disso. Em vez de simples teclas de memória, o computador nos oferece **variáveis**. Essas são como nomes para valores que queremos armazenar e utilizar posteriormente.

Além das operações básicas também temos as operações matemáticas (funções), como por exemplo o seno, *sine* em inglês. Para saber como uma função funciona podemos pedir ajuda ao ambiente, usando uma `?` ou o macro (funções especiais) `@doc`, e em seguida digitando o que queremos saber, como por exemplo em:

```
@doc sin
```

A saída desse comando indica a operação que a função realiza e ainda apresenta alguns exemplos:

```
sin(x)

Compute sine of x, where x is in radians.

See also sind, sinpi, sincos, cis, asin.

Examples


julia> round.(sin.(range(0, 2pi, length=9)'), digits=3)
1×9 Matrix{Float64}:
0.0  0.707  1.0  0.707  0.0  -0.707  -1.0  -0.707  -0.0
```

Ambos os comandos `? sin` e `@doc sin` possuem a mesma saída.

Notem que nem tudo que foi apresentado faz sentido no momento, mas já dá para entender o uso de uma função como `sin`. Vejamos agora a raiz quadrada:

```
@doc sqrt
```

```
sqrt(x)

Return \sqrt{x}.

Throws DomainError for negative Real arguments. Use complex negative arguments instead. Note
along the negative real axis.

The prefix operator √ is equivalent to sqrt.

See also: hypot
...
```

Nela vemos que é possível calcular a raiz como em:

```
sqrt(4)
```

```
2.0
```

```
sqrt(4.0)
```

```
2.0
```

Agora, observe que a documentação da função `big()` tem a seguinte ajuda:

```
big(T::Type)

Compute the type that represents the numeric type T with arbitrary precision. Equivalent to t

Examples


julia> big(Rational)
Rational{BigInt}
```

```
julia> big(Float64)
BigFloat

julia> big(Complex{Int})
Complex{BigInt}

big(x)

Convert a number to a maximum precision representation (typically BigInt or BigFloat). See Bi
some pitfalls with floating-point numbers.
```

A função `big()` permite criar números de grande magnitude, representados pelos tipos `BigInt` ou `BigFloat`. Essa função é particularmente útil quando você precisa trabalhar com números muito grandes que ultrapassam os limites dos tipos padrão, como `Int64` ou `Int32`. Ao utilizar números do tipo `BigInt`, eliminamos problemas de estouro (overflow), conforme podemos observar abaixo:

```
big(2) ^ 1002
```

```
428603442874506928379370019624000724224561924682213442977500155348140420449974448997279351526
```

2.2 Variáveis e Tipos de Dados

Como já introduzido, em Julia, temos o conceito de variáveis. Variáveis servem para armazenar dados diversos, como inteiros e floats. Podemos operar nas variáveis da mesma forma que operamos nos dados que elas guardam (veja o exemplo abaixo).

```
a = 7
2 + a
```

```
9
```

Quando escrevemos `a = 7`, estamos realizando uma operação chamada **atribuição**. O operador `=` em Julia (e na maioria das linguagens de programação) não representa igualdade matemática, mas sim uma instrução para armazenar o valor à direita na variável à esquerda. Podemos visualizar isso como se estivéssemos colocando o valor `7` dentro de uma caixa chamada `a`.

É importante destacar que as variáveis em Julia podem receber novos valores, e o tipo da variável é determinado pela última atribuição realizada. A função `typeof` pode ser usada para identificar o tipo da variável especificada.

```
a = 3
typeof(a)
```

```
Int64
```

```
a = a + 1
typeof(a)
```

```
Int64
```

A atribuição sempre acontece da direita para a esquerda: primeiro calcula-se o valor da expressão à direita, e depois esse valor é armazenado na variável à esquerda. No exemplo a seguir, a variável `b` começa com um valor de tipo inteiro. No entanto, após a operação de multiplicação, seu valor passa a ser do tipo ponto flutuante.

```
b = 3
b = b * 0.5
typeof(b)
```

```
Float64
```

A capacidade de alterar o tipo da variável é conhecida como **tipagem dinâmica**. Esta característica apresenta diversas vantagens, como a flexibilidade de reutilizar variáveis para armazenar diferentes tipos de dados ao longo do tempo e a menor verbosidade, pois não é necessário especificar o tipo de cada variável, o que melhora a legibilidade do código. Neste contexto, podemos observar que Julia possui vários tipos primitivos, sendo os principais:

```
typeof(1)
```

```
Int64
```

```
typeof(1.1)
```

```
Float64
```

```
typeof("Bom dia")
```

```
String
```

Falando em **strings**, eles são definidos por conjuntos de caracteres entre aspas como:

```
s1 = "Olha que legal"
s2 = "Outra String"
```

```
"Outra String"
```

Também é possível realizar operações com strings, como **concatenação**:

```
s1 = "Tenha um"
s2 = " Bom dia"
s3 = s1 * s2
```

```
"Tenha um Bom dia"
```

Ou repetição usando o operador de potência:

```
s = "Não vou mais fazer coisas que possam desagradar os meus colegas "
s ^ 10
```

```
"Não vou mais fazer coisas que possam desagradar os meus colegas Não vou mais fazer coisas qu
```

Para evitar que se digitem muitos caracteres, por vezes podemos usar *açucares sintáticos*.

```
x = 1
x = x + 1
x += 1  # '+= 1' equivale a '= x + 1', também funciona para os operadores *, - e /
```

```
3
```

O código acima utiliza comentários (tudo depois do `#`). Esses comentários são ignorados pelo interpretador e podem ser usados para tornar o código mais legível.

Ainda sobre variáveis, há algumas regras referentes aos seus nomes: devem começar com uma letra (ou com `_`), podem conter dígitos e não podem ser palavras reservadas. Vale ressaltar que Julia, por ser uma linguagem moderna, aceita caracteres unicode e emojis nos nomes, como por exemplo o Δ (`\Delta`).

```
Δ = 2
```

```
2
```

```
 = 5 # \:cat: <tab>
 = 3 # \:dog: <tab>
 = 20 # \:house: <tab>
```

```
20
```

Isso não adiciona nada do lado de algoritmos, mas é possível ter variáveis bem bonitinhas. A lista de figuras pode ser encontrada aqui.

2.3 Saída de Dados

Para imprimir informações no terminal, usamos as funções `print()` e o `println()`. A diferença entre elas é que a primeira não pula linha, enquanto que a segunda pula.

```
print("Hello ")
println("World!")
println("Ola, mundo!")
```

```
Hello World!
Ola, mundo!
```

O comando `println()` pode receber múltiplos argumentos, que serão convertidos em strings e concatenados automaticamente:

```
nome = "Maria"
idade = 25
println("Olá, meu nome é ", nome, " e tenho ", idade, " anos.")
```

```
Olá, meu nome é Maria e tenho 25 anos.
```

Para formatações mais complexas, Julia oferece **interpolação de strings**, onde podemos inserir variáveis e expressões diretamente dentro de uma string usando o cifrão `$`:

```julia
nome = "João"
altura = 1.75
println("$nome tem $altura metros de altura.")
```

```
João tem 1.75 metros de altura.
```

Também podemos incluir expressões dentro de chaves após o cifrão:

```julia
preco = 9.99
quantidade = 3
println("Total da compra: R\$ $(preco * quantidade)")
```

```
Total da compra: R$ 29.97
```

Para formatação numérica, podemos usar a função `@sprintf` ou a macro `@printf` do módulo `Printf` (detalhes sobre módulos na Seção 2.4):

```julia
using Printf

valor = 123.456
@printf("Valor formatado: %.2f\n", valor)  # Exibe com 2 casas decimais
```

```
Valor formatado: 123.46
```

Ou alternativamente:

```julia
valor = 123.456
s = @sprintf("Valor formatado: %.2f", valor)
println(s)
```

```
Valor formatado: 123.46
```

2.4 Arquivos Externos e Módulos

No exemplo anterior usamos a sintaxe `using Printf`. Esta é a sintaxe para importar um módulo em Julia. Os módulos são coleções organizadas de código que podemos utilizar em nossos programas. O módulo `Printf` faz parte da **biblioteca padrão** de Julia e oferece funções para formatação de tipos no estilo da linguagem C. Ao escrever `using Printf`, informamos o interpretador que queremos acessar as funções deste módulo, como `@printf` e `@sprintf`. Para descobrir quais funções estão disponíveis neste e em outros módulos, consulte a documentação oficial de Julia. A documentação específica do módulo `Printf` está disponível em https://docs.julialang.org/en/v1/stdlib/Printf/.

Julia vem com vários módulos padrão que podem ser úteis:

- `Statistics`: para cálculos estatísticos (média, mediana, etc.)
- `Dates`: para trabalhar com datas e horas
- `Printf`: para formatação avançada de texto
- `LinearAlgebra`: para operações de álgebra linear
- `Random`: para geração de números aleatórios

A comunidade de Julia também desenvolve diversos módulos (pacotes) que podem ser instalados para expandir as funcionalidades da linguagem. Vamos aprender a fazer isso mais adiante no curso.

Além de módulos, Julia permite carregar código de arquivos externos usando o comando `include()`. Este comando lê o arquivo especificado e executa todo seu conteúdo no contexto atual. Como resultado, todas as funções, variáveis e definições do arquivo tornam-se disponíveis no ambiente onde `include` foi chamado.

No Windows, os caminhos de arquivo tradicionalmente usam barras invertidas (\). Porém, em Julia, podemos usar tanto barras normais (/) quanto barras invertidas. Há um detalhe importante: quando usamos barras invertidas dentro de strings em Julia, precisamos duplicá-las. Isso ocorre porque a barra invertida sozinha (\) é um caractere especial em strings, usado para representar caracteres como `\n` (nova linha) ou `\t` (tabulação). Para indicar que queremos uma barra invertida literal, precisamos escrever duas (\\). Por esse motivo, caminhos com várias pastas tornam-se mais difíceis de ler:

```
# Caminho usando barras normais (recomendado)
include("C:/Users/MeuUsuario/Documentos/arquivo.jl")

# Mesmo caminho usando barras invertidas (mais complicado)
include("C:\\Users\\MeuUsuario\\Documentos\\arquivo.jl")
```

Se o arquivo estiver no mesmo diretório que seu script ou REPL atual, basta usar o nome do arquivo:

```
include("funcoes.jl")
```

Para arquivos em subdiretórios do diretório atual:

```
include("utilitarios/matematica.jl")
```

Para arquivos no diretório pai:

```
include("../exemplos.jl")
```

Vamos ver um exemplo prático. Suponha que você tenha criado um arquivo chamado `funcoes.jl` na pasta `C:/Projetos/Julia/` com o seguinte conteúdo:

```
function ola(nome)
    println("Olá ", nome)
end

function soma(a, b)
    return a + b
end
```

Agora você pode usar essas funções (mais sobre funções no Capítulo 4) no REPL ou em outro arquivo:

```
# No REPL ou em um arquivo na mesma pasta:
include("funcoes.jl")

# Ou com caminho completo:
include("C:/Projetos/Julia/funcoes.jl")

# Agora podemos usar as funções definidas em funcoes.jl
ola("Alfredo")         # Imprime: Olá Alfredo
resultado = soma(5, 3) # resultado = 8
println(resultado)     # Imprime: 8
```

Esta funcionalidade é especialmente útil para organizar seu código em múltiplos arquivos, permitindo que você divida programas maiores em partes menores.

2.4.1 O que é um arquivo `.jl`?

Um arquivo `.jl` é semelhante a um arquivo de texto `.txt`, porém com a extensão `.jl`. Embora seja possível abri-lo com um editor de texto simples como o Bloco de Notas, não é recomendado utilizá-lo para programação. Os arquivos `.jl` são arquivos de código-fonte da linguagem Julia e são geralmente editados com editores específicos para programação, como Visual Studio Code, Atom ou Sublime Text.

Não existe um editor de texto definitivamente superior aos demais, o importante é escolher aquele com o qual você se sinta mais confortável. Nossa recomendação é o Visual Studio Code, que oferece recursos muito mais avançados que um editor de texto comum e possui uma extensão dedicada à linguagem Julia, facilitando significativamente a escrita de código. Para começar a usar o Visual Studio Code com Julia, os tutoriais a seguir podem ser úteis:

- https://code.visualstudio.com/docs/getstarted/getting-started
- https://code.visualstudio.com/docs/languages/julia

2.5 Verifique seu Aprendizado

1. Qual a diferença entre os resultados obtidos pelos operadores `/` e `div` em Julia? Em quais situações cada um seria mais apropriado?
2. Por que a expressão `2.6 - 0.7 - 1.9` não resulta exatamente em zero? O que isso nos ensina sobre cálculos computacionais?
3. Explique o que significa 'tipagem dinâmica' e como isso afeta o comportamento das variáveis quando atribuímos diferentes tipos de valores a elas.
4. Use a função `big()` para calcular 2^{1000}. Compare este resultado com o que acontece ao tentar calcular 2^{1000} sem usar `big()`.
5. Armazene seu nome e sobrenome em variáveis separadas e depois combine-as para formar seu nome completo com um espaço entre elas. Demonstre também a operação de repetição de strings.
6. Crie as variáveis `a = 10`, `b = 3` e `c = 4.5`. Realize os seguintes cálculos: `a + b + c`, `a * b * c`, `a % b` e verifique o tipo do resultado de cada operação usando `typeof()`.

2.6 Explore por Conta Própria

1. Procure na documentação duas funções matemáticas que não foram mencionadas no capítulo e teste seu uso no REPL.
2. O que acontece quando você tenta dividir um número por zero em Julia? E quando calcula `0/0`? Teste e observe os resultados.
3. Experimente o operador Unicode (digite `\approx` seguido de **TAB**). Como ele se comporta ao comparar `0.1 + 0.2  0.3`?

4. Investigue a função `round()` e utilize-a para corrigir alguns dos problemas de precisão demonstrados no capítulo.

3 Estruturas de Controle e Tomada de Decisões

O objetivo deste capítulo é entender como um programa pode tomar decisões e alterar seu fluxo de execução. Vamos explorar os operadores de comparação, o tipo booleano, e as estruturas condicionais `if`, `else` e `elseif` em Julia.

3.1 Operadores de Comparação e o Tipo Booleano

Antes de estudarmos estruturas condicionais, precisamos entender os operadores de comparação e o tipo de dado que eles produzem: o tipo booleano (`Bool`). Uma variável booleana pode ter apenas dois valores possíveis: `true` (verdadeiro) ou `false` (falso). Vamos examinar os principais operadores de comparação em Julia:

```
# Igualdade: retorna true se os valores forem iguais
2 + 2 == 4
```

```
true
```

```
# Diferença: retorna true se os valores forem diferentes
3 != 8
```

```
true
```

```
# Menor que: retorna true se o primeiro valor for menor que o segundo
23 < 24
```

```
true
```

```
# Menor ou igual: retorna true se o primeiro valor for menor ou igual ao segundo
42 <= 44
```

```
true
```

```
# Maior que: retorna true se o primeiro valor for maior que o segundo
42 > 2
```

```
true
```

```
# Maior ou igual: retorna true se o primeiro valor for maior ou igual ao segundo
42 >= 42
```

```
true
```

É importante observar que em linguagens de programação, incluindo Julia, o sinal de igual (`=`) é usado para atribuição de valores a variáveis, enquanto o operador de igualdade (`==`) é usado para comparações.

Podemos verificar o tipo de uma expressão de comparação:

```
typeof(2 == 3)
```

```
Bool
```

Como esperado, o tipo é `Bool`, indicando um valor booleano.

3.1.1 Operadores Lógicos

Além dos operadores de comparação, Julia também oferece operadores lógicos que permitem combinar ou modificar valores booleanos:

```
# Operador NOT (negação): inverte o valor booleano
!true
```

```
false
```

```
!false
```

```
true
```

```
# Operador AND: retorna true apenas se ambos os valores forem true
true && true
```

```
true
```

```
true && false
```

```
false
```

```
# Operador OR: retorna true se pelo menos um dos valores for true
true || false
```

```
true
```

```
false || false
```

```
false
```

Esses operadores são essenciais para construir condições mais complexas em nossas estruturas condicionais.

3.2 Alterando o Fluxo de Execução com if-else

Até agora, nossos programas seguiam um fluxo de execução linear, com as instruções sendo executadas na ordem em que foram escritas. Veja o exemplo:

```
println("Oi")
println("um")
println("dois")
```

```
Oi
um
dois
```

A ordem de impressão será "Oi", "um" e "dois", exatamente na sequência em que os comandos foram escritos.

No entanto, muitas vezes precisamos que nosso programa tome decisões e execute diferentes blocos de código dependendo de certas condições. É aqui que entra a estrutura condicional `if`.

3.2.1 A Estrutura if

A estrutura **if** permite executar um bloco de código apenas se uma condição for verdadeira:

```
pandemia = true
println("Vou sair de casa?")
if pandemia == true
   println("Só vou sair de casa se for essencial")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

Neste exemplo, a mensagem "Só vou sair de casa se for essencial" só será impressa se a variável `pandemia` for igual a `true`.

Aqui está outro exemplo:

```
denominador = 1
if denominador != 0
   println("Sei fazer a divisão se não for por zero")
   println("O resultado da divisão de 30 por ", denominador, " é igual a ", 30/denominador)
end
```

```
Sei fazer a divisão se não for por zero
O resultado da divisão de 30 por 1 é igual a 30.0
```

O código dentro do bloco `if` só será executado se o denominador for diferente de zero, evitando assim um erro de divisão por zero.

3.2.2 Adicionando Alternativas com else

Frequentemente, queremos executar um bloco de código se uma condição for verdadeira e outro bloco se a condição for falsa. Para isso, usamos a estrutura **if-else**:

```
pandemia = true
println("Vou sair de casa?")
if pandemia == true
   println("Só vou sair de casa se for essencial")
else
   println("Balada liberada!!")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

Se a variável `pandemia` for `true`, será impressa a mensagem "Só vou sair de casa se for essencial". Caso contrário, será impressa a mensagem "Balada liberada!!".

3.2.3 Múltiplas Condições com elseif

E se tivermos mais de duas situações possíveis? Nesse caso, podemos usar a estrutura **if-elseif-else**:

```
pandemia = true
tenhoqueestudar = true
println("Vou sair de casa?")
if pandemia == true
   println("Só vou sair de casa se for essencial")
elseif tenhoqueestudar == true
   println("Melhor ficar em casa")
else
   println("Balada liberada")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

Neste exemplo, temos três caminhos possíveis:

1. Se houver pandemia, sair apenas se for essencial
2. Se não houver pandemia mas eu tiver que estudar, ficar em casa
3. Se não houver pandemia e eu não tiver que estudar, ir para a balada

A estrutura **if-elseif-else** avalia as condições na ordem em que aparecem. Assim que uma condição verdadeira é encontrada, o bloco correspondente é executado e as demais condições são ignoradas.

3.3 Verifique seu Aprendizado

1. Qual é a diferença entre o operador `=` e o operador `==` em Julia? Por que essa distinção é importante?

2. Explique a diferença entre **if-else** e **if-elseif-else**. Em quais situações você usaria cada um?
3. Considere a seguinte expressão booleana: `(a > b) && !(c == d)`. Explique em palavras o que ela significa.

3.4 Explore por Conta Própria

1. Pesquise sobre a avaliação em curto-circuito dos operadores lógicos `&&` e `||` em Julia. Como esse comportamento pode ser útil em programação?
2. Em Julia, além dos valores `true` e `false`, quais outros valores são considerados "verdadeiros" ou "falsos" em um contexto booleano?
3. Investigue o operador ternário (`?:`) em Julia e como ele pode ser usado como uma alternativa mais concisa para certas estruturas **if-else**.
4. Explore como as estruturas condicionais podem ser combinadas com funções para criar código mais modular e reutilizável.

4 Introdução às Funções

O objetivo deste capítulo é compreender o conceito de funções em programação e como elas são implementadas em Julia. Vamos explorar como criar nossas próprias funções, como elas podem receber parâmetros e retornar valores, além de introduzir o conceito de recursão.

4.1 Funções como Abstrações Naturais

Na aula anterior, já utilizamos algumas funções predefinidas em Julia. Funções são blocos de código que realizam tarefas específicas e podem ser reutilizados sempre que necessário. Elas nos permitem abstrair operações complexas em comandos simples, tornando o código mais legível e modular.

Vamos relembrar algumas das funções que já utilizamos:

- `typeof()` - Recebe um valor como parâmetro e retorna o seu tipo.
- `div()` - Recebe dois números e retorna a divisão inteira do primeiro pelo segundo.
- `print()` e `println()` - Imprimem valores no console, sendo que o segundo adiciona uma quebra de linha após a impressão.

Até agora, vimos exemplos de **chamadas de funções** como `sin(0.5)` ou `sqrt(4)`. Quando escrevemos o nome da função seguido de parênteses contendo argumentos, estamos "chamando" ou "invocando" essa função.

Mas como essas funções são criadas? Em Julia, podemos declarar nossas próprias funções usando a palavra-chave `function`:

```
function dobro(x)
    return x * 2
end
```

```
dobro (generic function with 1 method)
```

Aqui, `dobro` é o nome da função, `x` é um parâmetro (um valor que a função recebe), e `return x * 2` especifica o que a função deve calcular e retornar quando chamada. Fornecer parâmetros a uma função é opcional.

Após declarar a função, podemos chamá-la várias vezes com diferentes argumentos:

```
resultado1 = dobro(5)    # Chama a função com o argumento 5
println(resultado1)      # Imprime 10

resultado2 = dobro(3.5)  # Chama a função com o argumento 3.5
println(resultado2)      # Imprime 7.0
```

```
10
7.0
```

É importante entender a diferença entre declaração e chamada de funções:

- **Declaração**: Define como a função deve se comportar (ocorre uma vez)
- **Chamada**: Executa a função com valores específicos (pode ocorrer múltiplas vezes)

Funções como `sin()`, `sqrt()` e `big()` já vêm declaradas em Julia, por isso podemos utilizá-las diretamente.

4.1.1 Funções Chamando Outras Funções

Uma função pode chamar outra função, permitindo a composição de operações mais complexas:

```
function imprime(a)
   println("Vou imprimir ", a)
end

function imprimeduasvezes(a)
   imprime(a)
   imprime(a)
end
```

```
imprimeduasvezes (generic function with 1 method)
```

Testando nossa nova função:

```
imprimeduasvezes(13)
```

```
Vou imprimir 13
Vou imprimir 13
```

4.1.2 Acessando a Documentação de Funções

Podemos pedir ajuda ao interpretador para entender melhor como essas funções funcionam. Para isso, usamos o ponto de interrogação `?` ou o macro `@doc` antes do nome da função:

```
# Exemplos de como acessar a documentação
@doc typeof
```

```
@doc div
```

```
@doc println
```

Ao consultar a documentação, descobrimos que algumas funções como `div()` podem ser utilizadas com uma sintaxe alternativa, como por exemplo `\div`. Esse tipo de notação é particularmente útil para operações matemáticas.

4.1.3 Funções de Conversão

Uma categoria importante de funções em Julia são as funções de conversão, que transformam valores de um tipo em outro. Vejamos alguns exemplos:

```
# Converte uma string para um número em ponto flutuante
parse(Float64, "32")
```

```
32.0
```

```
# Converte um número em ponto flutuante para um inteiro (removendo a parte decimal)
trunc(Int64, 2.25)
```

```
2
```

```
# Converte um inteiro para um número em ponto flutuante
float(2)
```

```
2.0
```

```
# Converte um número para uma string
string(3)
```

```
"3"
```

```
# Converte um número em ponto flutuante para uma string
string(3.57)
```

```
"3.57"
```

4.1.4 Funções Matemáticas

Julia possui uma grande biblioteca de funções matemáticas prontas para uso. Aqui estão algumas das mais comuns:

Função	Descrição
`sin(x)`	Calcula o seno de (x) em radianos
`cos(x)`	Calcula o cosseno de (x) em radianos
`tan(x)`	Calcula a tangente de (x) em radianos
`deg2rad(x)`	Converte (x) de graus em radianos
`rad2deg(x)`	Converte (x) de radianos em graus
`log(x)`	Calcula o logaritmo natural de (x)
`log(b, x)`	Calcula o logaritmo de (x) na base (b)
`log2(x)`	Calcula o logaritmo de (x) na base 2
`log10(x)`	Calcula o logaritmo de (x) na base 10
`exp(x)`	Calcula o expoente da base natural de (x)
`abs(x)`	Calcula o valor absoluto de (x)
`sqrt(x)`	Calcula a raiz quadrada de (x)
`cbrt(x)`	Calcula a raiz cúbica de (x)
`factorial(x)`	Calcula o fatorial de (x)

Uma boa prática para se familiarizar com essas funções é experimentá-las com diferentes valores e verificar os resultados. Para funções mais complexas, é possível que já existam implementações prontas em Julia. Uma dica útil é pesquisar na internet usando palavras-chave como "julia lang hiperbolic sin" para encontrar a função desejada. Em geral, pesquisar em inglês tende a produzir melhores resultados.

4.1.5 Sobrecarga de Funções

Em Julia, podemos ter funções com o mesmo nome, mas com diferentes números ou tipos de parâmetros. Isso é chamado de "sobrecarga de funções":

```
function recebe(a)
  println("Recebi um parâmetro: ", a)
end

function recebe(a, b)
  println("Recebi dois parâmetros: ", a, " e ", b)
end
```

```
recebe (generic function with 2 methods)
```

O interpretador decide qual versão da função chamar com base nos argumentos fornecidos:

```
recebe(1)
```

```
Recebi um parâmetro: 1
```

```
recebe(1, 2)
```

```
Recebi dois parâmetros: 1 e 2
```

Também podemos chamar funções usando variáveis e expressões como argumentos:

```
a = 10
recebe(a)
recebe(a, a + 1)
```

```
Recebi um parâmetro: 10
Recebi dois parâmetros: 10 e 11
```

4.2 Funções que Retornam Valores

Até agora, vimos funções que apenas imprimem mensagens, mas não devolvem nenhum valor. O tipo de retorno dessas funções é `Nothing`, indicando que elas não produzem um valor que possa ser atribuído a uma variável.

No entanto, frequentemente queremos que nossas funções calculem e retornem valores. Para isso, usamos a palavra-chave `return`:

```
function soma1(a)
  return a + 1
end
```

```
soma1 (generic function with 1 method)
```

Agora podemos usar essa função em expressões e atribuições:

```
resultado = soma1(5)
println("O resultado é: ", resultado)
```

```
O resultado é: 6
```

```
# Também podemos usar o resultado em outras expressões
println("Resultado multiplicado por 2: ", soma1(5) * 2)
```

```
Resultado multiplicado por 2: 12
```

Podemos criar funções para cálculos mais complexos:

```
function hipotenusa(a, b)
  hip = sqrt(a^2 + b^2)
  return hip
end
```

```
hipotenusa (generic function with 1 method)
```

Testando nossa função:

```
# Calculando a hipotenusa de um triângulo 3-4-5
hipotenusa(3, 4)
```

```
5.0
```

4.3 Introdução à Recursão

Agora vamos explorar um conceito fundamental em programação: a recursão. Uma função recursiva é aquela que chama a si mesma como parte de sua execução. Isso pode parecer estranho à primeira vista, mas é uma técnica poderosa para resolver certos tipos de problemas.

Vamos começar com um exemplo simples: calcular o fatorial de um número. O fatorial de n (representado por $n!$) é o produto de todos os inteiros positivos menores ou iguais a n. Por exemplo, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

O fatorial pode ser definido recursivamente como:

- Caso base: $0! = 1$
- Caso recursivo: $n! = n \times (n-1)!$

Vamos implementar isso em Julia:

```
function fatorial(n)
  if n == 0
    return 1  # Caso base
  else
    return n * fatorial(n - 1)  # Chamada recursiva
  end
end
```

```
fatorial (generic function with 1 method)
```

Testando nossa função:

```
fatorial(5)
```

```
120
```

Para entender como a recursão funciona, vamos acompanhar passo a passo o cálculo de `fatorial(3)`:

1. Chamamos `fatorial(3)`
    - Como 3 não é igual a 0, executamos `return 3 * fatorial(2)`
2. Agora precisamos calcular `fatorial(2)`
    - Como 2 não é igual a 0, executamos `return 2 * fatorial(1)`

3. Agora precisamos calcular `fatorial(1)`
    - Como 1 não é igual a 0, executamos `return 1 * fatorial(0)`
4. Agora precisamos calcular `fatorial(0)`
    - Como 0 é igual a 0, retornamos 1
5. Agora podemos completar o cálculo de `fatorial(1)` $= 1 \times 1 = 1$
6. Agora podemos completar o cálculo de `fatorial(2)` $= 2 \times 1 = 2$
7. Finalmente, completamos o cálculo de `fatorial(3)` $= 3 \times 2 = 6$

A recursão tem duas partes fundamentais:

1. Um **caso base** que encerra a recursão (no nosso exemplo, quando $n = 0$)
2. Um **caso recursivo** que aproxima o problema do caso base (no nosso exemplo, reduzindo n em 1)

É necessário que a recursão sempre alcance o caso base, caso contrário, a função continuará chamando a si mesma indefinidamente, causando um erro de estouro de pilha (stack overflow).

4.3.1 Mais Exemplos de Recursão

Vamos implementar uma função recursiva para contagem regressiva:

```
function contagem(n)
    if n < 0
        println("Fim!")
    else
        print(n, " ")
        contagem(n - 1)
    end
end
```

```
contagem (generic function with 1 method)
```

Testando nossa função:

```
contagem(5)
```

```
5 4 3 2 1 0 Fim!
```

Podemos também usar recursão para calcular a soma dos primeiros n números inteiros:

```
function soma(n)
  if n == 0
    return 0  # Caso base
  else
    return n + soma(n - 1)  # Caso recursivo
  end
end
```

```
soma (generic function with 1 method)
```

Testando nossa função:

```
soma(10)
```

```
55
```

Outro exemplo interessante é o cálculo da soma dos termos da série harmônica:

```
function somaharmonica(atual, n)
  # Caso base: quando chegamos ao último termo
  if atual > n
    return 0.0
  else
    # Caso recursivo: somamos o termo atual e chamamos a função para o próximo termo
    return 1.0 / atual + somaharmonica(atual + 1, n)
  end
end
```

```
somaharmonica (generic function with 1 method)
```

Vamos calcular a soma dos 10 primeiros termos da série harmônica:

```
somaharmonica(1, 10)
```

```
2.9289682539682538
```

4.4 Verifique seu Aprendizado

1. Qual é a diferença entre uma função que imprime um valor e uma função que retorna um valor? Por que esta distinção é importante?
2. Explique o conceito de recursão em suas próprias palavras. Quais são os componentes essenciais de uma função recursiva?
3. Crie uma função que receba um número inteiro positivo e retorne a soma de seus dígitos. Por exemplo, para o número 123, a função deve retornar $1 + 2 + 3 = 6$.
4. Implemente uma função que calcule o n-ésimo número da **sequência de Fibonacci** usando recursão. Lembre-se que

   - $Fib(0) = 0$
   - $Fib(1) = 1$
   - $Fib(n) = Fib(n-1) + Fib(n-2)$, se $n > 1$.

5. Crie uma função que receba dois números como parâmetros e retorne o **máximo divisor comum (MDC)** entre eles usando o algoritmo de Euclides recursivamente.

4.5 Explore por Conta Própria

1. Pesquise sobre o conceito de "pilha de chamadas" (*call stack*) e como ele se relaciona com a recursão. Quais são as limitações práticas da recursão devido à pilha de chamadas?
2. Pense em como seria possível otimizar a função recursiva de Fibonacci para evitar cálculos repetidos.

   - Dica: pesquise sobre "memoização".

3. Explore funções com um número variável de argumentos em Julia usando a sintaxe de "splats" (`...`).
4. Procure como você pode definir valores padrão para parâmetros de funções em Julia.

5 Mais Problemas Envolvendo Recursão

No capítulo anterior, introduzimos o conceito de recursão e vimos como funções podem chamar a si mesmas para resolver problemas. Neste capítulo, vamos explorar mais alguns problemas que podem ser resolvidos utilizando recursão, o que nos ajudará a desenvolver nossa capacidade de "pensar recursivamente".

5.1 Pensando Recursivamente

Resolver problemas usando recursão requer uma mudança na forma como enxergamos os problemas. Em vez de pensar em todos os passos necessários para chegar à solução, focamos em:

1. Como expressar o problema em termos de uma versão menor do mesmo problema
2. Quando parar a recursão (caso base)

Vamos explorar quatro problemas que ilustram bem o poder da recursão: o problema das escadas, a potenciação por quadrados, o cálculo da raiz quadrada usando o método de Heron e o coeficiente binomial.

5.2 Problema das Escadas

Imagine uma escada com n degraus. Você pode subir 1 ou 2 degraus por vez. De quantas maneiras diferentes você pode chegar ao topo da escada?

Por exemplo, se temos uma escada com 3 degraus, existem 3 maneiras de subir:

- Dar três passos de 1 degrau: (1, 1, 1)
- Dar um passo de 1 degrau seguido de um passo de 2 degraus: (1, 2)
- Dar um passo de 2 degraus seguido de um passo de 1 degrau: (2, 1)

Como podemos pensar nesse problema recursivamente? Para chegar ao degrau n, devemos ter vindo do degrau $n-1$ (dando um passo de 1 degrau) ou do degrau $n-2$ (dando um passo de 2 degraus). Portanto, o número total de maneiras de chegar ao degrau n é a soma do número de maneiras de chegar ao degrau $n-1$ e ao degrau $n-2$.

Nossos casos base são:

- Se $n = 0$ (nenhum degrau): há 1 maneira (não subir)
- Se $n = 1$ (um degrau): há 1 maneira (dar um passo de 1 degrau)

Vamos implementar essa solução:

```
function maneiras_subir_escada(n)
    # Casos base
    if n == 0 || n == 1
        return 1
    else
        # Caso recursivo: soma das maneiras de chegar a partir de n-1 e n-2
        return maneiras_subir_escada(n - 1) + maneiras_subir_escada(n - 2)
    end
end
```

```
maneiras_subir_escada (generic function with 1 method)
```

Vamos testar nosso código para diferentes números de degraus:

```
for i in 1:10
    println("Escada com $i degraus: $(maneiras_subir_escada(i)) maneiras diferentes")
end
```

```
Escada com 1 degraus: 1 maneiras diferentes
Escada com 2 degraus: 2 maneiras diferentes
Escada com 3 degraus: 3 maneiras diferentes
Escada com 4 degraus: 5 maneiras diferentes
Escada com 5 degraus: 8 maneiras diferentes
Escada com 6 degraus: 13 maneiras diferentes
Escada com 7 degraus: 21 maneiras diferentes
Escada com 8 degraus: 34 maneiras diferentes
Escada com 9 degraus: 55 maneiras diferentes
Escada com 10 degraus: 89 maneiras diferentes
```

Note que estamos usando uma estrutura diferente, `for`. Não se preocupe com os detalhes dessa estrutura no momento, mas saiba que estamos apenas repetindo a execução de um bloco de código.

Se você olhar atentamente para essa sequência de resultados, perceberá que ela corresponde à famosa sequência de Fibonacci: $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \dots)$. Isso não é coincidência. O problema da escada e a sequência de Fibonacci compartilham a mesma estrutura recursiva.

5.3 Potenciação por Quadrados

Quando queremos calcular potências como a^n, a abordagem mais simples seria multiplicar a por si mesmo n vezes. Por exemplo, para calcular 2^8, faríamos $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$, realizando 7 multiplicações. Mas existe uma abordagem muito mais eficiente usando recursão.

O método de potenciação por quadrados que vamos apresentar consegue calcular o mesmo valor realizando apenas cerca de $\log_2 n$ operações. Por exemplo, para calcular 2^8, precisaríamos apenas de 3 multiplicações. Para números grandes, essa diferença é ainda mais significativa - calcular 2^{1000} exigiria 999 multiplicações com o método simples, mas apenas cerca de 10 multiplicações com nosso método recursivo.

A ideia baseia-se nas seguintes propriedades matemáticas:

- Se n é negativo: $a^n = 1/a^{-n}$ (invertemos o resultado da potência positiva)
- Se n é par: $a^n = (a^{n/2})^2$ (calculamos a "metade" da potência e elevamos ao quadrado)
- Se n é ímpar: $a^n = a \times a^{n-1}$ (multiplicamos por uma potência par, que sabemos calcular)

Como isso se traduz para o pensamento recursivo?

1. Se queremos calcular a^n e n é par:
    - Primeiro calculamos $a^{n/2}$ (um problema menor)
    - Depois multiplicamos esse resultado por si mesmo
2. Se queremos calcular a^n e n é ímpar:
    - Primeiro calculamos a^{n-1} (que é par e sabemos resolver pelo caso anterior)
    - Depois multiplicamos esse resultado por a
3. Se queremos calcular a^n e n é negativo:
    - Calculamos a^{-n} (sabemos resolver pelos casos anteriores)
    - Depois calculamos $1/a^{-n}$

Nossos casos base (onde a recursão para) são:

- Se $n = 0$, então $a^n = 1$ (qualquer número elevado a 0 é 1)
- Se $n = 1$, então $a^n = a$ (qualquer número elevado a 1 é ele mesmo)

Vamos implementar essa solução:

```
function potenciacao(base, expoente)
    # Resolvemos o expoente negativo primeiro
    if expoente < 0
        return 1 ÷ potenciacao(base, -expoente)
    end
```

```
    if expoente == 0
        return 1
    elseif expoente == 1
        return base
    end

    # Se o expoente for par
    if expoente % 2 == 0
        temp = potenciacao(base, expoente ÷ 2)
        return temp * temp
    else
        # Se o expoente for ímpar
        return base * potenciacao(base, expoente - 1)
    end
end
```

```
potenciacao (generic function with 1 method)
```

Vamos testar nossa função:

```
println(potenciacao(2, 10))  # Deve retornar 1024
println(potenciacao(3, 5))   # Deve retornar 243
```

```
1024
243
```

Para entender melhor como essa abordagem economiza operações, vamos traçar a execução de `potenciacao(2, 10)`:

Passo 1: `potenciacao(2, 10)`

- `expoente = 10` é par
- Precisamos calcular `potenciacao(2, 5)^2`

Passo 2: `potenciacao(2, 5)`

- `expoente = 5` é ímpar
- Precisamos calcular `2 * potenciacao(2, 4)`

Passo 3: `potenciacao(2, 4)`

- `expoente = 4` é par

- Precisamos calcular `potenciacao(2, 2)^2`

Passo 4: `potenciacao(2, 2)`

- `expoente = 2` é par
- Precisamos calcular `potenciacao(2, 1)^2`

Passo 5: `potenciacao(2, 1)`

- `expoente = 1` é ímpar
- Precisamos calcular `2 * potenciacao(2, 0)`

Passo 6: `potenciacao(2, 0)`

- caso base, retorna `1`

Valores retornados:

- `potenciacao(2, 0)` retorna `1`
- `potenciacao(2, 1)` retorna `2 * 1 = 2`
- `potenciacao(2, 2)` retorna `2^2 = 4`
- `potenciacao(2, 4)` retorna `4^2 = 16`
- `potenciacao(2, 5)` retorna `2 * 16 = 32`
- `potenciacao(2, 10)` retorna `32^2 = 1024`

Para os entusiastas de computação, quando falamos de **complexidade** estamos nos referindo à eficiência de um algoritmo em termos do número de operações realizadas. Para representar a quantidade de operações, utilizamos a notação *Big-O* (O-Grande), simbolizada como $O(\cdot)$.

Neste contexto, a abordagem que utilizamos consegue reduzir a complexidade de $O(n)$ (onde o número de operações cresce **linearmente** com o tamanho da entrada) para $O(\log n)$ (onde o número de operações cresce **logaritmicamente**, tornando o algoritmo muito mais eficiente para entradas grandes).

5.4 Cálculo da Raiz Quadrada (Método de Heron)

O Método de Heron (também conhecido como método babilônico) é um algoritmo antigo para calcular aproximações de raízes quadradas. A ideia é começar com uma estimativa e melhorá-la progressivamente.

Vamos calcular uma aproximação para $\sqrt{S}$. Se $x_0 > 0$ é a nossa estimativa inicial, podemos melhorar nossa estimativa usando a seguinte fórmula iterativa:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right).$$

Considere ε o erro em nossa estimativa de $\sqrt{S}$. Então, $S = (x_0 + \varepsilon)^2$. Expandindo o binômio temos

$$S = (x_0 + \varepsilon)^2 = x_0^2 + 2x_0\varepsilon + \varepsilon^2.$$

Podemos resolver a equação acima para ε.

$$\varepsilon = \frac{S - x_0^2}{2x_0 + \varepsilon} \approx \frac{S - x_0^2}{2x_0}, \quad (\varepsilon \ll x_0).$$

Assim, podemos compensar o erro e atualizar nossa estimativa antiga como

$$x_0 + \varepsilon \approx x_0 + \frac{S - x_0^2}{2x_0} = \frac{S + x_0^2}{2x_0} = \frac{\frac{S}{x_0} + x_0}{2} \equiv x_1$$

Como o erro calculado não foi exato, esta não é a resposta final, mas se torna nossa nova estimativa para usar na próxima iteração. O processo de atualização é repetido até que a precisão desejada seja obtida.

Vamos implementar esse método recursivamente:

```
function raiz_quadrada(S, x  = S / 2,   = 0.0001)
    x   = (x  + S / x ) / 2

    # Verificamos se a diferença entre as estimativas é menor que a precisão desejada
    if abs(x   - x ) <
        return x
    else
        return raiz_quadrada(S, x  ,  )
    end
end
```

```
raiz_quadrada (generic function with 3 methods)
```

A função recebe três parâmetros:

- `S`: o número do qual queremos a raiz quadrada
- `x` : nossa estimativa atual (por padrão, começamos com metade do número)
- ` ` : quão próximas duas estimativas consecutivas devem estar para considerarmos que encontramos a resposta (por padrão, estamos considerando `0.0001`)

Vamos testar nossa função:

```
println(raiz_quadrada(25))  # Deve ser próximo de 5
println(raiz_quadrada(2))   # Deve ser próximo de 1.4142...
```

```
5.000000000016778
1.4142135623746899
```

Observe o que acontece com o valor da estimativa para o cálculo da raíz quadrada de 25:

1. Primeira chamada: `x = 25/2 = 12.5`
2. Segunda chamada: `x = (12.5 + 25/12.5)/2 = 7.25`
3. Terceira chamada: `x = (7.25 + 25/7.25)/2 = 5.35`
4. Quarta chamada: `x = (5.35 + 25/5.35)/2 = 5.01`
5. Quinta chamada: `x = (5.01 + 25/5.01)/2 = 5.0`

5.5 Coeficiente Binomial

O coeficiente binomial $\binom{n}{k}$ (lê-se "n escolhe k") representa o número de maneiras de escolher k elementos de um conjunto de n elementos, sem considerar a ordem. Por exemplo, $\binom{5}{2}$ é o número de maneiras de escolher 2 elementos de um conjunto de 5 elementos.

O coeficiente binomial possui a seguinte definição recursiva (para $n > k$):

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Esta fórmula pode ser deduzida dividindo o problema em dois casos complementares. Consideremos um elemento X específico dentre as n alternativas disponíveis. Podemos classificar todos os possíveis subconjuntos de k elementos em duas categorias:

1. Subconjuntos que incluem X: Neste caso, como X já está selecionado, precisamos escolher apenas $k-1$ elementos adicionais dentre os $n-1$ elementos restantes. Isto corresponde a $\binom{n-1}{k-1}$ possibilidades.
2. Subconjuntos que não incluem X: Neste caso, devemos selecionar todos os k elementos dentre os $n-1$ elementos restantes (excluindo X). Isto corresponde a $\binom{n-1}{k}$ possibilidades.

O número total de subconjuntos possíveis é a soma destes dois casos, o que justifica a fórmula recursiva apresentada.

Podemos determinar os casos base através das seguintes propriedades:

1. $\binom{n}{0} = 1$ para qualquer $n \geq 0$ (há apenas uma maneira de escolher 0 elementos)
2. $\binom{n}{n} = 1$ para qualquer $n \geq 0$ (há apenas uma maneira de escolher todos os elementos)

Vamos implementar essa solução:

```
function coeficiente_binomial(n, k)
    if k == 0 || k == n
        return 1
    else
        return coeficiente_binomial(n - 1, k - 1) + coeficiente_binomial(n - 1, k)
    end
end
```

```
coeficiente_binomial (generic function with 1 method)
```

Vamos testar nossa função:

```
println(coeficiente_binomial(5, 2))  # Deve retornar 10
println(coeficiente_binomial(10, 4)) # Deve retornar 210
```

```
10
210
```

Parte II

Parte II: Estruturas Fundamentais e Boas Práticas

6 Mais algoritmos e introdução aos testes

Nessa aula, vamos ver algoritmos um pouco mais elaborados. Mas, sabendo que vamos usar algo com um maior grau de sofisticação, que tal pensar em testes?

De uma forma geral, para verificar o funcionamento de um programa, podemos escrever testes que verificam o funcionamento em algumas situações específicas.

Dado que o primeiro problema que queremos resolver é um algoritmo que encontra o n-ésimo número de Fibonacci. Por que não começar com testes?

Uma forma de se fazr testes, e de forma manual, mas isso não é reprodutível. A melhor maneira de se fazer testes, é de forma automatizada, ou seja criar código que teste código. Isso pode parecer complicado, mas vamos ver abaixo que não é.

Em uma busca rápida, podemos ver que a sequência de Fibonacci é definida da seguinte forma, os dois primeiros elementos F_1 e F_2 valem 1, em seguida temos a fórmula $F_n = F_{n-1} + F_{n-2}$. Mas, antes de pensar em resolver o problema vamos pensar em como testar.

Já sabemos os primeiros valores, além disso, através de uma busca rápida, podemos descobrir alguns valores da sequência como $F_5 = 5$ e $F_{12} = 144$. Supondo que a função para o cálculo do n-ésimo número de Fibonacci chamará fibo(). Podemos escrever o seguinte trecho de código:

```
function testafibo_versao1()
    if fibo(1) == 1
        println("Deu certo para 1")
    end
    if fibo(2) == 1
        println("Deu certo para 2")
    end
    if fibo(5) == 5
        println("Deu certo para 5")
    end
    if fibo(12) == 144
        println("Deu certo para 12")
    end
    println("Final dos testes")
end
```

```
testafibo_versao1 (generic function with 1 method)
```

A função de testes acima verifica se a função fibo() devolve o resultado correto para três casos. Mas, ela tem um defeito, ela imprime mensagens demais, o que pode ser ruim. Considerando isso, vamos ver o primeiro fundamento importante com relação a testes automatizados.

Se o teste passou, ele deve indicar apenas que deu certo!

Levando em conta o que foi escrito acima, podemos mudar o nosso teste para:

```
function testafibo()
    if fibo(1) != 1
        println("Não deu certo para 1")
    end
    if fibo(2) != 1
        println("Não deu certo para 2")
    end
    if fibo(5) != 5
        println("Não eu certo para 5")
    end
    if fibo(12) != 144
        println("Não deu certo para 12")
    end
    println("Final dos testes")
end
```

```
testafibo (generic function with 1 method)
```

Agora de posse da nossa função de testes, podemos pensar em escrever a nossa função de Fibonacci. Vamos ao caso fácil de n for menor que 2, a resposta é 1. Como vemos abaixo:

```
function fibo(n)
    if n <= 2
        return 1
    else
        # ainda não sabemos o que colocar aqui...
    end
end
```

```
fibo (generic function with 1 method)
```

Mas, a resposta está na própria definição da função, ou seja: $F_n = F_{n-1} + F_{n-2}$. Se o n for maior do que 2, temos que fazer a soma dos valores de Fibonacci de $n-1$ e de $n-2$. Ou seja:

```
function fibo(n)
    if n <= 2
        return 1
    else
        return fibo(n - 1) + fibo(n - 2)
    end
end

fibo(10)
```

```
55
```

É interessante notar que apesar de ser um dos exemplos clássicos de uso de recursão, o algoritmo acima é extremamente ineficiente. A razão é simples, cada vez que é feita a chamada, toda os valores de Fibonacci são recalculados para os valores de n e $n-1$.

Como Julia é uma linguagem moderna podemos usar o conceito de Memoização, que evita calcular o que já foi calculado. O Memoize tem que ser instalado no Julia com os comandos `import Pkg` e `Pkg.add("Memoize")`.

```
using Memoize
@memoize function fibo(n)
    if n <= 2
        return 1
    else
        return fibo(n - 1) + fibo(n - 2)
    end
end

fibo(10)
```

```
55
```

As diferenças de tempo das duas versões podem ser verificada com o comando @time. Da seguinte forma:

```
@time fibo(10)
```

```
  0.000001 seconds
```

```
55
```

Esse tipo de comando, que começa com @ é conhecido como anotação, e tem o poder de mudar o comportamente de partes do código.

Vamos ao segundo algoritmo da aula, o MDC (Máximo Divisor Comum). A ideia é usar o algoritmo de Euclides.

Basicamente ele diz que o MDC de dois números a e b, é igual ao MDC de b e r, onde $r = a\%b$. Quando esse resto for zero, chegamos a solução, que é b.

Vamos começar com os testes para alguns valores bem conhecidos. Por sinal começar pelos testes antes de escrever o código é uma boa prática de programação conhecida por TDD (Test Driven Design).

```
function testaMDC()
    if MDC(3298, 2031) != 1
        println("deu erro, para 3298 e 2031")
    end
    if MDC(120, 36) != 12
        println("deu erro, para 120 e 36")
    end
    if MDC(36, 120) != 12
        println("deu erro, para 36 e 120")
    end
    println("Acabaram os testes")
end
```

```
testaMDC (generic function with 1 method)
```

Vamos pensar na função agora. Dessa vez, se o resto for 0, temos que devolver o segundo termo. Caso contrário temos que continuar com a regra

```
function MDC(a, b)
    r = a % b
    if r == 0
        return b
```

```
    else
        return MDC(b, r)
    end
end

testaMDC()
```

```
Acabaram os testes
```

Até agora usamos o modo interativo do Julia para fazer os nosso códigos. Mas, existe oura forma bem mais reutilizável, ou seja escrever o texto em arqivos. Isso é relativamente simples, basta usar um editor de texto (puro) da sua preferência, como o notepad, nano, juno, atom, vscode ou outro e salvar um arquivo com a extensão .jl.

Mas, para que algo seja executado é importante colocar uma chamada ao final. Veja abaixo um possível arquivo mdc.jl.

```
function testeMDC()
    if mdc(70, 5) != 5
        println("Não funcionou para 70 e 5")
    end
    if mdc(13, 7) != 1
        println("Não funcionou para 13 e 7")
    end
    if mdc(127, 15) != 1
        println("Não funcionou para 127 e 15")
    end
    if mdc(20, 15) != 5
        println("Não funcionou para 20 e 15")
    end
    if mdc(42, 3) != 3
        println("Não funcionou para 42 e 3")
    end
    if mdc(42, 8) != 2
        println("Não funcionou para 42 e 8")
    end
    println("Final dos testes")
end

function mdc(a, b)
    r = a % b
    if r == 0
```

```
        return b
    else
        mdc(b, r)
    end
end

testeMDC()
println("O mdc entre 1227 e 321 é ", mdc(1227, 321))
```

```
Final dos testes
O mdc entre 1227 e 321 é 3
```

7 Testes automatizados e um pouco mais de código

Vamos começar o capítulo vendo uma forma mais simples de se rodar testes. Nos testes que vimos até agora sempre havia o teste de uma condição booleana associado a uma mensagem de erro quando não funcionasse. Mas, observando que a mensagem de erro geralmente está ligada à condição, por vezes a condição pode ser auto-explicativa.

Logo, uma forma elegante de expressar as condições pode ser útil na escrita dos testes. Para isso, vamos usar o módulo de testes. Em linguagens modernas, várias das situações repetitivas que enfrentamos podem ser evitadas usando alguma técnica mais moderna.

```
using Test
@testset "Modelo de testes" begin
    @test 2 == 1 + 1
    @test true
    @test !false
end
```

```
Test Summary:    | Pass  Total  Time
Modelo de testes |    3      3  0.1s
```

```
Test.DefaultTestSet("Modelo de testes", Any[], 3, false, false, true, 1.74113105242797e9, 1.7
```

No trecho acima primeiro indicamos que queremos fazer testes. Em seguida usamos o *test* que espera uma condição ou valor booleano. Finalmente todos os testes são reunidos em um *testset*.

Claro que o teste dá infomações relevantes quando falha:

```
using Test
@test 2 + 2 != 4
```

```
Test Failed at REPL[2]:1
  Expression: 2 + 2 != 4
   Evaluated: 4 != 4
```

Agora sim, vamos pensar em problemas algoritmicos novos. Que tal fazer a soma dos dígitos de um número inteiro. Ou seja, pensar em um número dígito à dígito. Vamos aos testes primeiro:

```
using Test
@testset "Teste da Soma de Dígitos" begin
    @test somaDig(0) == 0
    @test somaDig(1) == 1
    @test somaDig(100) == 1
    @test somaDig(123) == 6
    @test somaDig(321) == 6
    @test somaDig(99) == 18
end
```

Vamos agora tentar pensar em como "descascar" um número, dado o número 123, uma forma seria pegar o resto por 10 (ou seja 3) e depois dividir por 10 (ou seja 12), e assim por diante. Ou seja.

```
function somaDig(n)
    if n <=0 return 0
    else
        return n % 10 + somaDig(n ÷ 10)
    end
end

println(somaDig(1234))
```

```
10
```

Vamos agora a um outro problema clássico, a verificação se um número é ou não é primo. Na prática para fazer isso, temos a definição, um número n é primo apenas se for divisível apenas por 1 e por ele mesmo. Ou seja, nenhum número entre 2 e $n-1$ pode ser divisor de um número primo.

A forma de se fazer isso é relativamente simples. Vamos pensar em uma função que tenta dividir um número recursivamente, se conseguir devolve falso, se não conseguir devolve verdadeiro.

Vamos aos código:

```
function divide(n, i)
    if n % i == 0
        return false
```

```
    elseif i == n - 1
        return true
    else
        return divide(n, i + 1)
    end
end
```

```
divide (generic function with 1 method)
```

Que pode ser chamada por:

```
function éPrimo(n)
    return divide(n, 2)
end
```

```
éPrimo (generic function with 1 method)
```

Mais um exemplo, o método de Newton para o cálculo de raiz quadrada. Para achar a raiz de x, a partir de um chute inicial (por exemplos $y = x/2$), chegamos a um novo chute que é a média de y e x/y.

Mas, sim, vamos começar com os testes. Como estamos usando números do tipo *double* é bom sempre ter uma tolerância, por isso vamos usar uma comparação aproximada. Também poderiamos ter usado a função *isapprox* da linguagem Julia.

```
using Test
function quaseIgual(a, b)
    if abs(a - b) <= 1e-10
        return true
    else
        return false
    end
end


@testset "Teste da raiz pelo método de Newton" begin
    @test quaseIgual(3.0, raiz(3.0 * 3.0))
    @test quaseIgual(33.7, raiz(33.7 * 33.7))
    @test quaseIgual(223.7, raiz(223.7 * 223.7))
    @test quaseIgual(0.7, raiz(0.7 * 0.7))
    @test quaseIgual(1.0, raiz(1.0 * 1.0))
end
```

Note que como estamos comparando números em ponto flutuante, não usamos a comparação exata.

A solução final é:

```
function newton(c, n)
    q = n / c
    if quaseIgual(q, c)
        return q
    else
        return newton( (c + q) / 2.0, n)
    end
end


function raiz(n)
    a =  newton(n / 2.0, n)
    println("a raiz de ", n, " é ", a)
    return a
end
```

```
raiz (generic function with 1 method)
```

7.1 Funções caóticas

Vamos brincar um pouco agora com funções caóticas :), isso é, funções, que conforme o comportamento de uma constante k, apresentam resultados que podem convergir ou não. Isso é, a cada passo, quero saber o valor do próximo ponto aplicando a função novamente, isso é:

$$x_1 = f(x_0), x_2 = f(x_1), \ldots, x_n = f(x_{n-1})$$

As funções caóticas desempenham um papel significativo em diversas áreas da matemática e da física, com aplicações que vão desde a modelagem de crescimento populacional até a previsão de padrões climáticos. Elas também são fundamentais na análise de circuitos elétricos não lineares, onde pequenas variações nas condições iniciais podem levar a resultados drasticamente diferentes.

Para o nosso teste, a função f é extremamente simples: $x_{i+1} = x_i * (1 - x_i) * k$.

Implemente a função e imprima os 30 primeiros resultados. Comece com um valor de x entre 0 e 1, como 0.2. Use constantes $k = 2.1, 2.5, 2.8$ e 3.1 o que ocorre com $k = 3.7$?

Entregue o código e um pequeno relatório sobre o que acontece.

8 Uma outra forma de se fazer laços

Até o momento vimos que o computador é muito bom para fazer contas e repetições. Fizemos isso até agora com funções recursivas. Mas, existe um outro comando para isso, o while. A motivação é que enquanto alguma condição for válida, o computador continua repetindo os comandos.

O formato básico é o seguinte:

```
while condição
  # execute obloco
end
```

Enquanto a condição continuar verdadeira, o computador vai seguir repetindo o bloco que pode ser formado por várias intruções. Logo, para que a repetição, ou laço, não seja repetido indefinidamente, é essencial que algo ligado a condição seja atualizado no corpo do while.

Vejamos o exemplo simples da contagem regressiva:

```
n = 5
while n > 0
  println(n)
  n = n - 1
end
println("Acabou")
```

```
5
4
3
2
1
Acabou
```

Mas, vamos ver abaixo um caso onde o uso de while deixa o código mais Claro que com a recursão (onde é ruim fazer uma com vários parãmetros). Veja a resolução da série de Taylor abaixo:

```
 function sinTaylor2(x)
   i = 1
   termo = x
   soma = 0.0
   while i <= 15
     soma = soma + termo
     termo = -1 * termo * x * x / ((2 * i) * (2 * i + 1))
     i = i + 1
   end
   return soma
end
```

```
sinTaylor2 (generic function with 1 method)
```

Nela são calculados os 15 primeiros termos.

Observem a versão recursiva:

```
function sinTaylor(x)
    return sinTaylorRec(1, 15, x, 1, x)
end

function sinTaylorRec(i, n, x, sinal, termo)
    if n == i
      return 0.0
    else
      return sinal * termo +
         sinTaylorRec(i + 1, n, x, -1 * sinal, termo * x * x/ (2*i * (2*i+1)))
    end
end
```

```
sinTaylorRec (generic function with 1 method)
```

Podemos também fazer operações com os dígitos de um número inteiro, para isso operações como o resto da divisão por 10 e a divisão inteira por 10 são bastante úteis. Abaixo temos as duas versões que fazem a soma dos dígitos de um número inteiro.

```
using Test
function testaSD()
  @test sd(123) == 6
  @test sd(321) == 6
```

```
  @test sd(0) == 0
  @test sd(1001) == 2
  @test sd(3279) == 21
  println("Fim dos testes")
end

function sd(x)
  if x == 0
     return 0
  else
     d = x % 10
     return d + sd(div(x, 10))
  end
end

function sd1(x)
  soma = 0
  while x != 0
   d = x % 10
   soma = soma + d
   x = div(x, 10)
  end
  return soma
end
testaSD()
```

```
Fim dos testes
```

9 Aula de exercícios

9.1 Revisitando o cálculo do fatorial, recursivo e interativo

Agora que aprendemos a fazer também repetições com o comando while, sempre é bom pensar em qual o comando mais adequado. Vejamos o exemplo abaixo com duas versões da função para o cálculo do Fatorial.

```
function fatorial_recursivo(n::Int64) # Com o ::Int64 estamos definindo que o parâmetro da fu
    # Caso base do fatorial: 0! e 1! são iguais a 1
    if n == 0 || n == 1
        return 1
    # Chamada recursiva: n! = n * (n-1)!
    else
        return n * fatorial_recursivo(n - 1)
    end
end

function fatorial_iterativo(n::Int64)
    # Inicializa o resultado como 1 (já que o fatorial de 0 é 1)
    resultado = 1

    # No loop estamos fazendo a multiplicação: n * (n-1) * ... * 2
    while n > 1
        # Multiplica o resultado pelo valor atual de n
        resultado *= n

        # Decrementa n em 1 para continuar o cálculo do fatorial
        n -= 1
    end
    return resultado
end

println(fatorial_recursivo(3))
```

```
6
```

No código acima temos uma novidade, nos parâmetros da função, o tipo está sendo declarado expicitamente. No caso, estamos dizendo que o valor n que a função vai receber é de um tipo específico. Ou seja um Inteiro de 64 bits.

O estilo de código está um pouco diferente do que antes, pois foi escrito por outra pessoa. A monitora. Vemos que ela tem o hábito de usar nomes de variáveis maiores além do que usar contrações como += e *=.

9.2 Aproximação da raiz quadrada

Para o próximo exemplo, vamos ver o método de Newthon-Raphson para o cálculo da raiz quadrada. É um método recursivo no qual o próximo valor é baseado no valor anterior. Quanto mais chamadas forem feitas, mais próximo do valor final vai se chegar.

Mais informações sobre o método podem ser encontradas em aqui. Mas para o momento temos que pensar na seguinte implementação. Para se calcular a raiz, podemos usar a seguinte fórmula, a partir de um palpite inicial r, para o valor da raiz de x.

$$r_{n+1} = 0.5 * (r + x/r)$$

Como o código abaixo é mais complicado, foram usados comentários.

```
function aproxima_raiz(x::Float64, epsilon::Float64)::Float64
    if x < 0
        return nothing
    end

    # Chute inicial
    aproximacao = x/2
    melhor_aproximicao = aproximacao

    while true
        # Fórmula para aproximação de raiz quadrada utilizando o método de Newthon-Raphson
        melhor_aproximicao = 0.5 * (aproximacao + x/aproximacao)

        # Se a distância absoluta entre os dois pontos é menor do que epsilon, então podemos
        if abs(aproximacao - melhor_aproximicao) <= epsilon
            break
        end

        # Se a aproximação ainda não for boa o sufuciente, então atualizamos a aproximação pa
        aproximacao = melhor_aproximicao
```

```
    end

    return melhor_aproximicao

end
```

```
aproxima_raiz (generic function with 1 method)
```

Notem que foi introduzido um comando novo, o break, esse comando apenas interrompe a execução do while. Ou seja, força a saída do laço.

9.3 Verificar se um número é primo

No próximo exemplo, vamos verificar se um número é primo, ou seja, se os seus únicos divisores são 1 e o próprio. A forma mais simples de se fazer isso é procurando dividir o número por outros. Se algum dividir, o número não é primo.

```
function verifica_primo(num :: Int64)
    if num <= 1
        return false
    end
    i=2
    # pode ser melhorado com i<=num/2
    # ou também com i<= sqrt(num): baseado no fato que um número composto deve ter um fator r
    while i<num
        if num % i == 0
            return false
        end
        i+=1
    end
    return true
end
```

```
verifica_primo (generic function with 1 method)
```

Assim, como o comando break é usado para interromper a execução de um laço, o comando return, pode ser usado para terminar a execução de uma função, a qualquer momento.

9.4 Verificar se um número é palíndromo

Um número palíndromo é um número que é simétrico. Ou seja, a leitura dos dígitos da esquerda para a direita é igual a leitura dos dígitos na ordem inversa. Por exemplo, o número 121 é palíndromo, assim como o 11 e o 25677652. Os números de um dígito também são.

```
function e_palindromo(n::Int64)
    #=
        Guarda os dígitos de n que ainda devem ser invertidos
        A variável auxiliar é necessária para que o valor de n não seja, perdido, e possamos
    =#
    aux = n
    # Guarda a inversão do número n
    n_inv = 0

    #=
        Continuamos o while enquanto ainda há números a serem invertidos,
        ou seja, enquanto aux for maior que 0.
    =#
    while aux > 0
        # Coloca o último dígito de aux na variável que guarda a inversão
        resto = aux % 10
        n_inv= n_inv * 10 + resto

        # Retira o último dígito de aux
        aux = div(aux,10)
    end

    if n == n_inv
        println("O número $n é palíndromo")
    else
        println("O número $n não é palíndromo")
    end
end

e_palindromo(2002)
e_palindromo(1234)
```

```
O número 2002 é palíndromo
O número 1234 não é palíndromo
```

10 Revisitando a aula passada

Além de discutirmos o que vimos na aula passada. Nessa aula, vimos uma nova solução para o problema de verificar de um número é palíndromo.

Para isso usamos uma técnica um pouco diferente, ou seja, ao invés de inverter o número e compará-lo com o original. Verificamos se os seus extremos são iguais.

Observe o número 234432, o primeiro passo seria verificar que nos extremos, mais significativo e menos significativo, temos os números 2. Em seguida, podemos continuar com a verificação para o número 3443. Se em algum momento a verificação falhar o número não é palíndromo.

Seguem os testes e o código abaixo.

```
using Test

function testaPal()
  @test testaPal(1)
  @test testaPal(131)
  @test testaPal(22)
  @test testaPal(53877835)
  @test !testaPal(123)
  @test !testaPal(23452)
  println("Final dos testes")
end

function testaPal(n::Int64)
# o primeiro passo é encontrar um número com o mesmo número de dígitos de n
  pot10 = 1
  while pot10 < n
    pot10 = pot10 * 10
  end
  pot10 = div(pot10, 10)


  while n > 9
    d1 = n % 10
    d2 = div(n, pot10)
```

```
    if d1 != d2
      return false
    end
    n = div(n % pot10, 10)
    pot10 = div(pot10, 100)
  end
  return true
end
```

```
testaPal (generic function with 2 methods)
```

10.1 Aleatoreidade

Em julia temos a função rand() que devolve um número em ponto flutuante entre 0 e 1. Conforme os parâmetros, podemos ter outros tipos de número como:

```
rand(Int)  # devolve um inteiro
rand(1:10) # devolve um número entre 1 e 10
rand(Bool) # devolve verdadeiro ou falso
```

```
true
```

Mas, antes de ver um código com rand(). Vamos pensar em um problema da vida real. Imagine que temos que fazer um sorteio justo, e o único instrumento que possuímos para o sorteio é uma moeda viciada. Que tem como resultado muito mais faces do que coroas. Dá para usar essa moeda em um sorteio justo?

A ideia para resolver o problema é olhar para pares de sorteios. Ou seja, vamos ignorar sorteios onde tenhamos duas faces ou duas coroas. Nos outros, teremos uma coroa e uma face ou vice versa. As chances das duas serão de 50%. Logo podemos assim, corrigir a moeda viciada.

Para simplificar o exercício, a moeda pode devolver 0, ou 1, correspondentes a cara ou a coroa. Observe a seguinte função que simula uma moeda viciada.

```
function sorteio()
  if rand() > 0.90
    return 1
  else
    return 0
  end
end
```

```
sorteio (generic function with 1 method)
```

Pode se observar que a função devolve 0 na maior parte das vezes. Podemos inclusive ver isso, fazendo mil sorteios:

```
function verificaSorteio()
   cara = 0
   coroa = 0
   i = 0
   while i < 1000
     if sorteio() == 0
        cara = cara + 1
     else
        coroa = coroa + 1
     end
     i = i + 1
   end
   println("O número de caras foi: ", cara," e de coroas foi :", coroa)
end
```

```
verificaSorteio (generic function with 1 method)
```

Mas, podemos corrigir o sorteio da seguinte forma:

```
function sorteioBom()
   sorteio1 = sorteio()
   sorteio2 = sorteio()
   while sorteio1 == sorteio2 # se forem iguais, tente novamente
     sorteio1 = sorteio()
     sorteio2 = sorteio()
   end
   return sorteio1   # ao termos um diferente, podemos devolver o primeiro sorteio
end
```

```
sorteioBom (generic function with 1 method)
```

Podemos usar o verificaSorteio para ver a diferença.

```
function verificaSorteio()
   cara = 0
   coroa = 0
   i = 0
   while i < 1000
     if sorteioBom() == 0
        cara = cara + 1
     else
        coroa = coroa + 1
     end
     i = i + 1
   end
   println("O número de caras foi: ", cara," e de coroas foi :", coroa)
end
```

```
verificaSorteio (generic function with 1 method)
```

Podemos ainda aproximar o número de Euler ( ), constante matemática que é a base dos logaritmos naturais, usando uma simulação probabilística. A ideia por trás desse código é que o número médio de tentativas necessárias para que a soma de números aleatórios entre 0 e 1 ultrapasse 1 se aproxima do valor de . Isso é baseado em uma relação matemática que conecta essa situação ao número .

```
function calculaEuler(total)
    soma_tentativas = 0
    for i in 1:total
        soma = 0.0
        tentativas = 0
        while soma <= 1   # Continue gerando números até a soma ultrapassar 1
            soma += rand()     # Gera número aleatório entre 0 e 1
            tentativas += 1
        end
        soma_tentativas += tentativas     # Somar o número de tentativas necessárias
    end
    return soma_tentativas / total     # A média do número de tentativas será uma estimativa
end

println("Estimativa de e (1000 iterações): ", calculaEuler(1000))
println("Estimativa de e (100000 iterações): ", calculaEuler(100000))
println("Estimativa de e (100000000 iterações): ", calculaEuler(100000000))
```

```
Estimativa de e (1000 iterações): 2.733
Estimativa de e (100000 iterações): 2.71734
Estimativa de e (100000000 iterações): 2.71835781
```

Para terminar a aula vamos aplicar o método de Monte Carlo para o cálculo de Pi. Imaginem o primeiro quadrante, onde temos um semi-círculo de raio 1, dentro de um quadrado de lado 1. Podemos sortear valores, os que sairem dentro do círculo podem contar para a área desse. Mais informações podem ser vistas aqui (https://pt.wikipedia.org/wiki/M%C3%A9todo_de_Monte_Carlo)

```
function calculaPi(total)
   noAlvo = 0
   i = 0
   while i < total
     x = rand() / 2.0 # gera um número entre 0 e 0.5
     y = rand() / 2.0
     if sqrt(x * x + y * y) <= 0.5
       noAlvo = noAlvo + 1
     end
     i = i + 1
   end
   return 4 * (noAlvo / total)  # precisamos multiplicar para ter a área de 4 quadrantes
end

println(calculaPi(100))
println(calculaPi(1000000))
println(calculaPi(1000000000))
```

```
3.04
3.141788
3.141560052
```

11 Entrada de dados e o começo de listas

Nessa aula, temos dois tópicos principais, como fazer a entrada de dados, através de comandos de entrada e com argumentos na linha de comando. Além disso também veremos como tratar de um tipo especial de variável, onde é possível, guardar mais de um valor.

11.1 O comando input

Quando queremos inserir dados, em Julia, basta colocar dados. Mas, como podemos fazer para entrar dados em um programa comum?

Para isso temos o comando readline(), que interrompe a execução do programa e espera pela entrada de uma String, o que ocorre quando a tecla "enter" é pressionada.

```
println("Digite o seu nome")
resposta = readline()
println("O seu nome é: ", resposta)
```

Caso, ao rodar o programa, você digitar `Maria`, e pressionar a tecla enter, a resposta final do seu programa será `O seu nome é: Maria`.

Como o readline() lê Strings, se quisermos ler números, é necessário usar o comando parse. O comando parse de forma simples possui dois parâmetros, o primeiro corresponde ao tipo que se quer transformar, e o segundo o valor original.

```
println("Digite um inteiro")
valor = parse(Int64, readline())
println("O numero digitado foi ", valor)
```

Sabendo ler números do teclado, vamos a um exercício simples, ler uma sequência de números inteiros terminada por zero e devolver a sua soma.

```
function somaVarios()
    soma = 0.0
    println("Digite um número")
    n = parse(Float64, readline())
    while  n!=0
        soma = soma + n
        println("Digite um número")
        n = parse(Float64, readline())
    end
    println("A soma é: ", soma)
end
```

Observe o seguinte exemplo que calcula os quadrados dos números de uma lista terminada por zero.

```
function leQ()
  x = readline()
  n = parse(Float64, x)
  while n != 0
    println("$n ao quadrado é ", n * n)
    x = readline()
    n = parse(Float64, x)
  end
end
```

Notem que o readline também pode receber uma variável de arquivo para que dados sejam lidos diretamente. Mas, nesse caso temos que tomar Ocuidado para abrir (open()) e fechar (close()) o arquivo. Como abaixo:

```
function leQ()
    println("Digite um número")
    f = open("numeros.txt", "r+")
    x = readline(f)
    n = parse(Float64, x)
    while n != 0
        println("$n ao quadrado é ", n * n)
        println("Digite outro número")
        x = readline(f)
        n = parse(Float64, x)
    end
    close(f)
end
```

11.2 Lendo através da linha de comando

A outra forma de ler comandos é através da constante ARGS que é preparada na chamada de um programa. Para entender melhor isso, vamos ver o seguinte programa.

```
println(ARGS)
```

Se a linha acima está no arquivo args.jl, ao chamar julia args.jl com diversos parâmetros, teremos diversos resultados diferentes.

Por exemplo ao chamar:

julia args.jl 1 2 3 abc

Teremos como resposta

```
["1", "2", "3", "abc"]
```

Vamos analisar um pouco melhor essa resposta observando que cada parâmetro está em uma posição.

```
tam = length(ARGS)
println("O tamanho dos argumentos é: ", tam)
for i in 1:tam
    println(ARGS[i])
end
```

Olhando o código acima, podemos ver que a função length() devolve o número de argumentos, ou seja, o tamanho da lista ARGS. Além disso com os colchetes é possível acessar a cada posição da lista de forma individual.

O exemplo abaixo soma os parâmetros inteiros dados como argumentos. Ele também ilustra uma boa prática que é, sempre colocar o código em módulos, no caso abaixo em funções:

```
function SomaEntrada()
    tam = length(ARGS)
    s = 0
    i = 1
    while i <= tam
        valor = parse(Int, ARGS[i])
        println(valor)
        s = s + valor
        i = i + 1
```

```
    end
    println("A soma foi: ", s)
end
SomaEntrada()
```

A flexibilidade que temos ao usar listas é enorme! Por isso, listas ou vetores, merecem um tópico próprio.

11.3 Listas

Vamos primeiro brincar um pouco no console.

```
vetor = [1, 2, 3]
println(vetor[1])
println(length(vetor))
vetor[2] = vetor[2] + 1
vetor[1] = 2 * vetor[3]
println(vetor)
```

```
1
3
[6, 3, 3]
```

Como disse antes, o for foi feito para manipular vetores, vamos ver umas funções, a primeira que imprime os elementos de um vetor um por linha.

```
function imprimeVetor(v)
    for el in v
        println(el)
    end
end
```

Isso também pode ser feito por meio dos índices do vetor:

```
function imprimeVetor(v)
    for i in 1:lenght(v)
        println(v[i])
    end
end
```

Como cada posição é independente, podemos calcular a soma dos elementos ímpares de um vetor

```
function somaImpVetor(v)
    soma = 0
    for i in 1:length(v)
        if v[i] % 2 == 1
            soma = soma + v[i]
        end
    end
    return soma
end
```

Também vimos em aula alguns outros exemplos, como calcular a média dos elementos em um vetor.

```
function mediaV(v)
   soma = 0.0
   for i in v
      soma = soma + i
   end
   return soma / length(v)
end
```

Devolver a soma dos elementos ímpares de um vetor

```
function somaImpar(v)
    soma = 0
    for i in v
        if i % 2 == 1
            soma = soma + i
        end
    end
    return soma
end
```

Imprimir os números divisíveis por 5 de um vetor.

```
function imprimeDivisivelPor5(v)
    for i in v
        if i % 5 == 0
            println(i)
```

```
        end
    end
end
```

Com uma pequena variação e usando o comando push!() podemos ver como devolver um vetor com os números divisíveis por 5.

```
function devolveDivisivelPor5(v)
    x = []  # começa com um vetor vazio
    for i in v
        if i % 5 == 0
            push!(x, i)  # adiciona um elemento ao vetor x
        end
    end
    return x
end
```

11.3.1 Álgebra linear e Listas

A manipulação de listas é uma parte fundamental da álgebra linear, que estuda vetores e matrizes. Funções como o produto escalar de dois vetores são exemplos clássicos. Abaixo temos dois exemplos de produto escalar de dois vetores. lembrado esse é definido como a soma dos produtos de elementos em posições iguais.

```
function dotProduct(a, b)
    soma = 0
    if length(a) != length(b)
       return soma   # o produto não está definido se os tamanhos são diferentes
    end
    for i in 1:length(a)
        soma = soma + a[i] * b[i]
    end
    return soma
end
```

Acima vimos que um caso especial do uso do for, consiste em fazer Ofor varias entre 1 e um tamanho (1:lenght(a))

Observem a diferença na versão abaixo:

```
function dotProduct(a, b)
    soma = 0
    if length(a) != length(b)
       return soma   # o produto não está definido se os tamanhos são diferentes
    end
    i = 1
    for x in a
        soma = soma + x * b[i]
        i = i + 1
    end
    return soma
end
```

11.3.2 Exercício de permutação

Para terminar, vamos fazer uma função onde dado um vetor de inteiros de tamanho n, verifica se esse vetor é uma permutação dos números de 1 a n. Para isso, veremos se cada número de 1 a n está no vetor.

Mas, sem esquecer dos testes:

```
@testset "Verifica Permutação" begin
    @test permuta([1,2,3])
    @test permuta([3, 2, 1])
    @test permuta([1])
    @test permuta([2, 1])
    @test permuta([4, 2, 3, 1])
    @test !permuta([1, 1])
    @test !permuta([1, 3])
    @test permuta([])
end
```

e o código:

```
function permuta(v)
   tam = length(v)
   for i in 1:tam
      if  !(i in v)
         return false
      end
   end
```

```
    return true
end
```

Foi usado o comando in de Julia que verifica se um elemento está no vetor.

12 Exercícios com vetores

Os vetores permitem que sejam realizados algoritmos bem mais complexos, nesse capítulo veremos algums exercícios.

12.1 Permutação

Dado um vetor com inteiros, queremos verificar se esse vetor contém uma permutação. Para isso, temos que verificar em um vetor de tamanho n, se ele contém os números de 1 a n exatamente uma vez cada 1. O vetor [3, 1, 2] é uma permutação, pois tem tamanho 3 e os elementos de 1 a 3 aparecem uma vez.

Uma forma de se resolver esse problema é por meio de um indicador de passagem. Inicialmente vamos supor que o vetor é uma permutação, em seguida verificamos se todos os números entre 1 e n estão no vetor. Isso pode ser feito com comando in, que verifica se um elemento pertence ao vetor.

```
function permutação(l)
    perm = true
    tamanho = length(l)
    i = 1
    while i <= tamanho
        if !(i in l)
            perm = false
        end
        i += 1
    end
    return perm
end
```

```
permutação (generic function with 1 method)
```

Uma outra alternativa é verificar se para cada elemento do vetor, se ele está entre 1 e n, e é unico. Ou seja, verificamos se o primeiro elemento está entre 1 e n, e depois percorremos o vetor para ver se ele é único. Em seguida fazemos isso para os elementos seguintes. O código fica:

```
function permutação(l)
    perm = true
    tamanho = length(l)
    i = 1
    while i <= tamanho
        if (l[i] > tamanho || l[i] <= 0)
            perm = false
        end
        j = i +1
        while j <= tamanho
            if l[j] == l[i]
                perm = false
            end
            j += 1
        end
        i += 1
    end
    return perm
end
```

```
permutação (generic function with 1 method)
```

Uma outra alternativa é ter um vetor auxiliar onde contamos as ocorrências de cada número entre 1 e n. Ao final, todos os elementos desse vetor auxiliar tem que valer 1. Dessa vez, aproveitamos e já colocamos os testes automatizados.

```
using Test
function permutação(l)
    perm = true
    tamanho = length(l)
    aux = zeros(Int8, tamanho)
    for i in l
      if i < 1 || i > tamanho
        perm = false
      else
        aux[i] += 1
      end
    end
    for i in aux
      if i != 1
        perm = false
```

```
      end
    end
    return perm
end

@testset "Verifica Permutação" begin
    @test permutação([1,2,3])
    @test permutação([3, 2, 1])
    @test permutação([1])
    @test permutação([2, 1])
    @test permutação([4, 2, 3, 1])
    @test !permutação([1, 1])
    @test !permutação([1, 3])
    @test !permutação([4, 2, 3, -1])
    @test !permutação([5, 2, 3, 1])
    @test permutação([])
    @test !permutação([0, 3, 3])
    @test !permutação([2, 2, 2])
end
```

```
Test Summary:       | Pass  Total  Time
Verifica Permutação |   12     12  0.1s
```

```
Test.DefaultTestSet("Verifica Permutação", Any[], 12, false, false, true, 1.74113108948372e9
```

12.2 Histograma

Já que vimos o exemplo anterior onde "contamos" o número, podemos ir um pouco além e calcular o histograma de um vetor com números entre 1 e 10.

```
using Test

function histograma(l)
    result = [0,0,0,0,0,0,0,0,0,0]
    i = 1
    while i <= length(l)
        valor_atual = l[i]
        if valor_atual >= 1 && valor_atual <= 10
           result[valor_atual] += 1
        end
```

```
        i += 1
    end
    return result
end

@testset "Verifica Histograma" begin
    @test [1,0,0,0,0,0,0,0,0,0] == histograma([1])
    @test [0,0,0,0,0,0,0,0,0,0] == histograma([-1])
    @test [0,0,1,0,0,0,0,0,0,0] == histograma([3])
    @test [0,0,0,0,0,0,0,0,0,1] == histograma([10])
    @test [0,0,0,0,0,0,0,0,0,0] == histograma([11])
    @test [1,4,0,2,5,1,0,1,0,0] == histograma([5,6,5,4,5,5,4,2,8,2,1,2,5,2])
    @test [0,0,0,0,0,0,0,0,0,0] == histograma([])
    end
```

```
Test Summary:       | Pass  Total  Time
Verifica Histograma |    7      7  0.0s
```

```
Test.DefaultTestSet("Verifica Histograma", Any[], 7, false, false, true, 1.741131089810984e9
```

12.3 Modelando problemas com o computador

O computador pode ser uma ferramenta bem poderosa para a modelagem de problemas reais. Para isso vamos pegar o caso do problema dos aniversários. Esse problema também é conhecido pelo paradoxo do aniversário: Calcular a probabilidade de que em uma sala com n pessoas, pelo menos duas possuam a mesma data de aniversário. Esse problema pode ser resolvido usando probabilidade, por meio da qual se descobre que se a sala tem 23 pessoas a chance de duas terem a mesma data é de pouco mais de 50%.

Mas, também podemos modelar esse problema computacionalmente. Para isso, o primeiro passo é simplificar as datas, ao invés de mês e ano, podemos codificar os dias em um número entre 1 e 365, sendo que 1 corresponderia a primeiro de janeiro. Para resolver o problema, podemos sortear n datas, e ver se há alguma repetição, se houver encontramos duas pessoas com a mesma data.

Isso está representado na função experimento_niver abaixo. Mas, para saber a chance real, temos que repetr o experimento várias vezes. Na função main() abaixo, pedimos a quantidade de experimentos e o número de pessoas para executar a simulação.

```
function experimento_niver(n)
    repetiu = false
    i = 1
    nivers = []
    while i <= n && (repetiu == false)
        niver = rand(1:365)
        if niver in nivers
            repetiu = true
        end
        push!(nivers, niver)
        i += 1
    end
    return repetiu
end

function main()
    print("Quantos experimentos? ")
    quantas = readline()
    print("Quantas pessoas? ")
    npessoas = readline()
    quantas = parse(Int64, quantas)
    npessoas = parse(Int64, npessoas)
    sucessos = 0
    i = 1
    while i <= quantas
        if experimento_niver(npessoas)
            sucessos += 1
        end
        i += 1
    end
    println("A probabilidade estimada é ", 100*sucessos/quantas, "%")
end
main()
```

A parte interessante é que podemos com pequenas variações ter outros experimentos, como verificar se mais do que duas pessoas fazem aniversário na mesma data. Para isso, abaixo, contamos o número de repetições.

```
function experimento_niver(n)
    repetiu = 0
    i = 1
    nivers = []
```

```
    while i <= n
        niver = rand(1:365)
        if niver in nivers
            repetiu += 1
        end
        push!(nivers, niver)
        i += 1
    end
    return repetiu >= 2
end
```

```
experimento_niver (generic function with 1 method)
```

13 Modelando um problema maior

Nessa aula vamos modelar um jogo bem conhecido, o 21, ou BlackJack. Nele os jogadores devem tentar chegar mais perto da soma de cartas 21, sem estourar. Quem chegar mais perto ganha.

Cada jogador começa com duas cartas, sendo que as cartas tem o seu valor nominal, as figuras (J, Q, K), que valem 10. Além disso, o Ás, pode valer 1 ou 11. O que for mais vantajoso para o jogador.

Para começar vamos fazer uma simulação com um baralho, ou seja 52 cartas. Já que] para o jogo, não importa o naipe da carta, vamos supor que existem quatro cartas de cada. Para isso, vamos criar duas funções, uma que cria um baralho e o guarda em um vetor, e uma segunda que pega uma carta do baralho. Nessa segunda função temos que "retirar" a carta do vetor. Caso já não exista a carta do tipo desejado, temos que sortear uma nova carta.

```
function criaBaralho()
  cards = zeros(Int8, 13)
  i = 1
  while i < 14
    cards[i] = 4
    i += 1
  end
  return cards
end

function pegarCarta(cards)
  sorteio = rand(1:13)
  while cards[sorteio] == 0
    sorteio = rand(1:13)
  end
  cards[sorteio] -= 1
  if sorteio > 10  # se a carta for figura, ela vale 10
    sorteio = 10
  end
  return sorteio
end
```

```
pegarCarta (generic function with 1 method)
```

De posse dessas duas funções, podemos criar outras que simulam o comportamento dos jogadores. Vamos usar algumas estratégias simples, como o jogador que fica com as duas cartas que recebeu.

```
function jogador1(cards)
  carta1 = pegarCarta(cards)
  carta2 = pegarCarta(cards)
  if carta1 == 1 || carta2 == 1
    return carta1 + carta2 + 10
  else
    return carta1 + carta2
  end
end
```

```
jogador1 (generic function with 1 method)
```

Notem que acima, usamos a estratégia de usar o Ás da forma mais vantajosa.

Para os outros jogadores, vamos usar estratégias mais elaboradas, ou seja o jogador fica pegando cartas enquanto não chegar a um valor pré-determinado, como por exemplo 21, 19, 17, 15 e 13.

Como cada jogador pode ter um número grande de cartas e no caso dele ter um Ás, a conta tem que ser feita da maneira mais vantajosa, vamos usar uma função que recebe um vetor de cartas e calcula a soma.

```
function somaCartas(c)
  soma = 0
  temAz = false
  for i in c
    soma += i
    if c == 1
      temAz = true
    end
  end
  if soma <= 11 && temAz
      return soma + 10
  else
      return soma
  end
end
```

```
somaCartas (generic function with 1 method)
```

De posse do soma cartas, podemos modelar os jogadores.

```
function jogador2(cards)
  cartas = []
  push!(cartas, pegarCarta(cards))
  push!(cartas, pegarCarta(cards))
  while somaCartas(cartas) < 21
    push!(cartas, pegarCarta(cards))
  end
  return somaCartas(cartas)
end

function jogador3(cards)
  cartas = []
  push!(cartas, pegarCarta(cards))
  push!(cartas, pegarCarta(cards))
  while somaCartas(cartas) < 19
    push!(cartas,pegarCarta(cards))
  end
  return somaCartas(cartas)
end

function jogador4(cards)
  cartas = []
  push!(cartas, pegarCarta(cards))
  push!(cartas, pegarCarta(cards))
  while somaCartas(cartas) < 17
    push!(cartas,pegarCarta(cards))
  end
  return somaCartas(cartas)
end

  function jogador5(cards)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < 15
      push!(cartas,pegarCarta(cards))
    end
    return somaCartas(cartas)
```

```
  end

function jogador6(cards)
  cartas = []
  push!(cartas, pegarCarta(cards))
  push!(cartas, pegarCarta(cards))
  while somaCartas(cartas) < 13
    push!(cartas,pegarCarta(cards))
  end
  return somaCartas(cartas)
end
```

```
jogador6 (generic function with 1 method)
```

Agora que temos todos os jogadores, podemos modelar uma partida. Para isso criamos um baralho e fazemos com que cada jogador siga a sua estratégia

```
function partida()
  cards = criaBaralho()
  jogadores = zeros(Int8, 6)
  jogadores[1] = jogador1(cards)
  jogadores[2] = jogador2(cards)
  jogadores[3] = jogador3(cards)
  jogadores[4] = jogador4(cards)
  jogadores[5] = jogador5(cards)
  jogadores[6] = jogador6(cards)
end
```

```
partida (generic function with 1 method)
```

Não deu tempo de continuar, ficou para a próxima aula.

14 Continuando a modelagem

No capítulo anterior ficamos com uma partida, mas sem a verificação do vencedor, ou seja o jogador com o maior valor, menor ou igual a 21. Uma decisão de projeto é dizer que no caso de empate, os jogadores, com os maiores valores ganham e dividem o prêmio.

```
function partida()
  cards = criaBaralho()
  jogadores = zeros(Int8, 6)
  jogadores[1] = jogador1(cards)
  jogadores[2] = jogador2(cards)
  jogadores[3] = jogador3(cards)
  jogadores[4] = jogador4(cards)
  jogadores[5] = jogador5(cards)
  jogadores[6] = jogador6(cards)
  return jogadores
end
```

```
partida (generic function with 1 method)
```

Logo, a partida devolve a pontuação de cada jogador, para podermos verificar na rotina ganhador quem ganhou.

```
function ganhador(v)
    i = 1
    maximo = 0
    while i <= length(v)
        if v[i] > 21  # se estourou é como se tivesse o menor valor
            v[i] = 0
        end
        if v[i] > maximo
            maximo = v[i]  # encontra o vencedor
        end
        i = i + 1
    end
    result = zeros(Int64, length(v))
```

```
    i = 1
        while i <= length(v)
            if v[i] == maximo
                result[i] = 1
            end
            i = i + 1
        end
    return result
end
```

```
ganhador (generic function with 1 method)
```

A rotinha ganhador devolve um vetor com os vencedores, com 1 na posição de quem ganhou e zero na posição dos perdedores.

Uma das vantagens de se usar um computador é que podemos ter milhares de partidas de 21 para encontrar qual seria a melhor estratégia.

```
function porcentagem()
    i = 1
    porc = zeros(Int64, 6)
    while i < 100000
        porc = porc + ganhador(partida())
        i = i + 1
    end
    println(porc)
end
```

```
porcentagem (generic function with 1 method)
```

Ao simularmos o jogo 10000 vezes, podemos encontrar qual é a melhor estratégia dentre as que foram apresentadas.

O código acima ficou relativamente grande, e uma das coisas que podemos notar é que há muita duplicação nos códigos dos Jogadores a partir do segundo. Um dos maiores problemas de código é a duplicação. No caso acima, podemos evitá-la adicionando um parâmetro à função Jogador, de forma que esse seja o limite a ser considerado no laço. A função jogador2 fica assim:

```
function jogador2(cards, valor)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < valor
       push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end
```

```
jogador2 (generic function with 1 method)
```

Como a função tem um parâmetro novo, temos que acertar a partida. Mas, agora podemos usar todos os valores.

```
function partida()
   cards = criaBaralho()
   jogadores = zeros(Int8, 6)
   jogadores[1] = jogador1(cards)
   jogadores[2] = jogador2(cards, 21)
   jogadores[3] = jogador2(cards, 20)
   jogadores[4] = jogador2(cards, 19)
   jogadores[5] = jogador2(cards, 18)
   jogadores[6] = jogador2(cards, 17)
   return jogadores
end
```

```
partida (generic function with 1 method)
```

Notem que não há mudança na função ganhador, que continua funcionando.

Para terminar, podemos ter agora uma versão interativa que permite que um jogador humano jogue com o computador.

```
function partidaComHumano()
    cards = criaBaralho()
    humano = []
    computador = jogador2(cards, 19)
    push!(humano, pegarCarta(cards))
    push!(humano, pegarCarta(cards))
    println("O humano tem ", humano, " e soma ", somaCartas(humano))
```

```
    println("O humano quer mais cartas (S/N)?")
    resp = readline()
    while resp == "S" || resp == "s"
         push!(humano, pegarCarta(cards))
         println("O computador tem ", computador, " e soma ", somaCartas(computador))
         println("O humano tem ", humano, " e soma ", somaCartas(humano))
         println("O humano quer mais cartas (S/N)?")
         resp = readline()
    end
    println("O computador tem ", computador, " e soma ", somaCartas(computador))
    if somaCartas(computador) <= 21 && somaCartas(humano) <= 21
         if somaCartas(computador) > somaCartas(humano)
             println("Humano Perdeu")
         elseif somaCartas(computador) == somaCartas(humano)
             println("Empate")
         else
             println("Humano ganhou")
         end
    elseif somaCartas(computador) > 21 && somaCartas(humano) > 21
         println("os dois perderam")
    elseif somaCartas(computador) > 21
         println("Humano ganhou")
    else
         println("Computador ganhou")
    end
end
```

```
partidaComHumano (generic function with 1 method)
```

15 Boas práticas

Vamos começar apresentando 3 boas práticas de programação. Na verdade há uma área que cuida de desenvolvimento de software, a Engenharia de Software. Vamos a elas:

15.1 Uso de contratos

Sempre que possível o código deve ser modular, ou seja estar repartido em arquivos e ou funções. Cada tipo de função deve deixar claro quais são os seus parâmetros e o que ela devolve. Isso pode ser feito usando tipos.

```
function fatorial(n::Int64)::Int64
    if n < 2
        return 1
    else
        return n * fatorial(n - 1)
    end
end
```

```
fatorial (generic function with 1 method)
```

Com isso, fica claro o que a função recebe e devolve, e se for enviado um tipo diferente do esperado, temos em erro imediato.

15.1.1 Boa prática 1: Use tipos

15.2 Testes automatizados

Para evitar que apareçam erros, ou os populates bugs, uma forma eficaz é escrever código que verifica o funcionamento do código. Se isso for feito de forma automática, temos os testes automatizados.

```julia
using Test
function testaFat()
  @test fatorial(3) == 6
  @test fatorial(5) == 120
  @test fatorial(1) == 1
  @test fatorial(0) == 1
  @test fatorial(4) == 24
end
```

```
testaFat (generic function with 1 method)
```

15.2.1 Boa prática 2: Sempre que possível faça testes

15.3 Escreva código para humanos, não para computadores

Apesar dos computadores serem capazes de ler código nem sempre bem formatado, é bem difícil para humanos lerem código de forma não padrão. Por isso algumas dicas importantes são:

- Use identação. Com isso, os blocos ficam bem claros e é fácil identificar os laços, blocos de if e corpos de função;
- Escolha bem o nome das variáveis e funções, isso ajuda muito quem for ler o código
- Sempre que você identificar uma possibilidade de melhoria no código, implemente. Ainda melhor se você tiver testes automatizados, para verificar que a melhoria não quebrou o código.

15.3.1 Boa prática 3: Escreva código para que outros leiam

15.4 Aplicando as boas práticas

Vamos agora resolver o seguinte problema, aplicando as práticas acima. Dada um vetor com números reais, determinar os números que estão no vetor e o número de vezes que cada um deles ocorre na mesma.

Ao analizar o problema, vemos que temos como entrada um vetor de número reais, que pode conter repetições. Para determinar os números que estão no vetor, podemos usar um outro vetor de saída. Sendo que o de entrada e o de saída devem ser do tipo Float64. Além disso, para o vetor que fornece a quantidade de números temos um vetor de inteiros. De posse disso, já temos a assinatura da função.

```
function contHist(v::Vector{Float64}, el::Vector{Float64}, qtd::Vector{Int64})
end
```

```
contHist (generic function with 1 method)
```

De posse dessa assinatura, já podemos escrever os testes.

```
function verifica(v::Vector{Float64}, elementos::Vector{Float64},
     quant::Vector{Int64})
     el = Float64[]
     quan = Int64[]
     contHist(v, el, quan)
     if el == elementos && quan == quant
        return true
     else
        return false
     end
end

function testaLista()
  @test verifica([1.3, 1.2, 0.0, 1.3], [1.3, 1.2, 0.0], [2, 1, 1])
  @test verifica([1.0, 1.0, 1.0, 1.0], [1.0], [4])
  @test verifica([8.3], [8.3], [1])
  @test verifica([3.14, 2.78, 2.78], [3.14, 2.78], [1, 2])
end
```

```
testaLista (generic function with 1 method)
```

Finalmente, podemos escrever o código. A idea para escrever a solução é simples, vamos percorrer o vetor de entrada. Para cada elemento, temos duas possibilidades, se ele não tiver aparecido antes, temos que adicionar o número ao vetor saída e marcar 1 ocorrência. Se já apareceu, basta incrementar o número de ocorrências.

```
function contHist(v::Vector{Float64}, el::Vector{Float64}, qtd::Vector{Int64})
    for a in v
        if a in el
            i = 1
            while el[i] != a
               i += 1
            end
```

```
            qtd[i] += 1
        else
            push!(el, a)
            push!(qtd, 1)
        end
    end
end
```

```
contHist (generic function with 1 method)
```

Parte III

Parte III: Conceitos Avançados

16 Indo além de uma dimensão (Matrizes)

Até o momento trabalhamos com estruturas com mais de uma dimensão, mas sem olharmos muito bem o seu tipo. Nessa aula vamos procurar entender as diferenças entre elas e como isso pode ser usado ao nosso favor.

Vamos começar com as listas:

```
v = [1, 2, 3]
typeof(v)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

O tipo devolvido é: Vector{Int64} (alias for Array{Int64, 1}). No caso isso significa que v é um vetor de inteiros, ou um array de uma dimensão. Da mesma forma

```
v = zeros(Int64, 3)
typeof(v)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

Mas, vetores podem ser mais flexíveis, como por exemplo abaixo:

```
v = [1, 2.0, "três"]
typeof(v)
```

```
Vector{Any} (alias for Array{Any, 1})
```

Nesse caso o tipo de vetor, deixa de ser de inteiros e passa a ser "Any", ou seja Vector{Any} (alias for Array{Any, 1}).

Mais ainda, imaginem a seguinte situação:

```
a = [1, 2, 3]
push!(v, a)
typeof(v)
```

```
Vector{Any} (alias for Array{Any, 1})
```

Nesse caso, o vetor continua sendo do tipo Any, mas na quarta posição temos um vetor com três inteiros. Com isso podemos ver que as estruturas de vetores podem ser bem flexíveis. Mas, apesar disso, quando temos estruturas de tipos diferentes, com muita flexibilidade, geralmente há alguma penalidade de uso, geralmente no desempenho.

Por outro lado, podemos ter estruturas com mais de uma dimensão, no caso elas são denominadas matrizes. Elas podem ser criadas com a função zeros que já usamos acima.

```
m = zeros(Int64, 3, 2)
typeof(m)
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

Acima foi criada uma matriz de duas dimensões com 3 linhas e duas colunas. Seus elementos podem se acessados como em um vetor, mas agora com dois indíces.

```
m[1, 2]  = 10
```

```
10
```

```
function imprime(m::Array{Int64,2})
    println(m)
end
```

```
imprime (generic function with 1 method)
```

```
function imprime(m::Vector{Vector{Int64}})
    println(m[1])
    println(m[2])
end
```

```
imprime (generic function with 2 methods)
```

```
function imprime(m::Vector{Vector{Int64}})
    for i in m
        println(i)
    end
 end
```

```
imprime (generic function with 2 methods)
```

```
function imprime(m::Vector{Vector{Int64}})
    for i in m
        for j in m[i]
            println(j,"  ")
        end
    end
end
```

```
imprime (generic function with 2 methods)
```

```
function imprime(m::Vector{Vector{Int64}})
    for i in m
        print("|")
        for j in i
            print(j,"  ")
        end
        println("|")
    end
end
```

```
imprime (generic function with 2 methods)
```

```
function imprimeMatriz(m::Matrix{Int64})
    println(m)
end
```

```
imprimeMatriz (generic function with 1 method)
```

```
function imprimeMatriz(m::Matrix{Int64})
    i = 1
    while i < size(m)[1]
        println(m[1])
        i += 1
    end
end
```

```
imprimeMatriz (generic function with 1 method)
```

```
function imprimeMatriz(m::Matrix{Int64})
    i = 1
    while i < size(m)[1]
        j = 1
        while j < size(m)[2]
            print(m[i, j], " ")
            j += 1
        end
        println()
        i += 1
    end
end
```

```
imprimeMatriz (generic function with 1 method)
```

```
function preencheMatriz(m::Matrix{Int64})
    i = 1
    while i <= length(m)
        m[i] = rand(Int) % 10
        i += 1
    end
end
```

```
preencheMatriz (generic function with 1 method)
```

```
function criaIdentidate(tam::Int64)
    m = zeros(Int64, tam, tam)
    i = 1
    while i <= tam
        m[i, i] = 1
```

```
    end
    return m
end
```

```
criaIdentidate (generic function with 1 method)
```

Operações diretas com matrizes tipo +, - e *

17 Aula de exercícios sobre Strings

Nesta aula, vamos explorar funções que manipulam strings e criar testes para verificar sua correção. Em algumas funções, vamos notar que há diversas formas de se obter o mesmo resultado

17.1 Concatenação de letras

A primeira função `concatena` concatena as primeiras duas e as últimas duas letras de uma string.

```
function concatena(s::String)::String
    if length(s) < 2
        return "Erro: tamanho da string menor do que 2"
    end
    resposta = s[1:2]*s[end-1:end]
    return resposta
end
```

```
concatena (generic function with 1 method)
```

Awui utulizamos `s[1:2]` para obter as duas primeiras letras de s, que é uma forma mais concisa de acessar mais de um índice de um objeto. Alternativamente, poderíamos acessar esses dois índices separadamente com o comando `s[1]*s[2]`.

Para verificar se a função está funcionando corretamente, podemos utilizar o seguinte teste:

```
using Test

function testeConcatena()
  @test concatena("Ola Bom Dia") == "Olia"
  @test concatena("oi") == "oioi"
  @test concatena("tre") == "trre"
  @test concatena("a") == "Erro: tamanho da string menor do que 2"
  @test concatena("a123") == "a123"
end
```

```
testeConcatena (generic function with 1 method)
```

17.2 Inversão de String

Devemos criar uma função que interte uma string, retornando os caracteres na ordem reversa.

```
function inverte(s::String)::String
    # Inicializamos uma string vazia
    inversa=""

    # Intervalo de lenght(s) até 1, a passos de -1
    for i in length(s):-1:1
        # Concatena cada caractere na ordem inversa
        inversa*=s[i]
    end

    return inversa
end
```

```
inverte (generic function with 1 method)
```

Para obter o resultado que desejamos, fazemos um laço `for` que itera do último índice da string, representado por `length(s)`, até o primeiro, concatenando os caracteres nessa ordem na string de retorno. O laço é configurado para decrementar o índice a cada iteração, especificando -1 como passo. Isso nos permite acessar cada caractere da string de trás para frente. E em cada iteração, concatenamos o caractere atual, `s[i]`, à string `inversa`. Dessa forma, os caracteres são adicionados na ordem inversa.

Agora podemos criar uma função de teste para verificar o funcionamento da nossa função `inverte`.

```
using Test

function testeInverte()
  @test inverte("123") == "321"
  @test inverte("x") == "x"
  @test inverte("SOS") == "SOS"
  @test inverte("tres") == "sert"
end
```

```
testeInverte (generic function with 1 method)
```

17.2.1 Função reverse

É interessante notar que Julia já fornece uma função chamada `reverse`, que pode ser utilizada para inverter tanto vetores quanto strings. Por exemplo:

```
reversa = reverse("exemplo")
```

```
"olpmexe"
```

Neste exemplo, a função `reverse` recebe como parâmetro apenas o objeto a ser invertido, mas no caso de vetores, podemos ainda informar exatamente o intervalo que desejamos que seja invertido.

```
vetor = [1, 2, 3, 4, 5]
reversa = reverse(vetor, 2, 4)
```

```
5-element Vector{Int64}:
 1
 4
 3
 2
 5
```

17.3 Modificação de String

A terceira função modifica altera uma string que termina com “ing” para adicionar “ly” ou, caso contrário, adiciona “ing”.

```
function modifica(s::String)::String
    if length(s) < 3
        return "Erro: tamanho da string menor do que 3"
    end

    if s[end-2:end] == "ing"
        s = s*"ly"
    else
        s = s*"ing"
    end
```

```
    return s
end
```

```
modifica (generic function with 1 method)
```

Neste exemplo, verificamos manualmente os últimos três caracteres da string s. No entanto, Julia oferece uma função mais prática e legível chamada `endswith`, que podemos usar para simplificar essa verificação.

```
function modifica(s::String)::String
    if length(s) < 3
        return "Erro: tamanho da string menor do que 3"
    end

    if endswith(s, "ing")
        s = s*"ly"
    else
        s = s*"ing"
    end

    return s
end
```

```
modifica (generic function with 1 method)
```

Vamos então escrever o teste que verifica o correto funcionamento das funções anteriores

```
using Test
function testaModifica()
  @test modifica("doing") == "doingly"
  @test modifica("sing") == "singly"
  @test modifica("run") == "runing"
  @test modifica("talk") == "talking"
end
```

```
testaModifica (generic function with 1 method)
```

17.4 Rearranjo de letras

A segunda função `rearranja` recebe uma string e devolve uma string que contém as letras minúsculas primeiro, seguidas pelas letras maiúsculas.

Podemos verificar se uma letra é maiúscula ou minúscula usando a tabela ASCII, que codifica caracteres em números inteiros. Na tabela, as letras maiúsculas estão no intervalo de 65 a 90, e as letras minúsculas no intervalo de 97 a 122.

Para saber mais sobre a tabela ASCII você pode acessar essa página.

```
function rearranja(s::String)::String
    maiusculos=""
    minusculos=""

    for i in 1:length(s)
        if Int(s[i]) >= 65 && Int(s[i]) <= 90
            maiusculos = maiusculos*s[i]
        elseif Int(s[i]) >= 97 && Int(s[i]) <= 122
            minusculos = minusculos*s[i]
        end
    end

    return minusculos*maiusculos

end
```

```
rearranja (generic function with 1 method)
```

Uma abordagem mais legível é utilizar as funções `islowercase` e `isuppercase`, que verificam se uma letra é minúscula ou maiúscula, respectivamente.

```
function rearranja(s::String)::String
    maiusculos=""
    minusculos=""

    for i in 1:length(s)
        if isuppercase(s[i])
            maiusculos = maiusculos*s[i]
        elseif islowercase(s[i])
            minusculos = minusculos*s[i]
        end
```

```
    end

    return minusculos*maiusculos

end
```

```
rearranja (generic function with 1 method)
```

Podemos então escrever o teste para nossas funções.

```
using Test

function testaRearranja()
  @test rearranja1("PaRaLelO") == "aaelPRLO"
  @test rearranja1("ELEfantE") == "fantELEE"
  @test rearranja1("Olá") == "lO"
  @test rearranja1("13La2") == "aL"
  @test rearranja2("PaRaLelO") == "aaelPRLO"
  @test rearranja2("ELEfantE") == "fantELEE"
  @test rearranja2("Olá") == "láO"
  @test rearranja2("13La2") == "aL"
end
```

```
testaRearranja (generic function with 1 method)
```

17.5 Encontrar a maior palavra

Nossa última função deve receber uma lista de palavras e retornar a maior delas, junto de seu tamanho.

```
function maior_palavra(vetor::Vector{String})
    # Inicialmente, a maoior palavra que encontramos é uma string vazia
    maior_palavra = ""
    maior_tamanho = 0

    for palavra in vetor
        #  Verifica se a palavra atual é maior que a maior encontrada até agora
        if length(palavra) > maior_tamanho
            maior_palavra = palavra
```

```
            maior_tamanho = length(palavra)
        end
    end

    return maior_palavra, maior_tamanho

end
```

```
maior_palavra (generic function with 1 method)
```

Apesar de parecer correto, esse código não lida com o caso de haver mais de uma palavra com o maior tamanho. Como por exemplo:

```
vetor = ["boa", "bem", "oi"]
maior_palavra(vetor)
```

```
("boa", 3)
```

Nesse caso, apenas a palavra "boa" será retornada, mesmo que "bem" tenha o mesmo tamanho. Para consertar a função devemos alterar a variável em que guardamos a maior palavra, para que possamos armazenar mais de uma palavra, para isso vamos usar um vetor de strings.

```
function maiores_palavras(vetor::Vector{String})
    maiores_palavras = String[]
    maior_tamanho = 0

    for palavra in vetor
        # Se a palavra é maior do que o maior tamanho salvo,
        # então todas as palavras que estão no vetor maior_palavra são menores do que a palav
        if length(palavra) > maior_tamanho
            # Limpa o vetor e salva a palavra atual
            maiores_palavras = String[]
            push!(maiores_palavras, palavra)
            maior_tamanho = length(palavra)

        # Se é igual ao tamanho salvo, então é do mesmo tamanho que as palavras já salvas no
        # apenas damos push na palavra atual
        elseif length(palavra) == maior_tamanho
            push!(maiores_palavras, palavra)
        end
```

```
    end

    return maiores_palavras, maior_tamanho

end
```

```
maiores_palavras (generic function with 1 method)
```

Assim podemos escrever testes para esta última função .

```
using Test

function testeMaioresPalavras()
    vetor1 = ["gato", "elefante", "cachorro"]
    @test maiores_palavras(vetor1) == (["elefante", "cachorro"], 8)

    vetor2 = ["a", "ab", "abc"]
    @test maiores_palavras(vetor2) == (["abc"], 3)

    vetor3 = ["bem", "boa", "bom", "oi"]
    @test maiores_palavras(vetor3) == (["bem", "boa", "bom"], 3)

    vetor4 = ["", " ", "teste"]
    @test maiores_palavras(vetor4) == (["teste"], 5)

    vetor5 = String[]
    @test maiores_palavras(vetor5) == ([], 0)

    vetor6 = ["a", "ab", "abc", "xyz", "xy"]
    @test maiores_palavras(vetor6) == (["abc", "xyz"], 3)
end
```

```
testeMaioresPalavras (generic function with 1 method)
```

17.6 Retorno de múltiplos valores

Como visto no exercício anterior, Julia permite que uma função retorne múltiplos valores. Isso permite que você envie mais de um resultado ao chamar uma função, tornando o código mais conciso e fácil de entender. Essa funcionalidade é especialmente útil em situações onde

você precisa de mais de um resultado, como em operações matemáticas, decomposições, ou processamento de dados.

Para retornar múltiplos valores em Julia, você pode simplesmente separá-los por vírgulas. Aqui está um exemplo simples:

```
function troca(a,b)
    aux = a
    a = b
    b = aux

    return a, b
end
```

```
troca (generic function with 1 method)
```

Ao chamar essa função, você pode capturar os múltiplos valores retornados em variáveis separadas:

```
a, b = troca(1, 10)
```

```
(10, 1)
```

Parte IV

Em andamento...

A Respostas para a pergunta “O que é um computador?”

As respostas foram coletadas através do site https://www.bli.do/MAC115. Eis as respostas:

- “É uma máquina que utiliza certos padrões para processar informações. É uma evolução da teórica máquina de Turing e pode ser considerada uma extensão de nossa inteligência.”
- “Computador: configura uma máquina capaz de realizar processos binários de criação e análise, sendo esses modelados pelo próprio dispositivo (IA) ou por um programador. Um produto de passos.”
- “Um computador é uma máquina capaz de identificar códigos baseados em binariedade utilizando linguagem lógica.”
- “É um instrumento tecnológico que junta milhões de dados que diversas pessoas ao redor do mundo distribuem, possibilitando trabalhar de forma técnica criando novos dados e meios de alcançar respostas.”
- “Um dispositivo que vive na tomada.”
- “É um equipamento que”roda” um programa instalado que recebe dados, os processa e tem com saída o resultado, se tudo correr bem…”
- “Decodificador.”
- “Tem por finalidade ser uma ferramenta baseada em algoritmos e lógica de decisão que vez programada e orientada possa possibilitar a resolução de problemas por coleta de dados, informações e então geração de sistemas.”
- “Um computador é um dispositivo eletrônico caracterizado pela sua capacidade de processamento lógico. São construídos com semicondutores e pode apresentar várias peças diferentes, como fonte (ou bateria), placa mãe, processador, placa de vídeo, memórias e etc.”
- “um robô que vai controlar os humanos um dia.”
- “Dispositivo que faz contas.”
- Um computador é um dispositivo que computa, que faz contas, utilizando uma CPU (Central Processing Unit) para fazer cálculos utilizando o sistema binário. 0 e 1, que são utilizados em portas lógicas que são feitas de silício, um semicondutor que permite que cálculos utilizando esses dois dígitos sejam realizados.”
- “Uma máquina.”
- “É um negócio que tem ferramentas que ajudam a realizar diferentes tarefas.”
- “É um celular 2.0.”
- “O computador é uma máquina que obedece comandos.”

- “Uma calculadora.”
- “Máquina de Turing.”
- “Computador é, de uma forma bem simplificada, uma inteligência eletrônica composta de uma carcaça (hardware) que completa a parte do software, feita para responder comandos.”
- “A evolução da calculadora.”