

**MAROMO**

# FUNDAMENTOS DA LÓGICA E PROGRAMAÇÃO:

UM GUIA PRÁTICO PARA  
INICIANTES

2024

# Fundamentos da Lógica e Programação: Um Guia Prático para Iniciantes

Este material apresenta os fundamentos essenciais da lógica de programação, com o objetivo de apoiar o aluno em suas etapas iniciais no aprendizado da programação de computadores.

Versão: 1.06  
**Me. Marcos Roberto de Moraes - Maromo**  
2025

## Sumário

1	Introdução.....	8
2	Programação .....	10
2.1	Programas de Computador .....	10
2.2	Linguagem de Máquina .....	11
2.3	Linguagem de Programação.....	12
2.4	Compilador .....	14
2.5	Máquinas Virtuais.....	14
2.6	Containers .....	15
2.7	Criando um Programa em C – usando CLion .....	17
2.8	Exercícios de Fixação .....	19
3	Lógica de Programação .....	20
3.1	Lógica.....	20
3.2	Algoritmos e Implementação .....	22
3.2.1	Solução Simples para o Problema .....	23
3.2.2	Características de um Algoritmo .....	24
3.3	Representação dos Algoritmos .....	24
3.3.1	Fluxogramas .....	24
3.3.2	Vantagens do Fluxograma .....	25
3.3.3	Importância dos Fluxogramas .....	26
3.3.4	Pseudocódigo .....	27
3.3.5	Estrutura Geral de um Pseudocódigo.....	27
3.3.6	Importância do Pseudocódigo.....	29
3.4	Exercício de Fixação.....	29
4	Tipos de Dados .....	31

4.1	Constantes.....	32
4.1.1	Constantes Inteiras.....	32
4.1.2	Constantes Reais .....	33
4.2	Variáveis .....	34
4.2.1	Funcionamento da Memória e Variáveis.....	34
4.2.2	Uso de Variáveis em Java, C e Python .....	34
4.3	Tipos de variáveis .....	37
4.3.1	Tipos primitivos em C .....	37
4.3.2	Tipos primitivos em Java .....	38
4.4	Convenção para a nomenclatura .....	39
4.5	Regras para a nomenclatura .....	39
4.6	Exercício de Fixação.....	40
5	Operadores.....	41
5.1	Tabela Verdade .....	43
5.2	Exercícios de Fixação .....	44
6	Controle de Fluxo .....	47
6.1	Comandos de Decisão .....	47
6.2	Estrutura SE ENTÃO SENÃO / IF... THEN... ELSE .....	48
6.3	Exercícios de Fixação .....	52
6.4	Estruturas de Repetição .....	54
6.4.1	Tipos ou modelos de repetição .....	54
6.5	Exercícios de Fixação .....	57
7	Vetores e Matrizes .....	60
7.1	Vetores .....	60
7.2	Matrizes.....	62
7.3	Exercícios de Fixação .....	64
7.3.1	Exercícios com Vetores.....	64
7.3.2	Exercícios com Matrizes .....	65

8	Linguagem de Projeto de Programação .....	67
8.1	Linguagem PDL .....	68
8.2	Exercícios Complementares .....	68
9	Exercícios Complementares .....	71
9.1	Exercícios .....	71
10	Bibliografia .....	74

## **Lista de Tabelas**

Tabela 1 - Principais formas usadas em fluxogramas .....	25
Tabela 2 - Exemplo de uso de operador de verificação de igualdade em Java.....	42
Tabela 3 - Exemplo de uso do operador && em Java .....	42
Tabela 4 - Exemplo If/else .....	49
Tabela 5 - Cadeia de Ifs em Java .....	52
Tabela 6 - Código Fatorial em Java .....	57
Tabela 7 - Exemplo de um algoritmo para o cálculo da média de 4 notas .....	67

## **Lista de Figuras**

Figura 1 - Planilha Eletrônica .....	10
Figura 2 - Trecho do programa Chrome em Binário (K19) .....	12
Figura 3 - Imagem do processo de compilação .....	14
Figura 4 - Containers .....	15
Figura 5 - Logo do CLion .....	17
Figura 6 - Lógica Aristotélica.....	20
Figura 7 - Fila de Pessoas.....	22
Figura 8 - Receita (Fonte: obadenbaden.com.br).....	22
Figura 9 - Fluxograma .....	26
Figura 10 - Tipos de Dados .....	32
Figura 11 - Operadores Matemáticos.....	41
Figura 12 - Exemplo de Tabela da Verdade.....	43

## 1 Introdução

Ao longo de 25 anos dedicados ao ensino de disciplinas voltadas à Tecnologia da Informação, pude acumular experiências práticas e acadêmicas que contribuíram significativamente para o desenvolvimento deste material didático. A elaboração desta apostila foi cuidadosamente fundamentada em diversas fontes confiáveis, criteriosamente selecionadas, com o objetivo de proporcionar um conteúdo acessível, relevante e voltado à prática do aprendizado em algoritmos e lógica de programação.

Um dos principais diferenciais desta obra é a abordagem adotada, que difere da maioria dos livros tradicionais sobre o tema. Cada capítulo foi estruturado para apresentar problemas práticos que servem como base para a aplicação e consolidação do conhecimento teórico. Esses problemas não apenas auxiliam na compreensão dos conceitos, mas também estimulam o raciocínio lógico e o desenvolvimento de habilidades práticas essenciais na área de programação.

Além do conteúdo teórico, a apostila inclui uma série de exercícios de fixação, cuidadosamente planejados para complementar os tópicos abordados em cada capítulo. Esses exercícios têm como objetivo consolidar o aprendizado adquirido, incentivando o leitor a explorar diferentes perspectivas e soluções para os problemas apresentados. A prática constante é um dos pilares fundamentais para o domínio de algoritmos e lógica de programação, e esta apostila foi projetada para apoiar o leitor nesse processo.

Outro aspecto relevante deste material é o compromisso com a transparência e a ética acadêmica. O conteúdo apresentado inclui um compilado de informações disponíveis na Internet, devidamente referenciadas ao final da apostila, em conformidade com os princípios de propriedade intelectual e direitos autorais. Dessa forma, este material serve tanto como uma ferramenta de aprendizado quanto como uma base de consulta confiável e enriquecedora.

Este material é distribuído sob a licença Creative Commons, que permite a adaptação e a criação de obras derivadas para fins não comerciais. De acordo com os termos dessa licença, é obrigatório que os trabalhos derivados atribuam o devido crédito ao autor

original, mas não é necessário que sejam licenciados sob os mesmos termos. A intenção dessa escolha é promover o compartilhamento de conhecimento e a disseminação de boas práticas de ensino, respeitando os limites impostos pela licença.

### **Mensagem ao Leitor**

Reconhecendo a importância de um material constantemente atualizado e aprimorado, convido o leitor a colaborar com sugestões, apontamentos de eventuais erros ou dúvidas sobre o conteúdo apresentado. Sua contribuição é essencial para a evolução e melhoria deste trabalho. Caso deseje compartilhar suas observações, entre em contato através do e-mail: [professormoraes@gmail.com](mailto:professormoraes@gmail.com). Agradeço antecipadamente por sua colaboração, que ajudará a tornar esta apostila ainda mais eficiente e útil para a formação de futuros profissionais da área de Tecnologia da Informação.



**Professor Marcos Roberto de Moraes (Maromo)**

Mestre em Educação pela UNISAL-SP,  
Especialista em Administração de Sistemas de Informação pela UFLA-MG  
Graduado em Processamento de Dados pela UNIPINHAL - SP  
Professor da Fatec de Mogi Mirim e Itapira / Etec de Mogi Mirim-SP  
Faculdade Santa Lúcia de Engenharia de Computação de Mogi Mirim-SP

## 2 Programação

Neste capítulo, são introduzidos alguns conceitos fundamentais que servirão como base para o estudo de **lógica de programação e algoritmos**. As questões a seguir orientarão a compreensão inicial do tema:

1. O que são programas de computador?
2. O que é uma linguagem de máquina?
3. Para que servem as diferentes linguagens de programação?
4. O que é um Compilador? O que é o processo de compilação?
5. O que são Máquinas Virtuais?
6. **Containers**, o que são e para que servem.

A exploração desses tópicos permitirá ao leitor construir uma base sólida para o entendimento dos próximos conteúdos.

### 2.1 Programas de Computador

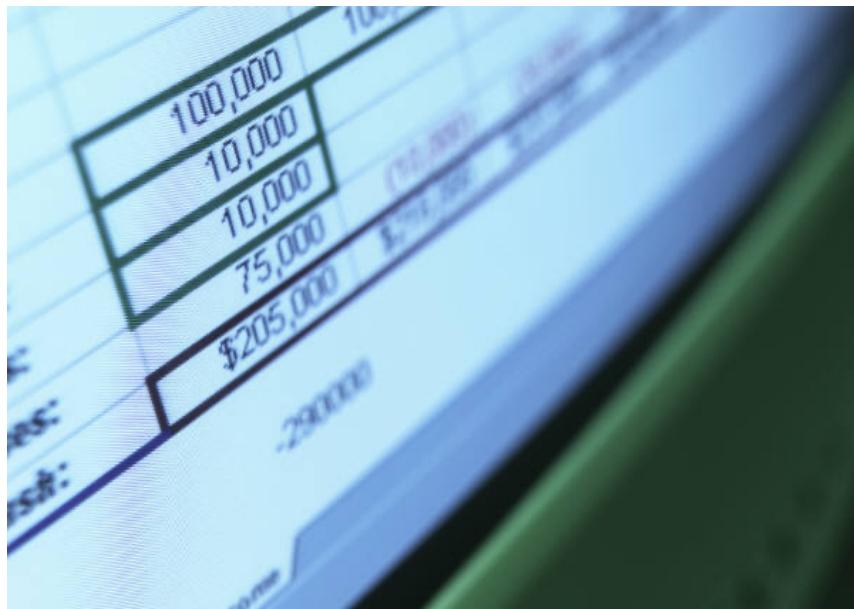


Figura 1 - Planilha Eletrônica

Um **programa de computador** é um conjunto de instruções organizadas logicamente com o objetivo de realizar uma tarefa específica quando interpretadas e executadas por um computador. Essas instruções são escritas em linguagens de programação, que servem como intermediárias entre a lógica humana e o processamento da máquina.

O termo "programa de computador" pode se referir tanto ao **código-fonte** — o texto legível por humanos escrito em uma linguagem de programação — quanto à sua forma **executável**, ou seja, o arquivo gerado após a conversão do código-fonte para linguagem de máquina, geralmente no formato **binário**. A forma binária é composta por sequências de zeros e uns, que são comprehensíveis para o hardware, mas extremamente difíceis de serem interpretadas por seres humanos.

Um programa pode ser simples, como um script para realizar cálculos básicos, ou extremamente complexo, como sistemas operacionais, aplicativos empresariais ou jogos interativos. De maneira geral, os programas funcionam como **instruções detalhadas** que guiam o computador no processamento de dados e na entrega de resultados desejados.

A criação de um programa envolve um processo que inclui a escrita do código-fonte, sua tradução para linguagem de máquina (geralmente por meio de um **compilador**) e a geração de um arquivo executável. Esse arquivo pode ser executado diretamente pelo sistema operacional, fazendo com que as instruções sejam interpretadas e executadas pelo processador.

Assim, programas de computador representam a base funcional dos sistemas computacionais modernos, permitindo que máquinas realizem desde operações simples até tarefas altamente complexas, facilitando a automação e a resolução de problemas em diversos domínios.

## 2.2 Linguagem de Máquina

Todo computador possui um **conjunto de instruções** que o processador é capaz de interpretar e executar diretamente. Essas instruções constituem o que chamamos de **código de máquina** e são representadas por **sequências de bits** (0 e 1), limitadas pela capacidade do registrador principal da **CPU** (Unidade Central de Processamento). Esse código, também conhecido como **código binário**, é a forma mais básica de linguagem compreendida pelo hardware do computador.

Na **Linguagem de Máquina**, as instruções são escritas como sequências de números binários que determinam operações específicas a serem realizadas pelo processador.

Cada número ou sequência binária corresponde a uma **operação** (como somar, subtrair, mover dados etc.) ou a um **endereço de memória**. Por exemplo, instruções simples como carregar dados em um registrador ou realizar uma operação aritmética são traduzidas em longas sequências de zeros e uns.

Apesar de ser extremamente eficiente para o computador, a Linguagem de Máquina é praticamente inviável para o uso direto por programadores humanos. Sua complexidade está na dificuldade de leitura, interpretação e manutenção do código, uma vez que qualquer erro pode ser difícil de identificar e corrigir. Podemos verificar essa complexidade na Figura 2 - Trecho do programa Chrome em Binário (K19) abaixo:

**Figura 2 - Trecho do programa Chrome em Binário (K19)**

## 2.3 Linguagem de Programação

Uma **linguagem de programação** é um sistema padronizado para expressar instruções que serão interpretadas e executadas por um computador. Ela consiste em um conjunto de **regras sintáticas e semânticas** que permite a criação de programas com instruções precisas sobre como os dados devem ser manipulados, armazenados ou transmitidos e quais ações devem ser realizadas em diferentes situações.

Escrever programas diretamente em **Linguagem de Máquina** é praticamente inviável devido à sua complexidade e dificuldade de manutenção. Para contornar essa limitação, surgiram as **linguagens de programação de alto nível**, que utilizam sintaxes mais próximas da linguagem humana, facilitando a compreensão e a produtividade dos programadores. Estas linguagens permitem que as instruções sejam escritas de maneira mais intuitiva e eficiente, e posteriormente são traduzidas para código de máquina por meio de **compiladores ou interpretadores**.

Abaixo, apresentamos um exemplo simples de um programa que imprime a frase "**Olá, Mundo!**" em três linguagens de programação amplamente utilizadas: **C**, **Python** e **Java**.

### ***Exemplo 1: Código em Linguagem C***

```
#include <stdio.h>

int main() {
    printf ("Olá, Mundo!\n");
    return 0;
}
```

- **Explicação:**

- `#include <stdio.h>`: Importa a biblioteca padrão de entrada e saída.
- `printf`: Função que imprime a mensagem na tela.
- `main()`: Ponto de entrada do programa.

### ***Exemplo 2: Código em Python***

```
print ("Olá, Mundo!")
```

- **Explicação:**

- A instrução `print` exibe a mensagem diretamente.
- Python é uma linguagem interpretada, de sintaxe simples e direta.

### ***Exemplo 3: Código em Java***

```
public class OlaMundo {
    public static void main(String[] args) {
        System.out.println("Olá, Mundo!");
    }
}
```

- **Explicação:**

- `public class OlaMundo`: Declara uma classe pública.
- `main`: Método principal que inicia a execução do programa.
- `System.out.println`: Imprime a mensagem na tela, com quebra de linha.

Como podemos observar, apesar de a **finalidade** dos três exemplos ser a mesma, as sintaxes e estruturas variam de acordo com a linguagem de programação utilizada. Isso ilustra como as linguagens de alto nível facilitam o desenvolvimento de programas, tornando-os **mais acessíveis, legíveis e manuteníveis** em comparação com o código em linguagem de máquina.

## 2.4 Compilador

Se o processador entende apenas Linguagem de Máquina, como ele irá interpretar o código escrito em uma Linguagem de Programação?

Um compilador é um programa de computador ou um conjunto de programas que, a partir de um código fonte escrito em uma linguagem de programação, cria um programa semanticamente equivalente, porém escrito em outra linguagem, objeto. Ou seja, o papel do compilador é, basicamente, “traduzir” um código em Linguagem de Programação para um código em Linguagem de Máquina.



Figura 3 - Imagem do processo de compilação

## 2.5 Máquinas Virtuais

Empresas que almejam disponibilizar suas aplicações em múltiplos sistemas operacionais — como Windows, Linux e macOS — e em diversas arquiteturas de processadores — tais como Intel, ARM e PowerPC — frequentemente enfrentam o desafio de desenvolver versões distintas do código-fonte para cada combinação de plataforma e arquitetura. Essa necessidade pode resultar em investimentos substancialmente elevados em desenvolvimento e manutenção, conforme destacado por Blanco e Lucrédio (2021).

Para mitigar esse desafio, uma abordagem eficaz é a utilização de **máquinas virtuais (VMs)**. As VMs atuam como uma camada intermediária entre o código compilado e a plataforma subjacente, permitindo que o código seja compilado para a máquina virtual, que, por sua vez, interpreta e executa as instruções no ambiente específico. Essa estratégia facilita a portabilidade e a compatibilidade das aplicações em diferentes sistemas operacionais e arquiteturas de hardware, conforme descrito por Blanco e Lucrédio (2021).

Além disso, o uso de máquinas virtuais oferece vantagens adicionais, como a capacidade de executar múltiplos sistemas operacionais simultaneamente em um único hardware físico, proporcionando flexibilidade e eficiência no uso de recursos, conforme discutido por Techtunes (2024).

Em resumo, a adoção de máquinas virtuais representa uma solução eficaz para empresas que buscam reduzir custos e complexidades associadas ao desenvolvimento e à manutenção de múltiplas versões de software para diferentes plataformas, ao mesmo tempo em que aprimoram a flexibilidade e a eficiência operacional.

## 2.6 Containers

Os **containers** representam uma tecnologia que permite empacotar uma aplicação juntamente com todas as suas dependências — como bibliotecas e configurações necessárias — em uma unidade padronizada de software. Essa abordagem assegura que a aplicação seja executada de maneira consistente em diversos ambientes, independentemente das variações na infraestrutura subjacente.

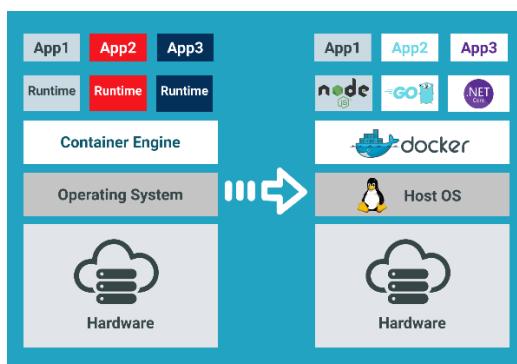


Figura 4 - Containers

### Vantagens dos Containers:

- **Portabilidade e Consistência:** Ao encapsular todas as dependências necessárias, os containers garantem que a aplicação funcione de forma uniforme em diferentes plataformas, eliminando problemas de compatibilidade.
- **Eficiência de Recursos:** Diferentemente das máquinas virtuais tradicionais, os containers compartilham o kernel do sistema operacional hospedeiro, o que reduz a sobrecarga e permite uma utilização mais eficiente dos recursos de hardware.
- **Escalabilidade e Agilidade:** Os containers podem ser iniciados e interrompidos rapidamente, facilitando a escalabilidade horizontal das aplicações para atender a variações na demanda de maneira ágil.
- **Isolamento de Processos:** Proporcionam um ambiente isolado para a execução de aplicações, aumentando a segurança e permitindo que múltiplos containers sejam executados simultaneamente sem interferência mútua.

### Containers vs. Máquinas Virtuais:

Embora tanto os containers quanto as máquinas virtuais (VMs) ofereçam formas de virtualização, eles diferem significativamente em sua arquitetura e utilização de recursos.

- **Máquinas Virtuais:** Cada VM inclui um sistema operacional completo, além da aplicação e suas dependências, resultando em maior consumo de recursos e tempos de inicialização mais longos.
- **Containers:** Compartilham o sistema operacional hospedeiro, necessitando apenas das bibliotecas e dependências específicas da aplicação, o que os torna mais leves e rápidos em comparação às VMs.

### Aplicações Práticas dos Containers:

- **Desenvolvimento e Testes:** Facilitam a criação de ambientes de desenvolvimento consistentes, permitindo que os desenvolvedores testem suas aplicações em condições idênticas às de produção.
- **Implantação de Aplicações:** Simplificam o processo de implantação, permitindo que as aplicações sejam distribuídas e executadas em diferentes ambientes de forma uniforme.
- **Arquitetura de Microsserviços:** Apoiam a decomposição de aplicações monolíticas em serviços menores e independentes, cada um executado em seu próprio container, facilitando a escalabilidade e a manutenção.

### Ferramentas Populares de Containerização:

- **Docker:** Uma plataforma amplamente utilizada que facilita a criação, distribuição e execução de containers, oferecendo uma interface amigável e um ecossistema robusto.
- **Kubernetes:** Uma plataforma open source para orquestração de containers, que automatiza a implantação, o dimensionamento e a gestão de aplicações em containers.

A adoção de containers tem transformado significativamente o desenvolvimento e a implantação de software, promovendo maior eficiência, flexibilidade e consistência em ambientes de TI modernos. Sua capacidade de isolar aplicações e suas dependências em um pacote portátil e leve os torna uma escolha ideal para arquiteturas nativas da nuvem e práticas de DevOps.

## 2.7 Criando um Programa em C – usando CLion



Figura 5 - Logo do CLion

Neste tópico, veremos como criar e executar um simples programa **Hello World** em linguagem C utilizando a **IDE CLion**, uma ferramenta popular para desenvolvimento em C e C++.

### Passo 1: Configurando o Ambiente no CLion

1. Abra o **CLion**. Caso ainda não tenha instalado, baixe e instale a IDE a partir do site oficial da **JetBrains**.
2. Certifique-se de ter instalado o **compilador GCC** ou **Clang**, pois o CLion utiliza um desses compiladores para processar o código C.
3. Crie um novo projeto **C** no CLion:
  - Vá em File > New Project.
  - Escolha a opção **C Executable**.

- Nomeie o projeto como **HelloWorld** e clique em **Create**.

## Passo 2: Escrevendo o Código em C

No arquivo **main.c** que é criado automaticamente no projeto, substitua o código existente pelo seguinte programa:

```
#include <stdio.h> // Biblioteca padrão de entrada e saída

int main(){
    printf("Hello, World!\n"); // Imprime a mensagem na tela
    return 0;                // Retorna 0 indicando sucesso
}
```

- **Explicação do Código:**

- `#include <stdio.h>`: Inclui a biblioteca padrão para funções de entrada e saída, como `printf`.
- `main()`: É a função principal do programa, onde a execução começa.
- `printf`: Função utilizada para exibir texto no console.
- `return 0`: Finaliza o programa retornando o valor 0, indicando execução bem-sucedida.

## Passo 3: Compilando o Programa

1. Clique no botão **Build** no menu superior do CLion, ou pressione **Ctrl + F9** (no Windows/Linux) ou **Cmd + F9** (no macOS).
  - O CLion compilará o programa utilizando o **GCC** ou **Clang**.
2. Verifique a aba **Build** no rodapé para garantir que a compilação foi concluída sem erros.

## Passo 4: Executando o Programa

1. Para executar o programa, clique no botão **Run** (ícone de triângulo verde) no menu superior, ou pressione **Shift + F10**.
2. O resultado será exibido no terminal integrado do CLion, conforme mostrado abaixo:

```
Hello, World!
```

## Resumo do Processo

- **Criação do Projeto:** Usando o CLion, configure um novo projeto **C Executable**.
- **Escrita do Código:** Implemente o programa **Hello World** no arquivo main.c.
- **Compilação:** Utilize o comando **Build** para compilar o código.
- **Execução:** Rode o programa e verifique a saída no terminal.

## 2.8 Exercícios de Fixação

- 1) Por que as linguagens de programação são preferíveis à linguagem de máquina para o desenvolvimento de software moderno? Explique, citando exemplos de código em C e a importância de compiladores ou máquinas virtuais.
- 2) Imagine que você é responsável por desenvolver um software que precisa rodar em sistemas operacionais diferentes (Windows, Linux, macOS) e em processadores variados (Intel, ARM). Quais tecnologias você utilizaria para resolver esse desafio? Justifique sua escolha.
- 3) Qual a principal diferença entre máquinas virtuais e containers? Em sua opinião, qual dessas tecnologias seria mais eficiente em um ambiente de desenvolvimento de software? Explique.
- 4) O programa "Hello World" foi apresentado em diferentes linguagens de programação, como C. Qual é a importância desse tipo de programa na aprendizagem inicial de lógica de programação? Como você adaptaria esse programa para incluir a entrada de um nome digitado pelo usuário?
- 5) Se você fosse criar um programa utilizando a linguagem C e executá-lo no CLion, quais seriam os passos principais para escrever, compilar e executar o código? Descreva o processo em suas próprias palavras.

### 3 Lógica de Programação

Neste capítulo são apresentados os conceitos de lógica, algoritmos e implementação.

#### 3.1 Lógica

A **lógica** é uma disciplina fundamental da filosofia que investiga os princípios e métodos que distinguem argumentos válidos de inválidos, ou seja, aqueles que possuem uma estrutura correta daqueles que não a possuem.

O estudo formal da lógica foi sistematizado pelo filósofo grego **Aristóteles** no século IV a.C. Ele desenvolveu a teoria do **silogismo**, um tipo de argumento constituído por proposições das quais se infere uma conclusão. Aristóteles considerava a lógica não como uma ciência, mas como um instrumento para o pensamento correto.



Figura 6 - Lógica Aristotélica

Um exemplo prático aplicado ao contexto do texto seria o uso do silogismo aristotélico para demonstrar como distinguir um argumento válido de um inválido.

Vamos construir um silogismo válido, respeitando os princípios da **lógica formal**:

**Exemplo de silogismo válido:**

- **Premissa maior:** Todos os seres humanos são mortais.
- **Premissa menor:** Sócrates é um ser humano.
- **Conclusão:** Logo, Sócrates é mortal.

Este é um exemplo de argumento válido porque a conclusão decorre necessariamente das premissas, mantendo a estrutura lógica correta.

Agora, vejamos um exemplo de argumento inválido (estrutura incorreta):

**Exemplo de silogismo inválido:**

- **Premissa maior:** Todos os cães são animais.
- **Premissa menor:** Todos os gatos são animais.
- **Conclusão:** Logo, todos os cães são gatos.

Aqui, a conclusão não decorre das premissas, pois o fato de cães e gatos serem animais não implica que eles sejam iguais. Este é um argumento inválido porque a estrutura não sustenta logicamente a conclusão.

Assim, aplicamos o conceito de Aristóteles de que a lógica serve como instrumento para avaliar a validade de argumentos, ajudando a **distinguir o que é coerente do que não é**.

A lógica pode ser aplicada de maneira prática na solução de problemas cotidianos, como identificar a idade da pessoa mais velha em um grupo de indivíduos organizados em uma fila. Esse problema exige uma sequência lógica de passos para alcançar a solução, seguindo os princípios de organização e análise sistemática.

**Problema:** Determinar a idade da pessoa mais velha em um grupo de indivíduos organizados em uma fila (Figura 7 - Fila de Pessoas), como em um banco ou cinema.



Figura 7 - Fila de Pessoas

Vamos elaborar uma estratégia para resolver este problema. Uma solução bem simples seria fazer o seguinte (Veja a seção a seguir - 22 Algoritmos e Implementação).

### 3.2 Algoritmos e Implementação

**Um algoritmo computacional é definido como uma sequência ordenada de passos que tem como objetivo resolver um problema específico ou atingir determinado resultado. Analogamente, pode-se pensar em um algoritmo como uma receita de bolo, onde cada etapa descreve o que deve ser feito para se obter o resultado desejado.**



Figura 8 - Receita (Fonte: obadenbaden.com.br)

Para solucionar um problema, assim como em qualquer receita, é necessário definir **como** os passos do algoritmo serão executados. Por exemplo, no problema apresentado anteriormente (Seção 3.1), que consiste em descobrir a maior idade entre um grupo de pessoas organizadas em uma fila, devemos determinar:

- **Como coletar as informações** sobre as idades das pessoas: isso pode ser feito perguntando diretamente a elas ou consultando registros disponíveis, como cadastros ou documentos.
- **Como armazenar as informações** coletadas: pode-se anotar os valores em um papel ou guardá-los em uma variável no computador.

A definição de como os passos de um algoritmo serão executados corresponde à **implementação do algoritmo**. Em resumo, o **algoritmo** especifica **o que deve ser feito**, enquanto a **implementação** descreve **como** as instruções do algoritmo serão efetivamente aplicadas.

### 3.2.1 Solução Simples para o Problema

A seguir, apresentamos uma solução algorítmica para determinar a maior idade entre as pessoas de uma fila:

#### 1. Inicialização:

- ✓ Pegue a idade da **primeira pessoa** da fila. Como esta é a única informação disponível até o momento, considere essa idade como sendo a maior e "armazene-a" em um local apropriado (por exemplo, uma variável).

#### 2. Iteração (para cada pessoa subsequente na fila):

- a. **Obtenha** a idade da pessoa atual.
- b. **Compare** a idade atual com a maior idade registrada até o momento. Esta comparação pode resultar em três situações:
  - i. A idade atual é **menor** que a maior registrada.
  - ii. A idade atual é **igual** à maior registrada.
  - iii. A idade atual é **maior** que a maior registrada.

- c. Se a idade atual for **maior**, atualize o valor da maior idade para ser igual à idade atual (CAELUM, 2011).

### 3.2.2 Características de um Algoritmo

Um algoritmo deve possuir as seguintes características:

1. **Sequência finita de passos:** A execução do algoritmo deve conter um número limitado de etapas.
2. **Reprodutibilidade:** Sempre que o algoritmo for executado com os mesmos dados de entrada, ele deve fornecer o mesmo resultado.
3. **Objetividade:** Cada etapa deve ser clara e não ambígua, garantindo que qualquer pessoa ou sistema que siga o algoritmo obtenha o resultado desejado.

Portanto, ao implementar um algoritmo como o descrito acima, garantimos uma solução lógica, sistemática e eficiente para resolver o problema proposto, independente do ambiente em que ele seja aplicado.

## 3.3 Representação dos Algoritmos

A representação de algoritmos é fundamental para facilitar a sua compreensão, desenvolvimento e implementação. Existem diversas formas de representar algoritmos, mas as duas mais amplamente utilizadas são: **Fluxogramas** e **Pseudocódigo**. Cada uma dessas formas possui características específicas que auxiliam na visualização e estruturação lógica do algoritmo, permitindo que ele seja compreendido de maneira mais clara tanto por programadores quanto por pessoas que não possuem conhecimentos avançados em programação.

### 3.3.1 Fluxogramas

O **fluxograma** é uma ferramenta visual utilizada para representar processos de maneira lógica e sequencial. Ele é frequentemente chamado de **diagrama de blocos**, pois utiliza **formas geométricas** padronizadas para ilustrar cada etapa do processo ou algoritmo. Essas formas geométricas são conectadas por setas, que indicam a direção do fluxo de execução.

O fluxograma facilita a interpretação e análise dos algoritmos, pois oferece uma **representação gráfica clara e objetiva** das ações que compõem o processo, bem como da relação entre elas. Isso o torna uma ferramenta didática amplamente adotada para o ensino de lógica de programação e desenvolvimento de sistemas.

### 3.3.2 Vantagens do Fluxograma

1. **Clareza Visual:** A representação gráfica facilita a identificação das etapas e decisões do algoritmo, tornando-o mais comprehensível.
2. **Identificação de Falhas:** Por ser visual, permite que erros ou redundâncias no processo sejam detectados com maior facilidade.
3. **Padronização:** O uso de símbolos padronizados permite que fluxogramas sejam compreendidos por profissionais de diversas áreas, facilitando a comunicação entre equipes.
4. **Versatilidade:** Pode ser utilizado em diferentes contextos, como modelagem de processos de negócio, documentação de sistemas e ensino de algoritmo.

Forma	Significado
	Terminador: Início e Fim do Fluxograma.
	Processo
	Entrada de Dados Manual
	Saída de Dados
	Decisão
	Conector
	Estrutura de Repetição PARA
	Sentido do fluxo

Tabela 1 - Principais formas usadas em fluxogramas

**Exemplo:**

Vamos supor que precisemos criar um algoritmo para somar dois valores (dados pelo usuário) e exibir o seu resultado.

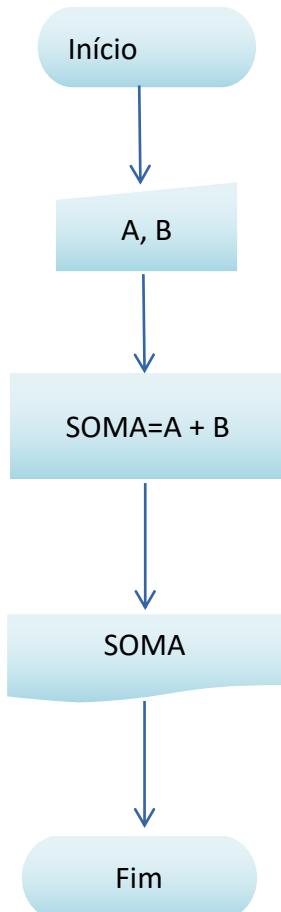


Figura 9 - Fluxograma

### 3.3.3 Importância dos Fluxogramas

Os fluxogramas desempenham um papel essencial no desenvolvimento e na documentação de algoritmos. Eles oferecem uma **visão clara e estruturada** das etapas necessárias para a resolução de um problema, permitindo que as instruções sejam organizadas de maneira lógica. Além disso, fluxogramas são amplamente utilizados na **engenharia de software**, na **modelagem de processos de negócios** e no **ensino de lógica de programação**, servindo como uma ponte entre a solução teórica de um problema e sua implementação prática.

**Conclusão:** O uso de fluxogramas como forma de representação de algoritmos promove uma abordagem visual eficiente, facilitando o entendimento das ideias subjacentes e a comunicação entre diferentes partes envolvidas no desenvolvimento do projeto.

### 3.3.4 Pseudocódigo

O pseudocódigo é uma forma textual de representação de algoritmos que combina simplicidade e clareza, assemelhando-se bastante à forma como os programas são escritos em linguagens de programação. Essa similaridade facilita sua aceitação entre programadores e estudantes, uma vez que permite uma transição quase direta para linguagens específicas de programação.

Ao contrário de linguagens formais, o pseudocódigo não segue uma sintaxe rígida, o que o torna flexível e adequado para descrever algoritmos de maneira intuitiva, porém com um nível de detalhe suficiente para sua implementação prática.

#### 3.3.4.1 Características do Pseudocódigo

- **Generalidade:** Sua estrutura é genérica, permitindo fácil adaptação para qualquer linguagem de programação.
- **Detalhamento:** Pode incluir elementos específicos, como a definição de variáveis e a criação de subalgoritmos.
- **Legibilidade:** Utiliza palavras e estruturas próximas à linguagem natural, facilitando o entendimento para pessoas que ainda estão aprendendo a programar.
- **Aceitação Amplia:** Por sua simplicidade e clareza, é amplamente aceito em ambientes educacionais e de documentação técnica.

### 3.3.5 Estrutura Geral de um Pseudocódigo

A estrutura básica para a representação de um algoritmo em pseudocódigo é a seguinte:

```

programa
{
    // Seção de declarações de variáveis (globais)
    // Exemplo:
    inteiro variavelGlobal1
    real variavelGlobal2

    // Função principal onde o programa inicia sua execução
    funcao inicio()
    {
        // Seção de declarações de variáveis locais, se necessário
        inteiro variavelLocal1
        real variavelLocal2

        // Bloco principal de instruções
        escreva("Olá, mundo!")
    }
}

```

```
// Exemplo de atribuição
variavelLocal1 = 10
variavelLocal2 = 3.5

// Exemplo de operação
escreva("A soma é: ", variavelLocal1 + variavelLocal2)
}
```

### Onde:

- **programa { ... }:** Define o escopo geral do algoritmo.
- **funcao inicio():** É o ponto de entrada do programa (função principal).
- **Declaração de Variáveis:**
  - Globais:** Podem ser declaradas logo após abrir o bloco programa, para que sejam acessíveis em todo o escopo.
  - Locais:** São declaradas dentro de funções (como inicio()), sendo acessíveis apenas nessas funções.
- **Comandos:** O corpo da função (funcao inicio()) contém a lógica que será executada, incluindo leitura de dados, escrita de resultados (com leia() e escreva()), uso de estruturas de controle (if, while, para, etc.) e assim por diante.

### Exemplo:

A soma de dois valores, agora representados em pseudocódigo.

```
programa
{
    // Declaração de variáveis globais
    inteiro valor1, valor2, soma

    // Função principal - ponto de entrada do programa
    funcao inicio()
    {
        escreva("Digite o primeiro valor: ")
        leia(valor1)

        escreva("Digite o segundo valor: ")
        leia(valor2)

        soma = valor1 + valor2

        escreva("A soma dos valores é: ", soma)
    }
}
```

### Explicação

- **Declaração de Variáveis:** As variáveis valor1, valor2 e soma são declaradas no escopo global, logo após abrir o bloco programa.

- **Função inicio():** É o ponto de entrada do programa em Portugol Studio. Nele acontecem:
  - a. **Entrada de dados:** O usuário digita dois valores, armazenados em valor1 e valor2.
  - b. **Processamento:** Soma dos valores ( $soma = valor1 + valor2$ ).
  - c. **Saída de dados:** Impressão do resultado pelo comando escreva.

A representação em pseudocódigo costuma ser simples e, na maioria dos casos, mostrase suficiente para descrever um algoritmo. Além disso, há vários interpretadores de Portugol, como o VisuAlg, e, em inglês, podem-se mencionar linguagens como C, Pascal e BASIC, cuja sintaxe se aproxima bastante de um pseudocódigo em inglês.

### 3.3.6 Importância do Pseudocódigo

O uso do pseudocódigo é uma prática essencial no **ensino de algoritmos** e no **planejamento de soluções computacionais**. Ele permite que programadores e estudantes se concentrem na **lógica** e na **estrutura** do algoritmo antes de se preocuparem com as particularidades sintáticas de uma linguagem específica.

Além disso, sua legibilidade o torna uma ferramenta eficiente para **documentação técnica e comunicação entre equipes**, facilitando a transposição de ideias complexas para linguagens de programação.

## 3.4 Exercício de Fixação

- 1) **Pesquise sobre as formas de representar algoritmos. O que é mencionado sobre a descrição narrativa?** Essa abordagem consiste em representar algoritmos utilizando textos descritivos em linguagem natural, detalhando cada etapa do processo de maneira sequencial. Embora seja uma forma intuitiva e de fácil compreensão para iniciantes, a descrição narrativa apresenta desvantagens, como a falta de padronização e a dificuldade de interpretação em algoritmos complexos. Assim, seu uso pode ser considerado interessante em situações simples, mas torna-se menos adequado para algoritmos maiores e mais estruturados, onde formas mais visuais ou formalizadas, como fluxogramas e pseudocódigo, são preferíveis.
- 2) Baixe o aplicativo **Portugol Studio** e execute o algoritmo apresentado acima. O **Portugol Studio** é uma ferramenta que permite criar e executar algoritmos em pseudocódigo na língua portuguesa de forma simples e

intuitiva. Para obter orientações detalhadas sobre como utilizar a ferramenta, consulte o *Capítulo 7 – Linguagem de Projeto de Programação*.

## 4 Tipos de Dados

Neste capítulo, apresentamos os conceitos fundamentais sobre tipos de dados, constantes e variáveis, além de explorar como ocorre a declaração e inicialização de variáveis em diferentes linguagens de programação, como **Java**, **C** e **Python**. Também discutimos as principais regras de declaração e as convenções amplamente adotadas no desenvolvimento de software, com foco em boas práticas que promovem organização e clareza no código.



**Figura 13 - Zebra (imagem construída em ASCII)**

**Nota sobre ASCII:** ASCII (acrônimo para American Standard Code for Information Interchange, que em português significa "Código Padrão Americano para o Intercâmbio de Informação") é uma codificação de caracteres de oito bits baseada no alfabeto inglês. Os códigos ASCII representam texto em computadores, equipamentos de comunicação, entre outros dispositivos que trabalham com texto. Desenvolvida a partir de 1960, grande parte das codificações de caracteres modernas a herdaram como base.

Os **dados** representam as informações processadas pelo computador. Alguns tipos de dados incluem: **numéricos** (inteiros e reais), **caracteres ou literais, lógicos, datas**, entre outros.

Os **tipos numéricos inteiros** são dados que representam números positivos, negativos e não fracionários. Exemplos: **100, -25, 0, 42**. Já os **tipos reais** são dados numéricos que incluem frações decimais, tanto positivos quanto negativos. Exemplos: **3.14, -12.5, 0.001, 25.75**.

Os **tipos caracteres** consistem em sequências de **letras, números e símbolos especiais**. Esse tipo de dado também é conhecido como **string, literal ou alfanumérico** e é sempre representado entre aspas. Exemplos: **"computador", "12345", "A@b#1!"**.

Os **tipos lógicos**, também chamados de **booleanos**, representam valores **binários: verdadeiro ou falso**. Nas linguagens de programação de alto nível, como Pascal, C/C++, Python e PHP, esses valores são representados pelos termos **true** e **false**, respectivamente.

**Nota:** Algumas linguagens tratam **datas** como um tipo especial. Para fins didáticos neste material, as datas serão representadas entre aspas, como se fossem do tipo **string**. Exemplos: "2024-06-01", "15/08/2023", "01-Jan-2025".

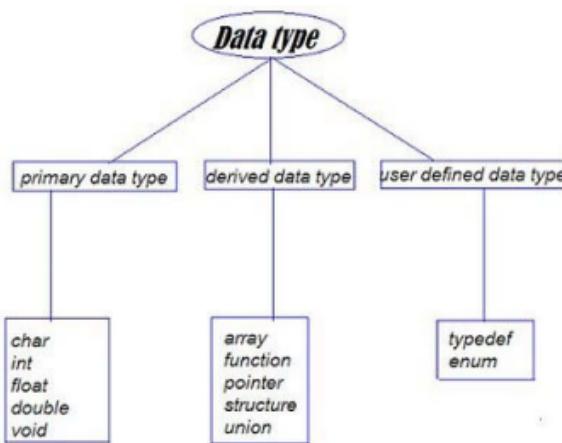


Figura 10 - Tipos de Dados

## 4.1 Constantes

**Constantes** são valores fixos que não podem ser alterados durante a execução de um programa. Elas são utilizadas em expressões para representar valores de diversos tipos, como inteiros, reais, caracteres ou booleanos. Em linguagens como **C** e **Java**, existem regras específicas e rígidas para a correta definição e uso de constantes. A seguir, apresentamos as principais regras para a escrita de constantes e exemplos para ilustrar esses conceitos.

### 4.1.1 Constantes Inteiras

As **constantes inteiras** representam valores numéricos sem ponto decimal, que podem ser positivos, negativos ou zero. Esses valores são escritos sem separação entre o **sinal** e o **valor numérico**.

**Exemplos válidos de constantes inteiras:**

- 2024
- -15
- +8
- 0
- -100000

**Exemplos de erros comuns na definição de constantes inteiras:**

- (*Erro: Não é permitido o uso de ponto decimal; trata-se de uma constante real, não inteira.*)
- **- 200** (*Erro: Não pode haver espaço entre o sinal e o valor numérico.*)
- **2E3** (*Erro: Notação de expoentes não é permitida para inteiros; somente em constantes reais.*)

#### 4.1.2 Constantes Reais

As **constantes reais** representam valores numéricos que possuem uma **parte fracionária**, separada da parte inteira por um ponto decimal. Elas podem ser escritas com ou sem sinal e também permitem a notação científica para representar valores muito grandes ou muito pequenos.

**Exemplos válidos de constantes reais:**

- 3.14
- -0.005
- +123.456
- **2.5E3** (*Equivale a  $2.5 \times 10^3 = 2500$* )
- **-1.2E-4** (*Equivale a  $-1.2 \times 10^{-4} = -0.00012$* )

**Exemplos de erros comuns:**

- **2,5** (*Erro: Utilizar vírgula em vez de ponto decimal.*)
- **1E** (*Erro: Exponencial incompleto; é necessário especificar o expoente.*)

## 4.2 Variáveis

Em computação, uma **variável** representa um **endereço da memória RAM** onde podemos armazenar e manipular dados de diversos tipos, como **inteiros**, **reais**, **strings** (texto), **booleanos** (verdadeiro ou falso) e até mesmo **objetos complexos**.

Ao **declarar uma variável**, atribuímos um **nome simbólico** a um endereço específico da memória RAM. Esse nome será utilizado no programa para acessar ou modificar as informações armazenadas nesse local. Por exemplo:

- Nome da variável: `idade`
- Valor armazenado: `25`
- Local na memória: Endereço hexadecimal, como `0x1A3B` (esse endereço é gerenciado automaticamente pelo compilador).

### 4.2.1 Funcionamento da Memória e Variáveis

Fisicamente, a **memória RAM** é organizada em posições consecutivas, cada uma com um **endereço único** (normalmente representado em hexadecimal). Quando utilizamos variáveis em um programa, o compilador associa o nome da variável a um endereço específico, facilitando a manipulação dos dados.

Por exemplo:

Endereço	Nome da Variável	Valor Armazenado
0x1A3B	idade	25
0x1A3C	altura	1.75

### 4.2.2 Uso de Variáveis em Java, C e Python

Para utilizar variáveis em linguagens como **Java**, **C** e **Python**, é necessário seguir duas etapas principais: **declaração** e **inicialização**.

## Declaração

Na declaração, informamos ao compilador (ou interpretador, no caso de Python) o tipo de dado que a variável irá armazenar e o seu nome simbólico.

### 4.2.2.1 Exemplo em Java

```
int idade; // Declara uma variável do tipo inteiro
double altura; // Declara uma variável do tipo real
String nome; // Declara uma variável do tipo string
```

### 4.2.2.2 Exemplo em C

```
int idade; // Declara uma variável do tipo inteiro
double altura; // Declara uma variável do tipo real
char nome[51]; // Declara um array de 51 caracteres (string em C)
```

### 4.2.2.3 Exemplo em Python

```
idade = 0 # Em Python, não declaramos tipos explicitamente
altura = 0.0 # Atribuir valores determina o tipo
nome = "" # Strings são declaradas entre aspas
```

## Inicialização

Na inicialização, atribuímos um valor inicial à variável.

### 4.2.2.4 Exemplo em Java

```
idade = 25;
altura = 1.75;
nome = "João";
```

### 4.2.2.5 Exemplo em C

```
idade = 25;
altura = 1.75;
strcpy(nome, "Maromo"); // Usa a função strcpy para copiar a string para o array nome
```

#### 4.2.2.6 Exemplo em Python

```
idade = 25
altura = 1.75
nome = "João"
```

#### Declaração e Inicialização Simultânea

É possível realizar as duas etapas ao mesmo tempo.

#### 4.2.2.7 Exemplo em Java

```
int idade = 25;
double altura = 1.75;
String nome = "João";
```

#### 4.2.2.8 Exemplo em C

```
int idade = 25;
double altura = 1.75;
char nome[51] = "João";
```

#### 4.2.2.9 Exemplo em Python

```
idade = 25
altura = 1.75
nome = "João"
```

#### Resumo

- **Constantes** representam valores fixos que não podem ser alterados durante a execução do programa. Existem constantes inteiras (sem ponto decimal) e reais (com ponto decimal e notação científica).
- **Variáveis** representam endereços de memória que armazenam dados manipuláveis pelo programa. O uso correto envolve declaração (tipo e nome) e inicialização (atribuição de valores).
- Linguagens como **Java** e **C** exigem a definição explícita do tipo de dado e seguem uma sintaxe mais rígida, enquanto **Python** permite a inferência de tipos dinamicamente, proporcionando maior flexibilidade, mas exigindo atenção especial à coerência dos tipos em tempo de execução.

## 4.3 Tipos de variáveis

Nas linguagens **Java** (ou até mesmo em linguagens como **C**, que também utilizam tipagem estática), cada variável deve obrigatoriamente possuir um tipo específico. Esse tipo determina a categoria de dados que a variável será capaz de armazenar, bem como as operações que poderão ser aplicadas a ela. Por esse motivo, costuma-se dizer apenas: “Qual é o tipo da variável?”, quando se deseja saber qual o tipo de dado associado à variável em questão.

Existem diversos tipos de dados em linguagens de programação, mas os mais básicos são conhecidos como **tipos primitivos**, pois são pré-definidos pela linguagem. Em Java, por exemplo, esses tipos incluem `int` (inteiro), `double` (ponto flutuante), `char` (caráter) e `boolean` (lógico). A vantagem do uso de tipos primitivos é a eficiência no acesso e na manipulação dos dados, pois eles são suportados de forma direta pela máquina virtual ou pelo compilador.

Além disso, a definição explícita do tipo de cada variável, presente em linguagens de tipagem estática como Java, permite ao compilador identificar inconsistências de tipo ainda em tempo de compilação, evitando problemas em tempo de execução e tornando o código mais robusto.

### 4.3.1 Tipos primitivos em C

#### 1) `char`

- Deve ter **pelo menos** 8 bits (1 byte).
- Pode ser signed ou unsigned por padrão, depende da implementação.

#### 2) `short int`

- Deve ter **pelo menos** 16 bits.
- Intervalo mínimo (se signed): -32767 a 32767
- `sizeof(short)` **não** pode ser maior que `sizeof(int)`.

#### 3) `int`

- Deve ter **pelo menos** o mesmo tamanho que short ou maior.
- Deve ter **pelo menos** 16 bits.
- Intervalo mínimo (se signed): -32767 a 32767.
- Na prática, compiladores de 32 bits ou 64 bits costumam adotar 32 bits para int.

#### 4) **long int**

- Deve ter **pelo menos** 32 bits.
- Intervalo mínimo (se signed): -2147483647 a 2147483647
- `sizeof(long)` **não** pode ser menor que `sizeof(int)`.

#### 5) **float**

- Normalmente segue o padrão **IEEE 754** de 32 bits em implementações modernas, mas a norma C não impõe tamanho exato (apenas requisitos de precisão).

#### 6) **double**

- Normalmente segue o padrão **IEEE 754** de 64 bits, mas, de novo, a norma apenas define requisitos mínimos de precisão (deve ser no mínimo tão preciso quanto float).

### 4.3.2 Tipos primitivos em Java

O Java define oito tipos primitivos, sendo quatro deles destinados à representação de valores inteiros: **byte**, **short**, **int** e **long**. A principal diferença entre esses tipos está no tamanho (em bytes) que cada um ocupa, o que determina o intervalo de valores que podem armazenar. Todos permitem representar números inteiros tanto positivos quanto negativos.

Além disso, há dois tipos voltados para a representação de números reais (ponto flutuante): **float** e **double**. O que distingue um do outro é a quantidade de bytes utilizada, o que afeta a precisão e o alcance dos valores possíveis. Ambos seguem o padrão **IEEE 754** para a formação de números em ponto flutuante.

Por fim, os dois últimos tipos primitivos são **char** e **boolean**. O tipo **char** serve para representar caracteres e, em termos de código, pode ser interpretado como um valor

numérico inteiro positivo (baseado na tabela Unicode). Já o tipo **boolean** aceita apenas dois valores lógicos: **true** ou **false**.

Tipo	Faixa de dados
byte (8bits)	-128 a 127
short (16bits)	-32768 a 32767
int (32bits)	-2147483648 a 2147483647
long (64bits)	-9223372036854775808 a 9223372036854775807
float (32 bits)	single-precision 32-bit IEEE 754 floating point
double (64 bits)	double-precision 64-bit IEEE 754 floating point
char (16 bits)	'\u0000' (ou 0) a '\uffff' (ou 65.535 inclusive)
Boolean	Assume true ou false (tamanho não definido)

#### 4.4 Convenção para a nomenclatura

Tanto em **Java** como em **C**, há uma convenção para a escrita dos nomes das variáveis, seguindo o padrão **Camel Case**. Nesse padrão, o nome da variável é escrito com a primeira letra de cada palavra em maiúscula, exceto a primeira, que permanece em minúscula.

Também é importante lembrar que ambas as linguagens são **Case Sensitive**, ou seja, fazem distinção entre letras maiúsculas e minúsculas. Assim, por exemplo, as variáveis `numeroDaConta` e `NumeroDaConta` seriam consideradas diferentes, pois a primeira começa com letra minúscula e a segunda, com maiúscula.

#### 4.5 Regras para a nomenclatura

As linguagens **Java** e **C** possuem regras muito semelhantes. Ao definir o nome de uma variável, devemos observar:

- 1) Não deve começar com um dígito;
- 2) Não pode ser igual a uma palavra reservada;
- 3) Não pode conter espaços em branco;
- 4) Pode ter qualquer tamanho;
- 5) Pode conter letras, dígitos e o caractere `_` (underscore). Em Java, pode conter também o caractere `$`.

Ambas as linguagens nos permitem escrever nomes de variáveis em qualquer idioma, pois aceitam caracteres Unicode. Portanto, são válidos nomes escritos com acentuação em português ou até em japonês, por exemplo:

```
int númeroDaConta;  
int アカウント番号;
```

Figura 19 - Exemplo de nomes de variáveis válidos utilizando caracteres especiais em outros idiomas.

Apesar de ser possível o uso de caracteres em outros idiomas, bem como os caracteres \$ (cifrão) e \_ (underscore), não é recomendável empregá-los. Evitar tais caracteres faz parte das boas práticas de programação, pois torna o código mais claro e padronizado.

## 4.6 Exercício de Fixação

1. Indique os nomes de variáveis que são válidos. Justifique os nomes inválidos.
  - a) tempo
  - b) nota\_final
  - c) us\$
  - d) char
  - e) 2dias
  - f) teste\_1
  - g) raio.do.circulo
  
2. Indique quais dos números abaixo são constantes inteiras (longas ou não) válidas. Justifique suas respostas.
  - a) 100
  - b) 2 345 123
  - c) 3.0
  - d) -35
  - e) - 234
  - f) 0L
  - g) 21

## 5 Operadores

Nas diferentes linguagens de programação utilizamos os operadores para manipularmos as variáveis de nossa aplicação. As linguagens Java e C possuem diversos operadores que são categorizados da seguinte forma:

- Aritmético (+, -, \*, /, %)
- Atribuição (=, +=, -=, \*=, /=, %=)
- Relacional (==, !=, <, <=, >, >=)
- Lógico (&&, ||)

Veja a seguir uma lista de operadores matemáticos básicos, listados em ordem de precedência. Use parênteses para outra ordem.

Operador	Objetivo
*, /, %	Multiplicação, Divisão, Módulo
+, -	Adição, subtração

### Exemplo

```
int x = 1;
int y = x + 10 * 100;      // multiplicação primeiro y = 1001
int z = (x + 10) * 100;    // adição primeiro z = 1100
```

Figura 11 - Operadores Matemáticos

Lista de operadores de atribuição e Incremento:

Operador	Objetivo
=	Atribuir valor
v++	Incremento variável v por 1.
v+=n	Incremento variável v por n.
v*=n	Multiplique variável v por n.
v-=n	Subtrair n da variável v.

Veja lista de operadores relacionais:

Operador	Objetivo
==	Verifica a igualdade.
!=	Verifica a desigualdade.
>	Mais de.
<	Menor que.
>=	Maior que ou igual a.
<=	Menor ou igual a.

```
public class ExemploIncremento {
    public static void main(String[] args) {
        int x = 0;
        int y = x++;
        // Pós-incremento: y recebe 0, depois x passa a valer 1

        System.out.println(x + " " + y);
        // Saída: x = 1, y = 0

        int z = ++x;
        // Pré-incremento: x passa a valer 2, depois z recebe 2

        System.out.println(x + " " + z);
        // Saída: x = 2, z = 2
    }
}
```

Tabela 2 - Exemplo de uso de operador de verificação de igualdade em Java

Veja a lista dos operadores lógicos:

Operador	Objetivo
&&	And condicional.
	OR condicional.
!	NOT condicional.

```
import java.util.Scanner;

public class ExemploVerificaIntervalo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Digite um valor inteiro: ");
        int x = scanner.nextInt();

        if ((x >= 100) && (x <= 200)) {
            System.out.println("X está entre 100 e 200");
        }

        scanner.close();
    }
}
```

Tabela 3 - Exemplo de uso do operador && em Java

## 5.1 Tabela Verdade

Os autores da K19 [K19, 2011] apresentam a tabela verdade com uma tabela matemática muito utilizada na Álgebra Booleana e faremos o uso dela para compreendermos melhor os operadores lógicos. Sendo A e B duas variáveis booleanas, confira como fica a tabela verdade para os operadores lógicos “E” (`&&`) e “OU” (`||`):

A	B	A e B	A ou B
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

V: Verdadeiro (true)  
 F: Falso (false)

Figura 12 - Exemplo de Tabela da Verdade

Rapidamente notamos que a operação “E” devolve `true` apenas quando A e B são `true`. Também notamos que a operação “OU” devolve `false` apenas quando A e B são `false`.

Vamos utilizar os exemplos de operadores lógicos dados anteriormente para ilustrarmos melhor como funciona a tabela verdade.

### Exemplo 1:

Seja A=5, B=3, C=2. Calcule o resultado da expressão lógica:

1)  $A + 7 > B * C$

Sendo A = 5

Então  $5 + 7 = 12$

E

$3 * 2 = 6$

Então a expressão é  $12 > 6$ , ou seja, o resultado é .verdadeiro. (Verdadeiro)

### Exemplo 2:

Suponha que temos três variáveis A = 5, B = 8 e C = 1, os resultados das expressões seriam:

Expressão	Operador	Operador em C	Expressão 2	Resultado
A = B	AND	&&	B > C	.falso.
A <> B	OR		B < C	.verdeiro.
A > B	NOT	!		.verdeiro.
A < B	AND	&&	B > C	.verdeiro.
A >= B	OR		B = C	.falso.
A <= B	NOT	!		.falso.

Operações Lógicas são utilizadas quando se torna necessário tomar decisões em um fluxograma ou diagrama de bloco. Num fluxograma, toda decisão terá sempre como resposta o resultado .verdeiro. ou .falso. (*true* ou *false*)

Uma estrutura de decisão ou desvio faz parte das técnicas de programação que conduzem a estruturas de programas que não são totalmente sequenciais. Com as instruções de SALTO ou DESVIO pode-se fazer com que o programa proceda de uma ou outra maneira, de acordo com as decisões lógicas tomadas em função dos dados ou resultados anteriores.

## 5.2 Exercícios de Fixação

### 1. Estoque Médio

Elabore um fluxograma e um programa que calculem o estoque médio de uma peça. Utilize a fórmula: **ESTOQUE MÉDIO = (QUANTIDADE MÍNIMA + QUANTIDADE MÁXIMA) / 2.**

### 2. Cálculo da Idade

Crie um fluxograma e um programa que calculem a idade aproximada de uma pessoa, considerando a seguinte fórmula: **IDADE = ANO ATUAL - ANO DE NASCIMENTO.**

### 3. Cálculo de Expressão (Quadrado de X)

Desenvolva um fluxograma e um programa que recebam um número como entrada e calculem o resultado da expressão: **X := X<sup>2</sup> + 2.**

### 4. Cálculo de Expressão com Multiplicação

Construa um fluxograma e um programa que leiam três valores (A, B e C) e calculem a seguinte expressão: **A := A + B \* C.**

## 5. Média Ponderada

Desenvolva um fluxograma e um programa que calculem a média ponderada de duas notas de um aluno. Considere os seguintes pesos:

- Nota 1: Peso 3.
- Nota 2: Peso 7.
- Fórmula:  $\text{MÉDIA} = (\text{NOTA1} * 3 + \text{NOTA2} * 7) / 10$ .

## 6. Classificação de Tipos de Dados

Analise os dados abaixo e classifique-os nos seguintes tipos:

- **I** (Inteiro).
- **R** (Real).
- **B** (Booleano).
- **L** (Literal).

**Dados:**

- a) 100
- b) 34.34
- c) 12.12
- d) "0"
- e) -34
- f) -9
- g) "10/11/09"
- h) "Maria"
- i) 0
- j) .verdadeiro.
- k) .falso.
- l) "
- m) 34
- n) -1.45
- o) "-30.3"

## 7. Conversão de Moeda

Desenvolva um fluxograma e um programa que:

- 1 Leiam a cotação atual do dólar.
- 2 Leiam um valor em dólares.
- 3 Converta o valor para reais.
- 4 Exiba o resultado da conversão.

## 8. Soma dos Quadrados

Crie um fluxograma e um programa que:

- 1 Leiam quatro números.

- 2 Calculem o quadrado de cada número.
- 3 Somem os quadrados.
- 4 Mostrem o resultado.

## 9. Cálculo de Comissão de Vendas

Elabore um algoritmo para calcular a comissão de vendedores, considerando:

A comissão será de 5% do total das vendas.

O programa deve receber:

- Identificação do vendedor.
- Código da peça.
- Preço unitário da peça.
- Quantidade vendida.

Exiba a comissão calculada.

## 10. Cálculo do Índice de Massa Corporal (IMC)

Desenvolva um algoritmo que:

- 1 Leia o peso de uma pessoa (em quilogramas).
- 2 Leia a altura da pessoa (em centímetros).
- 3 Calcule o Índice de Massa Corporal (IMC) utilizando a fórmula:  
**IMC = PESO / (ALTURA EM METROS)<sup>2</sup>.**
- 4 Exiba o valor calculado do IMC.

## 6 Controle de Fluxo

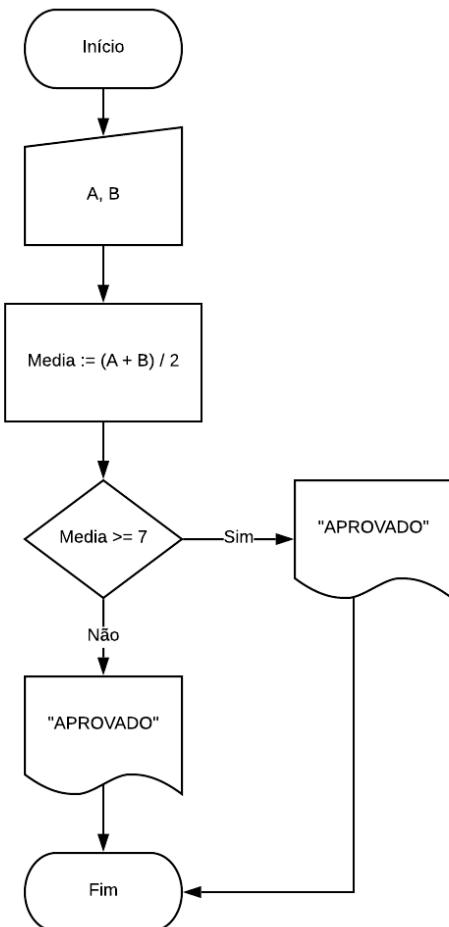
Conforme discutido até agora, entendemos o que é um programa e como ele funciona, assim como o modo de armazenamento dos dados na memória por meio de variáveis. Também abordamos a forma de realizar operações de diversos tipos utilizando os operadores fornecidos pelas linguagens de programação (DEITEL; DEITEL, 2016).

Neste capítulo, examinaremos as **instruções de controle de fluxo**, que incluem tanto as **instruções de decisão** quanto as **estruturas de repetição**. Segundo Kernighan e Ritchie (1988), essas estruturas permitem definir lógicas mais complexas dentro de um programa, direcionando o caminho de execução de acordo com condições estabelecidas ou repetições necessárias.

Entender o funcionamento adequado das instruções de decisão (por exemplo, `if`, `if-else`, `switch`) e de repetição (por exemplo, `for`, `while`, `do-while`) é crucial para desenvolver programas eficientes e corretos. Como afirmam Manzano e Oliveira (2012), essas construções tornam possível o desenvolvimento de algoritmos cada vez mais robustos, permitindo lidar com situações variadas em que o fluxo de execução depende de fatores como o valor de uma variável, o resultado de uma operação ou a necessidade de repetir um determinado bloco de código diversas vezes.

### 6.1 Comandos de Decisão

Por meio de um fluxograma que calcula a média de duas notas de um aluno, determina-se se ele foi aprovado ou não. A condição estabelecida é que, para ser aprovado, o aluno deverá atingir média **maior ou igual a sete**.



## 6.2 Estrutura SE ENTÃO SENÃO / IF... THEN... ELSE

A estrutura de decisão “SE/IF” normalmente vem acompanhada de um comando, ou seja, se determinada condição for satisfeita pelo comando SE/IF então execute determinado comando. Imagine um algoritmo que determinado aluno somente estará aprovado se sua média for maior ou igual a 7.0, como visto no fluxograma da **Erro! Fonte de referência não encontrada..**

```

SE MEDIA >= 7.0 ENTÃO
  APROVADO
SENÃO
  REPROVADO
  
```

**Veja o código na linguagem C.**

```

#include <stdio.h>
int main(){
  double n1, n2, media;
  
```

```

printf("Digite a primeira nota: ");
scanf("%lf", &n1);

printf("Digite a segunda nota: ");
scanf("%lf", &n2);

media = (n1 + n2) / 2;

printf("Média do aluno: %.2f\n", media);

if(media >= 7) {
    printf("Aprovado\n");
} else {
    printf("Reprovado\n");
}

return 0;
}

```

**Tabela 4 - Exemplo If/else**

### Explicação

- Incluímos stdio.h para usar funções de entrada e saída (printf, scanf).
- Declaramos as variáveis n1, n2 e media do tipo double.
- Lemos as notas do teclado com scanf.
- Calculamos a média e mostramos na tela.
- Verificamos se a média é maior ou igual a 7 para exibir “Aprovado” ou “Reprovado”.

### Veja na Linguagem Java

```

import java.util.Scanner;

public class MediaAluno {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        double n1, n2, media;
        System.out.print("Digite a primeira nota: ");
        n1 = scanner.nextDouble();

        System.out.print("Digite a segunda nota: ");
        n2 = scanner.nextDouble();

        media = (n1 + n2) / 2.0;

        System.out.println("Média do aluno: " + media);

        if(media >= 7) {
            System.out.println("Aprovado");
        } else {
            System.out.println("Reprovado");
        }

        scanner.close();
    }
}

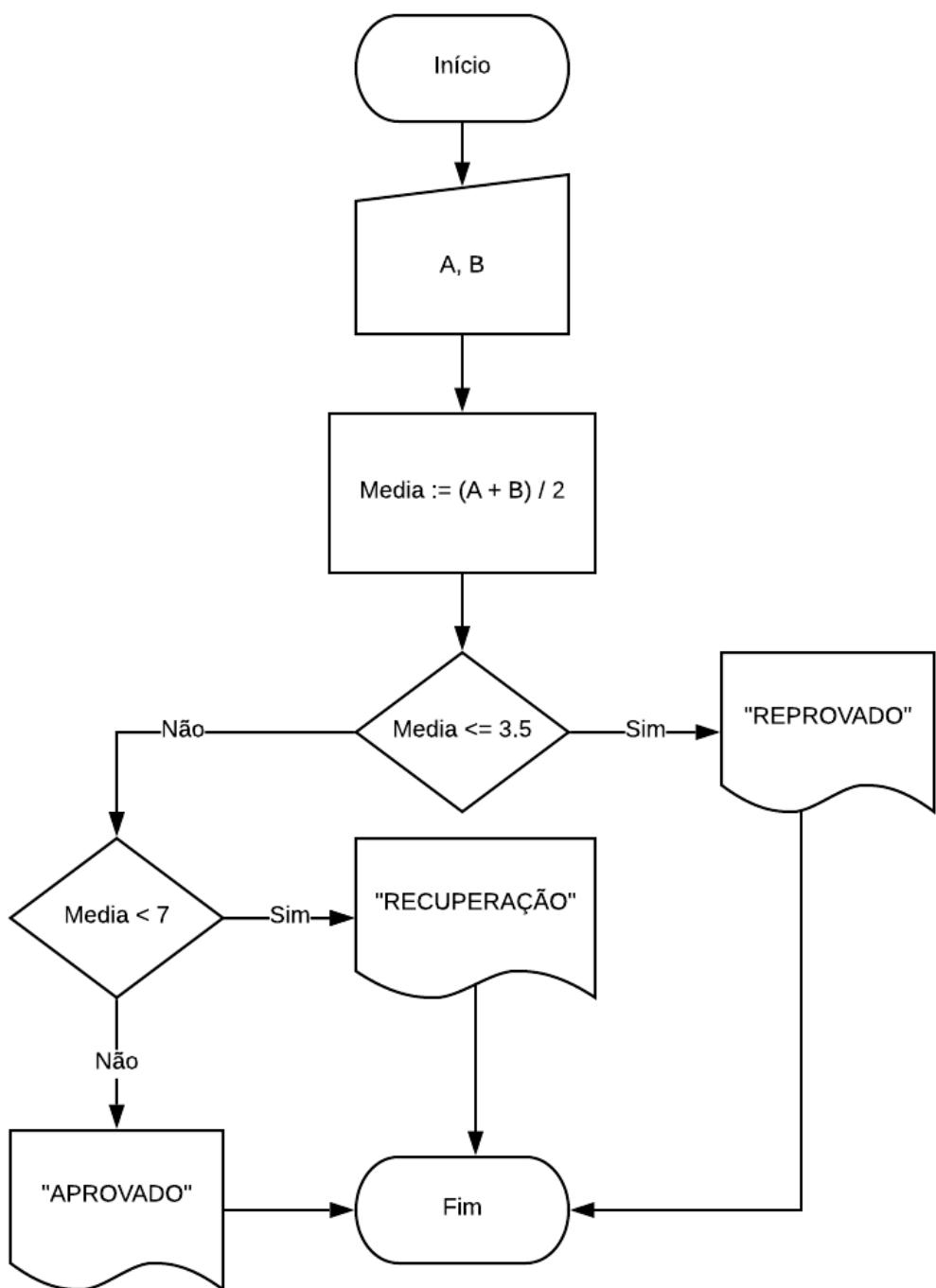
```

## Explicação

- Utilizamos a classe Scanner para ler valores do console.
- Declaramos as variáveis n1, n2 e media como double.
- Calculamos a média e a exibimos.
- Verificamos se a média é maior ou igual a 7 para imprimir “Aprovado” ou “Reprovado”.

Podemos também, dentro de uma mesma estrutura condicional, testar condições adicionais. Suponha que agora desejamos introduzir uma nova categoria de **RECUPERAÇÃO**. Nesse caso:

- Se a média for **menor ou igual a 3,5**, o aluno estará **REPROVADO**;
- Se a média estiver **acima de 3,5 e abaixo de 7**, o aluno estará em **RECUPERAÇÃO**;
- Se a média for **maior ou igual a 7**, sua situação será de **APROVADO**.



## Exemplo em Java

```

import java.util.Scanner;

public class ExemploMedia {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Digite a primeira nota: ");
        double nota1 = scanner.nextDouble();

        System.out.print("Digite a segunda nota: ");
        double nota2 = scanner.nextDouble();

        double media = (nota1 + nota2) / 2.0;
        System.out.println("Média do aluno: " + media);

        if(media <= 3.5){
            System.out.println("REPROVADO");
        } else if(media < 7.0){
            System.out.println("RECUPERAÇÃO");
        } else{
            System.out.println("APROVADO");
        }

        scanner.close();
    }
}

```

Tabela 5 - Cadeia de Ifs em Java

## 6.3 Exercícios de Fixação

### 1. Verificar Divisibilidade

Elabore um algoritmo que receba dois valores numéricos, **a** e **b**, e verifique se **a** é divisível por **b**. Exiba a mensagem:

- "É divisível", caso a divisão seja exata.
- "Não é divisível", caso contrário.

### 2. Saudação com Base na Hora

Elabore um algoritmo que leia a hora atual e exiba uma das seguintes mensagens, conforme o intervalo de tempo:

- "Bom dia", para horários entre 6h e 12h.
- "Boa tarde", para horários entre 12h e 18h.
- "Boa noite", para horários entre 18h e 6h.

A hora deve ser informada pelo usuário.

### 3. Classificação de Números em Variáveis

Elabore um algoritmo que leia um número:

- Caso seja positivo, armazene-o na variável **A**.
- Caso seja negativo, armazene-o na variável **B**.

#### 4. Verificação de Número Par ou Ímpar

Elabore um algoritmo que leia um número e determine sua paridade:

Caso seja par, armazene-o na variável **P**.

Caso seja ímpar, armazene-o na variável **I**.

#### 5. Condição para Exibição de Números

Elabore um algoritmo que leia uma variável numérica **N** e execute o seguinte:

Exiba o valor de **N**, caso seja maior que 100.

Caso contrário, exiba o valor zero.

#### 6. Cálculo do Peso Ideal

Elabore um algoritmo que leia a altura (**h**) e o sexo (**M** para masculino, **F** para feminino) de uma pessoa. O algoritmo deve calcular e exibir o peso ideal utilizando as fórmulas:

- Para homens: **Peso Ideal = (72.7 \* h) - 58.**
- Para mulheres: **Peso Ideal = (62.1 \* h) - 44.7.**

#### 7. Identificação do Maior Número

Elabore um algoritmo que leia três números distintos e exiba o maior deles.

#### 8. Ordenação de Números

Elabore um algoritmo que leia três números distintos e os exiba em ordem crescente.

#### 9. Cálculo de Salário Anual

Elabore um algoritmo que leia o número total de horas normais e o número total de horas extras trabalhadas em um ano. Considere:

**R\$ 29,00** para cada hora normal.

**R\$ 35,00** para cada hora extra.

Calcule e exiba o salário total recebido.

#### 10. Conversão de Temperatura

Elabore um algoritmo que leia uma temperatura em graus Celsius e a converta para Fahrenheit, utilizando a fórmula:  $F = (C * 9/5) + 32$ . Exiba o resultado da conversão.

## 6.4 Estruturas de Repetição

Em **Tecnologia da Informação (TI)**, uma **estrutura de repetição** é um recurso de desvio de fluxo de controle presente em linguagens de programação. Ela possibilita repetir a execução de determinadas instruções, desde que uma condição específica — por vezes chamada de "expressão de controle" ou "condição de parada" — seja **verdadeira**. Essa condição é processada e avaliada como um valor booleano (`true` ou `false`).

Sempre que a condição for verdadeira, o bloco de código associado à estrutura de repetição será executado. Ao término da execução desse bloco, a condição é novamente verificada, e, se ainda se mantiver verdadeira, o mesmo bloco volta a ser executado. Esse ciclo de verificação e execução continua até que a condição retorne **falso**, momento em que a estrutura de repetição é finalizada.

É importante observar que, se o bloco de código jamais modificar o estado da condição, a estrutura será executada indefinidamente, o que caracteriza um **laço infinito**. De modo semelhante, é possível que a própria condição seja sempre verdadeira, acarretando igualmente um loop infinito, pois o critério de parada não se concretiza em nenhum momento.

Algumas linguagens de programação especificam ainda uma palavra reservada para sair da estrutura de repetição de dentro do bloco de código, "quebrando" a estrutura. Também é oferecida por algumas linguagens uma palavra reservada para terminar uma iteração específica do bloco de código, forçando nova verificação da condição.

### 6.4.1 Tipos ou modelos de repetição

1. Repetição pré-testada;
2. Repetição pós-testada;
3. Repetição com variável de controle e;
4. Iteração de coleção.

#### **6.4.1.1 Repetição pré-testada**

A construção "enquanto" (também chamada "repetição pré-testada") é a mais difundida estrutura de repetição, e sua estrutura básica é a seguinte:

```
Enquanto (condição) Faça
    (bloco de código)
Fim Enquanto
```

#### **6.4.1.2 Repetição pós-testada**

A construção "repita enquanto" (também chamada "repetição pós-testada") é uma variação da construção anterior, e difere, pois a verificação da condição é feita após uma execução do bloco. Sua estrutura básica é a seguinte:

```
Repita
    (bloco de código)
Enquanto (condição)
```

#### **6.4.1.3 Repetição com variável de controle**

A construção "para" (ou "repetição com variável de controle") é uma estrutura de repetição que designa uma variável de controle para cada iteração do bloco, e uma operação de passo a cada iteração. Sua estrutura básica é a seguinte:

```
Para (V) De (vi) Até (vf) Passo (p) Faça
    (bloco de código)
Fim Para
```

Na construção acima, V é uma variável de controle, vi é o estado inicial de V e vf é o estado de saída da estrutura de repetição. O passo indica qual será o incremento entre cada iteração do código. No início da estrutura, vi é atribuído à V, e é verificado se V é igual a vf, a condição de parada. Caso não seja, o bloco de código é executado e então o passo é adicionado à V ( $V = V + p$ ). Segue-se então com nova verificação da condição de parada. O passo é opcional, e caso seja omitido assume-se incremento de uma unidade.

Esta estrutura é bastante utilizada para a iteração de vetores, em que cada iteração representa um índice do vetor. Nesse caso, para vetores multidimensionais é possível

aninhar este tipo de construção para as diversas dimensões associadas. Por exemplo, para uma estrutura bidimensional como uma matriz, que possui linhas e colunas, a estrutura é exemplificada abaixo:

```
Para (V) De (vi) Até (vf) Faça
    Para (Y) De (yi) Até (yf) Faça
        (bloco de código)
    Fim Para
Fim Para
```

#### **6.4.1.4 Iteração de coleção**

A estrutura "para cada" é usada para iterar itens de uma coleção, sendo uma especialização da estrutura "para". Menos flexível que a estrutura "para", esta estrutura torna implícita a atribuição inicial e o incremento do passo, e determina que a condição de parada é somente a situação na qual todos os elementos do conjunto já foram iterados.

Sua estrutura básica é:

```
Para Cada (item) De (conjunto) Faça
    (bloco de código)
Fim Para
```

Considere o seguinte problema: “Elaborar um diagrama de blocos de um programa que calcule o fatorial de um número **maior que 0** (zero) informado pelo usuário.” Recorde-se de que o fatorial de um número nnn (denotado por  $n!n!n!$ ) corresponde ao produto de todos os inteiros positivos de 1 até nnn. Por exemplo, se o usuário informar o número 5, temos:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 \quad \text{ou} \quad 5! = 1 \times 2 \times 3 \times 4 \times 5.$$

```
import java.util.Scanner;

public class Fatorial {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Digite um número inteiro maior que zero:");
        int numero = scanner.nextInt();

        if(numero <= 0) {
            System.out.println("Valor inválido! O número deve ser maior que zero.");
        }
    }
}
```

```

} else{
    long fatorial = 1;
    for (int i = 1; i <= numero; i++) {
        fatorial *= i;// Multiplicando sucessivamente
    }
    System.out.println("O fatorial de " + numero + " é: " + fatorial);
}

scanner.close();
}
}

```

Tabela 6 - Código Fatorial em Java

### Explicação do Código

- Leitura de Dados:** O programa solicita ao usuário que digite um número inteiro maior que zero.
- Validação:** Se o valor for menor ou igual a zero, o programa exibe uma mensagem de erro e encerra.

### Cálculo do Fatorial:

- A variável fatorial inicia em 1.
- Para cada valor de i de 1 até numero, fazemos fatorial \*= i.
- Ao final do laço, fatorial conterá o resultado de  $n!$
- Exibição do Resultado:** O programa exibe o valor do fatorial calculado.

## 6.5 Exercícios de Fixação

### 1. Determinar o Maior Número

Elabore um algoritmo que determine o maior valor entre vários números fornecidos. A condição de parada é a entrada do número **0**. O algoritmo deve continuar a execução até que o usuário insira o valor zero.

### 2. Grãos no Tabuleiro de Xadrez

Elabore um algoritmo que calcule a quantidade total de grãos de trigo que seriam necessários para preencher um tabuleiro de xadrez, conforme a seguinte regra:

- O primeiro quadro recebe **1 grão**.
  - Cada quadro subsequente recebe o dobro do número de grãos do quadro anterior.
- Exiba o total esperado de grãos.

### 3. Contagem com Múltiplos de 10

Elabore um algoritmo que conte de 1 a 100 e exiba uma mensagem "**Múltiplo de 10**" sempre que encontrar um número que seja múltiplo de 10.

#### **4. Números Ímpares entre 100 e 200**

Elabore um algoritmo que gere e exiba todos os números ímpares entre 100 e 200.

#### **5. Análise de 500 Valores**

Elabore um algoritmo que leia **500 números** fornecidos pelo usuário e realize as seguintes operações:

- Encontre o maior valor.
- Encontre o menor valor.
- Calcule a média dos valores fornecidos.

#### **6. Múltiplos de 5**

Elabore um algoritmo que leia diversos números fornecidos pelo usuário e exiba quantos deles são múltiplos de 5.

#### **7. Verificação de Número Primo**

Elabore um algoritmo que receba um número fornecido pelo usuário e determine se ele é primo ou não.

#### **8. Animal Mais Velho**

Elabore um algoritmo que leia a **idade** e a **espécie** de cada animal em um circo. Ao final, o algoritmo deve exibir a idade e a espécie do animal mais velho.

- O algoritmo deve perguntar ao usuário se deseja continuar inserindo dados.
- O fluxo termina quando o usuário digitar "n".

#### **9. Múltiplos de 6 entre Números Pares**

Elabore um algoritmo que identifique e exiba todos os números pares entre **1 e 168** que também sejam múltiplos de 6. O algoritmo deve listar cada número no resultado e exibir a quantidade total de números encontrados.

#### **10. Análise de Pessoas em uma Sala**

Elabore um algoritmo que leia o **Nome**, a **Idade** e o **Sexo** de cada pessoa presente em uma sala. Sabendo que a última pessoa é João e possui 33 anos, o algoritmo deve exibir ao final:

- 5 A quantidade total de pessoas do sexo masculino.
- 6 A quantidade total de pessoas do sexo feminino.
- 7 A quantidade de pessoas menores de 21 anos.
- 8 A média de idade da sala.

### 11. Cálculo de Potência

Elabore um algoritmo que receba uma base (**N**) e um expoente (**m**) fornecidos pelo usuário. O algoritmo deve calcular e exibir o valor da potência correspondente ( $N^m$ ).

### 12. Somatório de Valores Pares

Elabore um algoritmo que calcule e exiba o somatório de todos os números pares existentes entre **10** e **20**.

### 13. Potências de 2

Elabore um algoritmo que calcule e exiba as potências de **2**, variando o expoente de **0** a **10**. O resultado deve incluir os valores de  $2^0, 2^1, \dots, 2^{10}$ .

### 14. Soma de Quadrados

Elabore um algoritmo que leia um número inteiro **N** fornecido pelo usuário e calcule a soma dos quadrados dos números de **1** até **N**. **Exemplo:** Para **N = 3**, o resultado será  $1^2 + 2^2 + 3^2 = 14$ .

### 15. Fatorial de um Número

Elabore um algoritmo que leia um número inteiro **N** fornecido pelo usuário e calcule o seu fatorial. **Exemplo:** Para **N = 5**, o resultado será  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ .

## 7 Vetores e Matrizes

Neste capítulo, abordaremos o conceito e a utilização de **vetores** e **matrizes**, fundamentais para resolver problemas que envolvem o armazenamento e manipulação de dados em larga escala. Essas estruturas, amplamente utilizadas em algoritmos, permitem organizar informações de forma estruturada, proporcionando maior eficiência no processamento de dados.

De acordo com Deitel e Deitel (2016), vetores e matrizes são formas de armazenamento sequencial que possibilitam o acesso direto a qualquer elemento a partir de seu índice. Essas estruturas são particularmente úteis quando se trabalha com grandes quantidades de dados relacionados, como notas de alunos, resultados de medições ou estatísticas de vendas.

Ao longo deste capítulo, exploraremos as definições, aplicações e exemplos práticos dessas estruturas, além de propor exercícios de fixação para consolidar os conceitos apresentados.

### 7.1 Vetores

Um vetor é uma estrutura de dados unidimensional que armazena uma coleção de elementos do mesmo tipo em posições contíguas na memória. Cada elemento do vetor é acessado por meio de um índice, que varia de acordo com a linguagem de programação.

Por exemplo, considere um vetor que armazena as notas de 5 alunos. Em uma linguagem como C, o vetor poderia ser declarado e utilizado da seguinte forma:

```
#include <stdio.h>

int main() {
    float notas[5]; // Declaração de um vetor de 5 posições
    int i;

    // Entrada de dados
    for (i = 0; i < 5; i++) {
        printf("Digite a nota do aluno %d: ", i + 1);
        scanf("%f", &notas[i]);
    }
}
```

```

// Exibição das notas
printf("\nNotas dos alunos:\n");
for (i = 0; i < 5; i++) {
    printf("Aluno %d: %.2f\n", i + 1, notas[i]);
}

return 0;
}

```

### Explicação do código:

- Declaramos um vetor de 5 elementos do tipo float.
- Utilizamos um laço for para preencher o vetor com as notas inseridas pelo usuário.
- Exibimos os valores armazenados no vetor, utilizando o índice correspondente a cada elemento.

### Em Portugol

```

programa
{
    funcao inicio()
    {
        real notas[5] // Declaração de um vetor de 5 posições
        inteiro i

        // Entrada de dados
        para (i = 0; i < 5; i++)
        {
            escreva("Digite a nota do aluno ", (i + 1), ": ")
            leia(notas[i])
        }

        // Exibição das notas
        escreva("\nNotas dos alunos:\n")
        para (i = 0; i < 5; i++)
        {
            escreva("Aluno ", (i + 1), ":", notas[i], "\n")
        }
    }
}

```

### Explicação do código:

- Substituímos float por real, o tipo de dado correspondente em Portugol Studio.
- O vetor notas foi declarado com 5 posições.
- Utilizamos o comando leia para capturar as entradas do usuário e armazená-las no vetor.

- Para iterar sobre os elementos, utilizamos a estrutura `para`.
- A exibição das notas foi realizada com o comando `escreva`, que substitui o `printf` do C.

Este código reproduz exatamente o mesmo comportamento do exemplo em C, adaptado ao **Portugol Studio**.

## 7.2 Matrizes

Enquanto os vetores armazenam dados em uma única dimensão, as matrizes permitem armazenar dados em múltiplas dimensões, como tabelas ou grades. Uma matriz bidimensional, por exemplo, pode ser usada para armazenar as notas de vários alunos em diferentes disciplinas.

Considere o exemplo a seguir em linguagem Java, que calcula a média das notas de três alunos em quatro disciplinas:

```
import java.util.Scanner;

public class MatrizNotas {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        double[][] notas = new double[3][4];
        double[] medias = new double[3];

        // Entrada de dados
        for (int i = 0; i < 3; i++) {
            System.out.println("Digite as notas do aluno " + (i + 1) + ":");
            for (int j = 0; j < 4; j++) {
                System.out.print("Disciplina " + (j + 1) + ": ");
                notas[i][j] = scanner.nextDouble();
            }
        }

        // Cálculo das médias
        for (int i = 0; i < 3; i++) {
            double soma = 0;
            for (int j = 0; j < 4; j++) {
                soma += notas[i][j];
            }
            medias[i] = soma / 4;
        }

        // Exibição dos resultados
        for (int i = 0; i < 3; i++) {
            System.out.printf("Média do aluno %d: %.2f%n", i + 1, medias[i]);
        }

        scanner.close();
    }
}
```

### Explicação do código:

- Declaramos uma matriz de dimensões 3x4 para armazenar as notas de três alunos em quatro disciplinas.
- Utilizamos dois laços `for` aninhados para preencher a matriz com as notas inseridas pelo usuário.
- Calculamos a média das notas de cada aluno, somando os elementos da linha correspondente e dividindo pelo número de disciplinas.
- Exibimos as médias calculadas para cada aluno.

```

programa
{
    funcao inicio()
    {
        real notas[3][4] // Declaração da matriz 3x4
        real medias[3] // Declaração do vetor para armazenar as médias
        inteiro i, j

        // Entrada de dados
        para (i = 0; i < 3; i++)
        {
            escreva("Digite as notas do aluno ", (i + 1), ":\\n")
            para (j = 0; j < 4; j++)
            {
                escreva("Disciplina ", (j + 1), ": ")
                leia(notas[i][j])
            }
        }

        // Cálculo das médias
        para (i = 0; i < 3; i++)
        {
            real soma = 0
            para (j = 0; j < 4; j++)
            {
                soma = soma + notas[i][j]
            }
            medias[i] = soma / 4
        }

        // Exibição dos resultados
        para (i = 0; i < 3; i++)
        {
            escreva("Média do aluno ", (i + 1), ":", medias[i], "\\n")
        }
    }
}

```

### Explicação do Código:

#### 1) Declaração de Variáveis:

- `notas [3] [4]` é uma matriz para armazenar as notas de 3 alunos em 4 disciplinas.
- `medias [3]` é um vetor para armazenar a média de cada aluno.

## 2) Entrada de Dados:

- Um laço `para externo` percorre os alunos (linhas da matriz).
- Um laço `para interno` percorre as disciplinas (colunas da matriz).

A função `leia` captura as notas fornecidas pelo usuário.

## 3) Cálculo das Médias:

- A variável `soma` armazena a soma das notas de cada aluno.
- A média é calculada dividindo a soma por 4 (quantidade de disciplinas).

## 4) Exibição dos Resultados:

- As médias calculadas são exibidas utilizando o comando `escreva`.
- Este código é funcional e segue as boas práticas para o uso do Portugol Studio.

## 7.3 Exercícios de Fixação

Os exercícios a seguir foram elaborados para proporcionar uma prática consolidada dos conceitos de **vetores** e **matrizes**, fundamentais no estudo de algoritmos e estruturas de dados. Inicialmente, os problemas com vetores exploram o armazenamento e manipulação de dados unidimensionais, permitindo ao aluno compreender operações como soma, busca, contagem e identificação de valores extremos. Em seguida, os exercícios com matrizes ampliam esse conhecimento para estruturas multidimensionais, introduzindo aplicações práticas como cálculo de médias, transposição, busca de elementos e operações com diagonais. Esses desafios têm como objetivo reforçar a compreensão dos conceitos teóricos e estimular o raciocínio lógico para a resolução de problemas práticos em programação.

### 7.3.1 Exercícios com Vetores

Explorando o armazenamento e manipulação de dados unidimensionais.

#### 1. Soma de Elementos de um Vetor

- Desenvolva um algoritmo que leia 10 números fornecidos pelo usuário e armazene-os em um vetor. Em seguida, o programa deve calcular e exibir a soma de todos os elementos do vetor.

## 2. Busca de um Valor em um Vetor

- Elabore um algoritmo que leia 15 números e armazene-os em um vetor. Permita que o usuário informe um número e, em seguida, exiba a posição em que o número foi encontrado no vetor (ou uma mensagem informando que o número não foi encontrado).

## 3. Contagem de Números Pares em um Vetor

- Desenvolva um algoritmo que leia 20 números inteiros em um vetor. O programa deve contar e exibir quantos números pares existem nesse vetor.

## 4. Soma e Média dos Elementos de um Vetor

- Crie um algoritmo que leia 15 números reais em um vetor. O programa deve calcular e exibir a soma e a média dos valores armazenados no vetor.

## 5. Maior e Menor Valor em um Vetor

- Elabore um algoritmo que leia 12 números em um vetor. O programa deve identificar e exibir o maior e o menor valor armazenado no vetor.

### 7.3.2 Exercícios com Matrizes

## 6. Média de Cada Linha de uma Matriz

- Elabore um algoritmo que leia uma matriz 3x4 com notas de alunos em disciplinas. O programa deve calcular e exibir a média das notas de cada aluno.

## 7. Matriz Transposta

- Elabore um algoritmo que leia uma matriz 3x3 com valores inteiros fornecidos pelo usuário. O programa deve calcular e exibir a matriz transposta (troca das linhas pelas colunas).

## 8. Busca de um Valor em uma Matriz

- Elabore um algoritmo que leia uma matriz 5x5 com valores inteiros fornecidos pelo usuário. Em seguida, permita que o usuário informe um número e mostre se ele está presente na matriz (caso esteja, indique sua posição).

## 9. Soma das Diagonais de uma Matriz

- Elabore um algoritmo que leia uma matriz quadrada 4x4 e calcule a soma dos elementos da diagonal principal e da diagonal secundária. Exiba os resultados.

## 10. Maior e Menor Valor de uma Matriz

- Elabore um algoritmo que leia uma matriz 4x4 com valores inteiros. O programa deve identificar e exibir o maior e o menor valor armazenado na matriz.

## 8 Linguagem de Projeto de Programação

O **diagrama de blocos**, para o desenvolvimento da lógica de programação, é a primeira forma de notação gráfica. A etapa subsequente consiste em transcrever o diagrama para uma forma narrativa conhecida como **pseudocódigo**, frequentemente chamada de “português estruturado” ou até mesmo “**portugol**”.

Essa técnica baseia-se em uma **PDL – Program Design Language** (Linguagem de Projeto de Programação), utilizada como referência genérica para uma linguagem de projeto. Seu objetivo é apresentar uma notação para a elaboração de **algoritmos**, que servirão na definição, criação e desenvolvimento de diferentes linguagens computacionais, como: **Pascal, C/C++, C#, Java**, entre outras. A seguir, vemos um exemplo desse tipo de algoritmo, adaptado para a **sintaxe do Portugol Studio** (com base em [MANZANO, 2000]).

### Exemplo de Cálculo da Média de 4 Notas (Portugol Studio)

```
programa
{
    funcao inicio()
    {
        real n1, n2, n3, n4
        real media

        escreva("Entre com a nota 1:")
        leia(n1)

        escreva("Entre com a nota 2:")
        leia(n2)

        escreva("Entre com a nota 3:")
        leia(n3)

        escreva("Entre com a nota 4:")
        leia(n4)

        media = (n1 + n2 + n3 + n4) / 4
        escreva("A média é:", media)
    }
}
```

Tabela 7 - Exemplo de um algoritmo para o cálculo da média de 4 notas

O autor [MANZANO, 2000] observa que, embora uma **linguagem de programação de alto nível** possa ser compilada e executada diretamente em um computador, uma **PDL** (esteja ela escrita em português, inglês ou outro idioma) **não** pode ser compilada da

mesma forma. Contudo, há “Processadores de PDL” que conseguem traduzir essa linguagem para uma representação gráfica de projeto. Alguns processadores, como o **Portugol Studio**, além de fornecerem dados como tabela de referência cruzada, mapas de alinhamento e valores de variáveis, também permitem **executar** o código por meio de interpretação.

Você pode efetuar o download do **Portugol Studio** no seguinte endereço:

<https://lite.acad.univali.br/portugol/>

## 8.1 Linguagem PDL

Abaixo algumas das sintaxes que podem ser usadas na linguagem PDL: O formato básico do nosso pseudocódigo é o seguinte:

```
programa
{
    //Função:
    //Autor :
    //Data :

    //Seção de Declarações
    funcao inicio()
    {
        //Seção de Comandos
    }
}
```

- A palavra-chave **programa** delimita o bloco principal do seu pseudocódigo.
- A função **inicio()** é o ponto de entrada do programa, onde serão escritas as instruções (comandos) que deseja executar.

O Portugol Studio permite a inclusão de comentários: qualquer texto precedido de “//” é ignorado, até se atingir o final da sua linha. Por este motivo, os comentários não se estendem por mais de uma linha: quando se deseja escrever comentários mais longos, que ocupem várias linhas, cada uma delas deverá começar por “//”.

## 8.2 Exercícios Complementares

1. Pesquisar sobre o programa Portugol Studio. Procurar conhecer todas as estruturas lógicas conhecidas, como as estruturas de decisão e repetição na linguagem PDL.
2. **Soma de Números Pares até 100.** Enunciado:

- Escreva um algoritmo que calcule e exiba a soma de todos os números pares de 1 a 100.
- Resolução

```

programa
{
    funcao inicio()
    {
        inteiro soma = 0

        para (inteiro i = 1; i <= 100; i++)
        {
            se (i % 2 == 0)
            {
                soma = soma + i
            }
        }

        escreva("A soma dos números pares de 1 a 100 é:", soma)
    }
}

```

### 3. Tabuada de um Número. Enunciado:

- Elabore um algoritmo que receba um número inteiro e exiba a sua tabuada de 1 a 10.
- Resolução

```

programa
{
    funcao inicio()
    {
        inteiro numero

        escreva("Digite um número para ver sua tabuada: ")
        leia(numero)

        para (inteiro i = 1; i <= 10; i++)
        {
            escreva(numero, " x ", i, " = ", (numero * i), "\n")
        }
    }
}

```

### 4. Contagem Regressiva; Enunciado:

- Crie um algoritmo que exiba uma contagem regressiva de 10 a 1 e ao final mostre a mensagem "FIM!".
- Resolução

```

programa
{
    funcao inicio()
    {
        para (inteiro i = 10; i >= 1; i--)
        {

```

```

        escreva(i, "\n")
    }

    escreva("FIM!")
}
}
```

## 5. Cálculo de Fatorial. Enunciado:

- Elabore um algoritmo que receba um número inteiro positivo e calcule o seu fatorial.
- Resolução

```

programa
{
    funcao inicio()
    {
        inteiro numero, fatorial = 1

        escreva("Digite um número inteiro positivo: ")
        leia(numero)

        se (numero < 0)
        {
            escreva("O número deve ser positivo!")
        }
        senao
        {
            para (inteiro i = 1; i <= numero; i++)
            {
                fatorial = fatorial * i
            }
            escreva("O fatorial de ", numero, " é: ", fatorial)
        }
    }
}
```

## 9 Exercícios Complementares

Os exercícios são ferramentas essenciais para consolidar o aprendizado de qualquer conteúdo acadêmico, especialmente em disciplinas como lógica de programação e algoritmos. Este capítulo tem como objetivo proporcionar ao leitor uma série de desafios práticos, organizados por nível de dificuldade, que estimulam o desenvolvimento do raciocínio lógico e a aplicação direta dos conceitos abordados nos capítulos anteriores.

A prática contida aqui auxilia na fixação de conhecimentos fundamentais, como:

- Criação de algoritmos eficientes e corretos;
- Representação de soluções através de fluxogramas e pseudocódigos;
- Utilização de estruturas de controle de fluxo, como condições e laços de repetição;
- Manipulação de variáveis, tipos de dados e operadores matemáticos;
- Resolução de problemas aplicados a cenários do mundo real.

Os exercícios foram elaborados com base em uma progressão graduada de dificuldade, permitindo que estudantes iniciantes ganhem confiança nos primeiros desafios e avancem gradualmente para tarefas mais complexas e que exijam maior reflexão e análise. Além disso, muitos dos problemas apresentados incentivam a criatividade e o pensamento crítico, habilidades essenciais para profissionais da área de tecnologia.

Para aproveitar ao máximo este capítulo, recomenda-se:

1. Resolver os exercícios na ordem apresentada, respeitando a progressão de dificuldade.
2. Experimentar diferentes abordagens para a solução de um mesmo problema.
3. Revisar os conceitos teóricos apresentados anteriormente caso surjam dúvidas durante a execução.
4. Implementar as soluções utilizando linguagens de programação de sua preferência, como Java, Python ou C, para reforçar o aprendizado prático.

Ao final do capítulo, espera-se que os leitores estejam mais preparados para lidar com desafios da programação e possuam uma base sólida para avançar para tópicos mais especializados no campo da tecnologia.

### 9.1 Exercícios

1. Escreva um algoritmo para somar dois números informados pelo usuário.

2. Crie um programa que leia dois números e exiba o maior deles.
3. Desenvolva um algoritmo para calcular o dobro de um número.
4. Leia o valor de um produto e aplique um desconto de 10%.
5. Crie um algoritmo para calcular o quadrado de um número.
6. Escreva um programa que leia um número e verifique se ele é positivo ou negativo.
7. Leia a idade de uma pessoa e informe se ela é maior de idade.
8. Crie um programa que calcule a média de três números informados.
9. Desenvolva um algoritmo para converter temperaturas de Celsius para Fahrenheit.
10. Leia um número e verifique se ele é par ou ímpar.
11. Leia três números e exiba-os em ordem crescente.
12. Crie um algoritmo que calcule o fatorial de um número.
13. Escreva um programa para calcular o IMC de uma pessoa com base em sua altura e peso.
14. Desenvolva um algoritmo que calcule o valor de uma conta de luz com base no consumo em kWh.
15. Leia a altura e o sexo de uma pessoa e calcule seu peso ideal usando fórmulas específicas.
16. Crie um programa que leia quatro notas de um aluno e exiba sua situação (aprovado, recuperação, ou reprovado).
17. Desenvolva um algoritmo para calcular o salário líquido de um funcionário com base em descontos.
18. Crie um programa que leia dois números e exiba todos os números entre eles.
19. Leia a cotação do dólar e converta um valor em reais para dólares.
20. Escreva um programa que leia um número e verifique se ele é primo.
21. Crie um programa para resolver equações do segundo grau.
22. Desenvolva um algoritmo para calcular a média ponderada de várias notas, com pesos definidos pelo usuário.
23. Crie um programa que leia 10 números e mostre a soma apenas dos números pares.

24. Escreva um algoritmo para calcular o número de grãos de trigo em um tabuleiro de xadrez.
25. Desenvolva um programa que simule uma calculadora simples com as quatro operações básicas.
26. Crie um programa que leia 5 números e exiba o maior e o menor deles.
27. Leia uma sequência de números até o usuário digitar zero e exiba a soma dos números digitados.
28. Desenvolva um algoritmo para encontrar o MMC e o MDC de dois números.
29. Crie um programa para gerar os primeiros n números da sequência de Fibonacci.
30. Escreva um programa que calcule a soma dos dígitos de um número informado.
31. Escreva um algoritmo que leia 20 números inteiros em um vetor e determine quantos desses números são negativos.
32. Elabore um algoritmo que leia um vetor de 15 posições e exiba todos os números pares armazenados nele.
33. Crie um algoritmo que leia um vetor de 12 números reais e calcule o maior e o menor valor do vetor, exibindo suas respectivas posições.
34. Desenvolva um algoritmo que leia 10 números em um vetor e exiba o vetor em ordem inversa.
35. Escreva um algoritmo que leia dois vetores de 10 posições e gere um terceiro vetor contendo a soma dos elementos correspondentes dos dois primeiros vetores.
36. Elabore um algoritmo que leia um vetor de 15 posições contendo números inteiros e calcule a quantidade de números múltiplos de 3 e múltiplos de 5.
37. Desenvolva um algoritmo que leia um vetor de 8 números inteiros e calcule a média dos valores. Em seguida, exiba os números do vetor que são maiores que a média.
38. Crie um algoritmo que leia 10 nomes em um vetor e exiba os nomes que possuem mais de 5 caracteres.
39. Escreva um algoritmo que leia dois vetores de 5 posições cada e gere um terceiro vetor intercalando os elementos dos dois vetores.
40. Elabore um algoritmo que leia um vetor de 20 números inteiros e substitua todos os números negativos por zero. Em seguida, exiba o vetor resultante.

## 10 Bibliografia

CABRAL, G. **O que é Lógica**. Brasil Escola, 2012.

CAELUM. **CS-14 Algoritmos e Estrutura de Dados em Java**. Material Caelum, 2011.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, H. M.; DEITEL, P. J. **Java: Como Programar**. 10. ed. São Paulo: Pearson, 2016.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 9899:1999. *Information technology — Programming languages — C*. Geneva: ISO, 1999.

K19. **Lógica de Programação**. Apostila, 2011.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. 2. ed. New Jersey: Prentice Hall, 1988.

MANZANO, J. A. **Algoritmos – Estudo dirigido**. São Paulo: Érica, 2000.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. **Algoritmos: Lógica para Desenvolvimento de Programação de Computadores**. São Paulo: Érica, 2012.

ORACLE. **Java Language Specification**. Disponível em:  
<https://docs.oracle.com/javase/specs/>. Acesso em: 21 dez. 2024.

UFRN. **Algoritmo e Lógica de Programação**. 2004.