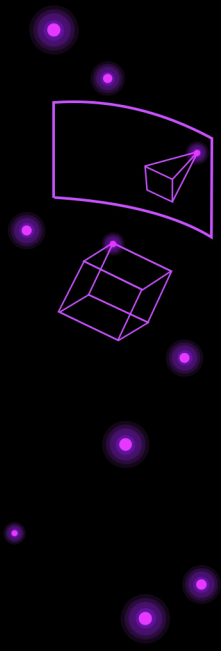
AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Autores:
Rafael Divino Ferreira Feitosa
Rafael Teixeira Sousa



Organizadores:
Taciana Novo Kudo
Deborah Silva Alves Fernandes
Renata Dutra Braga
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Autores:
Rafael Divino Ferreira Feitosa
Rafael Teixeira Sousa

Organizadores:
Taciana Novo Kudo
Deborah Silva Alves Fernandes
Renata Dutra Braga
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

2024

Revisão Técnica
Igor Gabriel Silva Batista
Yasmin de Freitas Pereira

Revisão Editorial
Ana Laura de Sene Amâncio Zara Brisolla

Capa
Iuri Vaz Miranda

Editoração Eletrônica
Luma Wanderley de Oliveira
Layane Grazielle Souza Dias

https://doi.org/10.5216/FEI.int.ebook.978-85-495-0952-9/2024

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Feitosa, Rafael Divino Ferreira
Introdução à lógica de programação [livro eletrônico] / autores Rafael Divino Ferreira Feitosa, Rafael Teixeira Sousa ; organizadores Taciana Novo Kudo...[et al.]. -- Goiânia, GO : Cegraf UFG, 2024.
PDF

Outros organizadores: Deborah Silva Alves Fernandes, Renata Dutra Braga, Cristiane Bastos Rocha Ferreira, Arlindo Rodrigues Galvão Filho.
Bibliografia.
ISBN 978-85-495-0952-9

1. Ciência da Computação 2. Linguagem de programação (Computadores) 3. Processamento de dados 4. Programação (Computadores) I. Feitosa, Rafael Divino Ferreira. II. Sousa, Rafael Teixeira. III. Kudo, Taciana Novo. IV. Fernandes, Deborah Silva Alves. V. Braga, Renata Dutra. VI. Ferreira, Cristiane Bastos Rocha. VII. Galvão Filho, Arlindo Rodrigues. VIII. Título.

24-217371 CDD-005.1

Índices para catálogo sistemático:

1. Lógica de programação : Computadores : Processamento de dados 005.1

Tábata Alves da Silva - Bibliotecária - CRB-8/9253

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Instituições responsáveis

Universidade Federal de Goiás (UFG)

Centro de Competência Embrapii em Tecnologias Imersivas, denominado AKCIT (Advanced Knowledge Center for Immersive Technologies)

Centro de Excelência em Inteligência Artificial (CEIA)

Instituições financiadoras

Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii)

Governo do Estado de Goiás

Empresas parceiras do AKCIT

Apoio

Universidade Federal de Goiás (UFG)

Pró-Reitoria de Pesquisa e Inovação (PRPI-UFG)

Instituto de Informática (INF-UFG)

Abreviaturas e Siglas

AKCIT	*Advanced Knowledge Center in Immersive Technology* - Centro de Competências em Tecnologias Imersivas
cm	Centímetro
CPU	*Central Processing Unit* - Unidade Central de Processamento
Embrapii	Empresa Brasileira de Pesquisa e Inovação Industrial
IDE	Ambiente Integrado de Desenvolvimento
kg	Quilos
m	Metro
RAM	*Random Access Memory* - Memória de Acesso Aleatório
SSD	*Solid State Drives* - Unidades de Estado Sólido
UFG	Universidade Federal de Goiás

Listas de Figuras

Sumário

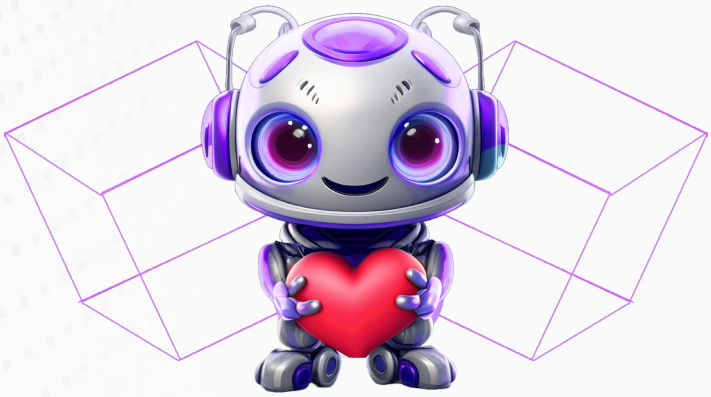
Apresentação

Prezado(a) Participante,

Seja bem-vindo(a) ao microcurso **Introdução à Lógica de Programação**!

Este Microcurso faz parte da Coleção Formação e Capacitação do Centro de Competências Imersivas, uma parceria entre a Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii) e a Universidade Federal de Goiás (UFG).

A sua oferta foi motivada com o objetivo de proporcionar aos(às) estudantes uma introdução abrangente à programação de computadores e à lógica de programação. Ao final do Curso, os(as) estudantes serão capazes de aplicar os princípios fundamentais da lógica de programação para resolver problemas computacionais básicos.



Desejamos um excelente estudo!

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Unidade I
Conceitos Básicos

Unidade I - Conceitos Básicos

1.1. A finalidade do computador

A principal finalidade de um computador é receber, manipular e armazenar dados. Essas funções são realizadas por meio de programas que, basicamente, são conjuntos de instruções que "dizem" ao computador o que fazer. Quando citamos o termo "computador", podemos estender esse conceito para qualquer dispositivo que permita a execução dessas funções como, por exemplo, *smartphones*, *smartwatches*, dispositivos embarcados e outros.

Receber dados

Os computadores recebem dados de várias formas. Pode ser por meio de um teclado, *mouse*, *scanner*, sensores ou até mesmo de outro computador via internet. Esses dados podem ser números, textos, imagens ou qualquer outra informação que possa ser digitalizada.

Manipular dados

Uma vez que os dados são recebidos, o computador precisa manipulá-los para produzir um resultado útil. Manipulação de dados pode incluir cálculos, comparação de valores, reorganização dos dados, entre outras operações. Tudo isso é feito seguindo as instruções contidas nos programas. Por exemplo, um programa pode somar uma série de números, ordenar uma lista de nomes em ordem alfabética ou transformar uma imagem colorida em uma versão em preto e branco.

Armazenar dados

Depois de manipulados, os dados podem ser armazenados para uso futuro. Os computadores armazenam dados em diferentes tipos de memória e dispositivos de armazenamento, como discos rígidos, *solid state drives* (SSDs), *pendrives*, CDs, DVDs e na nuvem. Isso permite que os dados sejam recuperados e reutilizados quando necessário.

Programas

Os programas são escritos em linguagens de programação, que são conjuntos de comandos e regras que permitem que os humanos criem essas instruções de uma maneira que o computador possa entender. Existem muitas linguagens de programação, cada uma com suas próprias características e usos específicos.

Por exemplo, uma simples tarefa como calcular a média de dois números pode ser programada em uma linguagem como Python da seguinte maneira:

```
# Este programa calcula a média de dois números

    # Recebe os números do usuário
    numero1 = float(input("Digite o primeiro número: "))
    numero2 = float(input("Digite o segundo número: "))

    # Calcula a média
    media = (numero1 + numero2) / 2

    # Mostra o resultado
    print("A média dos dois números é:", media)
```

Nesse exemplo, o programa:

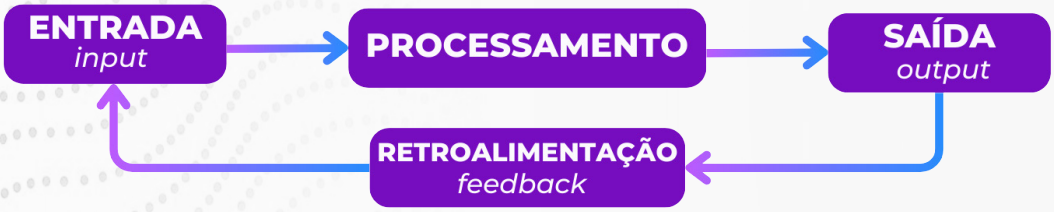
1. Recebe dois números do(a) usuário(a).
2. Manipula esses dados calculando a média.
3. Mostra o resultado ao(à) usuário(a).

Neste Curso, não iremos abordar a linguagem de programação Python. Para compreensão dos conceitos de lógica de programação, utilizaremos o Português Estruturado, mais conhecido como Portugol. O Portugol é uma linguagem de programação didática que utiliza uma sintaxe semelhante à linguagem natural, nesse caso, o português. Foi criada para ensinar os conceitos básicos de programação de forma mais acessível e intuitiva para iniciantes. O Portugol é usado frequentemente em ambientes educacionais para introduzir estudantes aos fundamentos da lógica de programação antes que eles avancem para linguagens de programação mais complexas e formais, como Python, Java, C++, entre outras.

1.2. O processamento de dados no computador

O computador realiza o processamento de dados por meio de uma sequência bem definida de etapas: entrada, processamento e saída. Na Figura 1, a seguir, essas etapas são ilustradas.

Figura 1 - Etapas de um processamento de dados



Fonte: autoria própria.

Vamos entender melhor como isso funciona.

Entrada de dados

Os dados são recebidos pelos dispositivos de entrada, que são periféricos conectados ao computador e permitem que os usuários insiram informações.

Exemplos de dispositivos de entrada incluem:

- Teclado: utilizado para digitar textos e comandos.
- *Mouse*: usado para navegar e selecionar itens na tela.
- *Scanner*: converte documentos físicos em formatos digitais.
- Microfone: captura áudio para ser processado pelo computador.
- Câmera: captura imagens e vídeos.

Processamento de dados

Uma vez que os dados são recebidos, o computador os processa. O processamento é realizado pela Unidade Central de Processamento (CPU), também conhecida como processador. O processador executa as instruções dos programas, realizando operações como cálculos matemáticos, tomadas de decisão e manipulação de dados.

Por exemplo, quando você digita um texto no teclado, o processador converte as teclas pressionadas em caracteres exibidos na tela.

Saída de dados

Após o processamento, os dados são enviados para os dispositivos de saída, que permitem que o(a) usuário(a) veja ou ouça os resultados do processamento. Exemplos de dispositivos de saída incluem:

- Monitor: exibe texto, imagens e vídeos.
- Impressora: produz cópias físicas de documentos digitais.

- Caixas de som: reproduzem sons e músicas.
- Projetor: exibe apresentações e vídeos em uma tela grande.

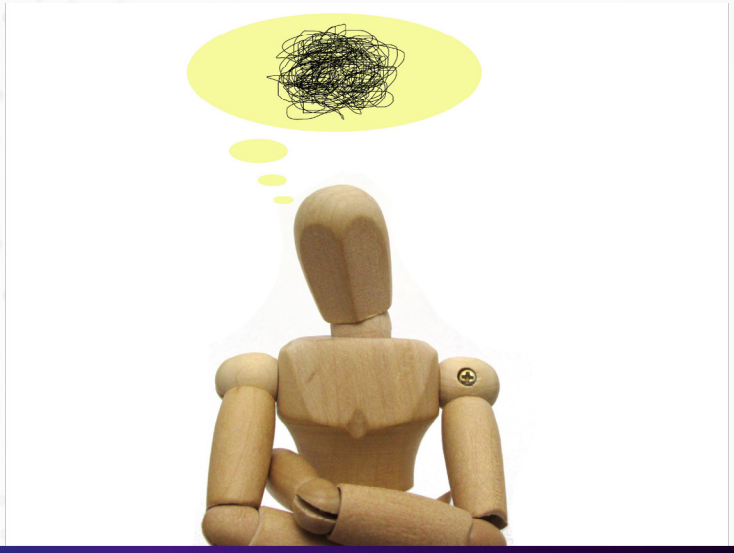
Retroalimentação

Em alguns casos, o computador pode utilizar a retroalimentação, onde os resultados das saídas são usados como novas entradas para o sistema, refinando e melhorando continuamente o processamento de dados. Isso é comum em sistemas de controle, como os termostatos, que ajustam a temperatura com base nos dados de sensores.

1.3. Etapas para o desenvolvimento de um programa

Desenvolver um programa de computador envolve várias etapas sistemáticas que ajudam a garantir que o problema seja resolvido de forma eficiente e eficaz. Vamos explorar essas etapas em detalhe (Figura 2).

Figura 2 - Etapas para o desenvolvimento de um programa



Fonte: autoria própria.

Algoritmo

Após a análise, a próxima etapa é a criação do algoritmo. Um algoritmo é uma sequência de passos sistemáticos para resolver o problema. Ele pode ser representado de várias formas, como:

Figura 3 - Representações de algoritmos

Fonte: autoria própria.

Imagine o seguinte problema:

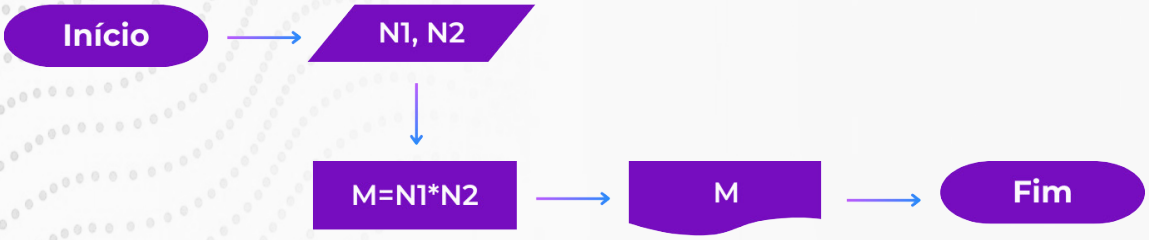
Faça um algoritmo para multiplicar dois números e mostrar o resultado.

A solução do problema citado poderia ser escrita utilizando as três ferramentas descritas anteriormente da seguinte forma:

- Descrição narrativa:
  1. Receber dois números que serão multiplicados
  2. Multiplicar os números
  3. Mostrar o resultado da multiplicação

- Fluxograma (Figura 4):

Figura 4 - Representação de um fluxograma



Fonte: autoria própria.

» Português estruturado:

```
algoritmo “MultiplicacaoDoisNumeros”
   // Este algoritmo multiplica dois números fornecidos pelo
   usuário e exibe o resultado.

   var
       numero1, numero2, resultado: Real

   inicio
       // Entrada de dados
       escreva(“Digite o primeiro número: “)
       leia(numero1)

       escreva(“Digite o segundo número: “)
       leia(numero2)

       // Processamento de dados
       resultado <- numero1 * numero2

       // Saída de dados
       escreva(“O resultado da multiplicação é: “, resultado)

   fimalgoritmo
```

Como citamos anteriormente, neste Curso, utilizaremos o Portugol para ensinar lógica de programação e resolver problemas computacionais. Por ser a ferramenta mais próxima das linguagens de programação, abstraindo suas complexidades inerentes, após este Curso, você estará apto(a) a fazer a transição para qualquer linguagem. Abordaremos a sintaxe do Portugol mais adiante.

Figura 5 - Codificação do algoritmo

Fonte: autoria própria.

Continuando com o exemplo anterior, em Python, a solução seria escrita da seguinte forma:

```
# Programa para multiplicar dois números fornecidos pelo usuário
  e exibir o resultado

  # Entrada de dados
  numero1 = float(input("Digite o primeiro número: "))
  numero2 = float(input("Digite o segundo número: "))

  # Processamento de dados
  resultado = numero1 * numero2

  # Saída de dados
  print("O resultado da multiplicação é:", resultado)
```

1.4. Lógica de programação

A lógica de programação é o uso correto das leis do pensamento e dos processos de raciocínio na programação de computadores. Vamos entender isso em mais detalhes a seguir.

Raciocínio abstrato

O raciocínio é algo abstrato e intangível, mas os seres humanos têm a capacidade de expressá-lo por meio da palavra falada ou escrita. Essas palavras se baseiam em um determinado idioma, que segue uma série de padrões conhecidos como gramática.

Por exemplo, podemos expressar um raciocínio lógico em diferentes idiomas, mas a essência do pensamento permanece a mesma. Vamos ver um exemplo simples:

- Em português: "Se estiver chovendo, leve um guarda-chuva."
- Em inglês: "*If it is raining, take an umbrella*."

Embora os idiomas sejam diferentes, a lógica por trás da frase é a mesma.

Lógica aplicada à programação

Algo similar ocorre com a lógica de programação, que pode ser representada em qualquer uma das inúmeras linguagens de programação existentes. A lógica de um algoritmo pode ser implementada em várias linguagens, mas a essência do raciocínio lógico permanece a mesma.

Vamos ilustrar isso com um exemplo simples de lógica de programação: verificar se um número é par ou ímpar.

Em Python:

```
# Verifica se um número é par ou ímpar

numero = int(input("Digite um número: "))

if numero % 2 == 0:
    print("O número é par.")
else:
    print("O número é ímpar.")
```

Em Portugol:

```
algoritmo "VerificarParOuImpar"
// Verifica se um número é par ou ímpar

var
    numero: Inteiro

inicio
    escreva("Digite um número: ")
    leia(numero)
```

```
        se numero % 2 = 0 entao
            escreva("O número é par.")
        senao
            escreva("O número é ímpar.")
        fimSe

    fimalgoritmo
```

Em JavaScript:

```
// Verifica se um número é par ou ímpar

    let numero = parseInt(prompt("Digite um número:"));

    if (numero % 2 === 0) {
        console.log("O número é par.");
    } else {
        console.log("O número é ímpar.");
    }
```

Padrões e convenções

Assim como os idiomas naturais seguem regras gramaticais, as linguagens de programação seguem suas próprias regras e convenções. Essas regras ajudam a garantir que o código seja compreensível e executável pelos computadores.

A lógica de programação é fundamentalmente sobre o uso correto das leis do pensamento para resolver problemas de forma sistemática. Independentemente da linguagem de programação utilizada, a lógica subjacente do algoritmo permanece a mesma. Compreender isso permite que os programadores possam aprender e trabalhar com diferentes linguagens de programação, aplicando os mesmos princípios lógicos.

1.5. Conceito de algoritmo

O objetivo principal do estudo da lógica de programação é a construção de algoritmos coerentes e válidos.

Definição de algoritmo

Um algoritmo pode ser definido como uma sequência de passos finita que visa atingir um objetivo bem definido. Algoritmos são fundamentais na programação porque fornecem uma forma clara e sistemática de resolver problemas.

Algoritmos no cotidiano

Apesar do nome pouco usual, algoritmos são comuns em nosso cotidiano. Um exemplo claro é uma receita de bolo. Uma receita é, essencialmente, um algoritmo, pois descreve uma série de ingredientes necessários e uma sequência de passos (ações) que devem ser seguidos para preparar o alimento desejado.

Vamos analisar uma receita de bolo como um exemplo de algoritmo.

Objetivo: Fazer um bolo de chocolate.

Ingredientes:

- 2 xícaras de farinha de trigo
- 1 xícara de açúcar
- 1 xícara de leite
- 1/2 xícara de óleo
- 1 xícara de chocolate em pó
- 3 ovos
- 1 colher de sopa de fermento em pó

Passos:

1. Pré-aqueça o forno a 180°C.
2. Em uma tigela, misture a farinha, o açúcar e o chocolate em pó.
3. Adicione o leite, o óleo e os ovos e misture bem até obter uma massa homogênea.
4. Acrescente o fermento em pó e misture delicadamente.
5. Despeje a massa em uma forma untada e enfarinhada.
6. Asse no forno pré-aquecido por cerca de 40 minutos ou até que um palito inserido no centro do bolo saia limpo.
7. Retire do forno e deixe esfriar antes de desenformar.

Algoritmo em programação

Da mesma forma que uma receita de bolo, um algoritmo em programação deve seguir uma sequência lógica de passos para resolver um problema específico. Vamos ver um exemplo simples de um algoritmo em programação que calcula a soma de dois números.

Objetivo: calcular a soma de dois números.

Passos:

1. Ler o primeiro número.
2. Ler o segundo número.
3. Somar os dois números.
4. Exibir o resultado.

A construção de algoritmos coerentes e válidos é crucial porque garante que o problema será resolvido de forma correta e eficiente. Um algoritmo mal definido pode levar a resultados incorretos ou a desperdício de recursos computacionais.

1.6. Tipos primitivos de dados

Para nos aproximarmos da maneira pela qual o computador manipula os dados, é útil dividi-los em quatro tipos primitivos, que serão os tipos básicos que usaremos na construção de algoritmos. A partir desses tipos primitivos, podemos criar estruturas mais complexas para manipular informações, chamadas estruturas de dados. Entretanto, as estruturas de dados são estudadas em cursos mais avançados e não fazem parte do escopo deste curso.

Tipos numéricos

Os tipos numéricos são usados para representar valores numéricos. Existem dois subtipos principais:

- Inteiro: representa números inteiros, ou seja, sem parte decimal. Exemplos incluem -3, 0, 42.
- Real: representa números reais, ou seja, com parte decimal. Exemplos incluem 3.14, -0.001, 42.0.

Tipo literal

O tipo literal é usado para representar sequências de caracteres, como palavras ou frases. Exemplos incluem “Olá, mundo!”, “12345”.

Tipo lógico

O tipo lógico é usado para representar valores de verdade, ou seja, verdadeiro ou falso.

Entender os tipos primitivos de dados é crucial porque eles são a base para a manipulação de dados em qualquer linguagem de programação. Cada tipo de dado tem suas próprias operações e métodos e escolher o tipo adequado é essencial para a eficiência e a corretude dos algoritmos.

À medida que avançarmos no curso, apresentaremos exemplos de aplicação de cada um desses tipos no contexto da resolução de problemas. Antes, precisamos entender que o computador manipula esses tipos de dados armazenando-os na memória principal (memória *Random Access Memory* [RAM]), que é a memória de trabalho. Esses dados não estão "soltos" na memória, mas em variáveis.

1.7. Conceito de variáveis

Um dado é classificado como variável quando tem a possibilidade de ser alterado em algum instante no decorrer do tempo. Durante a execução do algoritmo em que é utilizado, o valor do dado sofre alteração ou o dado é dependente da execução em um certo momento ou circunstância.

Variáveis no cotidiano

No dia a dia, encontramos muitos exemplos de dados variáveis, como:

- Cotação do dólar: o valor do dólar em relação à moeda local pode variar ao longo do tempo, dependendo de diversos fatores econômicos.
- Peso de uma pessoa: o peso de uma pessoa pode mudar devido a dietas, exercícios ou outras condições de saúde.
- Índice de inflação: a inflação é um indicador econômico que pode variar mensalmente, refletindo a mudança nos preços de bens e serviços.

Variáveis na programação

Quando um programa é executado, os dados que serão manipulados vão, conforme a necessidade, sendo armazenados na memória do computador. Uma variável é, portanto, um espaço na memória onde podemos armazenar valores que podem mudar durante a execução do programa.

Vejamos um exemplo onde uma pessoa realiza a soma de três números informados por nós. Este exemplo ilustra como as variáveis podem ser usadas para armazenar e manipular dados durante a execução de um algoritmo.

Descrição do algoritmo:

1. Informar os três números para o(a) usuário(a).
2. Somar o 1° e o 2° número.
3. Somar o resultado anterior ao 3° número.
4. Falar o resultado da soma.

1.8. Identificadores

Os identificadores são os nomes das variáveis, dos programas, das constantes, das rotinas, das unidades, enfim, de todos os elementos que fazem parte de um algoritmo. A escolha de bons identificadores é essencial para a clareza e a manutenção do código.

Regras básicas para a construção e nomeação de identificadores

1. Caracteres permitidos:
   - » Números (0...9)
   - » Letras maiúsculas (A...Z)
   - » Letras minúsculas (a...z)
   - » Caractere de sublinhado (_)

2. Caracteres não permitidos:
   - » Acentos (á, é, í, ó, ú)
   - » Cedilha (ç)
   - » Pontuações (., ;, :, ?)

3. Primeiro caractere:
   - » Deve ser sempre uma letra (A-Z, a-z) ou o sublinhado (_).

4. Espaços em branco e caracteres especiais:
   - » Não são permitidos espaços em branco.
   - » Não são permitidos caracteres especiais como (@, $, +, -, %, !, etc.).

5. Palavras reservadas:
   - » Não é permitido usar palavras reservadas da linguagem de programação como identificadores. Palavras reservadas são aquelas que têm um significado especial na linguagem, como var, inteiro, algoritmo, escreva, repita, etc.

1.9. Variáveis *vs*. constantes

Entendemos que um dado é constante quando não sofre nenhuma variação no decorrer do tempo. Seu valor é o mesmo desde o início até o fim da execução do algoritmo. Para ilustrar a diferença entre valores variáveis e constantes, vamos utilizar o exemplo de um algoritmo para cálculo da área de uma circunferência mais adiante.

Constantes

Uma constante é um valor fixo que não muda durante a execução do algoritmo. Por exemplo, na fórmula para calcular a área de uma circunferência, o valor de π (pi) é constante.

Variáveis

Uma variável é um valor que pode mudar durante a execução do algoritmo. No cálculo da área de uma circunferência, o raio (r) da circunferência é uma variável, pois pode ter diferentes valores.

Exemplo: Cálculo da área de uma circunferência

A fórmula para calcular a área de uma circunferência é:

$$\text{Área} = \pi * r^2$$

Onde:

- π é uma constante aproximadamente igual a 3,1416.
- r é o raio da circunferência, que é uma variável.

Assim, em qualquer programa que implemente o cálculo da área de uma circunferência, o valor de π poderá ser declarado como constante no início do programa e utilizado sempre que necessário.

1.10. Linguagens de programação

Para iniciantes em lógica de programação, escolher a linguagem certa para desenvolver seus projetos pode fazer uma grande diferença, pois cada uma possui particularidades e áreas de aplicação. Não se preocupe em decidir agora em qual linguagem de programação você irá se especializar após este Curso. O mais importante é desenvolver o pensamento algorítmico usando o Portugol. Aqui estão apenas algumas das principais linguagens de programação recomendadas para iniciantes, com uma vantagem, uma desvantagem, suas áreas de aplicação e o exemplo para multiplicação de dois números.

Python

Python é muito popular entre os iniciantes devido à sua sintaxe simples e clara, que é bastante intuitiva e próxima do inglês, facilitando a compreensão e a escrita de código. No entanto, como é uma linguagem interpretada, pode ser mais lenta em comparação com linguagens compiladas como C ou C++, o que a torna menos ideal para aplicações que exigem alta performance. Python é amplamente utilizado em ciência de dados,

aprendizado de máquina, desenvolvimento *web* (Django, Flask), automação de *scripts* e desenvolvimento de *software*.

```python
# Programa em Python para multiplicar dois números
num1 = float(input("Digite o primeiro número: "))
num2 = float(input("Digite o segundo número: "))
resultado = num1 * num2
print("O resultado da multiplicação é:", resultado)
```

JavaScript

JavaScript é essencial para o desenvolvimento *web*, permitindo a criação de páginas dinâmicas e interativas. É executado diretamente no navegador, o que facilita o aprendizado prático. No entanto, pode ser um pouco confuso para iniciantes devido ao seu comportamento assíncrono e à forma como lida com o contexto de execução. JavaScript é utilizado principalmente no desenvolvimento *front-end* de *websites*, mas, também, é muito usado no *back-end* com tecnologias como Node.js®, além de desenvolvimento de aplicativos móveis com *frameworks* como React Native®.

```javascript
// Programa em JavaScript para multiplicar dois números
var num1 = parseFloat(prompt("Digite o primeiro número: "));
var num2 = parseFloat(prompt("Digite o segundo número: "));
var resultado = num1 * num2;
console.log("O resultado da multiplicação é: " + resultado);
alert("O resultado da multiplicação é: " + resultado);
```

Java

Java é conhecido por sua portabilidade, seguindo o princípio "*write once, run anywhere*" (escreva uma vez, rode em qualquer lugar), sendo altamente portátil entre diferentes sistemas operacionais. Contudo, a sintaxe do Java pode ser mais verbosa, o que pode tornar os programas longos e mais difíceis de ler para iniciantes. Java é amplamente utilizado no desenvolvimento de aplicações empresariais, aplicativos Android®, sistemas de grande escala e desenvolvimento de servidores *web*.

```java
// Programa em Java para multiplicar dois números
import java.util.Scanner;

public class Multiplicacao {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite o primeiro número: ");
        double num1 = scanner.nextDouble();
```

```
            System.out.print("Digite o segundo número: ");
            double num2 = scanner.nextDouble();
            double resultado = num1 * num2;
            System.out.println("O resultado da multiplicação é: "
    + resultado);
        }
    }
```

C

Aprender C oferece uma compreensão profunda dos fundamentos da programação, pois muitas outras linguagens são baseadas em C. Entretanto, C requer a gestão manual de memória, o que pode levar a erros complexos e problemas de segurança, como vazamentos de memória e corrupção de dados. C é amplamente utilizado em sistemas operacionais, desenvolvimento de *software* de sistema, programação embarcada, e desenvolvimento de jogos de baixo nível.

```
// Programa em C para multiplicar dois números
#include <stdio.h>

int main() {
    double num1, num2, resultado;
    printf("Digite o primeiro número: ");
    scanf("%lf", &num1);
    printf("Digite o segundo número: ");
    scanf("%lf", &num2);
    resultado = num1 * num2;
    printf("O resultado da multiplicação é: %.2lf\n", resul-
tado);
    return 0;
}
```

Ruby

Ruby é conhecida por sua sintaxe limpa e fácil de ler, além de ter uma comunidade acolhedora e recursos abundantes para iniciantes. Apesar disso, Ruby é menos popular no mercado de trabalho comparado com Python ou JavaScript, o que pode limitar as oportunidades de emprego. Ruby é principalmente usada no desenvolvimento *web*, especialmente com o *framework Ruby on Rails*, que facilita a criação de aplicações *web* robustas e escaláveis.

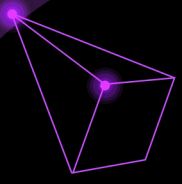
```ruby
# Programa em Ruby para multiplicar dois números
print “Digite o primeiro número: “
num1 = gets.chomp.to_f
print “Digite o segundo número: “
num2 = gets.chomp.to_f
resultado = num1 * num2
puts “O resultado da multiplicação é: #{resultado}”
```

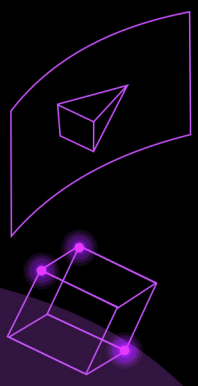
Cada linguagem de programação tem seus pontos fortes e fracos, bem como áreas de aplicação específicas. A escolha da melhor linguagem para começar depende dos objetivos e interesses do iniciante. Python é, geralmente, recomendado pela sua simplicidade e versatilidade em áreas como ciência de dados e desenvolvimento *web*. JavaScript é ótimo para quem deseja entrar no desenvolvimento *web*. Java oferece robustez e portabilidade para aplicações empresariais e Android®. C proporciona uma base sólida em conceitos fundamentais e é essencial para desenvolvimento de sistemas e *software* de baixo nível. Ruby é acessível e amigável, ideal para desenvolvimento *web* com *Ruby on Rails*.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Unidade II
Operadores e Estrutura Sequencial





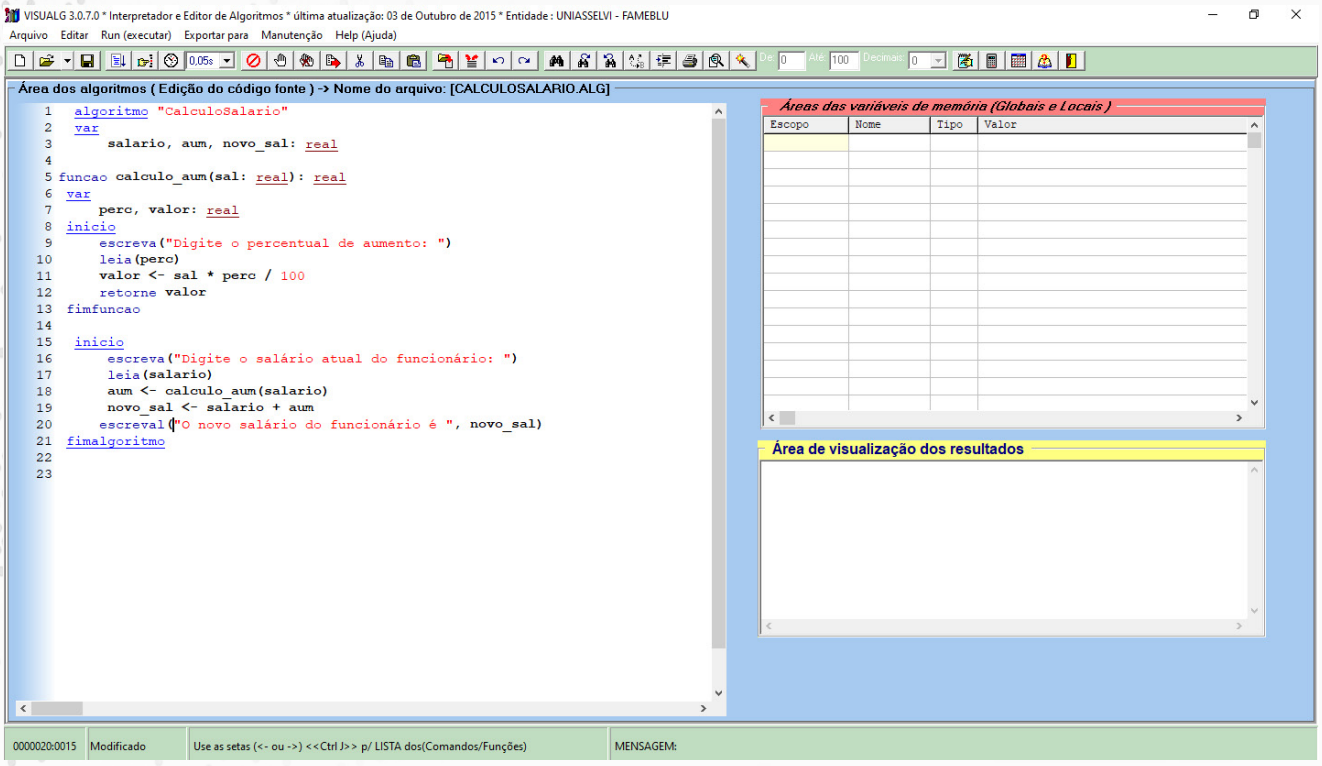
Unidade II - Operadores e Estrutura Sequencial

Nesta Unidade, iniciaremos a construção dos algoritmos em Portugol. Na Unidade anterior, vimos alguns algoritmos para ilustrar os conceitos básicos de lógica de programação. Entretanto, não detalhamos a explicação linha a linha dos códigos. Faremos isso nesta Unidade.

2.1. VisuAlg

Para codificação, execução e teste dos algoritmos em Portugol utilizaremos uma ferramenta chamada VisuAlg que pode ser baixada em sua versão mais estável (3.0.6.5) em https://sourceforge.net/projects/visualg30/. A seguir, podemos visualizar a interface da ferramenta (Figura 6).

Figura 6 - Interface da ferramenta educacional VisuAlg para codificação de algoritmos



Fonte: VisuAlg.

O VisuAlg é uma ferramenta educacional de desenvolvimento que permite a criação e execução de algoritmos usando a linguagem de programação Portugol. Desenvolvido

especialmente para iniciantes, o VisuAlg facilita o aprendizado de lógica de programação por meio de uma sintaxe próxima do português, o que torna o processo de codificação mais intuitivo e acessível.

Principais funcionalidades

1. Ambiente Integrado de Desenvolvimento (IDE): o VisuAlg oferece um ambiente completo para escrever, testar e depurar algoritmos. Sua interface gráfica é simples e intuitiva, ideal para quem está começando.
2. Linguagem Portugol: utiliza uma linguagem de programação didática e estruturada, com comandos em português, o que ajuda os iniciantes a se familiarizar com conceitos básicos de programação sem a barreira da língua inglesa.
3. Sintaxe simples: a sintaxe do VisuAlg é projetada para ser clara e fácil de entender, permitindo que os(as) estudantes foquem na lógica de programação e no desenvolvimento de algoritmos.
4. Execução passo a passo: permite a execução de algoritmos passo a passo, facilitando a compreensão de como o algoritmo funciona e ajudando na identificação de erros e na correção de *bugs*.
5. Recursos educacionais: inclui várias funções didáticas, como a possibilidade de visualizar a memória durante a execução, o que ajuda os(as) estudantes a entenderem como as variáveis são manipuladas.

Benefícios do VisuAlg

1. Facilidade de aprendizado: ideal para quem está começando a aprender programação, especialmente estudantes do ensino médio e de cursos introdutórios de computação.
2. Didático e intuitivo: a linguagem Portugol e a interface gráfica amigável tornam o VisuAlg uma ferramenta didática que ajuda a reforçar conceitos básicos de lógica e de algoritmos.
3. Gratuito e acessível: o VisuAlg é uma ferramenta gratuita, o que a torna acessível para escolas, universidades e autodidatas.

O VisuAlg é uma excelente ferramenta para quem deseja iniciar no mundo da programação. Com uma interface amigável e uma linguagem intuitiva, ele facilita a compreensão de conceitos fundamentais de lógica de programação, preparando os estudantes para aprender linguagens mais complexas no futuro. Ao utilizar o VisuAlg, os(as) iniciantes podem desenvolver suas habilidades de resolução de problemas e lógica de uma maneira acessível e eficaz.

2.2. Overview

Vamos iniciar esta Unidade com um exemplo completo de um algoritmo em Portugol. O programa será simples, mas cobrirá vários conceitos importantes, como variáveis, constantes, entrada e saída de dados e processamento que discutimos brevemente na Unidade anterior. O objetivo do algoritmo é receber o nome e o ano de nascimento do(a) usuário(a) e, após cumprimentá-lo cordialmente, mostrar qual sua idade no ano atual. Veja, a seguir, o código e a explicação linha a linha.

```
1.  algoritmo “PrimeiroPrograma”
2.  var
3.      nome: caractere
4.      ano_nasc, idade, ano_atual: inteiro
5.
6.  inicio
7.      ano_atual <- 2024
8.      escreva(“Digite seu nome: “)
9.      leia(nome)
10.     escreva(“Digite o ano do seu nascimento (AAAA): “)
11.     leia(ano_nasc)
12.     idade <- ano_atual - ano_nasc
13.     escreval(“Olá “, nome, “!”)
14.     escreval(“Em “, ano_atual, “ você completa “, idade,
“ anos.”)
15.     escreva(“Parabéns!!!”)
16. fimalgoritmo
```

Linha 1:

```
algoritmo “PrimeiroPrograma”
```

Define o início do algoritmo e dá um nome a ele. Em Portugol, cada programa é iniciado com a palavra reservada **algoritmo**, seguida do nome do algoritmo entre aspas.

Linha 2:

```
var
```

Indica o início da seção de declaração de variáveis. Todas as variáveis a serem usadas no algoritmo devem ser declaradas aqui.

Linha 3:

```
nome: caractere
```

Declara a variável nome do tipo **caractere**, que corresponde ao tipo texto. Essa variável armazenará o nome do(a) usuário(a).

Linha 4:

```
ano_nasc, idade, ano_atual: inteiro
```

Declara as variáveis **ano_nasc** e **idade** do tipo **inteiro**. **ano_nasc** armazenará o ano de nascimento do(a) usuário(a) e idade armazenará a idade calculada. Declara também a constante **ano_atual**, que receberá um valor fixo na **linha 7**.

Linha 6:

```
inicio
```

Marca o início do bloco principal do algoritmo, onde as instruções serão executadas.

Linha 7:

```
aano_atual <- 2024
```

Atribui o valor 2024 à constante **ano_atual**. No Visualg, constantes podem ser definidas na seção **const**, antes da seção **var**. Entretanto, neste Curso, vamos declará-las junto com as variáveis e definir seu valor fixo no corpo no algoritmo.

Linha 8:

```
escreva("Digite seu nome: ")
```

Exibe a mensagem "Digite seu nome: " na tela para solicitar a entrada do(a) usuário(a).

Linha 9:

```
leia(nome)
```

Lê um valor digitado pelo(a) usuário(a) e o armazena na variável **nome**.

Linha 10:

```
escreva("Digite o ano do seu nascimento (AAAA): ")
```

Exibe a mensagem "Digite o ano do seu nascimento (AAAA): " na tela para solicitar a entrada do(a) usuário(a).

Linha 11:

```
leia(ano_nasc)
```

Lê um valor digitado pelo(a) usuário(a) e o armazena na variável **ano_nasc**.

Linha 12:

```
idade <- ano_atual - ano_nasc
```

Calcula a idade subtraindo o ano de nascimento **ano_nasc** do ano atual **ano_atual** e armazena o resultado na variável idade. O operador <- é usado para atribuição em Portugol.

Linha 13:

```
escreval("Olá ", nome, "!")
```

Exibe a mensagem "Olá", seguida do nome do(a) usuário(a) e um ponto de exclamação. Em Portugol, a concatenação de texto e variáveis é feita dentro do comando escreva.

Linha 14:

```
escreval("Em ", ano_atual, " você completa ", idade, " anos.")
```

Exibe a mensagem “Em 2023 você completa X anos.” onde X é a idade calculada. Em Portugol, a concatenação de texto e variáveis é feita dentro do comando escreva.

Linha 15:

```
escreva(“Parabéns!!!”)
```

Exibe a mensagem “Parabéns!!!” na tela.

Linha 16:

```
fimalgoritmo
```

Define o fim do algoritmo. Em Portugol, cada programa termina com a palavra reservada **fimalgoritmo**.

Agora, vamos falar sobre o fluxo de criação de um algoritmo utilizando o Portugol, citando e explicando cada componente da estrutura sequencial.

2.3. Lógica sequencial dos comandos

A estrutura sequencial é a forma mais básica de organizar um algoritmo, onde as instruções são executadas uma após a outra, na ordem em que aparecem. Essa estrutura é fundamental para a construção de algoritmos e é utilizada quando não há necessidade de desvios ou repetições no fluxo de execução que veremos nas próximas Unidades.

A estrutura sequencial de um algoritmo corresponde ao conjunto de ações primitivas que será executado em uma sequência linear de cima para baixo e da esquerda para a direita, isto é, na mesma ordem que foram escritas. Essa é a forma mais básica e direta de organizar um algoritmo.

Características da estrutura sequencial

- Linearidade: as instruções são executadas uma após a outra, sem desvios ou saltos.
- Ordem de execução: as ações são executadas na mesma ordem em que aparecem no código.
- Simplicidade: ideal para tarefas simples e diretas onde o fluxo de execução não varia.

Como já vimos anteriormente, qualquer algoritmo escrito em Portugol terá, no mínimo, a estrutura a seguir, que chamamos de bloco principal do algoritmo.

```
algoritmo "NomeAlgoritmo"
   var
      ...
   inicio
      ...
   fimalgoritmo
```

Onde vemos o sinal de reticências, iremos acrescentar comandos de entrada, processamento e saída, que veremos a seguir. Observe a ausência de acentuação nas palavras que, mesmo escritas em português, seguem a convenção da língua inglesa, utilizada por padrão em outras linguagens de programação e não são acentuadas. Entretanto, devemos ter atenção a um fato: nem todas as linhas de código começam alinhadas à esquerda. Algumas estão recuadas em relação à margem esquerda e damos, a isso, o nome de indentação.

A indentação refere-se ao espaçamento usado no início das linhas de código dentro de um bloco para indicar hierarquia e estrutura. Isso quer dizer que, quando fazemos a indentação de uma linha, ela "está dentro" do bloco mais à esquerda. É uma prática comum em praticamente todas as linguagens de programação, onde o código dentro de estruturas de controle, como *loops*, condições e funções, é recuado para a direita. Em linguagens de programação que utilizam o sinal de ponto-e-vírgula (;) para delimitar o final de uma linha de código, a indentação, do ponto de vista de execução, é opcional. Entretanto, quando não há o ponto-e-vírgula, como é o caso do Portugol e do Python, a indentação é obrigatória para garantir a correta execução do código.

Importância da indentação em Lógica de Programação

1. Legibilidade: a indentação melhora significativamente a legibilidade do código. Quando os blocos de código são claramente diferenciados por recuos, torna-se mais fácil para os(as) programadores(as) entenderem a estrutura e o fluxo lógico do programa.
2. Manutenção: código bem indentado é mais fácil de manter e modificar. Os(As) Programadores(as) podem rapidamente localizar e corrigir erros ou fazer alterações necessárias, pois a estrutura do código é mais evidente.
3. Colaboração: em projetos onde múltiplos(as) programadores(as) colaboram, uma indentação consistente garante que todos(as) possam ler e entender o código de outros membros da equipe sem dificuldades adicionais.
4. Erro de sintaxe: algumas linguagens de programação, como Python, utilizam a indentação como parte da sintaxe. Uma indentação incorreta pode levar a erros de sintaxe e ao funcionamento incorreto do programa.

5. Estrutura lógica: a indentação ajuda a visualizar a estrutura lógica do código, destacando blocos de código que pertencem juntos. Isso é crucial para a compreensão de como as diferentes partes do código interagem e afetam umas às outras.

2.4. Declaração de variáveis e constantes

No Portugol, variáveis e constantes podem ser de diferentes tipos de dados, dependendo da natureza dos valores que precisam armazenar. Vamos explorar os tipos mais comuns e como declarar cada um deles.

Tipos

1. Inteiro: armazena números inteiros, positivos ou negativos, sem a parte decimal.
   - » Exemplo: 10, -5, 0

```
var
        idade: inteiro
```

2. Real: armazena números reais, ou seja, números com parte decimal. Também pode ser utilizado para armazenar números inteiros. As partes inteira e fracionada são separadas pelo ponto (.). A vírgula, em programação, tem outras funções conforme a linguagem.
   - » Exemplo: 3.14, -2.5, 0.0

```
var
        salario: real
```

3. Caracter: armazena sequências de caracteres, como palavras ou frases. Os valores armazenados nas variáveis devem ser colocados entre aspas (“ e ”).
   - » Exemplo: “João”, “A”

```
var
        nome: caracter
```

4. Lógico: armazena valores booleanos, ou seja, verdadeiro ou falso.
   - » Exemplo: Verdadeiro, Falso

```
var
        esta_empregado: logico
```

A declaração de constantes segue as mesmas regras para declaração de variáveis. Entretanto, após a declaração, o valor fixo da constante é informado, utilizando-se o símbolo de atribuição <-.

```
   idade <- 30
      salário <- 10000.00
      nome <- "João"
      esta_empregado <- verdadeiro
```

A utilização de variáveis requer algumas observações, que serão discutidas a seguir.

2.5. Diretrizes para declaração de variáveis

Quando trabalhamos com variáveis na programação, existem algumas regras e boas práticas que devem ser seguidas para garantir que o código seja eficiente, legível e livre de erros. Vamos abordar essas regras detalhadamente, com exemplos práticos. Não se preocupe, pois retornaremos para a construção de algoritmos de forma detalhada, linha a linha, posteriormente. Os exemplos, a seguir, se fazem necessários para visualizarmos o fluxo dos dados quando manipulamos as variáveis.

Uma variável guarda apenas um valor por vez

Cada variável pode armazenar apenas um valor de cada vez. Se atribuirmos um novo valor a uma variável, o valor anterior será substituído.

Exemplo em Portugol:

```
algoritmo "ExemploVariavel"
   var
       x: inteiro
   inicio
       x <- 10
       escreval("Valor de x: ", x)  // exibe 10
       x <- 20
       escreval("Valor de x: ", x)  // exibe 20 (o valor ante-
   rior foi substituído)
   fimalgoritmo
```

Tipos primitivos e tipagem forte

Não é possível armazenar um dado de tipo primitivo diferente em uma variável declarada para outro tipo. Isso é conhecido como tipagem forte. Uma variável declarada como um tipo numérico não pode armazenar um texto, por exemplo.

Exemplo em Portugol:

```
algoritmo “TipagemForte”
   var
       numero: inteiro
   inicio
       numero <- 10       // Tipo inteiro
       numero <- “dez”    // Tipo literal
   fimalgoritmo
```

Identificadores únicos

Devemos utilizar identificadores diferentes para cada variável. Isso evita confusão e erros durante a execução do programa. Identificadores são os nomes dados às variáveis. Caso duas ou mais variáveis sejam declaradas com o mesmo nome, mesmo de tipos diferentes, ocorrerá um erro de execução.

Exemplo em Portugol:

```
algoritmo “IdentificadoresUnicos”
   var
       idade: inteiro
       altura: real
   inicio
       idade <- 25
       altura <- 1.75
       escreval(“Idade: “, idade)
       escreval(“Altura: “, altura)
   fimalgoritmo
```

Tamanho dos identificadores

Os nomes dos identificadores podem possuir qualquer tamanho, mas devem ser escolhidos com cuidado para garantir a legibilidade do código. Perceba que, no código a seguir, foram declaradas duas variáveis, uma com o identificador completo e outra com o identificador abreviado. Dependendo da abreviação, pode ocorrer dúvidas sobre qual informação deve ser armazenada na variável.

Exemplo em Portugol:

```
algoritmo “TamanhoIdentificadores”
   var
       nome_do_usuario: caracter
       id_usu: inteiro
   inicio
       nome_do_usuario <- “Alice”
       id_usu <- 30
```

```
        escreval(“Nome: “, nome_do_usuario)
        escreval(“Idade: “, id_usu)
    fimalgoritmo
```

Nomes sugestivos para identificadores

Apesar de os nomes dos identificadores serem escolhidos pelo(a) programador(a), recomenda-se que se utilize nomes sugestivos, ou seja, que possam representar objetivamente o que será armazenado na variável identificada por ele(a). Isso melhora a legibilidade e a manutenção do código.

Exemplo em Portugol:

```
algoritmo “NomesSugestivos”
   var
       nome: caracter
       idade: inteiro
   inicio
       nome <- “João”
       idade <- 28
       escreval(“Nome: “, nome)
       escreval(“Idade: “, idade)
   fimalgoritmo
```

Boas práticas na nomeação de variáveis

- Use nomes descritivos: nomes como **numero_de_alunos**, **soma_total** e **media_notas** são autoexplicativos.
- Siga as convenções de nomenclatura: em Portugol, é comum usar nomes descritivos e claros.
- Evite abreviações desnecessárias: abreviações podem ser confusas. Prefira nomes completos, a menos que a abreviação seja amplamente reconhecida.

Além do que discutimos anteriormente, acerca de boas práticas na nomeação de variáveis, é preciso seguir regras para criação de identificadores. Mas, atenção! Como já discutimos, os identificadores não nomeiam apenas as variáveis de um programa.

2.6. Comando de entrada

O comando de entrada é usado para receber dados fornecidos pelo(a) usuário(a) durante a execução de um algoritmo. Esses dados são, então, armazenados em variáveis para serem processados posteriormente. No Portugol, o comando de entrada é feito utilizando-se a instrução **leia**.

A sintaxe básica do comando de entrada em Portugol é:

```
leia(variavel)
```

Onde **variavel** é a variável que receberá o valor fornecido pelo(a) usuário(a).

Podemos usar o comando **leia** para ler diferentes tipos de dados.

Vamos ver um exemplo prático no qual solicitamos ao(à) usuário(a) que insira seu nome. Nesse exemplo, é demonstrado o uso do comando de entrada **leia** para ler uma variável do tipo caracter.

```
algoritmo "EntradaNome"
   // Este programa solicita ao usuário que insira seu nome

   var
       nome: caracter

   inicio
       // Entrada de dados
       leia(nome)
   fimalgoritmo
```

Apesar de ilustrar o uso do comando **leia**, o exemplo anterior possui pouca aplicação prática, pois não interage com o(a) usuário(a); apenas recebe o valor digitado, sem mostrar nenhuma mensagem ou processar esse valor e é encerrado. O comando de saída pode ser utilizado para resolver parte desse problema.

2.7. Comando de saída

O comando de saída é utilizado para exibir informações ao(à) usuário(a) durante a execução de um algoritmo. Esse comando permite que o programa mostre resultados, mensagens ou quaisquer dados que sejam importantes para o(a) usuário(a) visualizar.

No Portugol, utilizamos a instrução **escreva** para exibir informações na tela. A instrução **escreva** pode ser seguida por uma ou mais variáveis ou mensagens a serem exibidas, sempre separadas por vírgula (,) que também é o símbolo de concatenação.

A operação de concatenação de textos é usada para unir dois ou mais textos (sequências de caracteres) e/ou variáveis em um único texto. Essa operação é fundamental em muitas situações de programação, especialmente quando se deseja combinar várias partes de texto em uma só, como mensagens personalizadas, construção de frases ou formatação de dados.

Existe uma variação para o comando de saída, o **escreval**. Esse comando é utilizado quando se deseja exibir uma mensagem e/ou valor na tela e, em seguida, pular uma linha. Isso é útil quando temos dois ou mais comandos de saída em sequência para que esses não sejam exibidos na tela em uma mesma linha.

Mas, atenção! Diferentemente dos comandos próprios do Portugol, todo o texto colocado dentro do comandos escreva ou escreval pode e deve ser acentuado para seguir as regras ortográficas e passar uma boa impressão na comunicação com os(as) usuários(as).

A sintaxe básica para usar o comando de saída é a seguinte:

```
escreva(valor1, valor2, ..., valorN)
escreval(valor1, valor2, ..., valorN)
```

Onde valor1, valor2, ..., valorN são os valores que serão exibidos na tela. Esses valores podem ser variáveis, constantes ou mensagens de texto. Podemos usar o comando **escreva** para exibir diferentes tipos de dados.

Melhorando o exemplo anterior, vamos acrescentar uma mensagem pedindo ao(à) usuário(a) que digite seu nome e depois exibir uma saudação.

```
algoritmo "MensagemBoasVindas"
   // Este programa exibe uma mensagem de boas-vindas ao usuário

   var
       nome: caracter

   inicio
       // Entrada de dados
       escreva("Digite seu nome: ")
       leia(nome)

       // Saída de dados
       escreva("Bem-vindo(a), ", nome, "!")

   fimalgoritmo
```

2.8. Comentários

Você deve ter percebido, em vários códigos, a presença de textos explicativos antecedidos por duas barras inclinadas à direita (//). Chamamos esse recurso de comentários.

Comentários são anotações no código que não são executadas pelo programa. Eles são usados para explicar e documentar o código, tornando-o mais legível e compreensível para outros(as) programadores(as) (ou para você mesmo no futuro). Comentários são extremamente úteis para descrever a finalidade das variáveis, o funcionamento das funções e qualquer lógica complexa.

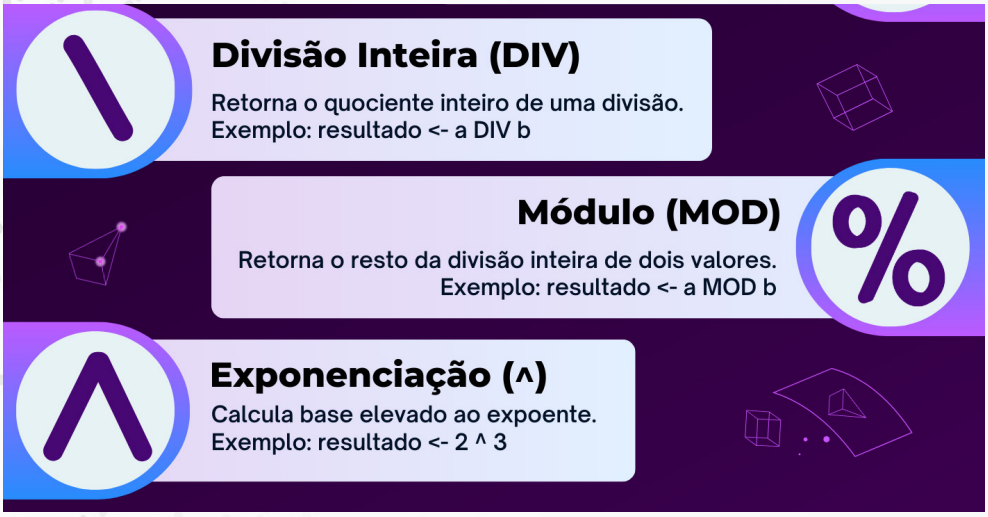
No Portugol, os comentários são precedidos por dois sinais de barra (//).

Tudo o que for escrito após esses sinais, até o final da linha, será considerado um comentário e não será executado pelo programa.

2.9. Operadores

Operadores aritméticos são símbolos utilizados para realizar operações matemáticas em variáveis e constantes. Eles são fundamentais para a manipulação de dados numéricos em algoritmos. No Portugol, os operadores aritméticos mais comuns são: adição, subtração, multiplicação, divisão e módulo (Figura 7).

Figura 7 - Operadores aritméticos



Fonte: autoria própria.

Outro recurso importante para resolução de expressões aritméticas é o uso de parênteses para determinar a precedência das operações. Em programação, a precedência dos operadores determina a ordem em que as operações são realizadas. O uso de parênteses pode alterar essa ordem, garantindo que certas operações sejam executadas primeiro. Vamos explorar a precedência padrão dos operadores aritméticos e como os parênteses influenciam essas operações.

Precedência padrão dos operadores aritméticos

Sem o uso de parênteses, a ordem de precedência dos operadores aritméticos em Portugol é a seguinte:

1. Exponenciação (^) maior precedência
2. Multiplicação (*) e divisão (/)
3. Adição (+) e subtração (-) menor precedência

Exemplos

Sem parênteses:

```
resultado <- 3 + 4 * 2
```

Explicação: a multiplicação tem maior precedência que a adição, então a operação 4 * 2 é realizada primeiro, seguida pela adição 3 + 8. O resultado final é 11.

Com parênteses:

```
resultado <- (3 + 4) * 2
```

Explicação: os parênteses têm maior precedência, então a operação 3 + 4 é realizada primeiro, seguida pela multiplicação 7 * 2. O resultado final é 14.

Vamos criar um algoritmo que demonstre o uso dos operadores aritméticos realizando operações básicas com dois números fornecidos pelo(a) usuário(a).

```
1.  algoritmo “CalculadoraSimples”
2.  // Este programa realiza operações aritméticas básicas com dois números fornecidos pelo usuário
3.
4.  var
5.      numero1, numero2: real
6.      soma, subtracao, multiplicacao, divisao, divisao_int, modulo: real
7.
8.  inicio
9.      // entrada de dados
10.     escreva(“Digite o primeiro número: “)
11.     leia(numero1)
12.
13.     escreva(“Digite o segundo número: “)
14.     leia(numero2)
15.
16.     // operações aritméticas
17.     soma <- numero1 + numero2
18.     subtracao <- numero1 - numero2
19.     multiplicacao <- numero1 * numero2
20.     divisao <- numero1 / numero2
21.     divisao_int <- numero1 DIV numero2
22.     modulo <- numero1 MOD numero2
23.
24.     // saída de dados
25.     escreval(“A soma dos números é: “, soma)
26.     escreval(“A subtração dos números é: “, subtracao)
27.     escreval(“A multiplicação dos números é: “, multiplicacao)
28.     escreval(“A divisão dos números é: “, divisao)
29.     escreval(“A divisão inteira dos números é: “, divisao_int)
30.     escreva(“O módulo dos números é: “, modulo)
31. fimalgoritmo
```

Exercícios

1. Faça um algoritmo que receba o nome e o valor de um produto qualquer, calcule e mostre o valor das prestações, sabendo que o seu valor é dividido em 5x sem juros.
2. Escreva um algoritmo que determine o consumo médio de combustível de um automóvel após recebidas a distância total percorrida e a quantidade de combustível gasto.
3. Escreva um algoritmo que leia o nome do aluno e as 4 notas bimestrais do ano, calcule e mostre o nome do aluno e as suas médias aritméticas anual e semestrais.
4. Faça um algoritmo que receba o preço de custo de um produto e a margem de lucro sobre o mesmo em porcentagem (%), calcule e mostre o preço de venda já com o lucro incluído.
5. Escreva um algoritmo que leia o nome do vendedor, o seu salário base mensal e o valor do total de vendas realizadas por ele durante o mês. Sabendo que este vendedor recebe comissão de 15% sobre o valor das vendas efetuadas por ele e que o seu salário final é a composição do salário base mais o valor da comissão, calcule e mostre o nome do vendedor, seu salário final e a porcentagem recebida a mais em relação ao seu salário base mensal.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Unidade III
Estruturas Condicionais

Unidade III - Estruturas Condicionais

Uma estrutura condicional ou de seleção permite a execução de um grupo de ações (bloco) quando determinadas condições, representadas por expressões lógicas e/ou relacionais, sejam ou não satisfeitas. Dentro dos algoritmos, essas estruturas possibilitam, por meio da tomada de decisão, desviar o fluxo de execução dos comandos de acordo com a necessidade.

Antes de criarmos algoritmos utilizando estruturas condicionais, vamos aprender sobre operadores lógicos aritméticos. As operações com esses operadores sempre resultam em um valor lógico (verdadeiro ou falso) fazendo com que uma ação seja executada (em caso de verdadeiro) ou não (em caso de falso).

3.1. Operadores relacionais

Os operadores relacionais são usados para comparar valores. A seguir, estão os principais operadores relacionais, suas descrições e exemplos do cotidiano (Figura 8).

Figura 8 - Operadores relacionais

	DESCRIÇÃO E EXPRESSÃO	EXEMPLO	RESULTADO
=	**Igualdade** idade1 = idade2	Comparar se duas idades são iguais	Verdadeiro se idade1 for igual a idade2
<>	**Diferença** altura1 <> altura2	Comparar se duas alturas são diferentes	Verdadeiro se altura1 for diferente de altura2
>	**Maior que** altura1 > altura2	Verificar se uma pessoa é mais alta	Verdadeiro se altura1 for maior que altura2
<	**Menor que** temp1 < temp2	Verificar se a temperatura está mais baixa	Verdadeiro se temp1 for menor que temp2
>=	**Maior ou igual** nota >= nota_minima	Verificar se a nota é suficiente	Verdadeiro se nota for maior ou igual a nota_minima
<=	**Menor ou igual** velocidade <= limite	Verificar se a velocidade está dentro do limite	Verdadeiro se velocidade for menor ou igual a limite

Fonte: autoria própria.

Mais exemplos:

1. Igualdade (=)
   - Situação: Verificar se duas senhas são iguais.
   - Condições: Uma pessoa está tentando fazer login em um site e precisa verificar se a senha digitada é igual à senha cadastrada.
   - Expressão relacional: **senha_digitada = senha_cadastrada**

2. Diferença (<>)
   - Situação: Decidir se dois amigos têm alturas diferentes.
   - Condições: Dois amigos estão se medindo para ver se um é mais alto que o outro.
   - Expressão relacional: **altura_amigo1 <> altura_amigo2**

3. Maior que (>)
   - Situação: Verificar se uma criança pode andar em um brinquedo de parque de diversões.
   - Condições: A criança deve ter mais de 1,20 metros de altura para andar no brinquedo.
   - Expressão relacional: **altura_crianca > 1.20**

4. Menor que (<)
   - Situação: Determinar se uma pessoa pode comprar uma passagem de ônibus com desconto.
   - Condições: O desconto é oferecido para pessoas com menos de 18 anos.
   - Expressão relacional: **idade_pessoa < 18**

5. Maior ou igual (>=)
   - Situação: Verificar se um aluno passou de ano na escola.
   - Condições: O aluno deve ter uma média final maior ou igual a 6.0 para passar.
   - Expressão relacional: **media_final >= 6.0**

6. Menor ou igual (<=)
   - Situação: Verificar se uma pessoa pode entrar em um evento com um limite de idade.
   - Condições: A entrada é permitida para pessoas com até 25 anos.
   - Expressão relacional: **idade_pessoa <= 25**

3.2. Operadores lógicos

Os operadores lógicos são usados para combinar expressões relacionais. Abaixo estão os principais operadores lógicos, suas descrições e exemplos do cotidiano (Figura 9).

Figura 9 - Operadores lógicos

	DESCRIÇÃO E EXPRESSÃO	EXEMPLO	RESULTADO
e	**Conjunção** tem_carteira E maior_de_idade	Verificar se uma pessoa tem carteira de motorista e é maior de idade	Verdadeiro se ambos tem_carteira e maior_de_idade forem verdadeiros
ou	**Disjunção** esta_chovendo OU esta_nevando	Verificar se está chovendo ou nevando	Verdadeiro se pelo menos um de esta_chovendo ou esta_nevando for verdadeiro
não	**Negação** NAO esta_doente	Verificar se uma pessoa não está doente	Verdadeiro se esta_doente for falso

Fonte: autoria própria.

Mais exemplos:

1. Operador E
   - Situação: verificar se uma pessoa pode participar de um sorteio.
   - Condições: a pessoa precisa ser maior de 18 anos de idade e ter comprado um bilhete.
   - Expressão lógica: **idade >= 18 E comprou_bilhete**

2. Operador OU
   - Situação: decidir se vai ao parque.
   - Condições: vai ao parque se for fim de semana ou estiver de férias.
   - Expressão lógica: **fim_de_semana OU de_ferias**

3. Operador NÃO
   - Situação: verificar se um equipamento não está quebrado.
   - Condições: o equipamento está funcionando, se não estiver quebrado.
   - Expressão lógica: **NAO esta_quebrado**

3.3. Condicional simples

Quando precisamos testar uma condição antes de executar uma ação, usamos uma seleção simples, que segue a seguinte sintaxe em Portugol:

```
se (condicao) entao
        // Bloco de código se a condição for verdadeira
    fimse
```

No exemplo anterior, o comando somente será executado se o teste da condição retornar verdadeiro. O resultado de uma condição é a comparação entre dois ou mais valores, podendo ser verdadeiro ou falso. As condições são, geralmente, expressões que utilizam operadores relacionais e lógicos para comparar valores.

Vamos criar um exemplo onde verificamos se um aluno está aprovado com base em sua média. Utilizaremos a estrutura condicional **se...então**.

```
1.  algoritmo "VerificacaoAprovacao"
2.  // Este programa verifica se um aluno está aprovado com
    base em sua média
3.
4.  var
5.      nota1, nota2, nota3, media: real
6.
7.  inicio
8.      // Entrada de dados
9.      escreva("Digite a primeira nota: ")
10.     leia(nota1)
11.
12.     escreva("Digite a segunda nota: ")
13.     leia(nota2)
14.
15.     escreva("Digite a terceira nota: ")
16.     leia(nota3)
17.
18.     // Processamento de dados
19.     media <- (nota1 + nota2 + nota3) / 3
20.
21.     escreval("A média do aluno é: ", media)
22.
23.     // Estrutura condicional simples para verificar apro-
    vação
24.     se (media >= 7) entao
25.         escreva("Status: Aprovado")
26.     fimse
27.
28. fimalgoritmo
```

No exemplo anterior, a **linha 25** será executada somente se o aluno obtiver média igual ou superior a 7. Entretanto, se essa média for inferior a 7 nada será exibido na tela. Então, precisamos utilizar uma estrutura condicional composta.

3.4. Condicional composta

A estrutura condicional composta é utilizada quando existem situações em que há a necessidade de contemplar duas alternativas que dependem da condição. Quando é preciso executar comandos se a condição for verdadeira ou outros comandos se for falsa, utilizamos o comando **senao**. A sintaxe é a seguinte:

```
se (condicao) entao
        // Bloco de código se a condição for verdadeira
    senao
        // Bloco de código se a condição for falsa
    fimse
```

No exemplo anterior, os comandos dentro do bloco **entao** serão executados se a condição for verdadeira. Se a condição for falsa, os comandos dentro do bloco **senao** serão executados.

Vamos criar o exemplo anterior, onde verificamos se um aluno está aprovado ou reprovado com base em sua média. Utilizaremos a estrutura condicional composta **se... então...senao**.

```
1.  algoritmo "VerificacaoAprovacao"
2.  // Este programa verifica se um aluno está aprovado ou re-
    provado
3.
4.  var
5.      nota1, nota2, nota3, media: real
6.
7.  inicio
8.      // Entrada de dados
9.      escreva("Digite a primeira nota: ")
10.     leia(nota1)
11.
12.     escreva("Digite a segunda nota: ")
13.     leia(nota2)
14.
15.     escreva("Digite a terceira nota: ")
16.     leia(nota3)
17.
18.     // Processamento de dados
19.     media <- (nota1 + nota2 + nota3) / 3
20.
21.     escreval("A média do aluno é: ", media)
22.
```

```
23.     // Estrutura condicional composta para verificar apro-
vação
24.     se (media >= 7) entao
25.         escreva(“Status: Aprovado”)
26.     senao
27.         escreva(“Status: Reprovado”)
28.     fimse
29.
30. fimalgoritmo
```

3.5. Condicional encadeada

Quando temos um algoritmo onde um comando ou mais comandos devem ser executados apenas quando todas condições necessárias são satisfeitas, podemos encadear essas condições usando a estrutura **se...entao...se**. A sintaxe no Portugol para essa estrutura é:

```
se condicao1 entao
        se condicao2 entao
            se condicao3 entao
                se condicao4 entao
                    // Bloco de código se todas as condições fo-
rem verdadeiras
                fimse
            fimse
        fimse
    fimse
```

A estrutura descrita anteriormente segue padrões de encadeamento e não possui nenhum desvio para a falsidade (**senao**). Nesse caso, a forma como foi apresentada não é usual em algoritmos práticos. Podemos utilizar o operador lógico E - eventualmente, os operadores OU e NÃO - e escrevê-la simplificadamente, em Portugol, da seguinte forma:

```
se (condicao1) E (condicao2) E (condicao3) E (condicao4) entao
        // Bloco de código se todas as condições forem verdadei-
ras
    fimse
```

Perceba que, ao encadear duas ou mais condições com os operadores lógicos, precisamos escrever todas as condições entre parênteses individuais.

Incrementando o algoritmo anterior, para verificação da aprovação do aluno, podemos acrescentar a leitura do limite de faltas no ano e quantas faltas o aluno possui. Em seguida, podemos efetuar duas verificações, de nota e falta, para determinar se o algoritmo satisfaz as duas condições, simultaneamente, para ser aprovado.

Algoritmo em Portugol:

```
algoritmo “VerificacaoAprovacao”
// Este programa verifica se um aluno está aprovado com base em suas notas e faltas

var
    nota1, nota2, nota3, media: real
    limite_faltas, faltas: inteiro

inicio
    // Entrada de dados
    escreva(“Digite a primeira nota: “)
    leia(nota1)

    escreva(“Digite a segunda nota: “)
    leia(nota2)

    escreva(“Digite a terceira nota: “)
    leia(nota3)

    escreva(“Digite o limite de faltas no ano: “)
    leia(limite_faltas)

    escreval(“Digite o número de faltas do aluno: “)
    leia(faltas)

    // Processamento de dados
    media <- (nota1 + nota2 + nota3) / 3

    // Estrutura condicional para verificar aprovação
    se (media >= 7) E (faltas <= limite_faltas) entao
        escreva(“O aluno está aprovado.”)
    senao
        escreva(“O aluno está reprovado.”)
    fimse
fimalgoritmo
```

Em outro caso, podemos nos deparar com a situação em que as condições são excludentes, ou seja, quando existe um conjunto encadeado de condições, no qual apenas uma condição pode ser satisfeita, sendo que as demais não precisam ser verificadas.

Suponha um algoritmo que teste o valor de uma variável X, podendo ela assumir apenas um de quatro valores possíveis e executar um comando diferente.

```
se (x = 1) entao
       // Bloco de código se x for igual a 1
fimse
   se (x = 2) entao
       // Bloco de código se x for igual a 2
fimse
   se (x = 3) entao
       // Bloco de código se x for igual a 3
fimse
   se (x = 4) entao
       // Bloco de código se x for igual a 4
fimse
```

No exemplo anterior, todas as condições serão verificadas, mesmo após uma ser satisfeita, causando uma sobrecarga desnecessária na execução do algoritmo. A solução para otimização da estrutura desse algoritmo seria encadeá-la por meio da estrutura **se... senao...se**. Essa estrutura permite que, após encontrado o primeiro valor verdadeiro para o teste das condições, a estrutura seja interrompida, conforme exemplo a seguir.

```
se (x = 1) entao
       // Bloco de código se x for igual a 1
   senao
    se (x = 2) entao
           // Bloco de código se x for igual a 2
       senao
        se (x = 3) entao
               // Bloco de código se x for igual a 3
           senao
            se (x = 4) entao
                   // Bloco de código se x for igual a 4
            fimse
        fimse
    fimse
fimse
```

Por fim, vamos ampliar ainda mais o exemplo anterior, para verificação da aprovação do aluno, utilizando a estrutura condicional encadeada para determinar se o aluno está:

- Aprovado;
- Reprovado por nota;
- Reprovado por falta;
- Reprovado por falta e nota.

```
1.  algoritmo “verificacao_aprovacao”
2.  // Este programa calcula a média de quatro notas, verifica o limite de
faltas e determina se o aluno está aprovado ou reprovado
3.
4.  var
5.      n1b, n2b, n3b, n4b, media: real
6.      lim_faltas, tot_faltas: inteiro
7.
8.  inicio
9.      escreva(“Digite as quatro notas bimestrais: “)
10.     leia(n1b, n2b, n3b, n4b)
11.
12.     escreva(“Digite o limite de faltas da disciplina: “)
13.     leia(lim_faltas)
14.
15.     escreva(“Digite a quantidade de faltas do aluno: “)
16.     leia(tot_faltas)
17.
18.     media <- (n1b + n2b + n3b + n4b) / 4
19.     escreval(“Média do aluno: “, media)
20.
21.     se (media >= 7) E (tot_faltas <= lim_faltas) entao
22.         escreva(“O aluno foi aprovado!”)
23.     senao
24.         se (media < 7) E (tot_faltas <= lim_faltas) entao
25.             escreva(“O aluno foi reprovado por nota!”)
26.         senao
27.             se (media >= 7) E (tot_faltas > lim_faltas) entao
28.                 escreva(“O aluno foi reprovado por falta!”)
29.             senao
30.                 se (media < 7) E (tot_faltas > lim_faltas) entao
31.                     escreva(“O aluno foi reprovado por falta e
nota!”)
32.                 fimse
33.             fimse
34.         fimse
35.     fimse
36. fimalgoritmo
```

Linhas 1-2:

```
1.  algoritmo “verificacao_aprovacao”
2.  // Este programa calcula a média de quatro notas, verifica o limite de faltas e determina se o aluno está aprovado ou reprovado
```

Definimos o nome do algoritmo e adicionamos um comentário explicando seu propósito.

Linhas 4-6:

```
4.  var
5.      n1b, n2b, n3b, n4b, media: real
6.      lim_faltas, tot_faltas: inteiro
```

Declaramos as variáveis necessárias para armazenar as notas, a média e as faltas.

Linhas 8-10:

```
8.  inicio
9.      escreva(“Digite as quatro notas bimestrais: “)
10.     leia(n1b, n2b, n3b, n4b)
```

Solicitamos ao(à) usuário(a) que insira as quatro notas bimestrais e as lemos.

Linhas 12-13:

```
12.     escreva(“Digite o limite de faltas da disciplina: “)
13.     leia(lim_faltas)
```

Solicitamos ao(à) usuário(a) que insira o limite de faltas da disciplina e lemos o valor.

Linhas 15-16:

```
15.     escreva(“Digite a quantidade de faltas do aluno: “)
16.     leia(tot_faltas)
```

Solicitamos ao(à) usuário(a) que insira a quantidade de faltas do aluno e lemos o valor.

Linhas 18-19:

```
18.     media <- (n1b + n2b + n3b + n4b) / 4
19.     escreval(“Média do aluno: “, media)
```

Calculamos a média das quatro notas e exibimos o resultado.

Linhas 21-35:

```
21.     se (media >= 7) E (tot_faltas <= lim_faltas) entao
22.         escreva(“O aluno foi aprovado!”)
23.     senao
24.         se (media < 7) E (tot_faltas <= lim_faltas) entao
25.             escreva(“O aluno foi reprovado por nota!”)
26.         senao
27.             se (media >= 7) E (tot_faltas > lim_faltas) entao
28.                 escreva(“O aluno foi reprovado por falta!”)
29.             senao
30.                 se (media < 7) E (tot_faltas > lim_faltas) entao
31.                     escreva(“O aluno foi reprovado por falta e
nota!”)
32.                 fimse
33.             fimse
34.         fimse
35.     fimse
```

Verificamos as condições de aprovação e exibimos a mensagem apropriada para cada situação:

- Aprovado: média maior ou igual a 7.0 e faltas dentro do limite.
- Reprovado por nota: média menor que 7.0 e faltas dentro do limite.
- Reprovado por falta: média maior ou igual a 7.0 e faltas acima do limite.
- Reprovado por falta e nota: média menor que 7.0 e faltas acima do limite.

Linha 36:

```
36. fimalgoritmo
```

Indicamos o fim do algoritmo.

3.6. Escolha

Quando um conjunto de valores precisa ser testado e ações diferentes são associadas a esses valores, podemos utilizar a estrutura de seleção de múltipla escolha criada com a palavra de comando **escolha...caso**. Essa estrutura permite que o algoritmo selecione uma entre várias alternativas com base no valor de uma variável.

A sintaxe básica da estrutura de múltipla escolha em Portugol é a seguinte:

```
escolha (variavel)
        caso valor1
            // Bloco de código para valor1
        caso valor2
            // Bloco de código para valor2
        caso valor3
            // Bloco de código para valor3
        ...
        outrocaso
            // Bloco de código padrão se nenhum dos casos for sa-
    tisfeito
    fimescolha
```

Para exemplificar o uso da estrutura **escolha...caso**, vamos construir um algoritmo que, dado um número de 1 a 7 informado pelo(a) usuário(a), é mostrado, na tela, o dia correspondente da semana ou uma mensagem de erro, se for outro número.

```
1.  algoritmo “verificacao_dia_semana”
2.  // Este programa verifica o dia da semana com base em um
    número de 1 a 7
3.
4.  var
5.      dia: inteiro
6.
7.  inicio
8.      escreva(“Digite um número de 1 a 7 para representar o
    dia da semana: “)
9.      leia(dia)
10.
11.     escolha (dia)
12.         caso 1
13.             escreva(“Segunda-feira”)
14.         caso 2
15.             escreva(“Terça-feira”)
16.         caso 3
17.             escreva(“Quarta-feira”)
18.         caso 4
19.             escreva(“Quinta-feira”)
20.         caso 5
21.             escreva(“Sexta-feira”)
22.         caso 6
23.             escreva(“Sábado”)
24.         caso 7
25.             escreva(“Domingo”)
26.         outrocaso
27.             escreva(“Número inválido! Por favor, digite
    um número de 1 a 7.”)
28.     fimescolha
29.
30. fimalgoritmo
```

Linhas 1-2:

```
1.  algoritmo “verificacao_dia_semana”
2.  // Este programa verifica o dia da semana com base em um
    número de 1 a 7
```

Define o nome do algoritmo como “verificacao_dia_semana” e diciona um comentário explicando que o programa verifica o dia da semana com base em um número de 1 a 7.

Linhas 4-5:

```
4.  var
5.      dia: inteiro
```

Inicia a declaração das variáveis e declara a variável **dia** do tipo **inteiro**, que armazenará o número digitado pelo(a) usuário(a).

Linhas 7-9:

```
7.  inicio
8.      escreva("Digite um número de 1 a 7 para representar o
dia da semana: ")
9.      leia(dia)
```

Inicia o bloco principal do algoritmo e exibe a mensagem solicitando ao(à) usuário(a) que digite um número de 1 a 7. Em seguida, lê o número digitado pelo(a) usuário(a) e armazena na variável **dia**.

Linhas 11-28:

```
11.     escolha (dia)
12.         caso 1
13.             escreva("Segunda-feira")
14.         caso 2
15.             escreva("Terça-feira")
16.         caso 3
17.             escreva("Quarta-feira")
18.         caso 4
19.             escreva("Quinta-feira")
20.         caso 5
21.             escreva("Sexta-feira")
22.         caso 6
23.             escreva("Sábado")
24.         caso 7
25.             escreva("Domingo")
26.         caso contrario
27.             escreva("Número inválido! Por favor, digite
um número de 1 a 7.")
28.     fimescolha
```

Inicia a estrutura de seleção múltipla escolha baseada no valor de dia e testa todos os possíveis valores esperados. Na **linha 26**, se o dia não for nenhum dos valores entre 1 e 7, exibe "Número inválido! Por favor, digite um número de 1 a 7" e finaliza a estrutura.

Linhas 30:

```
30. fimalgoritmo
```

Linha 30: Indica o fim do algoritmo.

É importante saber algumas regras de restrições do uso da estrutura **escolha...caso**:

- Em Portugol, a variável testada em uma estrutura de seleção de múltipla escolha pode ser apenas dos tipos primitivos inteiro, lógico e caracter (com apenas um caractere);

- Em Portugol, no caso dos valores testados serem do tipo caracter, devem possuir apenas um caractere e serem colocados na lista entre aspas (“ e ”).

Exercícios

1. Escreva um algoritmo que receba um número inteiro e diga se este é par ou ímpar.
2. Escreva um algoritmo que verifique a validade de uma senha fornecida pelo(a) usuário(a). A senha válida é o número 1234. Caso a senha seja válida deve ser mostrada a mensagem “Acesso permitido”, caso contrário, se a senha for inválida, deve ser mostrada a mensagem “Acesso negado”.
3. Construa um algoritmo que receba as duas médias bimestrais obtidas por um aluno durante o semestre. Ao final deve ser mostrada sua média semestral (aritmética) e a sua situação de acordo com a tabela, a seguir.

MÉDIA	SITUAÇÃO
Maior ou igual a 6,0	**Aprovado**
Menor que 6,0	**Reprovado**

4. Tendo como dados de entrada, a altura e o sexo de uma pessoa, construa um algoritmo que calcule o peso ideal, utilizando as fórmulas abaixo, onde H é a altura da pessoa:
   - se o sexo for masculino: (72.7*H)-58;
   - se o sexo for feminino: (62.1*H)-44.7.
5. Faça um algoritmo que leia três valores inteiros e diferentes e mostre-os em ordem decrescente. Utilize, para tal, uma seleção encadeada.
6. Para participar da “Categoria Ouro” do 1º Campeonato Mundial de Bolinha de Gude o jogador deve pesar entre 70 kg (inclusive) e 80 kg (inclusive) e medir de 1,75 m (inclusive) a 1,90 m (inclusive). Construa um algoritmo que leia a altura e o peso de um jogador e determine se o jogador está apto a participar do campeonato, mostrando uma das seguintes mensagens conforme cada situação:
   - “Recusado por peso”: se somente o peso do jogador for inválido;
   - “Recusado por altura”: se somente a altura do jogador for inválida;

» “Recusado por peso e altura”: se a altura e o peso do jogador forem inválidos;

» “Aceito”: se a altura e o peso do jogador estiverem dentro da faixa especificada.

7. Um mercado está vendendo frutas de acordo com a seguinte tabela de preços:

	PREÇO POR QUILO	
	ATÉ 5KG	ACIMA DE 5KG
Morango	R$ 5,00	R$ 4,00
Maçã	R$ 3,00	R$ 2,00

Se o cliente comprar mais de 8 kg em frutas ou o valor total da compra ultrapassar R$35,00, receberá ainda um desconto de 20% sobre esse total. Escreva um algoritmo que receba a quantidade (em kg) de morangos e a quantidade (em kg) de maçãs adquiridas e escreva o valor a ser pago pelo cliente.

8. Faça um algoritmo que receba a altura e o peso de uma pessoa. De acordo com a tabela a seguir, verifique e mostre a classificação dessa pessoa.

ALTURA	PESO		
	ATÉ 60	ENTRE 60 E 90	ACIMA DE 90
Menores que 1,20	A	D	G
De 1,20 a 1,70	B	E	H
Maiores que 1,70	C	F	I

9. Escreva um algoritmo que receba quatro valores: I, A, B e C. Desses valores, I é inteiro positivo podendo ter os valores 1, 2 ou 3; A, B e C são reais. Escreva os números A, B e C, obedecendo a tabela a seguir. Suponha que o valor digitado para I seja sempre válido (1, 2 ou 3) e que os números digitados sejam diferentes.

VALOR DE I	FORMA DE ESCREVER
1	A, B e C em ordem crescente
2	A, B e C em ordem decrescente
3	O maior fica entre os outros números

10. Escreva um algoritmo que apresente o menu a seguir, permita que o(a) usuário(a) escolha a opção desejada, receba os dados necessários para executar a operação e mostre o resultado. Verifique a possibilidade de a opção escolhida ser inválida e não se preocupe com restrições como, por exemplo, salário negativo.

» Menu de opções:

1 - Imposto a ser pago e salário final

2 - Novo salário com aumento

3 - Classificação

» Digite a opção desejada:

Na opção 1: receber o salário de um funcionário, calcular e mostrar o valor do imposto e o salário final usando as regras a seguir:

SALÁRIO	IMPOSTO
Menor que R$ 1.100,00	5%
De R$ 1.100,00 a 3.000,00	10%
Acima de R$ 3.000,00	15%

Na opção 2: receber o salário de um funcionário, calcular e mostrar o valor do novo salário com aumento, usando as regras a seguir:

SALÁRIO	AUMENTO
Maior que R$ 3.000,00	R$ 450,00
De R$ 2.000,00 (inclusive) a R$: 3.000,00 (inclusive)	R$ 300,00
De R$ 1.500,00 (inclusive) a R$: 2.000,00	R$ 250,00
Menor que R$ 1.500,00	R$ 100,00

Na opção 3: receber o salário de um funcionário e mostrar sua classificação usando a tabela a seguir:

SALÁRIO	CLASSIFICAÇÃO
Até R$ 1.500,00 (inclusive)	Mal remunerado
Maior que R$ 1.500,00	Bem remunerado

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Unidade IV
Estruturas de Repetição

Unidade IV - Estruturas de Repetição

Nosso dia a dia é repleto de rotinas que, embora, muitas vezes, se tornem cansativas e pouco agradáveis, são necessárias para nos proporcionar previsibilidade e segurança. Grande parte dos nossos dias se repete com ações e efeitos similares, como acordar, trabalhar, estudar e dormir. Essas ações cotidianas podem ser comparadas às entradas de um algoritmo, enquanto os resultados dessas ações representam as saídas: mesmas entradas resultam em mesmas saídas. No entanto, na vida real, existem variáveis externas, como imprevistos e mudanças no ambiente, que podem alterar os resultados esperados, criando uma dinâmica que, apesar de previsível, é repleta de pequenas variações e desafios.

Nesta Unidade, vamos abordar as estruturas de repetição em lógica de programação, que podem ser comparadas a essa rotina diária e aos enredos dos filmes de comédia "Groundhog day" (1993), de terror "A morte te dá parabéns" (2017) e ação "No limite do amanhã" (2014). Assim como nos algoritmos, onde um bloco de código pode ser executado repetidamente até que uma condição seja satisfeita, esses filmes mostram personagens que revivem o mesmo dia várias vezes até resolverem um problema específico. Phil Connors (Bill Murray), em "Groundhog day", Tree Gelbman (Jessica Rothe), em "A morte te dá parabéns", e Major William "Bill" Cage (Tom Cruise), em "No limite do amanhã", todos enfrentam repetição constante de eventos, muito semelhante às nossas rotinas diárias. Eles precisam ajustar suas ações (entradas) para alterar os resultados (saídas) e, finalmente, quebrar o ciclo. Assim como na programação, onde ajustes são feitos para atingir o resultado desejado, esses personagens ajustam suas estratégias para superar os desafios impostos pelo ciclo repetitivo, refletindo a interação contínua entre a previsibilidade das rotinas e as variações impostas por fatores externos.

Uma estrutura de repetição é utilizada quando um trecho de algoritmo precisa ser repetido. O número de repetições pode ser fixo ou estar vinculado a uma condição, como discutimos anteriormente. Por exemplo, em um algoritmo para calcular a média de um aluno, esse precisa ser executado apenas uma vez. Mas, para calcular a média de 50 alunos, precisaríamos escrevê-lo 50 vezes, o que tornaria o código extremamente extenso e com linhas repetidas.

A solução do problema é utilizar a mesma sequência de comandos de leitura, escrita e cálculo para todos os alunos. Para cada aluno, ao final do processamento de suas notas, o algoritmo retrocede para o início do bloco de repetição, repetindo esse comportamento para cada aluno.

Quando agrupamos os comandos em uma estrutura ou laço de repetição, também estamos realizando um *loop* ou *looping*.

Existem três tipos de laços de repetição e vamos discutir cada um separadamente:

1. Repetição com variável de controle;
2. Repetição com teste no início;
3. Repetição com teste no final.

4.1. Repetição com variável de controle

A principal característica da estrutura de repetição com variável de controle é a sua utilização quando se sabe o número de vezes que um trecho de algoritmo deve ser repetido. Assim, a quantidade de repetições é definida *a priori*, logo no início da estrutura. Para fazer isso, precisamos utilizar uma variável exclusivamente declarada para controlar o número máximo de iterações.

A sintaxe básica da estrutura **para** em Portugol é a seguinte:

```
para variavel_de_controle de valor_inicial ate valor_final faca
        // Bloco de código a ser repetido
  fimpara
```

Onde:

- **variavel_de_controle:** variável que controla o número de repetições.
- **valor_inicial:** valor inicial da variável de controle.
- **valor_final:** valor final da variável de controle.
- Bloco de código: código que será executado repetidamente enquanto a variável de controle estiver dentro do intervalo definido.

Uma característica da variável de controle é que ela deve ser do tipo **inteiro**, pois, a cada repetição, ela é incrementada em uma unidade até atingir o limite estabelecido pelo valor final. Em Portugol, quando o valor inicial é maior que o valor final da variável de controle, a cada repetição, ela será decrementada, ou seja, ao invés de ser acrescida uma unidade, será retirada uma unidade.

Observe os trechos simplificados a seguir, que demonstram o comportamento da estrutura **para**:

```
para i de 1 ate 10 faca
        escreval(i)
  fimpara
```

No trecho anterior, o comando **escreval** será executado 10 vezes. Na 1ª repetição, a variável **i** receberá o valor 1, na 2ª, receberá 2, e assim por diante, até a 10ª repetição, onde seu valor passará a ser 10 e o laço será interrompido. Assim, os valores de **i** escritos na tela serão: 1, 2, 3, 4, 5, 6, 7, 8, 9 e 10.

```
para j de 1 ate 20 passo 2 faca
        escreval(j)
   fimpara
```

No trecho anterior, a variável de controle **j** inicia com valor 1 e o laço de repetição será executado até que o seu valor chegue a 20. Por padrão, a estrutura de repetição **para** incrementa em uma unidade o valor da variável de controle a cada iteração. Entretanto, incluímos o comando **passo** para determinar que o incremento a cada execução seja de 2 em 2. Assim, os valores de j serão: 2, 4, 6, 8, 10, 12, 14, 16, 18 e 20.

```
para cont de 10 ate 1 passo -1 faca
        escreval(cont)
   fimpara
```

No trecho anterior, o comando **escreval** também será executado 10 vezes. Entretanto, na 1ª repetição, a variável **cont** receberá o valor 10, na 2ª, receberá 9, e assim por diante, até a 10ª repetição, onde seu valor passará a ser 1 e o laço será interrompido. Observe que, nesse caso, há o decréscimo da variável de controle, pois o seu valor inicial é maior que o seu valor final e utilizamos o passo no valor de decréscimo -1. Assim, os valores de **cont** serão: 10, 9, 8, 7, 6, 5, 4, 3, 2 e 1.

Agora que entendemos o mecanismo de repetição, vamos para um exemplo completo. Vamos criar um algoritmo em Portugol que efetue a soma de todos os números ímpares, múltiplos de 3 e que se encontrem no conjunto dos números de 1 até 500.

```
1.  algoritmo “SomaImparesMultiplosDe3”
2.  // Este programa soma todos os números ímpares que são
múltiplos de 3 entre 1 e 500
3.  var
4.      i, soma: inteiro
5.  inicio
6.      soma <- 0
7.      para i de 1 ate 500 faca
8.          se (i mod 2 <> 0) e (i mod 3 = 0) entao
9.              soma <- soma + i
10.         fimse
11.     fimpara
12.     escreva(“A soma dos números é: “, soma)
13. fimalgoritmo
```

Linhas 1-2:

```
1. algoritmo “SomaImparesMultiplosDe3”
2. // Este programa soma todos os números ímpares que são múltiplos de
3 entre 1 e 500
```

Define o nome do algoritmo como “SomaImparesMultiplosDe3” e adiciona um comentário explicando que o programa soma todos os números ímpares múltiplos de 3 entre 1 e 500.

Linhas 3-4:

```
3. var
   4. i, soma: inteiro
```

Inicia a declaração das variáveis e declara as variáveis **i** e **soma** do tipo **inteiro**.

Linha 5:

```
5. inicio
```

Inicia o bloco principal do algoritmo.

Linha 6:

```
6. soma <- 0
```

Inicializa a variável soma com 0. Essa variável armazenará a soma dos números ímpares múltiplos de 3. Essa inicialização é necessária, pois precisamos ter um valor inicial para somá-lo aos demais a cada verificação.

Linha 7 :

```
7. para i de 1 ate 500 faca
```

Inicia a estrutura de repetição **para**, que fará com que a variável i assuma valores de 1 até 500.

Linha 8:

```
8. se (i % 2 <> 0) e (i % 3 = 0) entao
```

Dentro do bloco de repetição, verifica se o valor atual de **i** é ímpar (**i mod 2 <> 0**) e se é múltiplo de 3 (**i mod 3 = 0**).

Linha 9:

```
9. soma <- soma + i
```

Se ambas as condições forem verdadeiras, adiciona o valor atual de **i** à variável soma.

Linha 10:

```
10. fimse
```

Finaliza a estrutura condicional **se**.

Linha 11:

```
11. fimpara
```

Finaliza a estrutura de repetição **para**.

Linha 12:

```
12. escreva("A soma dos números ímpares múltiplos de 3 de 1 até 500 é: ",
    soma)
```

Exibe a soma dos números ímpares múltiplos de 3 de 1 até 500.

Linhas 13:

```
13. fimalgoritmo
```

Indica o fim do algoritmo.

Vamos, agora, elaborar um algoritmo em Portugol que receba a quantidade de alunos de uma turma e as notas semestrais de cada um, calcule e mostre a média anual de cada aluno e a média anual da turma.

```
1.  algoritmo “MediaAnualTurma”eddd
2.  // Este programa calcula a média anual de cada aluno e a média anual da turma
3.  var
4.      qtd_alunos, i, j: inteiro
5.      nota1, nota2, media_anual_aluno, soma_medias, media_anual_turma: real
6.  inicio
7.      escreva(“Digite a quantidade de alunos na turma: “)
8.      leia(qtd_alunos)
9.
10.     soma_medias <- 0
11.     para i de 1 ate qtd_alunos faca
12.         escreva(“Digite a nota do primeiro semestre do aluno “, i, “: “)
13.         leia(nota1)
14.         escreva(“Digite a nota do segundo semestre do aluno “, i, “: “)
15.         leia(nota2)
16.
17.         media_anual_aluno <- (nota1 + nota2) / 2
18.         escreval(“A média anual do aluno “, i, “ é: “, media_anual_aluno)
19.
20.         soma_medias <- soma_medias + media_anual_aluno
21.     fimpara
22.     media_anual_turma <- soma_medias / qtd_alunos
23.     escreva(“A média anual da turma é: “, media_anual_turma)
24. fimalgoritmo
```

Linhas 1-2:

```
1.  algoritmo “MediaAnualTurma”
2.  // Este programa calcula a média anual de cada aluno e a média anual da turma
```

Define o nome do algoritmo como “MediaAnualTurma” e adiciona um comentário explicando que o programa calcula a média anual de cada aluno e a média anual da turma.

Linhas 3 -5:

```
3.  var
4.      qtd_alunos, i, j: inteiro
5.      nota1, nota2, media_anual_aluno, soma_medias, media_anual_turma: real
```

Inicia a declaração das variáveis e declara as variáveis **qtd_alunos**, **i** e **j** do tipo **inteiro**. Também declara as variáveis **nota1**, **nota2**, **media_anual_aluno**, **soma_medias** e **media_anual_turma** do tipo **real**.

Linha 6:

```
6. inicio
```

Inicia o bloco principal do algoritmo.

Linhas 7-8:

```
7.      escreva(“Digite a quantidade de alunos na turma: “)
8.         leia(qtd_alunos)
```

Exibe a mensagem solicitando a quantidade de alunos na turma e recebe o valor digitado pelo(a) usuário(a) e armazena na variável **qtd_alunos**.

Linha 10:

```
10. soma_medias <- 0
```

Inicializa a variável **soma_medias** com 0. Essa variável armazenará a soma das médias anuais dos alunos.

Linha 11:

```
11. para i de 1 ate qtd_alunos faca
```

Inicia a estrutura de repetição **para**, que fará com que a variável i assuma valores de 1 até **qtd_alunos**.

Linhas 12-15:

```
12.         escreva(“Digite a nota do primeiro semestre do aluno “, i, “:
“)
13.            leia(nota1)
14.            escreva(“Digite a nota do segundo semestre do aluno “, i,
“: “)
15.            leia(nota2)
```

Solicita ao(à) usuário(a) que insira as notas do primeiro e do segundo semestre do aluno **i** e lê os valores para as variáveis **nota1** e **nota2**.

Linhas 17-18:

```
17.         media_anual_aluno <- (nota1 + nota2) / 2
18.            escreval(“A média anual do aluno “, i, “ é: “, media_anual_
aluno)
```

Calcula a média anual do aluno **i**, armazena o resultado na variável **media_anual_aluno** e exibe seu valor.

Linha 20:

```
20. soma_medias <- soma_medias + media_anual_aluno
```

Adiciona a média anual do aluno i à variável **soma_medias**.

```
21. fimpara
```

Finaliza a estrutura de repetição **para**.

Linhas 22-23:

```
22.     media_anual_turma <- soma_medias / qtd_alunos
23.     escreva("A média anual da turma é: ", media_anual_turma)
```

Calcula a média anual da turma, armazena o resultado na variável **media_anual_turma** e exibe seu valor.

Linha 24:

```
24. fimalgoritmo
```

Indica o fim do algoritmo.

No exemplo anterior, foi aplicado o conceito de variável acumuladora na variável **soma_medias** na **linha 10**. Como proposto, o algoritmo deveria ser capaz de calcular a média individual de cada aluno e a média geral da turma. Para isso, a variável **soma_medias** foi inicializada com o valor 0 e, a cada repetição, a média de cada aluno, presente na variável **media_anual_aluno**, foi somada à **soma_medias** na linha 20. Ao final, o valor acumulado na variável **soma_medias** foi dividido pela quantidade de alunos presente na variável **qtd_alunos**, obtendo a média anual em **media_anual_turma**.

Teste de mesa

Quando implementamos algoritmos que utilizam estruturas de repetição, é comum que múltiplas variáveis assumam valores diferentes a cada iteração, dificultando encontrar algum erro de lógica, caso exista. Em outras linguagens de programação, executamos o *debug* das instruções, utilizando ferramentas próprias que facilitam esse trabalho. Em algoritmos, utilizamos o teste de mesa.

O teste de mesa é uma técnica utilizada para validar e verificar a lógica de um algoritmo ou programa antes de sua implementação prática. Ele consiste em simular a execução do código manualmente, linha por linha, registrando o valor das variáveis e o fluxo de controle em uma tabela ou "mesa". Essa técnica é especialmente útil para identificar e corrigir erros lógicos, falhas de execução e inconsistências na lógica de programação.

Importância em lógica de programação

1. Detecção precoce de erros: o teste de mesa permite identificar erros lógicos e de execução antes mesmo de compilar e executar o programa. Isso economiza tempo e esforço na fase de depuração.
2. Compreensão da lógica do algoritmo: por meio do teste de mesa, programadores(as) e desenvolvedores(as) podem entender melhor como o algoritmo se comporta em diferentes cenários. Isso é crucial para garantir que o algoritmo funcione conforme o esperado.
3. Validação da correção: ao simular a execução do algoritmo com diferentes conjuntos de dados de entrada, o teste de mesa ajuda a validar se o algoritmo produz os resultados corretos em todas as situações.
4. Aprimoramento do pensamento lógico: realizar testes de mesa regularmente ajuda os(as) programadores(as) a desenvolverem uma abordagem mais meticulosa e estruturada para resolver problemas, aprimorando suas habilidades de pensamento lógico.
5. Documentação e comunicação: o teste de mesa fornece uma documentação clara e detalhada do comportamento do algoritmo. Isso é útil para comunicação entre membros da equipe de desenvolvimento e para futuros referenciais durante a manutenção do código.

Como realizar um teste de mesa

1. Preparação:
   - » Escolha um conjunto de dados de entrada representativo.
   - » Desenhe uma tabela com colunas para cada variável, o ponto de controle (linha do código) e a saída.
2. Simulação:
   - » Execute o algoritmo manualmente, linha por linha.
   - » Atualize os valores das variáveis e registre o fluxo de controle na tabela.
3. Análise:
   - » Compare os resultados obtidos na tabela com os resultados esperados.
   - » Identifique discrepâncias e determine a causa dos erros.

4. Correção:
   - » Corrija os erros identificados no algoritmo.
   - » Repita o teste de mesa até que o algoritmo funcione corretamente em todas as situações testadas.

Vamos realizar um teste de mesa para um algoritmo simples que calcula a soma dos números de 1 a 5.

Algoritmo em Portugol:

```
1.  algoritmo “SomaNumeros”
2.  var
3.      i, soma: inteiro
4.
5.  inicio
6.      soma <- 0
7.      para i de 1 ate 5 faca
8.          soma <- soma + i
9.      fimpara
10.     escreva(“A soma é: “, soma)
11. fimalgoritmo
```

Teste de mesa:

LINHA	i	SOMA	COMENTÁRIO
1			Declaração do algoritmo
2			Declaração das variáveis
3			Início do bloco principal
6		0	Inicializa soma com 0
7	1	0	Início do loop: i <- 1
8	1	1	soma <- soma + 1 (0 + 1)
7	2	1	Próxima iteração do loop: i <- 2
8	2	3	soma <- soma + 2 (1 + 2)
7	3	3	Próxima iteração do loop: i <- 3
8	3	6	soma <- soma + 3 (3 + 3)
7	4	6	Próxima iteração do loop: i <- 4
8	4	10	soma <- soma + 4 (6 + 4)
7	5	10	Próxima iteração do loop: i <- 5
8	5	15	soma <- soma + 5 (10 + 5)

9	6	15	Fim do loop: i > 5
10		15	Exibe o resultado: "A soma é: 15"
11			Fim do algoritmo

O exemplo de teste de mesa apresentado é bem detalhado em relação à última coluna de comentário, o que o torna extenso. À medida que você avançar nos estudos de lógica de programação e se tornar mais experiente, esses comentários podem ser suprimidos, mantendo apenas o registro dos valores das variáveis.

Exercícios

1. Faça um algoritmo que escreva a tabuada de multiplicação completa dos números de 1 a 10, utilizando uma estrutura de repetição.
2. Elabore um algoritmo que determine o valor de S, em que:

   S = 1/1 – 2/4 + 3/9 – 4/16 + 5/25 – 6/36 + ... - 10/100.
3. Uma rainha requisitou os serviços de um monge e disse-lhe que pagaria qualquer preço. O monge, necessitando de alimentos, perguntou à rainha se o pagamento poderia ser feito com grãos de trigo dispostos em um tabuleiro de xadrez, de tal forma que o primeiro quadro contivesse apenas um grão e os quadros subsequentes, o dobro do quadro anterior. A rainha considerou o pagamento barato e pediu que o serviço fosse executado, sem se dar conta de que seria impossível efetuar o pagamento. Faça um algoritmo para calcular o número de grãos que o monge esperava receber.
4. Construa um algoritmo que receba um número inteiro maior que 1, verifique se o número fornecido é primo ou não e mostre mensagem de número primo ou de número não primo. Um número é primo quando é divisível naturalmente apenas por 1 e por ele mesmo, ou seja, quando não há resto na divisão.
5. Construa um algoritmo que leia um conjunto de dados contendo altura e sexo (“M” para masculino e “F” para feminino) de 50 pessoas e, depois, calcule e escreva:
   - a maior e a menor altura do grupo;
   - a média de altura das mulheres;
   - o número de homens e a diferença porcentual entre eles e as mulheres.
6. Calcule o imposto de renda de um grupo de dez contribuintes, considerando que os dados de cada contribuinte (número de CPF, número de dependentes e renda mensal) são valores fornecidos pelo(a) usuário(a). Para cada contribuinte, será feito um desconto de 5% do salário mínimo por dependente. Observe que deve ser fornecido o valor atual do salário mínimo para que o algoritmo calcule

os valores corretamente. Os valores da alíquota para cálculo do imposto são:

RENDA LÍQUIDA	ALÍQUOTA
Até 2 salários mínimos (inclusive)	Isento
2 a 3 salários mínimos (inclusive)	5%
3 a 5 salários (inclusive)	10%
5 a 7 salários (inclusive)	15%
Acima de 7 salários mínimos	20%

4.2. Repetição com teste no início

A repetição com teste no início **enquanto** é utilizada quando não se sabe o número de vezes que um trecho do algoritmo deve ser repetido. Entretanto, esse tipo de estrutura também pode ser utilizado quando se conhece esse número, simulando o comportamento da estrutura **para**.

A principal característica dessa estrutura é o teste de uma condição realizado no início, ou seja, antes da primeira repetição. Enquanto a condição for satisfeita (verdadeira), os comandos da estrutura serão executados. Como é realizado um teste condicional, podem existir situações em que o teste condicional da estrutura de repetição, que fica no início, resulte em um valor falso logo na primeira comparação. Nesses casos, os comandos de dentro da estrutura de repetição não serão executados nenhuma vez.

A sintaxe básica da estrutura **enquanto** em Portugol é a seguinte:

```
enquanto condicao faca
        // Bloco de código a ser repetido
    fimenquanto
```

Onde **condicao** se refere a uma expressão lógica e/ou relacional e é verificada antes de cada iteração. Enquanto essa condição for verdadeira, o bloco de código dentro da estrutura **enquanto** será executado. Quando a condição se torna falsa, a repetição é interrompida. Se for falsa logo no início do laço (primeiro teste), a estrutura é interrompida, sem executar nenhuma vez.

Diferentemente da estrutura **para**, a estrutura **enquanto** não implementa um contador nativamente. Deve-se tomar o cuidado para que, em algum momento da repetição, a condição seja falsa, de acordo com as necessidades, para que essa seja interrompida, evitando o *loop* infinito.

Veja, a seguir, alguns trechos de algoritmos usando a estrutura **enquanto** que simulam o funcionamento da estrutura **para**:

```
cont <- 1
   enquanto cont <= 5 faca
       escreval("Repetição nº ", cont)
       cont <- cont + 1
   fimenquanto
```

No trecho anterior, foi estabelecida a condição para o laço ser repetido, enquanto o valor da variável **cont** fosse menor ou igual a 5. Foi tomado o cuidado de implementar uma instrução para que, a cada execução, o valor da variável **cont** fosse incrementado em 1 unidade (cont <- cont + 1), garantindo, assim, que, em algum momento, a condição se tornasse falsa, interrompendo a repetição.

```
x <- 1
y <- 5
   enquanto x < y faca
       x <- x + 2
       y <- y + 1
   fimenquanto
```

No trecho anterior, foi estabelecida a condição onde o laço será repetido até que o valor da variável **x** não fosse superior (maior) ao valor da variável **y**. Foi tomado o cuidado de implementar instruções para que, a cada execução, os valores das variáveis fossem incrementados (**x <- x + 2** e **y <- y + 1**). Caso contrário, a condição sempre seria verdadeira e o laço nunca seria interrompido.

Observe, na tabela a seguir, o teste de mesa simplificado realizado para ilustrar os valores das variáveis **x** e **y**, simultaneamente, a cada repetição.

X	1	3	5	7	9
Y	5	6	7	8	9

Podemos observar que o trecho se repetiu cinco vezes, sendo encerrado quando a condição (**x < y**) se tornou falsa, ou seja, **x** se tornou igual a **y**.

Vamos, agora, a um exemplo completo de um algoritmo que recebe as notas semestrais de cada aluno de uma turma, calcula e mostra a média anual de cada aluno e a média anual da turma. Não se sabe a quantidade total de alunos da turma. Portanto, foi implementado um laço de repetição com uma condição finalizadora para interromper sua execução após o último aluno.

```
1.  algoritmo “MediaAnualTurma”
2.  // Este programa calcula a média anual de cada aluno e a média anual
    da turma
3.  var
4.      cont, ms1, ms2, maluno, smedia, mturma: real
5.      opcao: caracter
6.  inicio
7.      smedia <- 0
8.      cont <- 0
9.      escreva(“Deseja calcular a média de um aluno? (S ou N): “)
10.     leia(opcao)
11.
12.     enquanto opcao = “S” faca
13.         cont <- cont + 1
14.         escreva(“Digite as médias semestrais do “, cont, “º aluno: “)
15.         leia(ms1, ms2)
16.         maluno <- (ms1 + ms2) / 2
17.         escreval(“A média anual do “, cont, “º aluno foi “, maluno)
18.         smedia <- smedia + maluno
19.         escreva(“Deseja calcular a média de mais um aluno? (S ou N):
    “)
20.         leia(opcao)
21.     fimenquanto
22.
23.     se (cont > 0) entao
24.         mturma <- smedia / cont
25.         escreva(“A média anual da turma foi “, mturma)
26.     fimse
27.
28. fimalgoritmo
```

Agora que você já está mais familiarizado com os algoritmos e já implementou vários códigos a partir dos exercícios, vamos explicar os códigos mais diretamente.

Na **linha 9** é perguntado ao(à) usuário(a) se deseja calcular a média de um aluno:

- Caso a resposta, armazenada na variável **opcao**, seja diferente de “S” (Sim) temos uma situação onde a estrutura de repetição não será executada nenhuma vez.
- Caso a resposta para a variável **opcao** seja “S”, o laço **enquanto** será iniciado para receber as notas e efetuar o cálculo da média, finalizando novamente com a pergunta que verifica se será calculada a média de mais algum aluno na **linha 19**.

Quando a variável **opcao** receber algum valor diferente de “S”, será verificado se foi calculada a média de pelo menos um aluno (**cont > 0**) na **linha 23** para prosseguir com o cálculo da média geral da turma na **linha 24** (**mturma <- smedia/cont**).

Exercícios

1. Anacleto tem 1,50 m de altura e cresce 2 cm por ano, enquanto Felisberto tem 1,10 m de altura e cresce 3 cm por ano. Construa um algoritmo que calcule e mostre quantos anos serão necessários para que Felisberto seja maior que Anacleto.

2. Um pecuarista deseja obter o maior peso entre os animais de seu rebanho, mas não sabe a quantidade total de animais. Faça um algoritmo que receba os pesos de todos os animais utilizando um laço de repetição e, após a leitura do último, diga quantos animais o rebanho possui e qual o maior peso. Considere como condição finalizadora, para encerrar a leitura, o peso do animal sendo 0.
3. Elabore um algoritmo que seja capaz de obter o quociente inteiro da divisão de dois números fornecidos, sem utilizar a operação de divisão (/).
4. Foi feita uma pesquisa de audiência de canal de TV em várias casas de uma certa cidade em um determinado dia. Para cada casa pesquisada, foi coletado o número do canal (5, 7, 10 ou 12) e o número de pessoas que o estavam assistindo naquela casa. Faça um algoritmo que receba esses dados da pesquisa, calcule e mostre a porcentagem de audiência de cada emissora. Termine a entrada dos dados da pesquisa informando o número do canal como sendo 0.
5. Faça um algoritmo que receba o valor do salário mínimo atual. Receba também a quantidade de quilowatts consumidos e o código do tipo de consumidor de um conjunto de consumidores de quantidade inicial desconhecida, calcule e mostre:
    - o valor de cada quilowatt, sabendo que o quilowatt custa 1/1000 do salário mínimo informado;
    - o valor a ser pago por cada consumidor, incluindo o acréscimo do tipo de consumidor. O acréscimo encontra-se na tabela a seguir:

CÓDIGO	TIPO	% ACRÉSCIMO SOBRE O GASTO
1	Residencial	5
2	Comercial	10
3	Industrial	15

    - o faturamento geral da empresa;
    - a quantidade de consumidores que pagam entre R$500,00 e R$1.000,00 de conta de energia.
    - Termine a entrada de dados do conjunto de consumidores com a quantidade de quilowatts igual a 0.

4.3. Repetição com teste no final

A repetição com teste no final **repita** é utilizada quando não se sabe o número de vezes que um trecho do algoritmo deve ser repetido, embora, também, possa ser utilizada quando se conhece esse número.

Da mesma forma que a estrutura **enquanto**, o **repita** baseia-se na análise de uma condição. Entretanto, a estrutura é repetida até a condição tornar-se verdadeira, ou seja, é executada enquanto a condição for falsa. Essa diferença entre o **enquanto** e o **repita** acontece, pois, no **repita** o teste condicional é realizado ao final da estrutura. Assim, os comandos do laço de repetição serão executados pelo menos uma vez, mesmo que o primeiro teste da estrutura seja verdadeiro.

A sintaxe básica da estrutura **repita** em Portugol é a seguinte:

```
repita
      // Bloco de comandos a ser repetido
   ate condicao
```

Veja, a seguir, alguns trechos de algoritmos usando a estrutura **repita** que simulam o funcionamento da estrutura **para**.

```
cont <- 1
   repita
       escreval("Repetição Nº ", cont)
       cont <- cont + 1
   ate cont > 5
```

No trecho anterior, foi estabelecida a condição finalizadora onde laço será repetido até que o valor da variável **cont** seja maior que 5. Foi tomado o cuidado de implementar uma instrução para que, a cada execução, o valor da variável **cont** fosse incrementado em 1 unidade (**cont <- cont + 1**), garantindo, assim, que, em algum momento, a condição se tornasse falsa, interrompendo a repetição.

```
x <- 1
   y <- 5
   repita
      x <- x + 2
      y <- y + 1
   ate x >= y
```

No trecho anterior, foi estabelecida a condição onde o laço será repetido até que o valor da variável **x** se torne superior (maior) ou igual ao valor da variável **y**. Para evitar um *loop* infinito, foram implementadas instruções para que, a cada execução, os valores das variáveis sejam incrementados (**x ← x + 2 e y ← y + 1**). Caso contrário, a condição sempre seria falsa e o laço nunca seria interrompido.

Observe, na tabela a seguir, o teste de mesa simplificado realizado para ilustrar os valores das variáveis x e y, simultaneamente, a cada repetição.

X	1	3	5	7	9	11
Y	5	6	7	8	9	10

Podemos observar que o trecho se repetiu seis vezes (1 x a mais que o mesmo código implementado usando a estrutura **enquanto**), sendo encerrado quando a condição (**x < y**) se tornou falsa, ou seja, **x** se tornou igual a **y**.

Vamos demonstrar o funcionamento do **repita** usando um jogo de adivinhação simples. Imagine uma brincadeira entre dois colegas, na qual um pensa um número qualquer e o outro deve realizar "chutes" até acertar o número imaginado. Como dica, a cada tentativa, será dito se o chute foi alto (número maior que o imaginado) ou baixo (número menor que o imaginado). Devemos elaborar um algoritmo dentro deste contexto que leia o número imaginado e os chutes, dando as dicas de chute alto ou baixo a cada tentativa, e, ao final, mostre quantos chutes foram realizados para se descobrir o número.

```
1.  algoritmo "Adivinhacao"
2.  // Este programa permite que o usuário adivinhe um número imagina-
    do
3.  var
4.      num, chute, ntent: inteiro
5.
6.  inicio
7.      ntent <- 0
8.      escreva("Digite um número para ser adivinhado: ")
9.      leia(num)
10.     repita
11.         escreva("Tente adivinhar o número: ")
12.         leia(chute)
13.         se (chute > num) entao
14.             escreval("Chutou alto!")
15.         senao
16.             se (chute < num) entao
17.                 escreval("Chutou baixo!")
18.             fimse
19.         fimse
20.         ntent <- ntent + 1
21.     ate chute = num
22.     escreva("Foram ", ntent, " tentativas para acertar o número")
23. fimalgoritmo
```

Linha 1:

```
algoritmo "Adivinhacao"
```

Define o nome do algoritmo como "Adivinhacao".

Linha 2:

```
// Este programa permite que o usuário adivinhe um número imaginado
```

Adiciona um comentário explicando que o programa permite que o(a) usuário(a) adivinhe um número imaginado.

Linhas 3-4:

```
var
       num, chute, ntent: inteiro
```

Declara as variáveis **num**, **chute**, e **ntent** do tipo **inteiro**.

- **num** será o número que deve ser adivinhado.
- **chute** armazenará os palpites do(a) usuário(a).
- **ntent** contará o número de tentativas.

Linha 6:

```
inicio
```

Inicia o bloco principal do algoritmo.

Linha 7:

```
ntent <- 0
```

Inicializa a variável **ntent** com 0, representando o número inicial de tentativas. Essa inicialização é necessária, pois, precisamos ter um valor inicial para incrementá-lo a cada tentativa.

Linhas 8-9:

```
escreva("Digite um número para ser adivinhado: ")
   leia(num)
```

Exibe a mensagem solicitando que o(a) usuário(a) digite um número para ser adivinhado, lê o número digitado pelo(a) usuário(a) e armazena na variável **num**.

Linha 10:

```
repita
```

Inicia a estrutura de repetição **repita**, que continuará executando até que a condição **chute = num** seja verdadeira.

Linhas 11-12:

```
escreva("Tente adivinhar o número: ")
   leia(chute)
```

Exibe a mensagem solicitando que o(a) usuário(a) tente adivinhar o número, lê o palpite do(a) usuário(a) e armazena na variável **chute**.

Linhas 13-19:

```
se (chute > num) entao
        escreval("Chutou alto!")
        senao
            se (chute < num) entao
                 escreval("Chutou baixo!")
            fimse
    fimse
```

Verifica se o palpite (**chute**) é maior que o número (**num**). Se for verdadeiro, executa a linha 14, exibindo a mensagem "Chutou alto!". Se a condição anterior for falsa, verifica se o palpite é menor que o número. Se for verdadeiro, exibe a mensagem "Chutou baixo!" e finaliza a estrutura condicional **se**.

Linha 20:

```
ntent <- ntent + 1
```

Incrementa a variável ntent em 1, contabilizando a tentativa.

Linha 21:

```
ate chute = num
```

Finaliza a estrutura de repetição repita quando o palpite for igual ao número (chute = num).

Linha 22:

```
escreva("Foram ", ntent, " tentativas para acertar o número")
```

Exibe o número total de tentativas realizadas pelo(a) usuário(a) para adivinhar o número.

Linha 23:

```
fimalgoritmo
```

Indica o fim do algoritmo.

As estruturas de repetição **enquanto** e **repita**, em Portugol, são usadas para executar um bloco de código várias vezes, mas elas têm diferenças importantes em como e quando a condição de término é verificada. Essas diferenças influenciam se e como elas podem ser intercambiáveis.

A estrutura **enquanto** verifica a condição no início do *loop*. Se a condição for falsa desde o início, o bloco de código não será executado nenhuma vez. A estrutura **repita** verifica a condição no final do *loop*. Isso garante que o bloco de código seja executado pelo menos uma vez, independentemente da condição inicial.

Na prática, se você tem certeza de que o bloco de código deve ser executado pelo menos uma vez, ambas as estruturas podem ser usadas de forma intercambiável. Por exemplo, o *loop* **repita** sempre executa o bloco pelo menos uma vez, então, se a condição inicial de um **enquanto** é sempre verdadeira na primeira verificação, elas podem ser intercambiáveis.

Entretanto, se há a possibilidade de a condição inicial ser falsa, o **enquanto** não executará o bloco de código, mas o **repita** executará pelo menos uma vez. Nesse caso, elas não são intercambiáveis.

Exercícios

1. Uma agência de publicidade quer prestar serviços somente para as maiores companhias (em números de funcionários) em cada uma das classificações: grande, média, pequena e microempresa. Para tal, deve ser fornecido pelo(a) usuário(a) um conjunto de dados com o código, o número de funcionários e o porte da empresa. Construa um algoritmo que liste o código da empresa com maiores recursos humanos dentro de cada categoria. Utilize como finalizador o código de empresa igual a 0.

2. No planeta Alpha vive a criatura Blobs, que come precisamente 1/2 de seu suprimento de comida disponível todos os dias. Escreva um algoritmo que leia a capacidade inicial de suprimento de comida em quilos, e calcule quantos dias passarão antes que os Blobs comam todo esse suprimento até atingir 1 quilo ou menos.
3. Escreva um algoritmo que leia um valor X e um valor Z (se Z for menor que X deve ser lido um novo valor para Z). Em seguida conte e mostre quantos números inteiros devemos somar em sequência, a partir do X (inclusive), (X, X+1, X+2, X+3, ...) para que a soma ultrapasse o valor de Z o mínimo possível.
4. Foi realizada uma pesquisa sobre algumas características físicas da população de uma certa região, a qual coletou os seguintes dados referentes a cada habitante para análise:
   - idade;

- sexo (“M” para masculino ou “F” para feminino);
- cor dos olhos (“A” para azuis, “V” para verdes ou “C” para castanhos);
- cor dos cabelos (“L” para loiros, “C” para castanhos ou “P” para pretos).

Faça um algoritmo que determine e escreva:

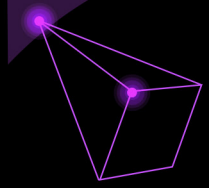
- a maior idade dos habitantes;
- a porcentagem, entre apenas os indivíduos do sexo masculino, cujo sexo é masculino e a idade está entre 18 e 35 anos, inclusive;
- a porcentagem, entre o total de indivíduos, cujo sexo seja feminino, que idade esteja entre 18 e 35 anos, inclusive, e que tenha olhos e cabelos castanhos.

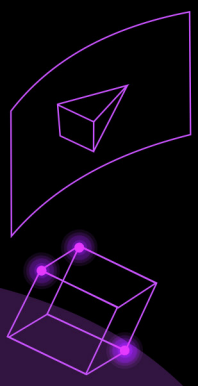
O final do conjunto de habitantes é reconhecido pelo valor - 1 para a idade.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Unidade V
Vetores e Matrizes





Unidade V - Vetores e Matrizes

Em muitas situações do mundo real, os tipos primitivos **caracter**, **inteiro**, **real** e **lógico** para representação de dados em algoritmos são escassos, tornando necessária a criação de novos tipos ou tipos "construídos" à medida em que se fazem necessários.

Fazendo uma analogia de variáveis representadas por gavetas, anteriormente, as gavetas (variáveis) podiam comportar apenas um objeto (valor) por vez, do mesmo tipo de foi criada. Nesta Unidade, esse conceito será atualizado, permitindo que uma gaveta (variável) comporte não apenas um único valor, mas um conjunto de valores, desde que ela seja dividida em compartimentos, aproximando-se assim do conceito utilizado na nossa realidade. Damos o nome de estruturas de dados para esse novo tipo de variável.

Uma variável simples é interpretada como um único elemento, enquanto uma estrutura de dados é caracterizada como um conjunto. Quando uma determinada estrutura de dados é composta de variáveis do mesmo tipo primitivo, temos um conjunto homogêneo de dados. Iremos abordar duas estruturas homogêneas: vetores e matrizes. Entretanto, existem outras dezenas de estruturas de dados homogêneas (e também heterogêneas - tipos diferentes de dados) com aplicações avançadas e específicas em programação. Esse assunto, você estudará à medida que seguir nos estudos. Os vetores são conhecidos também como variáveis compostas unidimensionais.

5.1. Vetores unidimensionais

Para entendermos o conceito de variáveis compostas unidimensionais, imaginemos um edifício com um número finito de andares. O edifício representa uma estrutura de dados e seus andares são partições dessa estrutura. Os andares são uma segmentação direta do prédio, compondo o que chamamos de estrutura composta unidimensional, ou seja, uma estrutura de dados que possui apenas uma dimensão. Essas estruturas também são conhecidas como vetores ou *arrays*: um conjunto de valores do mesmo tipo que possuem o mesmo identificador (variável) e são alocadas sequencialmente na memória. Como as variáveis têm o mesmo nome, o que os distingue é um índice que referencia sua localização dentro da estrutura.

Em Portugol, a declaração de vetores é feita especificando o nome do vetor, seu tipo e o número de elementos que ele pode armazenar. A sintaxe para declarar um vetor em Portugol é a seguinte:

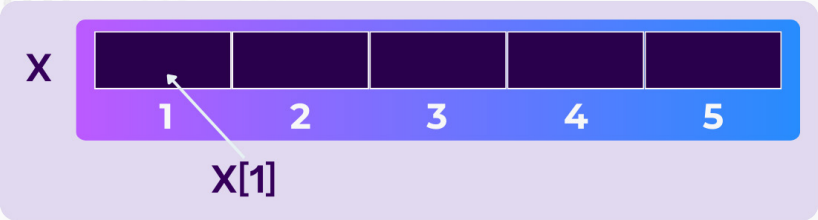
```
var
        nome_vetor: vetor[1..n] de tipo_dado
```

Onde:

- **nome_vetor**: nome da variável composta unidimensional. Segue as mesmas regras de criação de identificadores.
- **1..n**: intervalo que define o tamanho do vetor, onde n é o número de elementos.
- **tipo_dado**: tipo dos dados que o vetor armazenará (**inteiro**, **real**, **caracter** ou **lógico**).

Supondo que precisamos declarar um vetor **x** de cinco posições do tipo primitivo **inteiro**, ficaria da seguinte forma:

```
var
      x: vetor[1..5] de inteiro
```



O vetor **x** é dividido em cinco "compartimentos" e cada um desses é referenciado pelo seu respectivo índice (1, 2, 3, 4 ou 5). A 1ª posição do vetor é identificada pela composição do identificador do vetor (**x**) e o número do índice (1) entre colchetes ([e]), sendo **x[1]**.

Para atribuirmos valores diretamente para um vetor, devemos, obrigatoriamente, informar em qual das posições o valor será armazenado, ou seja, devemos informar o índice do vetor.

Exemplos:

```
1.  algoritmo "InicializacaoVetor"
2.  var
3.      x: vetor[1..5] de inteiro
4.
5.  inicio
6.      x[1] <- 10
7.      x[2] <- 20
8.      x[3] <- 30
9.      x[4] <- 40
10.     x[5] <- 50
11. fimalgoritmo
```

Utilizando a analogia da gaveta, além de especificarmos em qual gaveta o objeto será guardado, devemos também informar em qual compartimento da gaveta esse objeto deverá ser colocado: é necessário informar o nome do vetor e a posição que receberá o valor atribuído, seja na atribuição direta de valor (como no exemplo) ou recebendo uma valor informado pelo(a) usuário(a).

Observe que, no exemplo acima, são utilizadas cinco linhas para preencher todo o vetor **x**. Podemos utilizar uma estrutura de repetição **para** com o objetivo de melhorar e tornar mais flexível o código. No exemplo a seguir, é mostrada essa modificação.

```
1.  algoritmo "InicializacaoVetor"
2.  var
3.      x: vetor[1..5] de inteiro
4.      i: inteiro
5.
6.  inicio
7.      para i de 1 ate 5 faca
8.          x[i] <- i * 10
9.      fimpara
10. fimalgoritmo
```

Podemos, então, escalar esse código, alterando o valor 5 da **linha 7** para, por exemplo, 1.000.000 sem qualquer esforço extra, diferentemente do exemplo anterior, que teríamos que repetir as linhas.

Como os valores do vetor seguem uma lógica (10 em 10 até 50), basta multiplicarmos o valor de **i** por 10, a cada iteração, na **linha 8** para preencher o vetor com os mesmos valores do exemplo anterior.

Podemos pedir ao(à) usuário(a) para digitar os valores do vetor e, logo em seguida, exibí-los, também utilizando a estrutura **para**, como no exemplo a seguir.

```
1.  algoritmo "InicializaExibiVetor"
2.  var
3.      x: vetor[1..5] de inteiro
4.      i: inteiro
5.
6.  inicio
7.      // Preenchendo o vetor x
8.      para i de 1 ate 5 faca
9.          escreva("Digite o ", i, "º valor de x: ")
10.         leia(x[i])
11.     fimpara
12.
13.     // Mostrando os valores do vetor x
14.     escreval("Os valores de x são:")
15.     para i de 1 ate 5 faca
16.         escreval("x[", i, "] = ", x[i])
17.     fimpara
18. fimalgoritmo
```

Observe, na Tabela a seguir, o teste de mesa das **linhas 8-11** do trecho anterior. Vamos entender melhor a aplicação de vetores em exemplos práticos.

PASSO	i	INSTRUÇÃO	ENTRADA USUÁRIO	VETOR X
1	1	escreva(”Digite o 1° valor de x: “) leia (x[1])	10	x [1] =10 x [2]= x [3]= x [4]= x [5]=
2	2	escreva(”Digite o 2° valor de x: “) leia (x[2])	20	x [1] =10 x [2]=20 x [3]= x [4 = x [5]=
3	3	escreva(”Digite o 3° valor de x: “) leia (x[3])	30	x [1] =10 x [2]=20 x [3]=30 x [4] = x [5]=
4	4	escreva(”Digite o 4° valor de x: “) leia (x[4])	40	x [1] =10 x [2]=20 x [3]=30 x [4 =40 x [5]=
5	5	escreva(”Digite o 5° valor de x: “) leia (x[5])	50	x [1] =10 x [2]=20 x [3]=30 x [4 =40 x [5]=50

Imagine um algoritmo que receba as médias finais de uma classe com 10 alunos, calcule e mostre a média geral da turma. Com os conhecimentos que você adquiriu neste Curso, até o momento, seria possível implementar esse algoritmo. Entretanto, após calcular a média da turma, se adicionarmos a funcionalidade de mostrar a quantidade de médias dos alunos que foram acima da média da turma, você precisaria utilizar um vetor para armazenar as médias dos alunos. Caso contrário, teria que recalcular todas as médias, pedindo ao(à) usuário(a) para informar novamente as notas. Veja, a seguir, como ficaria esse algoritmo.

```
1.  algoritmo “MediaTurma”
2.  var
3.      cont, qAcima: inteiro
4.      mediasAlunos: vetor[1..10] de real
5.      somaMedias, mediaTurma: real
6.
7.  inicio
8.      somaMedias <- 0
9.      qAcima <- 0
10.
11.     // Leitura das médias dos alunos e cálculo da soma
    das médias
12.     para cont de 1 ate 10 faca
13.         escreva(“Digite a média do “, cont, “º aluno: “)
14.         leia(mediasAlunos[cont])
15.         somaMedias <- somaMedias + mediasAlunos[cont]
16.     fimpara
17.
18.     // Cálculo da média da turma
19.     mediaTurma <- somaMedias / 10
20.     escreval(“A média geral da turma foi “, mediaTurma)
21.
22.     // Contagem dos alunos com média acima da média da
    turma
23.     para cont de 1 ate 10 faca
24.         se mediasAlunos[cont] > mediaTurma entao
25.             qAcima <- qAcima + 1
26.         fimse
27.     fimpara
28.
29.     escreval(qAcima, “ alunos obtiveram média acima da
    média da turma”)
30. fimalgoritmo
```

Linha 2-3:

```
var
    cont, qAcima: inteiro
```

Declara as variáveis **cont** e **qAcima** do tipo **inteiro**.

- **cont** será usado como índice nos *loops*.
- **qAcima** contará quantos alunos têm média acima da média da turma.

Linha 4:

```
mediasAlunos: vetor[1..10] de real
```

Declara o vetor **mediasAlunos** com 10 elementos do tipo **real**.

Linha 5:

```
somaMedias, mediaTurma: real
```

Declara as variáveis **somaMedias** e **mediaTurma** do tipo **real**.

- **somaMedias** acumula a soma das médias dos alunos.
- **mediaTurma** armazenará a média geral da turma.

Linha 8:

```
somaMedias <- 0
```

Inicializa a variável **somaMedias** com 0. Essa inicialização é necessária, pois precisamos ter um valor inicial para somá-lo às demais médias a cada aluno.

Linha 9:

```
qAcima <- 0
```

Inicializa a variável **qAcima** com 0.

Linha 12:

```
para cont de 1 ate 10 faca
```

Inicia a estrutura de repetição para receber as notas dos 10 alunos.

Linha 13:

```
escreva("Digite a média do ", cont, "º aluno: ")
leia(mediasAlunos[cont])
```

Solicita ao(à) usuário(a) que digite a média do aluno cont e lê o valor digitado pelo(a) usuário(a) e armazena no vetor **mediasAlunos[cont]**.

Linha 15:

```
somaMedias <- somaMedias + mediasAlunos[cont]
```

Adiciona o valor lido à variável **somaMedias**.

Linha 19:

```
mediaTurma <- somaMedias / 10
```

Calcula a média da turma dividindo **somaMedias** por 10.

Linha 20:

```
escreval("A média geral da turma foi ", mediaTurma)
```

Exibe a média geral da turma.

Linha 23:

```
para cont de 1 ate 10 faca
```

Inicia a estrutura de repetição para contar as médias dos alunos superiores à média da turma.

Linha 24:

```
se mediasAlunos[cont] > mediaTurma entao
```

Verifica se a média de cada aluno é maior que a média da turma.

Linha 25:

```
qAcima <- qAcima + 1
```

Se a condição for verdadeira, incrementa **qAcima**.

Linha 29:

```
escreval(qAcima, " alunos obtiveram média acima da média da turma")
```

Exibe a quantidade de alunos que obteve média acima da média da turma.

Agora, vamos construir um algoritmo que preencha automaticamente um vetor de 100 elementos inteiros, colocando 1 na posição corresponde a um índice par e 0 a um índice ímpar.

```
1.  algoritmo “PreencherVetor100”
2.  var
3.      v: vetor[1..100] de inteiro
4.      i: inteiro
5.
6.  inicio
7.      // Preenchendo o vetor com 1 em índices pares e 0 em
    índices ímpares
8.      para i de 1 ate 100 faca
9.          se (i mod 2 = 0) entao
10.             v[i] <- 1
11.         senao
12.             v[i] <- 0
13.         fimse
14.     fimpara
15.
16.     // Mostrando os valores do vetor
17.     para i de 1 ate 100 faca
18.         escreval(“v[“, i, “] = “, v[i])
19.     fimpara
20. fimalgoritmo
```

Vamos direto às linhas mais importantes do algoritmo:

Linha 8:

```
para i de 1 ate 100 faca
```

Inicia a estrutura de repetição para, que irá de 1 até 100.

Linha 9:

```
se (i mod 2 = 0) entao
```

Verifica se o índice **i** é par (**i mod 2 = 0**), ou seja, se o resto da divisão de **i** por 2 é igual a 0.

Linha 10:

```
v[i] <- 1
```

Se o índice **i** for par, coloca 1 na posição **i** do vetor.

Linha 12:

```
v[i] <- 0
```

Caso contrário, ou seja, se o índice **i** for ímpar, coloca 0 na posição **i** do vetor.

Linha 17:

```
para i de 1 ate 100 faca
```

Inicia a estrutura de repetição **para**, que irá de 1 até 100, para exibir os valores do vetor.

Linha 18:

```
escreva("v[", i, "] = ", v[i])
```

Exibe o valor de cada elemento do vetor.

Exercícios

1. Desenvolva um algoritmo que permita a leitura de um vetor de 30 números inteiros e gere um segundo vetor com os mesmos dados, só que de maneira invertida, ou seja, o primeiro elemento ficará na última posição, o segundo na penúltima posição e assim por diante.
2. Construa um algoritmo que leia uma série de 50 notas de uma turma de alunos e calcule quantas são acima, abaixo e iguais à média dessas notas.
3. Crie um algoritmo que leia um vetor de 10 elementos. A seguir, troque o 1° elemento do vetor com o último, o 2° com o penúltimo, e assim por diante, até a troca do 5° com o 6° elemento do vetor. Ao final, escreva os valores do vetor modificado.
4. Elabore um algoritmo que receba do(a) usuário(a) 10 letras diferentes do alfabeto e as armazene em um vetor. Em seguida, receba uma letra qualquer, também informada pelo(a) usuário(a), e realize uma busca no vetor. Caso a encontre, mostre em que posição ela está; caso não a encontre, mostre uma mensagem.
5. Um apostador da Mega Sena realizou uma aposta e deseja saber quantos números ele acertou. Faça um algoritmo que receba os números sorteados no concurso da loteria e a aposta realizada e mostre quantos pontos ele fez.

5.2. Vetores bidimensionais

A compreensão do conceito de variáveis compostas multidimensionais, também chamadas de matrizes, pode ser realizada por meio da analogia do edifício, utilizada anteriormente para entender os vetores. Além do acesso pelo elevador até um determinado andar, temos também a divisão desse andar em apartamentos. Para chegar até um desses apartamentos, não basta só o número do andar, precisamos também do número do apartamento.

Como vimos anteriormente, os vetores unidimensionais têm como principal característica a necessidade de apenas um índice para endereçamento – são estruturas com apenas uma dimensão. Já os vetores bidimensionais possuem duas dimensões: também são conhecidas como matrizes e precisam de mais de um índice, como no caso do edifício dividido em andares, que por sua vez, são divididos em apartamentos. Os vetores são capazes de possuir um número infinito de dimensões, mas trabalharemos apenas com vetores de duas dimensões neste Curso: vetores bidimensionais – divididos em linhas e colunas. Por convenção, vamos adotar, nesta Unidade, o termo matriz para se referir a um vetor bidimensional. Isso será útil para a continuidade dos estudos, pois diversos autores o adotam em convenção.

Em Portugol, a declaração de matrizes é feita especificando o nome da matriz, seu tipo e o número de linhas e colunas. A sintaxe para declarar uma matriz, em Portugol, é a seguinte:

```
var
      nome_matriz: vetor[1..n, 1..m] de tipo_dado
```

Onde:

- **nome_matriz**: nome da variável composta bidimensional. Segue as mesmas regras de criação de identificadores.
- **1..n**: intervalo que define o número de linhas da matriz.
- **1..m**: intervalo que define o número de colunas da matriz.
- **tipo_dado**: tipo dos dados que a matriz armazenará (**inteiro**, **real**, **caracter** ou **lógico**).

Supondo que precisamos declarar uma matriz bidimensional **x** do tipo **inteiro**, onde o tamanho da 1ª dimensão (linha) é 3 e o da 2ª dimensão (coluna) é 5, ou seja, uma matriz 3x5, ficaria da seguinte forma:

```
var
      x: vetor[1..3, 1..5] de inteiro
```

A matriz **x** é dividida em 15 “compartimentos” e cada um desses é referenciado pelos seus respectivos índices de linha e coluna. A 1ª posição da matriz é identificada pela composição do identificador da matriz e os respectivos números dos índices da linha e coluna entre colchetes ([e]), sendo **x[1,1]** e a 14ª posição é identificada por **x[3,4]**.

As ações de atribuir, preencher ou mostrar todos os valores de uma matriz segue o mesmo princípio do vetores, mas com duas diferenças principais:

1. Uma posição da matriz é identificada por dois índices, linha e coluna; portanto,
2. Precisamos implementar duas estruturas de repetição para aninhadas, ou seja, uma dentro da outra para percorrer todas as posições da matriz.

Como falamos anteriormente, precisamos de duas estruturas de repetição e, consequentemente, duas variáveis de controle diferentes, pois a matriz é bidimensional. A regra é: para cada dimensão, temos uma repetição. Isso se faz necessário, pois, por exemplo, para cada linha, existem cinco colunas no exemplo anterior: ao terminar de preencher ou mostrar todos os valores da 1ª linha, serão mostrados os valores na 2ª linha, reiniciando na 1ª coluna e assim por diante.

Veja, a seguir, um exemplo de algoritmo que recebe os valores digitados pelo(a) usuário(a) e os armazena em uma matriz 3x2 e depois mostra esses valores.

```
1.  algoritmo “RecebeMostraMatriz”
2.  var
3.      matriz: vetor[1..3, 1..2] de inteiro
4.      i, j: inteiro
5.
6.  inicio
7.      // Leitura dos valores da matriz
8.      para i de 1 ate 3 faca
9.          para j de 1 ate 2 faca
10.             escreva(“Digite o valor da matriz[“, i, “][“, j,
“]: “)
11.             leia(matriz[i, j])
12.         fimpara
13.     fimpara
```

```
14.
15.     // Exibição dos valores da matriz
16.     escreval("Valores da matriz:")
17.     para i de 1 ate 3 faca
18.         para j de 1 ate 2 faca
19.             escreva(matriz[i, j], " ")
20.         fimpara
21.         escreval("")
22.     fimpara
23. fimalgoritmo
```

Linha 3:

```
matriz: vetor[1..3, 1..2] de inteiro
```

Declara a matriz com três linhas e duas colunas do tipo **inteiro**.

Linha 4:

```
i, j: inteiro
```

Declara as variáveis **i** e **j** do tipo **inteiro**, que serão usadas como índices nos *loops*. **i** para as linhas e **j** para as colunas,

Linha 8:

```
para i de 1 ate 3 faca
```

Inicia a estrutura de repetição **para**, que irá de 1 até 3 para percorrer as linhas da matriz.

Linha 9:

```
para j de 1 ate 2 faca
```

Inicia a estrutura de repetição **para**, que irá de 1 até 2 para percorrer as colunas da matriz.

Linha 10:

```
escreva("Digite o valor para matriz[", i, "][", j, "]: ")
```

Exibe a mensagem solicitando ao(à) usuário(a) que digite o valor para a posição **[i, j]** da matriz.

Linha 11:

```
leia(matriz[i, j])
```

Lê o valor digitado pelo(a) usuário(a) e armazena na posição [i, j] da matriz.

Linha 17:

```
para i de 1 ate 3 faca
```

Inicia a estrutura de repetição **para**, que irá de 1 até 3 para percorrer as linhas da matriz.

Linha 18:

```
para j de 1 ate 2 faca
```

Inicia a estrutura de repetição **para**, que irá de 1 até 2 para percorrer as colunas da matriz.

Linha 19:

```
escreva(matriz[i, j], " ")
```

Exibe o valor da posição **[i, j]** da matriz, dentro da mesma linha da matriz, separando-o com um espaço em branco.

Linha 21:

```
escreval("")
```

Exibe uma nova linha para formatar a saída, garantindo que cada linha da matriz seja exibida separadamente.

Observe, na tabela a seguir, o teste de mesa das **linhas 8-12** do algoritmo citado anteriormente.

PASSO	i	j	INSTRUÇÃO	ENTRADA USUÁRIO	MATRIZ
1	1	1	escreva (“Digite o valor para matriz [1] [1]: “) leia (matriz[1,1])	8	matriz[1, 1] = 8 matriz[1, 2] = matriz[2, 1] = matriz[2,2] = matriz[3,1] = matriz[3,2] =
2	1	2	escreva (“Digite o valor para matriz [1] [2]: “) leia (matriz[1,2])	4	matriz[1, 1] = 8 matriz[1, 2] = 4 matriz[2, 1] = matriz[2,2] = matriz[3,1] = matriz[3,2] =
3	2	1	escreva (“Digite o valor para matriz [2] [1]: “) leia (matriz[2,1])	12	matriz[1, 1] = 8 matriz[1, 2] = 4 matriz[2, 1] = 12 matriz[2,2] = matriz[3,1] = matriz[3,2] =
4	2	2	escreva (“Digite o valor para matriz [2] [2]: “) leia (matriz[2,2])	2	matriz[1, 1] = 8 matriz[1, 2] = 4 matriz[2, 1] = 12 matriz[2,2] = 2 matriz[3,1] = matriz[3,2] =
5	3	1	escreva (“Digite o valor para matriz [3] [1]: “) leia (matriz[3,1])	90	matriz[1, 1] = 8 matriz[1, 2] = 4 matriz[2, 1] = 12 matriz[2,2] = 2 matriz[3,1] = 90 matriz[3,2] =
6	3	2	escreva (“Digite o valor para matriz [3] [2]: “) leia (matriz[3,2])	-32	matriz[1, 1] = 8 matriz[1, 1] = 4 matriz[1, 1] = 12 matriz[1, 1] = 2 matriz[1, 1] = 90 matriz[1, 1] = -32

Para praticarmos o uso de matrizes para armazenamento e processamento de dados, vamos utilizar como exemplo um algoritmo que recebe valores numéricos digitados pelo(a) usuário(a) e preenche uma matriz M (2x2), calcula e mostra a matriz R, resultante da multiplicação dos elementos de M pelo seu maior elemento. Para isso, o algoritmo terá quatro passos:

1. Receber os dados da matriz M;
2. Encontrar o maior valor da matriz M;
3. Preencher a matriz R com a multiplicação de M pelo seu maior valor;
4. Mostrar os valores da matriz resultante M.

```
1.  algoritmo “MatrizOperacoes”
2.  var
3.      M, R: vetor[1..2, 1..2] de inteiro
4.      maior, i, j: inteiro
5.
6.  inicio
7.      maior <- 0
8.      escreval(“Digite os valores da matriz M:”)
9.
10.     // Passo 1: Leitura dos valores da matriz M
11.     para i de 1 ate 2 faca
12.         para j de 1 ate 2 faca
13.             escreva(i, “ª linha “, j, “ª coluna: “)
14.             leia(M[i, j])
15.         fimpara
16.     fimpara
17.
18.     // Passo 2: Encontrar o maior valor na matriz M
19.     para i de 1 ate 2 faca
20.         para j de 1 ate 2 faca
21.             se (M[i, j] > maior) entao
22.                 maior <- M[i, j]
23.             fimse
24.         fimpara
25.     fimpara
26.
27.     // Passo 3: Calcular a matriz R com os valores de M
    multiplicados pelo maior valor
28.     para i de 1 ate 2 faca
29.         para j de 1 ate 2 faca
30.             R[i, j] <- maior * M[i, j]
31.         fimpara
32.     fimpara
33.
34.     // Passo 4: Exibir os valores da matriz R
35.     escreval(“Valores da matriz R:”)
36.     para i de 1 ate 2 faca
37.         para j de 1 ate 2 faca
38.             escreval(i, “ª linha “, j, “ª coluna da ma-
    triz: “, R[i, j])
39.         fimpara
40.     fimpara
41. fimalgoritmo
```

Vamos direto às linhas mais importantes do algoritmo:

Linha 7:

```
maior <- 0
```

Inicializa a variável **maior** com 0. Essa inicialização é necessária pois precisamos ter um valor de referência para comparar os valores da matriz **M** e atualizar a variável **maior** com o maior valor a cada verificação.

Linha 11-12:

```
para i de 1 ate 2 faca
       para j de 1 ate 2 faca
```

Inicia as estruturas de repetição **para**, para percorrer as linhas e colunas da matriz **M** preenchendo com os valores informados pelo(a) usuário(a).

Linha 13-14:

```
escreva(i, "ª linha ", j, "ª coluna: ")
   leia(M[i, j])
```

Exibe a mensagem solicitando ao(à) usuário(a) que digite o valor , lê o valor digitado pelo(a) usuário(a) e armazena na posição **[i,j]** da matriz **M**.

Linha 19-20:

```
para i de 1 ate 2 faca
       para j de 1 ate 2 faca
```

Inicia as estruturas de repetição **para**, para percorrer as linhas e colunas da matriz **M** para descobrir qual o maior valor.

Linha 21-22:

```
se (M[i, j] > maior) entao
       maior <- M[i, j]
```

Verifica se o valor na posição **[i,j]** da matriz **M** é maior que o valor atual de **maior**. Se sim, atualiza **maior** com o valor de **M[i,j]**.

Linha 28-32:

```
para i de 1 ate 2 faca
       para j de 1 ate 2 faca
           R[i, j] <- maior * M[i, j]
       fimpara
   fimpara
```

Inicia a estrutura de repetição **para**, para percorrer as linhas e colunas da matriz R calculando o valor de **R[i,j]** obtido pela multiplicação de **maior** pelo valor de **M[i,j]**.

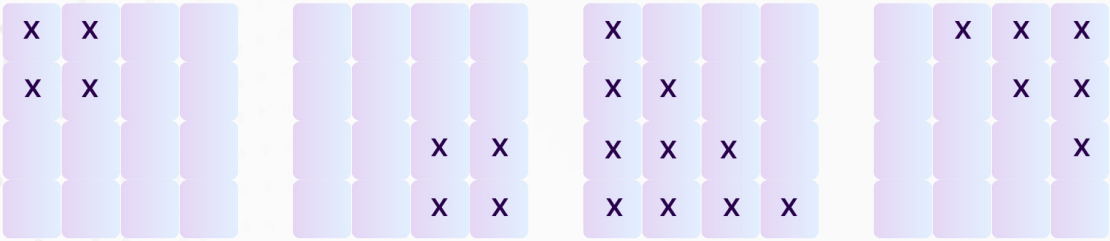
Linha 36-40:

```
para i de 1 ate 2 faca
        para j de 1 ate 2 faca
            escreval(i, "ª linha ", j, "ª coluna da matriz: ", R[i, j])
        fimpara
    fimpara
```

Percorre a matriz R, mostrando os valores obtidos da multiplicação.

Exercícios

1. Crie um algoritmo que receba os valores de uma matriz D 5x5. Considere que o(a) usuário(a) não informará valores duplicados. A seguir, leia um número X qualquer e escreva uma mensagem indicando se o valor de X existe ou não na matriz.
2. Escreva um algoritmo que leia uma matriz G 5x5 e preencha automaticamente dois vetores, de cinco elementos, SL e SC que contenham, respectivamente, as somas das linhas e das colunas da matriz G. Ao final, escreva os valores dos vetores SL e SC.
3. Construa um algoritmo que leia uma matriz M 6x6. Em seguida, copie para um vetor V de tamanho 6 o menor elemento de cada linha da matriz M, em suas respectivas posições. Após o término da cópia, mostre os valores do vetor V.
4. Elabore um algoritmo que leia uma matriz A 4x4, calcule e escreva as somas dos elementos marcados com o X nos esquemas a seguir.



5. Em uma loteria, a aposta é realizada em cartelas divididas em cinco linhas e cinco colunas preenchidas com os números de 1 a 25. Em cada concurso é realizado o sorteio de cinco números dentro do intervalo da cartela (1 a 25). Faça um algoritmo que receba uma aposta codificada em uma matriz APOSTA 5x5 preenchida pelo jogador de forma que, onde for informado o número 1, identifique os valores apostados, e o número 0 identifique os valores que não foram apostados. Em seguida, receba os valores sorteados em um vetor SORTEIO de cinco elementos, conte e mostre quantos pontos foram obtidos pelo jogador em sua aposta. Implemente o recurso de não permitir que o jogador aposte em mais de cinco números da cartela.

CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Unidade VI
Funções

Unidade VI - Funções

Até agora, aprendemos a construir algoritmos em apenas um bloco, o bloco principal, construído com a estrutura **algoritmo...fimalgoritmo**. Entretanto, em diversos casos, precisamos tratar a resolução de problemas computacionais mais complexos de forma modularizada. Isso significa que dividimos o problema em partes e propomos a solução dessas partes.

Modularizar significa dividir algo em partes menores, no nosso caso, os algoritmos. As funções, uma das formas de se modularizar algoritmos, são blocos de instruções que realizam tarefas específicas. Esses comandos são carregados uma vez e podem ser executados quantas vezes forem necessárias.

Dividir o problema em pequenas tarefas resulta em algoritmos menores, mais organizados e reutilizáveis. Geralmente, os algoritmos são executados linearmente, uma linha após a outra até o fim. No entanto, a utilização de funções permite a realização de desvios na execução dos algoritmos. Esses desvios são efetuados quando uma função é chamada pelo bloco principal, alterando o fluxo de execução conforme necessário.

Para esse conceito não ficar muito abstrato, vamos utilizar o exemplo a seguir para melhor entendimento.

```
1.  algoritmo “CalculoSalario”
2.  var
3.      salario, aum, novo_sal: real
4.
5.  funcao calculo_aum(sal: real): real
6.  var
7.      perc, valor: real
8.  inicio
9.      escreva(“Digite o percentual de aumento: “)
10.     leia(perc)
11.     valor <- sal * perc / 100
12.     retorne valor
13. fimfuncao
14.
15. inicio
16.     escreva(“Digite o salário atual do funcionário: “)
17.     leia(salario)
18.     aum <- calculo_aum(salario) //Desvio do fluxo normal de execu-
ção
19.     novo_sal <- salario + aum
20.     escreval(“O novo salário do funcionário é “, novo_sal)
21. fimalgoritmo
```

O objetivo do algoritmo anterior é calcular o novo salário de um funcionário acrescido de um aumento. Para resolver o problema foi criado o algoritmo principal (**linhas 15-21**) e uma função (**linhas 5-13**). O bloco principal é executado linearmente até a **linha 18**. A partir desse ponto, existe um desvio do fluxo normal de execução do algoritmo para realizar uma chamada à função **calculo_aum**, que recebe, como parâmetro, o salário atual informado pelo(a) usuário(a).

Até a conclusão da execução da função, o algoritmo principal é suspenso. A execução só volta para o algoritmo principal quando o comando **retorne** é executado (**linha 12**). Este comando é responsável por "devolver" ao algoritmo principal o valor calculado dentro da função. A execução do algoritmo principal é retomada exatamente do ponto em que foi interrompida (**linha 18**). Nessa linha, a variável **aum** recebe o valor devolvido pela função e a execução segue linearmente até o fim.

A princípio, podemos imaginar que essa solução, ao contrário do que foi dito anteriormente, necessita de mais linhas de código, tornando-a mais extensa e complexa. Entretanto, ao criarmos uma função que realiza o cálculo do aumento do salário, podemos utilizá-la quantas vezes forem necessárias ao longo do algoritmo, sem a necessidade de reescrevê-la. Devemos apenas chamá-la. Se assim fizermos, também podemos reescrever o código dessa função apenas uma vez, sem se preocupar em atualizá-la no algoritmo principal, pois todas as suas chamadas serão direcionadas para o mesmo código único.

Observe que, no exemplo anterior, temos declaração de variáveis em dois locais diferentes. Na **linha 3** declaramos as variáveis utilizadas no bloco da função, chamadas de variáveis locais ou de escopo local. Na **linha 7** declaramos as variáveis utilizadas no bloco principal do algoritmo, chamadas de variáveis globais ou de escopo global. Antes de estudarmos a sintaxe e exemplos do uso de funções em algoritmos, vamos entender melhor o conceito de escopo de variáveis.

6.1. Escopo de variáveis

Como já falamos, as funções são consideradas pequenos algoritmos dentro de um algoritmo principal. Elas possuem variáveis próprias utilizadas em suas instruções internas, chamadas de variáveis locais. As variáveis locais recebem esse nome porque podem ser utilizadas apenas dentro das funções. Quando a execução de uma função chega ao fim, as variáveis declaradas dentro dela são destruídas e seus conteúdos perdidos.

Variáveis declaradas fora de qualquer função são chamadas de variáveis globais. Essas variáveis podem ser utilizadas em qualquer ponto do programa, inclusive dentro das funções. Elas são destruídas apenas quando a execução do algoritmo principal chega ao fim. No entanto, não é aconselhável a utilização excessiva de variáveis globais, pois isso pode tornar a manutenção e a busca por erros em programas mais difíceis. Da mesma forma, é desaconselhado que se use o mesmo identificador para uma variável local e outra global no mesmo código. Apesar de ser possível, sem gerar erros de execução, fazendo isso, podemos tornar mais difícil o gerenciamento das variáveis e a manutenção do código.

6.2. Declaração de funções

A seguir, apresentamos a sintaxe básica para definir funções em Portugol:

```
funcao nome_da_funcao(param_1: tipo_1,..., param_n: tipo_n): tipo_retorno
   var
       variaveis_locais: tipo
   inicio
       // Instruções da função
       retorne valor_retorno
   fimfuncao
```

Onde:

- **nome_da_funcao**: identificador da função que será utilizada para chamá-la.
- **param_*: tipo_***: lista de pares identificador/tipo dos parâmetros que a função recebe (pode ser vazia).
- **tipo_retorno**: tipo do valor (**valor_retorno**) que a função retorna (**inteiro**, **real**, **caractere** ou **lógico**).
- **variaveis_locais**: variáveis locais usadas apenas dentro da função.
- **valor_retorno**: valor que será retornado pela função.

Os identificadores das funções não podem ser iguais aos identificadores utilizados nas variáveis globais e locais e seguem as mesmas regras de criação dos identificadores de variáveis. Os parâmetros das funções são opcionais, mas, se declarados na criação da função, passam a ser obrigatórios. Não existe receita de bolo: o problema a ser resolvido é que dirá se a função projetada para tal receberá ou não parâmetros. Entretanto, sempre é esperado um retorno da função, com o valor obtido no processamento. Veja, a seguir, a sintaxe de uma requisição a uma função com passagem de parâmetros.

```
algoritmo “NomeDoAlgoritmo”
   var
       // Declaração de variáveis
   inicio
       // Instruções do algoritmo
       ...
       retorno_funcao <- nome_funcao(param_1, param2, ..., pa-
   ram_n)
       ...
       // Continuação das instruções do algoritmo
   fimalgoritmo
```

Onde:

- **retorno_funcao**: variável global declarada no bloco principal do algoritmo que recebe o retorno do processamento da função.
- **nome_funcao**: identificador da função declarada.

» **param_***: lista de variáveis globais declaradas no bloco principal do algoritmo que contém valores que serão passados à função para processamento e retorno.

Vamos utilizar o exemplo a seguir para demonstrar e explicar a declaração e o uso de duas funções em um algoritmo: uma função para calcular o aumento salarial de um funcionário e a outra para calcular o imposto de renda retido na fonte pagadora.

```
1.  algoritmo "CalculoSalarioImposto"
2.  var
3.      salario, perc_aum, novo_salario, imposto: real
4.
5.  funcao calc_aum(sal, perc: real): real
6.  var
7.      aumento: real
8.  inicio
9.      aumento <- sal * perc / 100
10.     retorne aumento
11. fimfuncao
12.
13. funcao calcular_imposto(novo_sal: real): real
14. var
15.     imp: real
16. inicio
17.     se (novo_sal <= 2259.20) entao
18.         imp <- 0
19.     senao
20.         se (novo_sal <= 2828.65) entao
21.             imp <- novo_sal * 0.075 - 169.44
22.         senao
23.             se (novo_sal <= 3751.05) entao
24.                 imp <- novo_sal * 0.15 - 381.44
25.             senao
26.                 se (novo_sal <= 4664.68) entao
27.                     imp <- novo_sal * 0.225 - 662.77
28.                 senao
29.                     imp <- novo_sal * 0.275 - 896
30.                 fimse
31.             fimse
32.         fimse
33.     fimse
34.     retorne imp
35. fimfuncao
36.
37. inicio
38.     escreva("Digite o salário do funcionário: ")
39.     leia(salario)
40.     escreva("Digite o percentual de aumento: ")
41.     leia(perc_aum)
42.
43.     novo_salario <- salario + calc_aum(salario, perc_aum)
44.     imposto <- calcular_imposto(novo_salario)
45.
46.     escreval("O novo salário é: ", novo_salario)
47.     escreval("O imposto de renda é: ", imposto)
48. fimalgoritmo
```

Linha 3:

```
salario, perc_aum, novo_salario, imposto: real
```

Declara as variáveis globais **salario**, **perc_aum**, **novo_salario** e **imposto** do tipo **real**.

Linha 5:

```
funcao calc_aum(sal, perc: real): real
```

Define a função **calc_aum** que recebe dois argumentos do tipo **real** (**sal** e **perc**) e retorna um valor do tipo **real**.

Linha 7:

```
aumento: real
```

Declara a variável local **aumento** do tipo **real** que será retornada pela função **calc_aum**.

Linha 9:

```
aumento <- sal * perc / 100
```

Calcula o valor do aumento multiplicando o salário pelo percentual de aumento e dividindo por 100.

Linha 10:

```
retorne aumento
```

Retorna o valor calculado para o algoritmo principal.

Linha 13:

```
funcao calcular_imposto(novo_sal: real): real
```

Define a função **calcular_imposto** que recebe um argumento do tipo **real** (**novo_sal**) e retorna um valor do tipo **real**.

Linha 15:

```
imp: real
```

Declara a variável local imposto do tipo **real** que será retornado pela função **calcular_imposto**.

Linhas 17-33:

```
    se (novo_sal <= 2259.20) entao
        imp <- 0
    senao
        se (novo_sal <= 2828.65) entao
            imp <- novo_sal * 0.075 - 169.44
        senao
            se (novo_sal <= 3751.05) entao
                imp <- novo_sal * 0.15 - 381.44
            senao
                se (novo_sal <= 4664.68) entao
                    imp <- novo_sal * 0.225 - 662.77
                senao
                    imp <- novo_sal * 0.275 - 896
                fimse
            fimse
        fimse
     fimse
```

Calcula o imposto de renda retido na fonte baseado no valor do salário, calculando o percentual das faixas de tributação e subtraindo a parcela a deduzir.

Linha 34:

```
    retorne imp
```

Retorna o valor do imposto calculado para o algoritmo principal.

Linha 43:

```
    novo_salario <- salario + calc_aum(salario, perc_aum)
```

Chama a função **calc_aum** com os argumentos **salario** e **perc_aum**, e armazena o valor retornado em **novo_salario**.

Linha 44:

```
    imposto <- calcular_imposto(novo_salario)
```

Chama a função **calcular_imposto** com o argumento **novo_salario**, e armazena o valor retornado em **imposto**.

Linha 46:

```
escreval("O novo salário é: ", novo_salario)
```

Exibe o novo salário do funcionário.

Linha 47:

```
escreval("O imposto de renda é: ", imposto)
```

Exibe o valor do imposto de renda retido na fonte.

Exercícios

1. Implemente a seguinte função em Portugol:

NOME	dobra
DESCRIÇÃO	Recebe um número inteiro NUM informado pelo usuário, o multiplica por 2 e mostra na tela o resultado
PARÂMETROS	- num (inteiro): número digitado pelo usuário
RETORNO	dobro (inteiro)

Escreva um algoritmo para ler um valor **inteiro num** e, utilizando a função **dobra**, calcule e retorne o dobro do número informado para uma variável qualquer do algoritmo principal e mostre na tela.

2. Implemente a seguinte função em Portugol:

NOME	maiorMenor
DESCRIÇÃO	Recebe dois números N1 e N2 maiores que 0 e retorna o maior deles ou 0 se iguais
PARÂMETROS	- n1 e n2 (real): números maiores que 0 digitados pelo usuário
RETORNO	- maior (real): o maior número entre N1 e N2 ou 0 se N1 igual a N2

Escreva um algoritmo que leia dois números **n1** e **n2** maiores que 0 e, utilizando a função **maiormenor**, retorne o maior número ou 0 se iguais para uma variável qualquer do algoritmo principal e mostre-o na tela.

3. Implemente a seguinte função em Portugol:

NOME	somatorio
DESCRIÇÃO	Recebe um número N inteiro e positivo e retorna o somatório dos números inteiros entre 1 e N
PARÂMETROS	- n (inteiro): número inteiro e positivo digitado pelo usuário
RETORNO	- soma (inteiro): somatório dos números inteiros entre 1 e N

Escreva um algoritmo que receba um número n inteiro e positivo e, utilizando a função **somatorio**, retorne o somatório dos números inteiros entre **1** e **n** para uma variável qualquer do algoritmo principal e escreva o resultado.

4. Implemente as seguintes funções em Portugol:

NOME	areaCircunferencia
DESCRIÇÃO	Recebe o raio R de uma circunferência e retorna a área $(A=n*R)^2$, onde n=3,1416
PARÂMETROS	- r (real): raio de uma circunferência digitado pelo usuário
RETORNO	- area (real): área de uma circunferência de raio R

NOME	comprimentoCircunferencia
DESCRIÇÃO	Recebe o raio R de uma circunferência e retorna o comprimento da circunferência (C=2*R*n, onde n=3,1416)
PARÂMETROS	- r (real): raio de uma circunferência digitado pelo usuário
RETORNO	- circunferência (real): comprimento de uma circunferência de raio R

Escreva um algoritmo que receba o raio de uma circunferência r e, utilizando as funções areacircunferencia e comprimentocircunferencia, retorne a área e o comprimento de uma circunferência de raio r para quaisquer variáveis do algoritmo principal e exiba os resultados.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

INTRODUÇÃO A LÓGICA DE PROGRAMAÇÃO

Unidade VII
Encerramento

Unidade VII - Encerramento

A lógica de programação é a base fundamental para o desenvolvimento de *softwares* e sistemas computacionais. Ela envolve o uso correto das leis do pensamento e dos processos de raciocínio na criação de algoritmos, que são sequências finitas de instruções com o objetivo de resolver problemas específicos. A lógica de programação é crucial para a construção de programas eficientes e robustos, permitindo que o(a) programador(a) desenvolva soluções coerentes e válidas.

A lógica de programação é essencial em razão de:

- Organização do pensamento: auxilia os(as) programadores(as) a organizarem suas ideias de forma clara e estruturada, facilitando a compreensão e a implementação de soluções para problemas complexos.
- Eficiência no desenvolvimento: promove a criação de algoritmos eficientes, que utilizam os recursos computacionais de maneira otimizada, melhorando o desempenho dos programas.
- Redução de erros: um bom entendimento da lógica de programação ajuda a minimizar erros durante o desenvolvimento, resultando em *softwares* mais confiáveis e de alta qualidade.
- Reutilização de código: incentiva a prática de modularização, onde funções são criadas para realizar tarefas específicas. Isso permite a reutilização de código, tornando o desenvolvimento mais ágil e menos propenso a erros.
- Base para aprendizado de linguagens: fornece a base necessária para aprender diversas linguagens de programação, já que os conceitos fundamentais de lógica são aplicáveis a todas elas.

A lógica de programação abrange vários conceitos e estruturas que são essenciais para o desenvolvimento de algoritmos eficientes. Os principais conteúdos incluem:

- Tipos de dados: conhecimento dos diferentes tipos de dados, como **inteiros**, **reais**, **literais** e **lógicos**, e a importância de escolher o tipo adequado para cada variável em um algoritmo.
- Variáveis e constantes: compreensão das diferenças entre variáveis (dados que podem mudar durante a execução do programa) e constantes (dados que permanecem inalterados). É crucial para a correta manipulação de dados e para a organização do código.

- Entrada e saída de dados: métodos para receber dados do(a) usuário(a) e apresentar os resultados. A interação com o(a) usuário(a) é fundamental para a usabilidade do programa.
- Operadores aritméticos e lógicos: utilizados para realizar cálculos e comparar valores. Compreender como utilizar operadores aritméticos (como soma, subtração, multiplicação, divisão) e operadores lógicos (como **E**, **OU**, **NÃO**) é essencial para a construção de condições e expressões em algoritmos.
- Estruturas de controle: incluem estruturas sequenciais, condicionais (como **se...então** e **se...então...senão**) e de repetição (**para**, **enquanto** e **repita**). Essas estruturas controlam o fluxo de execução do programa, permitindo a tomada de decisões e a repetição de ações de acordo com condições específicas.
- Vetores e matrizes: estruturas de dados que permitem armazenar e manipular conjuntos de valores de forma organizada. Vetores são unidimensionais, enquanto matrizes podem ter múltiplas dimensões, facilitando o trabalho com grandes volumes de dados relacionados.
- Funções: ferramentas para modularizar o código. Funções são subprogramas que realizam uma tarefa específica e retornam um valor, promovendo a reutilização de código e a organização do programa. Utilizadas para dividir um programa em partes menores e mais gerenciáveis.
- Modularização: técnica de dividir um programa em partes menores e mais gerenciáveis, conhecidas como módulos ou sub-rotinas. A modularização facilita a manutenção e o entendimento do código, além de promover a sua reutilização.

A lógica de programação é, portanto, um componente vital na formação de qualquer programador(a). Dominar esses conceitos e estruturas permite a criação de algoritmos claros, eficientes e robustos, essenciais para o desenvolvimento de *software* de qualidade. A prática contínua e o estudo aprofundado desses tópicos são fundamentais para se tornar um(a) programador(a) competente e bem-sucedido(a).

AKCIT

CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

SOBRE O E-BOOK

Tipografia: Montserrat

Publicação: Cegraf UFG
Câmpus Samambaia, Goiânia - Goiás. Brasil. CEP 74690-900
Fone: (62) 3521-1358
https://cegraf.ufg.br