Pós-Graduação em Automação de Testes de Software

ALGORITMOS E LÓGICA DA PROGRAMAÇÃO

07336894690

Curso de Pós-Graduação Latu Sensu em Automação de Testes de Software em parceria entre Julio de Lima Consultoria e Treinamentos de Testes e Qualidade de Software e a Faculdade VINCIT.

Dados comerciais:
Julio de Lima Consultoria e Treinamentos de Testes e Qualidade de Software LTDA
Rua Antônio Rosa Felipe, 203
Jardim Universo
Araçatuba, São Paulo
CEP 16056-808
CNPJ 18.726.544/0001-09
IE 177.428.549.110

Contato:
info@pgats.com.br
(18) 99793-6246

OBJETIVO GERAL

Fornecer aos alunos uma compreensão sólida dos conceitos fundamentais e das técnicas necessárias para a resolução de problemas utilizando algoritmos e lógica de programação com foco em automação de testes.

CONTEÚDO

1. INTRODUÇÃO À LÓGICA E SUA CONEXÃO COM O PENSAMENTO COMPUTACIONAL

1.1. Introdução

A lógica de programação é a base para qualquer profissional que deseja trabalhar com automação de testes de software. Compreender conceitos fundamentais de algoritmos permite criar scripts eficientes, resolver problemas com clareza e estruturar soluções de maneira organizada.

No contexto do pensamento computacional, a lógica de programação aprofunda um de seus pilares fundamentais: **os algoritmos**. Enquanto o pensamento computacional nos ensina a decompor problemas, reconhecer padrões e desenvolver soluções estruturadas, a lógica entra como a ferramenta essencial para transformar esse raciocínio em instruções claras e executáveis por um computador.

Uma forma de tornar esse aprendizado mais intuitivo é usar uma analogia simples que nos acompanhará ao longo do e-book: **"Montando um Quebra-Cabeça"**. Assim como um quebra-cabeça requer peças bem encaixadas, a lógica de

programação exige um raciocínio estruturado para que todas as partes do código se conectem corretamente.

Neste e-book, vamos explorar diversos conceitos relacionados à lógica de programação que são essenciais para a automação de testes.

Alguns dos principais tópicos abordados incluem:

- **Tipos de Dados e Operadores**: Entender como diferentes tipos de dados são manipulados e como operadores são usados para realizar operações aritméticas, lógicas e de comparação.

- **Estruturas Condicionais**: Compreender como tomar decisões no código com base em condições específicas, utilizando comandos como se, senao e escolha-caso.

- **Estruturas de Repetição (Laços)**: Explorar como repetir blocos de código com laços como para e enquanto, permitindo que tarefas sejam automatizadas.

- **Funções e Modularização de Código**: Aprender a dividir o código em partes menores e reutilizáveis, facilitando a manutenção e clareza do código.

- **Estruturas de Dados Básicas (Listas)**: Aprofundar-se no uso de listas, uma das principais estruturas de dados para armazenar e manipular coleções de dados em programação.

Cada um desses tópicos será abordado detalhadamente para ajudar você a entender como construir soluções lógicas e

estruturadas, fundamentais para a automação de testes eficaz.

1.2. Portugol e o Portugol WebStudio

Para facilitar o aprendizado da lógica de programação, uma excelente ferramenta é o Portugol, uma pseudolinguagem que permite escrever algoritmos utilizando uma sintaxe semelhante ao português. O Portugol é amplamente utilizado no ensino introdutório de programação, pois reduz a barreira linguística e ajuda os iniciantes a focar nos conceitos fundamentais sem a complexidade de uma linguagem de programação formal.

O Portugol WebStudio (https://portugol.dev/) é uma plataforma online que permite escrever, testar e executar códigos em Portugol diretamente no navegador. Ele oferece um ambiente amigável para praticar lógica de programação sem necessidade de instalação de softwares adicionais. A interface intuitiva do Portugol WebStudio facilita a experimentação de algoritmos e a visualização imediata dos resultados.

Por que utilizar o Portugol?

O uso do Portugol é especialmente vantajoso para quem está iniciando na programação, pois:

- **Sintaxe simplificada**: Por ser escrito em português estruturado, facilita o entendimento dos comandos e conceitos.
- **Foco na lógica**: Permite aprender a estruturar algoritmos sem se preocupar com a sintaxe complexa de linguagens como Java, Python ou C.

- **Ambiente interativo**: O Portugol WebStudio proporciona um espaço para testar códigos rapidamente e visualizar a saída do programa.
- **Ideal para ensino**: Muitas instituições de ensino utilizam o Portugol como introdução à programação antes de avançar para linguagens mais formais.

Principais comandos do Portugol

O Portugol possui comandos básicos que ajudam a estruturar programas de maneira intuitiva.

Alguns dos principais incluem:

- Início e fim do programa:
- Variáveis e entrada de dados:
- Estruturas condicionais:
- Laços de repetição

Com esses conceitos básicos, é possível implementar algoritmos, facilitando posteriormente a transição para linguagens de programação utilizadas na automação de testes.

Usaremos o Portugol Webstudio para realizar os exercícios no decorrer da deste e-book.

2. TIPOS DE DADOS E OPERADORES

2.1. Introdução

Em qualquer programa, utilizamos **variáveis** para armazenar e manipular informações. Podemos imaginar uma variável como uma **caixa com um rótulo**, onde guardamos um valor específico. Se necessário, podemos trocar o conteúdo dessa caixa, mas sempre respeitando o tipo de dado adequado.

Os principais tipos de dados que usaremos são:

- **Inteiro**: Representa valores numéricos sem casas decimais.
- **Real**: Representa valores numéricos com casas decimais.
- **Cadeia (String)**: Cadeia de caracteres, usada para armazenar caracteres alfanuméricos.
- **Lógico (Booleano)**: Representa valores lógicos, podendo ser verdadeiro ou falso.
- **Lista (Array)**: Armazena uma coleção ordenada de elementos.

Podemos pensar nos **tipos de dados** como diferentes tipos de objetos que podem ser guardados em caixas:

- Um **número inteiro** é como uma bola de gude – um valor fixo e bem definido.
- Um **número real** é como o seu salário.
- Uma **cadeia** é como um bilhete – um conjunto de letras formando uma mensagem.

- Um **logico** é como um interruptor – pode estar ligado (true) ou desligado (false).
- Uma **lista** é como uma prateleira com vários itens organizados.

Exemplo

```
programa
{
    funcao inicio()
    {
        // Declaração de variáveis
        inteiro idade = 25           // Tipo Número inteiro
        real dividindoIdade = idade / 2 // Real (ex. 12,5)
        cadeia nome = "Ana"          // Tipo cadeira
        logico ativo = verdadeiro     // Tipo lógico
        cadeia frutas[2] = {"maçã", "banana"} // Tipo Lista
        cadeia nomes[] = {"Júlio", "Pri", "Isa", } // Tipo
Lista



        // Exibição do resultado
        escreva("Idade: ", idade, "\n")
        escreva("Dividing a Idade: ", dividindoIdade, "\n")
        escreva("Nome: ", nome, "\n")
        escreva("Ativo: ", ativo, "\n")
        escreva("Primeira Fruta: ", frutas[0], "\n")
        escreva("Último Nome: ", nomes[2], "\n")
    }
}
```

2.2. Operadores

Os operadores permitem manipular os valores armazenados nas variáveis. A seguir, vemos alguns dos principais operadores.

Operadores Matemáticos
Usados para realizar operações com números:

```
programa
{
    funcao inicio()
    {
        // Declaração de variáveis
        inteiro a = 10
        inteiro b = 3

        // Operações matemáticas
        inteiro soma = a + b
        inteiro subtracao = a - b
        inteiro multiplicacao = a * b
        real divisao = a / b
        inteiro resto = a % b

        // Exibição dos resultados
        escreva("Soma: ", soma, "\n")
        escreva("Subtração: ", subtracao, "\n")
        escreva("Multiplicação: ", multiplicacao, "\n")
        escreva("Divisão: ", divisao, "\n")
        escreva("Resto: ", resto, "\n")
    }
}
```

Operador de Concatenção
Usado para juntar cadeias:

```
programa
{
    funcao inicio()
    {
        // Declaração de variáveis
        cadeia nome = "Ana"
        cadeia sobrenome = "Silva"
```

```
        // Concatenação de cadeias
        cadeia nomeCompleto = nome + " " + sobrenome

        // Exibição do resultado
        escreva("Nome completo: ", nomeCompleto)
    }
}
```

Operadores Lógicos
Usados para avaliar expressões booleanas:

```
programa
{
    funcao inicio()
    {
        // Declaração de variáveis logicos
        logico maiorDeIdade = verdadeiro
        logico temCarteira = falso

        // Operações lógicas
        logico podeDirigir = maiorDeIdade e temCarteira
        logico podeFazerCadastro = maiorDeIdade ou
temCarteira
        logico naoTemCarteira = nao temCarteira

        // Exibição dos resultados
        escreva("Pode dirigir: ", podeDirigir, "\n")
        escreva("Pode fazer cadastro: ", podeFazerCadastro,
"\n")
        escreva("Não tem carteira: ", naoTemCarteira, "\n")
    }
}
```

2.4. Conclusão

Compreender os tipos de dados e operadores é essencial para organizar e manipular informações dentro de um programa. Escolher os tipos corretos melhora a clareza e evita erros no código.

2.5. Perguntas

1. Qual tipo de dado representa um valor verdadeiro ou falso?
   a) real
   b) logico
   c) cadeia
   d) inteiro

2. Qual operador é usado para concatenar cadeias?
   a) +
   b) -
   c) *
   d) e

3. Qual operador retorna verdadeiro apenas se ambas as condições forem verdadeiras?
   a) e
   b) ou
   c) nao
   d) +

Respostas: b, a, a

3. ESTRUTURAS CONDICIONAIS

3.1. Introdução

As estruturas condicionais são fundamentais para a lógica de programação, permitindo que um código tome decisões com base em condições específicas. Sem elas, um programa seguiria um fluxo fixo, incapaz de se adaptar às entradas fornecidas pelo usuário ou ao contexto da execução.

As principais estruturas condicionais em Portugol são:

- se, senao se e senao
- escolha-caso

3.2. Analogias

Podemos comparar as estruturas condicionais a um quebra-cabeça: se uma peça encaixa, usamos; caso contrário, testamos outra. No código, se uma condição for verdadeira, um bloco é executado; se não, seguimos para a próxima alternativa disponível.

Outro exemplo seria um semáforo:

- **Se** o sinal estiver verde, podemos atravessar.
- **Se** estiver amarelo, devemos decidir rapidamente se seguimos ou paramos.
- **Caso contrário**, paramos, pois o sinal está vermelho.

3.3. Estruturas Condicionais em Portugol

3.3.1. Se-Senao

A estrutura se avalia uma expressão lógica. Se for verdadeira, executa um bloco de código. O senao permite definir uma alternativa para o caso contrário.

```
programa
{
    funcao inicio()
    {
        inteiro idade = 17

        se (idade >= 18)
            escreva("Você é maior de idade.")
        senao
            escreva("Você é menor de idade.")
    }
}
```

3.3.2. Senao Se

Quando temos mais de uma condição a avaliar, podemos usar senao se.

```
programa
{
    funcao inicio()
    {
        inteiro nota = 85

        se (nota >= 90)
            escreva("Conceito A")
        senao se (nota >= 80)
            escreva("Conceito B")
        senao se (nota >= 70)
            escreva("Conceito C")
        senao
            escreva("Reprovado")
    }
}
```

3.3.3. Escolha-Caso

Para condições envolvendo vários casos fixos, o escolha-caso é uma alternativa mais organizada do que vários se-senao encadeados.

```
programa
{
    funcao inicio()
    {
        inteiro diaSemana = 2

        escolha (diaSemana)
        {
            caso 1:
                escreva("Domingo")
                pare
            caso 2:
                escreva("Segunda-feira")
                pare
            caso 3:
                escreva("Terça-feira")
                pare
            caso contrario:
                escreva("Dia inválido")
                pare
        }
    }
}
```

3.4. Boas Práticas

- Evite condições desnecessárias: Se já sabemos que um valor é verdadeiro, não precisamos compará-lo explicitamente com verdadeiro.

```
programa
{
    funcao inicio()
    {
        logico estaLogado = verdadeiro

        se (estaLogado)
          escreva("Usuário autenticado")
    }
}
```

3.5. Perguntas para Revisão

1. Qual estrutura é utilizada para executar um bloco de código baseado em uma condição?
    - a) Para
    - b) Enquanto
    - c) Se-Senao
    - d) Escolha-Caso
2. O que acontece se a condição de um se for falsa e não houver um senao?
    - a) O programa gera um erro
    - b) Nenhum bloco de código é executado
    - c) O próximo se é executado automaticamente
    - d) O se é executado mesmo assim
3. Quando é mais adequado usar escolha-caso?
    - a) Quando temos muitas condições booleanas

- b) Quando temos uma variável com vários valores fixos a comparar
- c) Quando não queremos usar se
- d) Apenas para números

Respostas: b, b, b

3.6. Conclusão

As estruturas condicionais são fundamentais para a tomada de decisões dentro de um programa. Compreender bem cada uma delas permite escrever códigos mais eficientes e organizados. O uso correto dessas estruturas torna o código mais legível, evitando redundâncias e melhorando a manutenção do software.

4. ESTRUTURAS DE REPETIÇÃO (LAÇOS)

4.1. Introdução

Ao escrever um programa, muitas vezes precisamos repetir uma mesma ação várias vezes. Imagine que você deseja exibir uma mensagem na tela dez vezes ou verificar uma lista de itens um por um. Se tivéssemos que escrever o mesmo comando repetidamente, o código ficaria longo, difícil de entender e sujeito a erros.

Para resolver esse problema, utilizamos as estruturas de repetição, também chamadas de laços. Elas permitem que um bloco de código seja executado várias vezes sem a necessidade de escrever comandos repetitivos. Isso torna os programas mais eficientes, organizados e fáceis de manter.

Os laços podem ser divididos em dois principais tipos:

1. Laço com número fixo de repetições → Quando sabemos exatamente quantas vezes uma ação precisa ser executada, utilizamos o para (for).
2. Laço baseado em uma condição → Quando não sabemos de antemão quantas vezes uma ação será repetida e a repetição depende de uma condição, usamos o enquanto (while).

Essas estruturas são fundamentais para a programação, permitindo a automação de tarefas repetitivas e tornando o código mais flexível e poderoso.

4.2. Analogias

Para entender melhor a importância dos laços, vejamos algumas situações do dia a dia que funcionam de maneira semelhante:

4.2.1. Organizando um quebra-cabeça

Imagine que você tem um quebra-cabeça de 500 peças e precisa encontrar as peças das bordas primeiro. Você pode adotar a seguinte estratégia:

- Examinar cada peça, verificando se ela tem um lado reto.
- Continuar até encontrar todas as bordas.

Essa abordagem é parecida com um laço de repetição, onde seguimos uma regra repetitiva até alcançar um objetivo específico.

4.2.2. Estudando para uma prova

Suponha que você tenha que estudar até sentir que domina o conteúdo. Você pode seguir este processo:

- Ler o material.
- Resolver exercícios.
- Verificar se já aprendeu o suficiente.
- Se ainda houver dúvidas, repetir o processo.

Esse ciclo é similar ao funcionamento de um laço enquanto, pois a repetição depende de uma condição (neste caso, sentir que aprendeu o conteúdo).

Esses exemplos mostram como os laços são úteis para resolver problemas de forma estruturada, evitando trabalho manual repetitivo.

4.3. Exemplos de Uso

Agora, vamos ver como os laços funcionam na prática utilizando Portugol.

4.3.1. Laço com número fixo de repetições (Para - For)

Imagine que queremos colocar cinco peças de um quebra-cabeça na mesa, uma de cada vez. Como já sabemos que esse processo deve ocorrer cinco vezes, utilizamos o laço para:

```
programa
{
    funcao inicio()
    {
        para (inteiro contador = 1; contador <= 5;
contador++)
        escreva("Peça ", contador, " colocada na mesa.\n")
    }
}
```

Explicação:

- O laço começa com contador = 1 e continua até contador= 5.
- A cada repetição, a variável i aumenta automaticamente em 1.
- O comando escreva exibe a mensagem indicando que a peça foi colocada.

O resultado exibido será:

Peça 1 colocada na mesa.

Peça 2 colocada na mesa.

Peça 3 colocada na mesa.

Peça 4 colocada na mesa.

Peça 5 colocada na mesa.

4.3.2. Laço baseado em uma condição (Enquanto - While)

Agora, suponha que estamos tentando encaixar uma peça no quebra-cabeça e podemos tentar no máximo três vezes. Como não sabemos quantas tentativas serão necessárias, usamos o laço enquanto:

```
programa
{
    funcao inicio()
    {
        inteiro tentativa = 0

        enquanto (tentativa < 3) {
            escreva("Tentativa ", tentativa + 1, " de
encaixar a peça.")
            tentativa = tentativa + 1
        }

    }
}
```

Explicação:

- Definimos tentativa = 0 antes do laço iniciar.
- O laço executa enquanto tentativa for menor que 3.
- A cada repetição, tentativa aumenta, garantindo que o laço pare após três tentativas.

A saída será:

Tentativa 1 de encaixar a peça.

Tentativa 2 de encaixar a peça.

Tentativa 3 de encaixar a peça.

Se tivéssemos um critério diferente, como "continuar tentando até encaixar a peça corretamente", poderíamos modificar a condição do enquanto para refletir isso.

4.4. Atenção: O Perigo dos Laços Infinitos

Um erro comum ao usar laços é esquecer de atualizar a variável de controle. Isso pode fazer com que a condição do laço nunca se torne falsa, gerando um laço infinito, onde o programa nunca para de executar.

Exemplo de erro:

```
programa
{
    funcao inicio()
    {
        inteiro tentativa = 0

        enquanto (tentativa < 3) {
            escreva("Tentativa ", tentativa + 1, " de
encaixar a peça.")
        }

    }
}
```

O que está errado?

A variável tentativa nunca é alterada dentro do laço. Como a condição tentativa < 3 será sempre verdadeira, o programa entrará em um ciclo infinito e nunca terminará.

Correção

Certifique-se de que a variável usada na condição do enquanto está sendo modificada corretamente dentro do laço!

4.5. Conclusão

As estruturas de repetição são ferramentas essenciais na programação, permitindo executar ações de forma automática e eficiente.

- O para é ideal quando já sabemos quantas vezes um código deve ser repetido.
- O enquanto é útil quando a repetição depende de uma condição específica.
- Devemos ter cuidado para evitar laços infinitos, garantindo que a condição do laço possa ser alterada corretamente.

Nos próximos capítulos, exploraremos funções e estruturas de dados, aprofundando ainda mais o entendimento da lógica de programação.

4.6. Perguntas para Praticar

1. Qual das opções abaixo melhor descreve a diferença entre um laço para e um laço enquanto?

   a) O para é usado para repetições fixas, enquanto o enquanto é usado para repetições condicionais.
   b) O enquanto é mais rápido do que o para.
   c) O para só pode ser usado para contar números.
   d) O para nunca termina, enquanto o enquanto sempre termina.

2. O que acontece se um laço enquanto nunca tiver sua condição alterada?

   a) O programa rodará apenas uma vez.
   b) O laço entrará em um ciclo infinito.
   c) O código será ignorado.
   d) O programa exibirá um erro imediatamente.

3. Se quisermos exibir uma mensagem 100 vezes, qual laço é mais adequado?

   a) Enquanto
   b) Para
   c) Se
   d) Escolha

Respostas: a, b, b

5. FUNÇÕES E MODULARIZAÇÃO DE CÓDIGO

5.1. Introdução

Imagine que você trabalha em uma confeitaria e precisa preparar um bolo. Você pode seguir uma receita do começo ao fim toda vez que alguém pedir um, ou pode facilitar seu trabalho criando misturas pré-preparadas para agilizar o processo. Se cada bolo exigisse medir a farinha, bater os ovos e misturar os ingredientes separadamente, você perderia tempo e poderia cometer erros.

Agora pense na programação. Se tivermos que repetir os mesmos cálculos ou operações em várias partes do código, corremos o risco de erros e dificuldades de manutenção. Para evitar isso, utilizamos **funções**, que são blocos de código reutilizáveis que realizam tarefas específicas.

As funções permitem que um programa fique mais organizado e eficiente, evitando repetições desnecessárias. Além disso, elas facilitam a manutenção, pois se for necessário corrigir ou alterar um procedimento, basta modificar a função em um único lugar, e todas as partes do programa que a utilizam serão beneficiadas.

5.2. Analogias

Vamos reforçar essa ideia com mais algumas analogias:

- **Montagem de um móvel**: Imagine que você precisa montar várias cadeiras iguais. Você pode seguir todo o processo do zero para cada uma ou criar um método eficiente, separando peças e parafusos previamente. Da mesma forma, na programação, as funções permitem executar tarefas repetitivas sem reescrever tudo de novo.
- **Receitas de culinária**: Uma receita pode ser usada várias vezes, apenas trocando ingredientes específicos (como frutas em um bolo). Assim como uma receita, uma função pode receber **parâmetros**, que permitem personalizar sua execução sem precisar modificar seu código interno.
- **Chamadas telefônicas**: Toda vez que você liga para um amigo, a operadora já tem um processo automatizado para conectar a chamada. Você não precisa se preocupar com como a ligação funciona internamente; basta digitar o número e a ação acontece. As funções seguem esse princípio: você as chama quando precisa, sem se preocupar com os detalhes internos.

5.3. Como as funções funcionam?

Uma função tem três partes principais:

1. **Declaração** – Definimos o que a função faz.
2. **Parâmetros** (opcional) – Permitem passar informações para personalizar sua execução.
3. **Retorno** (opcional) – Permite que a função envie um resultado de volta.

5.3.1. Criando uma função de soma

```
programa
{
    funcao inteiro soma(inteiro a, inteiro b) {
      retorne a + b
    }

    funcao inicio()
    {
        inteiro resultado = soma(10, 5)
        escreva("A soma é ", resultado)
    }
}
```

O que acontece aqui?

- Criamos a função soma(a, b), que recebe dois números e retorna sua soma.
- Chamamos essa função no programa principal e armazenamos o resultado em resultado.
- O programa exibe "A soma é 15".

Agora, sempre que precisarmos somar dois números, basta chamar soma(x, y), sem precisar reescrever a lógica.

5.3.2. Criando uma função sem retorno

Nem todas as funções precisam retornar valores. Algumas apenas executam comandos, elas são conhecidas como procedimentos. Veja um exemplo de função que exibe uma mensagem de boas-vindas:

```
programa
{
    funcao mostraSaudacao(cadeia nome) {
      escreva("Olá, ", nome, "\n")
    }

    funcao inicio()
    {
        mostraSaudacao("Júlio")
        mostraSaudacao("Pri")
        mostraSaudacao("Isa")
    }
}
```

Explicação:

- A função mostraSaudacao recebe um nome e exibe uma saudação personalizada.
- Como ela apenas imprime uma mensagem e não retorna nada, não usamos retorna.
- Chamamos a função algumas vezes, passando diferentes nomes, e ela exibe saudações personalizadas para cada um.

5.4. Conclusão

As funções são essenciais para organizar e modularizar o código, tornando-o mais reutilizável e fácil de manter. Além disso:

- Evitam repetições desnecessárias.
- Tornam o código mais claro e organizado.
- Facilitam a manutenção e depuração.
- Permitem reaproveitamento de código em diferentes partes do programa.

Ao programar, sempre que perceber que está repetindo um mesmo bloco de código, considere transformá-lo em uma função.

5.5. Perguntas e desafios

1. Por que devemos usar funções em um programa?
a) Para tornar o código mais longo.
b) Para organizar melhor e evitar repetições.
c) Para dificultar a leitura do código.
d) Para aumentar o consumo de memória.

2. Qual é a principal diferença entre uma função com e sem retorno?
a) Funções com retorno podem ser chamadas apenas uma vez.
b) Funções sem retorno não podem receber parâmetros.
c) Funções com retorno enviam um valor de volta, enquanto as sem retorno apenas executam uma ação.
d) Funções sem retorno sempre imprimem algo na tela.

3. Desafio prático: Escreva uma função chamada dobro, que recebe um número e retorna o seu dobro. Depois, no

programa principal, peça para o usuário digitar um número, chame a função e exiba o resultado.

Dica: Use leia() para capturar a entrada do usuário.

Nos próximos capítulos, exploraremos estruturas de dados, que nos ajudarão a armazenar e manipular.

Respostas: b, c

6. LISTAS

6.1. Introdução

As listas (também chamadas de **arrays**) são uma das estruturas de dados mais essenciais na programação. Elas permitem armazenar diversos elementos de maneira **sequencial**, facilitando a organização, recuperação e manipulação das informações.

Ao programar, frequentemente precisamos trabalhar com conjuntos de dados. Seja para armazenar nomes de clientes, produtos de um estoque, notas de alunos ou até mesmo os dias da semana, as listas são a escolha ideal para lidar com essas informações.

6.2. Analogias

Para entender melhor como funcionam as listas, imagine uma **fila de assentos em um cinema**. Cada assento possui um número e, quando alguém compra um ingresso, recebe um número correspondente à sua cadeira. Dessa forma, a pessoa sabe exatamente onde deve se sentar.

Do mesmo modo, em uma lista, cada elemento ocupa uma posição única identificada por um **índice**. Isso permite que os elementos sejam acessados diretamente, sem a necessidade de percorrer toda a estrutura.

6.3. Criando e Acessando Listas

Em Portugol, podemos criar uma lista definindo seu tamanho e os elementos que ela armazenará.

6.3.1. Criando uma Lista de Compras

```
programa
{
    funcao inicio()
    {
      cadeia listaCompras[4] = {"arroz", "feijão",
"macarrão", "leite"}
      escreva(listaCompras[0], "\n")
      escreva(listaCompras[2], "\n")
    }
}
```

No exemplo acima:

- Criamos uma lista chamada listaCompras contendo quatro elementos.
- Acessamos diretamente os elementos desejados por meio de seus índices.

Lembre-se de que a contagem dos índices começa em **zero**, ou seja:

- listaCompras[0] → "arroz"
- listaCompras[1] → "feijão"
- listaCompras[2] → "macarrão"
- listaCompras[3] → "leite"

Se tentarmos acessar um índice que não existe, o programa pode apresentar um erro, pois a posição solicitada está fora dos limites da lista.

6.4 Percorrendo Listas com Laços

Trabalhar com listas se torna muito mais eficiente quando usamos **laços de repetição**. Em vez de acessar cada

elemento individualmente, podemos percorrer toda a lista de forma automatizada.

6.4.1. Usando um Laço "para"

```
programa
{
    funcao inicio()
    {
      cadeia listaCompras[4] = {"arroz", "feijão",
"macarrão", "leite"}
      para (inteiro contador = 0; contador < 4; contador++)
        escreva(listaCompras[contador], "\n")

    }
}
```

Saída:

arroz

feijão

macarrão

leite

O que acontece nesse código?

1. Criamos uma lista com quatro elementos.
2. Utilizamos um laço **para** para percorrer a lista.
3. A variável i começa em 0 e aumenta até 3, que são os índices válidos da lista.
4. A cada repetição, escreva(listaCompras[contador]) exibe o elemento correspondente ao índice atual.

Esse método é muito mais prático do que escrever escreva(listaCompras[0]), escreva(listaCompras[1]) e assim por diante manualmente.

6.5. Aplicações Práticas

O uso de listas e laços é extremamente útil em diversas situações. Vamos ver alguns exemplos práticos.

6.5.1. Lista de Alunos

Imagine um programa que armazena os nomes de alunos de uma turma e exibe todos na tela.

```
programa
{
    funcao inicio()
    {
      cadeia alunos[5] = {"Ana", "Bruno", "Carlos",
"Diana", "Eduardo"}
      para (inteiro contador = 0; contador < 5; contador++)
        escreva("Aluno ", contador, ": ", alunos[contador],
"\n")
    }
}
```

Saída:

Aluno 1: Ana

Aluno 2: Bruno

Aluno 3: Carlos

Aluno 4: Diana

Aluno 5: Eduardo

Aqui, utilizamos contador + 1 para exibir a posição do aluno de forma mais intuitiva (começando em 1 em vez de 0).

Uma forma melhorada de contar os elementos na lista seria usando uma biblioteca que os conta automaticamente:

```
programa
{
    inclua biblioteca Util --> u

    funcao inicio()
    {
      cadeia listaCompras[] = {"arroz", "feijão",
"macarrão", "leite"}
      para (inteiro contador = 0; contador <
u.numero_elementos(listaCompras); contador++)
        escreva(listaCompras[contador], "\n")

    }
}
```

Nesse exemplo, a função u.numero_elementos recebe uma lista como parâmetro de entrada e retorna um inteiro com a quantidade de items encontrados na lista, nesse caso, 4.

6.5.2. Calculando a Média de Notas

Suponha que temos um conjunto de notas e queremos calcular a média da turma.

```
programa
{
    inclua biblioteca Util --> u

    funcao inicio()
    {
      real notas[] = {8.5, 7.0, 9.2, 6.8}
      real soma = 0

      para (inteiro contador = 0; contador <
u.numero_elementos(notas); contador++)
          soma = soma + notas[contador]

      real media = soma / u.numero_elementos(notas)

      escreva("A média da turma é: ", media)
    }
}
```

Saída:

A média da turma é: 7.875

Aqui, utilizamos um laço para somar todas as notas e, ao final, dividimos pelo número total de elementos para obter a média.

6.6. Conclusão

As listas são ferramentas essenciais para armazenar e organizar informações de forma estruturada. Quando

combinadas com laços, elas tornam a manipulação de grandes conjuntos de dados muito mais eficiente.

- As listas permitem armazenar elementos em **sequência ordenada**.
- Cada elemento da lista pode ser acessado diretamente pelo seu **índice**.
- O uso de **laços de repetição** torna o processamento de listas muito mais prático e eficiente.

6.7. Perguntas

1. Qual estrutura permite armazenar múltiplos elementos em uma sequência ordenada?
   a) Variável simples
   b) Lista (Array)
   c) Condicional
2. Como podemos acessar um elemento específico dentro de uma lista?
   a) Através de seu índice
   b) Percorrendo toda a lista
   c) Criando uma nova variável para cada elemento
3. Qual comando permite percorrer automaticamente todos os elementos de uma lista?
   a) escolha-caso
   b) para
   c) se-senao

Respostas: b, a, b

No próximo capítulo, veremos a conclusão do que aprendemos ao longo deste e-book.

7. CONCLUSÃO

Ao longo deste e-book, exploramos os conceitos essenciais de lógica de programação, que são fundamentais para quem deseja aprofundar-se na automação de testes de software. Começamos com a compreensão dos tipos de dados e operadores, peças chave para manipular informações e construir um código funcional. Aprendemos como escolher a peça certa para cada situação, seja um número, uma cadeia, ou um valor lógico, e como operadores podem conectá-las para resolver problemas de forma eficaz.

As estruturas condicionais e de repetição, ou laços, fornecem os mecanismos para que o código tome decisões e execute tarefas de forma repetitiva e eficiente. Com esses conceitos, o código deixa de ser uma sequência linear e se transforma em uma lógica capaz de se adaptar às necessidades de diferentes cenários. O pensamento computacional, com seu foco em decomposição e organização, se reflete diretamente na forma como lidamos com decisões e repetições no código.

Falamos também sobre funções e modularização, ferramentas poderosas que nos permitem organizar e reutilizar o código, evitando repetições desnecessárias e melhorando a manutenção. Ao dividir o código em blocos reutilizáveis, conseguimos não só reduzir a complexidade, mas também facilitar a depuração e a escalabilidade das nossas soluções.

Por fim, o estudo das listas e sua combinação com laços nos proporcionou uma maneira eficiente de organizar dados e percorrê-los de forma automatizada. Listas são estruturas de dados fundamentais que, quando usadas corretamente, se tornam indispensáveis para manipulação de grandes volumes

de informações, como as que frequentemente encontramos em testes de software.

Em resumo, todos esses conceitos que abordamos neste e-book são peças fundamentais para construir uma base sólida em programação. Cada conceito, desde os tipos de dados até as estruturas de controle, é uma ferramenta que, quando bem aplicada, torna a programação mais eficiente, clara e estruturada. Compreender a lógica de programação é entender como construir soluções inteligentes, organizadas e reutilizáveis.

Com as ferramentas e o conhecimento adquiridos aqui, você está mais preparado para enfrentar desafios complexos na automação de testes de software. A jornada do aprendizado da programação é contínua, e, à medida que você aplica esses conceitos em situações do mundo real, verá como o pensamento computacional se torna ainda mais crucial para criar soluções escaláveis e eficientes.

Agora, você está pronto para dar os próximos passos em sua jornada de programação e automação de testes. Lembre-se de que cada novo código escrito é uma peça adicional no grande quebra-cabeça da programação, e a prática constante é a chave para o sucesso.

REFERÊNCIAS

1. **COMPUTATIONAL THINKING FOR PROBLEM SOLVING NA COURSERA**
   https://www.coursera.org/learn/computational-thinking-problem-solving

2. **PORTUGOL WEBSTUDIO**
   https://portugol.dev/

3. **ALGORITMOS E LÓGIA DA PROGRAMAÇÃO 3.A EDIÇÃO**
   Souza et al. (2019)