ALGORITMOS E PROGRAMAÇÃO ESTRUTURADA

Marcio Aparecido Artero

CONHECENDO A DISCIPLINA

Caro aluno, bem-vindo ao estudo dos algoritmos e da programação estruturada. Este livro representa um marco na sua caminhada, pois, ao apropriar-se dos conhecimentos que serão apresentados e discutidos, você dará um passo importante para se tornar um profissional engajado em soluções tecnológicas.

A evolução da computação (hardware e software) teve como norte a necessidade de resolver problemas. Atualmente, vivemos em um cenário no qual o hardware alcançou altos níveis de desempenho, entretanto, para que ele seja de fato útil, são necessários softwares que realizem tarefas de forma automatizada e precisa. Esses programas de computadores são construídos seguindo um conjunto de regras e técnicas bastante específicas, portanto, nessa era digital, é

imprescindível que profissionais de diversas áreas aprendam essas ferramentas, para que possam contribuir e evoluir em sua área de atuação.

Diante dessa demanda por profissionais capazes de solucionar problemas, na primeira unidade deste livro você terá a oportunidade de conhecer e compreender os fundamentos de algoritmos e das linguagens de programação. Na segunda unidade, você aprenderá as estruturas de decisão e repetição. O conteúdo da terceira unidade abrirá para você uma gama de possibilidades, pois você conhecerá o que são, quais os tipos e para que servem as funções e recursividade. Na última unidade, você aprenderá técnicas que lhe permitirão organizar e otimizar seu programa, por meio das estruturas de dados com listas, filas e pilhas.

Nessa jornada, o aprendizado e domínio dos algoritmos e da programação estruturada só terão sucesso com o seu empenho em estudar e implementar todos os exemplos e exercícios que serão propostos ao longo do livro. A cada unidade você aumentará seu repertório de técnicas, podendo se aventurar em problemas cada vez mais desafiadores. Para que possamos começar nosso trabalho, convidamos você a se desafiar de forma que alcance excelentes resultados e seja um profissional diferenciado nessa área de grandes oportunidades.

Bons estudos!

NÃO PODE FALTAR

CONCEITOS DE ALGORITMOS E PROGRAMAÇÃO

Marcio Aparecido Artero

INTRODUÇÃO AOS ALGORITMOS

Você estudará os conceitos introdutórios relacionados a algoritmos e linguagens de programação.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Iniciamos aqui a primeira unidade do livro *Algoritmos e Programação Estruturada*. Aproveite ao máximo o conteúdo que nele será desenvolvido, o qual proporcionará a você a oportunidade de ser um excelente profissional.

É certo que a tecnologia é fascinante e com ela aprimoramos técnicas e elevamos o nível de conhecimento para solucionar os mais diversos tipos de problemas. Nada melhor que conhecer e compreender o que são os algoritmos, as linguagens de programação e a estrutura de um programa de computador e, assim, caminhar para a execução das mais diversas tarefas computacionais.

Nesta unidade, você terá o prazer de conhecer uma empresa de tecnologia de informação cujo foco principal é o desenvolvimento de softwares para instituições de ensino. Com a grande demanda de negócios no setor educacional, a empresa criou um projeto para contratação de estagiários para trabalhar com programação e atender à demanda de mercado. Você, um profissional exemplar na área de tecnologia da informação, foi incumbido de treinar os estagiários.

A empresa não exigiu experiência para os candidatos, e por esse motivo você deverá apresentar as definições e aplicações dos algoritmos, as inovações e os diferentes paradigmas para a área de programação, além dos componentes e das estruturas utilizadas na linguagem de programação C.

Após esse treinamento inicial, o estagiário orientado por você terá a oportunidade de saber reconhecer os conceitos e a estrutura dos algoritmos e das linguagens de programação.

Para fazer valer a proposta desta unidade, na primeira seção você terá a oportunidade de estudar os conceitos e a introdução aos algoritmos e das linguagens de programação, assim como os componentes de um programa de computador. Na segunda seção, serão apresentados a você os componentes e elementos da linguagem de programação C, e

você terá a oportunidade de aprender a respeito de variáveis, um conceito fundamental para a prática de programação. Na terceira seção, serão apresentadas as operações e expressões para um programa de computador.

PRATICAR PARA APRENDER

Caro aluno, algoritmo é uma sequência finita de passos que podem levar à criação e execução de determinada tarefa com a intenção de resolver um problema (FORBELLONE; EBERSPÄCHER, 2005). Assim, é necessário entender as definições de um algoritmo, suas aplicações e seus tipos antes de avançar para os próximos níveis deste material.

A fim de colocarmos os conhecimentos a serem aprendidos em prática, vamos analisar a seguinte situação-problema: você inicia agora o seu trabalho em uma empresa responsável por criar um software de locação de filmes on-line. Você foi incumbido de criar uma funcionalidade para o software, que consiste em detectar se um filme pode ou não ser locado pelo cliente, com base em sua idade e na classificação indicativa do filme.

Antes de partir para a ação e implementar a funcionalidade em uma linguagem de programação, construa um algoritmo que receba como entradas a idade do cliente e a classificação indicativa dos filmes que ele pretende locar. Logo após, o programa deve processar essas informações e mostrar na tela do computador um dos possíveis resultados: “Este filme não é indicado para sua faixa etária” ou “Este filme é indicado para sua faixa etária”. O algoritmo deverá ser elaborado nas formas descritas a seguir:

- Linguagem natural.
- Diagrama de blocos (fluxograma).
- Pseudocódigo.

CONCEITO-CHAVE

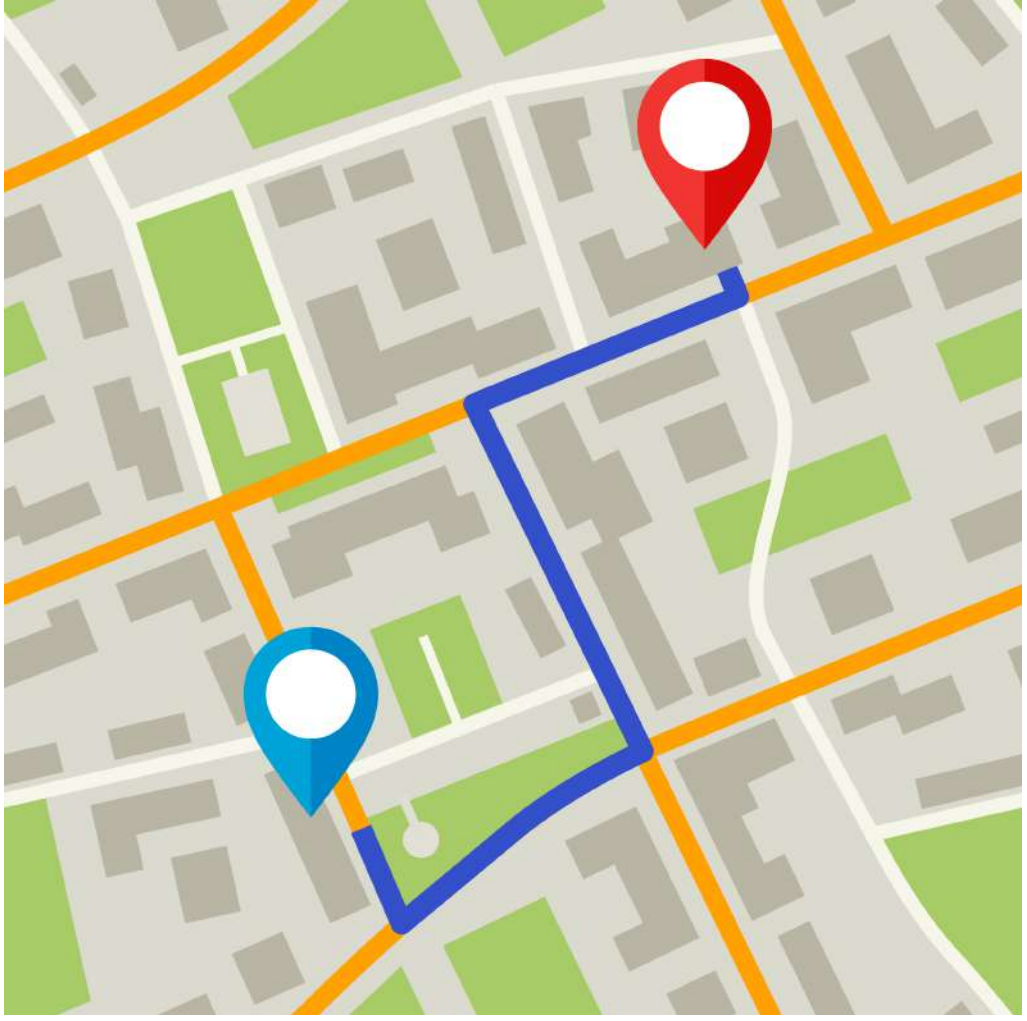
Você já deve ter se deparado muitas vezes com situações em que se deve realizar uma escolha e tomar uma decisão para resolver determinado problema. Por exemplo, ao ir ao mercado e encontrar duas marcas de um mesmo produto, qual produto você escolhe? O produto com o menor preço? O produto que você já está habituado a comprar? Ou um produto mais caro, importado? São caminhos distintos e, dependendo da escolha, você terá um impacto no orçamento.

Além disso, para executar uma ação, muitas vezes é necessário seguir uma sequência de passos para chegar a um resultado. Em uma receita de um prato, por exemplo, você tem uma sequência de passos a seguir para obter o resultado e, caso você mude algo, pode chegar a um objetivo diferente. No contexto computacional funciona de forma similar: ao elaborar um programa, você deve antes elaborar o algoritmo, que nada mais é do que uma sequência de passos necessária para alcançar o objetivo de um programa.

Quando você utiliza algum aplicativo de navegação por GPS (os mais conhecidos são Google Maps e Waze), você insere o destino final e o aplicativo faz a escolha da melhor rota, de acordo com menor congestionamento, menor caminho ou menor tempo de viagem, e você pode configurar qual das opções você deseja – por trás disso há um algoritmo com todas essas alternativas.

A Figura 1.1 ilustra um exemplo de aplicação desse tipo de abordagem. Imagine que você está no ponto azul e quer chegar ao vermelho: você terá algumas alternativas de rota, e o programa fará a escolha conforme os critérios estabelecidos por você.

Figura 1.1 | Exemplo de aplicação de algoritmos



Ver anotações 0

Fonte: Shutterstock.

A partir de agora você vai desmistificar como funcionam os algoritmos e quais são as suas aplicações dentro da programação. Para isso, você conhecerá conceitos, aplicações e tipos de algoritmos.

CONCEITOS, APLICAÇÕES E TIPOS DE ALGORITMOS

Berg e Figueiró (1998) descrevem algoritmos como uma sequência ordenada de passos que deve ser seguida para atingir um objetivo. Nesse sentido, os algoritmos nortearão você a descobrir qual o **melhor percurso para solucionar um problema computacional**. A elaboração de algoritmos é um passo importante para o desenvolvimento de um programa de computador (ou software), pois, com base na construção de algoritmos para a resolução de um problema, é possível traduzir o algoritmo para alguma linguagem de programação.

Conforme mencionado, para qualquer tarefa a ser executada no dia a dia podemos desenvolver um algoritmo. Como exemplo, tomemos a sequência de passos para o cozimento de arroz, que pode ser a seguinte:

1. Acender o fogo; 2. Refogar os temperos;

3. Colocar o arroz na panela; 4. Colocar a água; 5. Abaixar o fogo;

6. Esperar o ponto; 7. Desligar o fogo; 8. Servir o arroz.

Podemos, ainda, criar um algoritmo um pouco mais detalhado para preparar o cozimento do arroz:

1. Comprar o arroz; 2. Analisar a qualidade;

3. Realizar a pré-seleção para o cozimento; 4. Preparar os temperos;

5. Pegar a panela; 6. Acender o fogo;

7. Colocar os temperos na panela para refogar; 8. Adicionar o arroz;

9. Colocar a água na medida considerada ideal para a quantidade de arroz colocada;

10. Baixar o fogo; 11. Fechar a panela com a tampa;

12. Aguardar o ponto; 13. Desligar o fogo; 14. Servir o arroz.

Observe que não existe uma única forma de elaborar um algoritmo, porém, existe uma sequência lógica para a execução da tarefa. O passo 8, “Adicionar o arroz”, só pode ser realizado após pegar a panela (passo 5). Todavia, podem-se criar outras formas e sequências para alcançar o mesmo objetivo.

Para melhor entendimento dos algoritmos é necessário dividi-los em três partes:

- **Entrada**: dados de entrada do algoritmo. No caso do algoritmo para cozimento do arroz, seriam os insumos (ingredientes) necessários para o preparo do arroz.
- **Processamento**: são os procedimentos necessários para chegar ao resultado final (o cozimento do arroz).
- **Saída**: resultado ao qual o algoritmo quer chegar após o processamento dos dados de entrada (arroz pronto para ser servido).

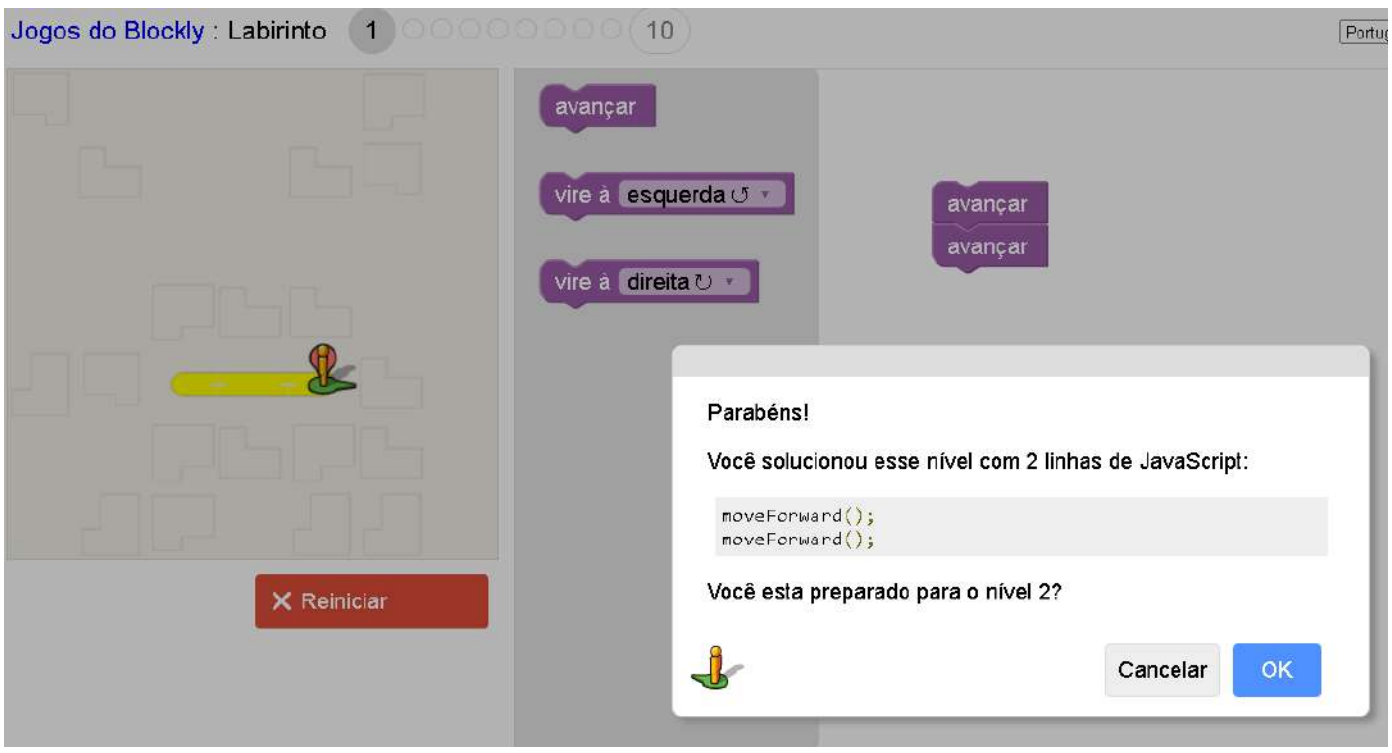
A seguir, você vai entender o funcionamento dos algoritmos usando a linguagem natural, os diagramas de blocos (em algumas literaturas são conhecidos como fluxograma) e os pseudocódigos.

EXEMPLIFICANDO

Blockly Games é um software gratuito, composto por um conjunto de jogos educacionais com enfoque no ensino de programação (BLOCKLY GAMES, 2020).

A Figura 1.2 ilustra um exemplo de jogo no qual o aluno precisa levar o avatar do Google Maps ao seu destino. Para isso, ele deve usar o conjunto de comandos disponíveis na plataforma. Neste exemplo, bastou utilizar dois comandos “Avançar” para que objetivo fosse atingido. Ao resolver o problema, o software informa a quantidade de linhas de códigos em outra linguagem, chamada Javascript. Todavia, para o treino de lógica é muito interessante.

Figura 1.2 | Jogo Labirinto Blockly para treino de lógica



Fonte: captura de tela do jogo Labirinto Blockly elaborada pelo autor.

Agora é com você . Veja se consegue completar todos os desafios do jogo “Labirinto ”.

Para visualizar o objeto, acesse seu material digital.

Ver anotações 0

LINGUAGEM NATURAL

Segundo Santos (2001), a linguagem natural é uma forma de comunicação entre as pessoas de diversas línguas, que pode ser falada, escrita ou gesticulada, entre outras formas de comunicação. A linguagem natural é uma grande contribuição para o desenvolvimento de uma aplicação computacional, pois pode direcionar de forma simples e eficiente as descrições dos problemas e suas soluções.

O algoritmo para cozimento de arroz, apresentado anteriormente, é um exemplo de uso da linguagem natural. Para reforçar esse conceito, podemos considerar o cadastro das notas dos alunos de um curso. O problema é o seguinte: o usuário (possivelmente o professor) deverá entrar com dois valores que representam as notas de um aluno em cada bimestre, e o computador retornará o resultado da média desses valores (média das notas). Então, se a média for maior ou igual a 6.0 (seis), o aluno está aprovado; caso contrário, está reprovado.

Para realizar a solução desse problema, podemos fazer uso da seguinte estrutura:

```
Início.
Entrar com a nota do aluno no primeiro bimestre.
Entrar com a nota do aluno no segundo bimestre.
Realizar a soma das duas notas e dividir o resultado por dois
(isso corresponde à média do aluno).
Armazenar o valor da média encontrada.
Mostrar na tela o valor da média.
Se a média for maior ou igual a seis, mostrar na tela que o
aluno está aprovado.
Senão, mostrar na tela que o aluno está reprovado.
Fim.
```

Observe que a linguagem natural é muito próxima da nossa linguagem.

Antes de iniciar a explicação acerca do diagrama de blocos e pseudocódigo, vamos entender sucintamente o que são variáveis e atribuições, para que você, aluno, tenha condições de interpretar e avançar nos seus estudos de algoritmos.

As **variáveis**, como o próprio nome sugere, consistem em algo que pode sofrer variações. Elas estão relacionadas à identificação de uma informação, por exemplo, o nome de um aluno, suas notas bimestrais, entre outras. A **atribuição**, representada em pseudocódigo por meio do símbolo ←, tem a função de indicar valores para as variáveis, ou seja, atribuir informação para variável. Por exemplo:

idade ← 8

nome_aluno ← “márcio”

nome_professor ← “josé”

nota_aluno ← 9.5

O exemplo apresentado indica que o número “8” está sendo atribuído como valor (informação) para a variável “idade”. Analogamente, o texto “márcio” está atribuído como valor para a variável “nome_aluno” e o valor real “9.5” está sendo atribuído como valor para a variável “nota_aluno”.

Dando sequência ao seu estudo de algoritmos, veja agora o funcionamento dos diagramas de blocos, que também podem ser chamados de fluxogramas.

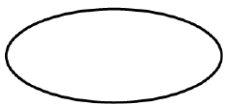
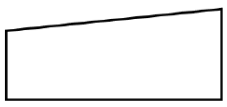
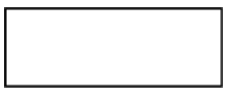
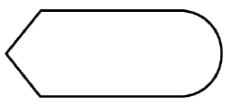
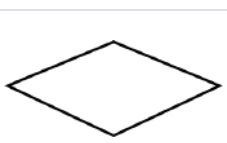
DIAGRAMA DE BLOCOS (FLUXOGRAMA)

Segundo Manzano, Franco e Villar (2015), podemos caracterizar diagrama de blocos como um conjunto de símbolos gráficos, em que cada um desses símbolos representa ações específicas a serem executadas pelo computador. Vale lembrar que o diagrama de blocos determina a linha de raciocínio utilizada pelo desenvolvedor para resolver problemas. Ao escrever um diagrama de blocos, o

desenvolvedor deve estar ciente de que os símbolos utilizados estejam em harmonia e sejam de fácil entendimento. Para que os diagramas de blocos tenham certa coerência, os seus símbolos foram padronizados pela ANSI (*American National Standards Institute* ou, em português, Instituto Americano de Padronização). O Quadro 1.1 mostra os principais símbolos utilizados para se descrever um algoritmo.

Ver anotações

Quadro 1.1 | Descrição e significados dos principais símbolos no diagrama de blocos

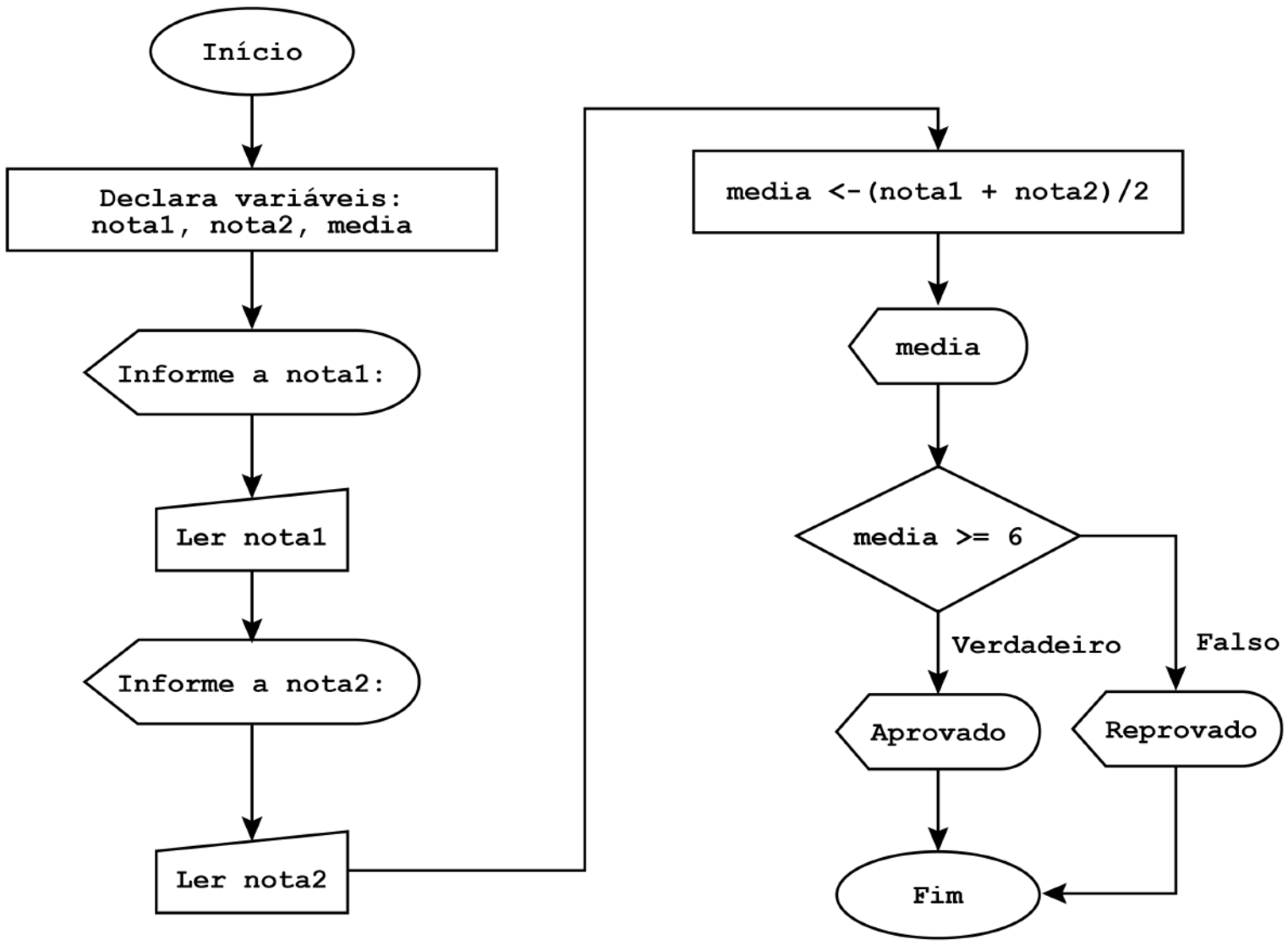
Símbolo	**Significado**	**Descrição**
	Terminal	Representa o início ou o fim de um fluxo lógico.
	Entrada/Saída de dados	Determina a entrada manual de dados por meio de um teclado ou a saída de dados.
	Processo	Representa operações/ações do algoritmo.
	Exibição	Mostra o resultado de uma ação, geralmente por meio da tela do computador.
	Condição	Mostra direções que o algoritmo deve seguir, conforme o resultado de uma condição. São os desvios condicionais.
	Fluxo	Sentido (direção) dos passos do algoritmo.

Fonte: adaptado de Manzano, Franco e Villar (2015).

Ao utilizar os símbolos do diagrama de blocos com suas respectivas instruções, você vai aprendendo e desenvolvendo cada vez mais a sua lógica em relação à resolução de problemas. A Figura 1.3 ilustra o

diagrama com o algoritmo descrito em linguagem natural, que calcula a média de notas e a situação de um aluno, representado por meio de um diagrama de blocos.

Figura 1.3 | Diagrama de blocos (fluxograma)



Fonte: elaborada pelo autor.

ASSIMILE

Algumas dicas para construir um diagrama de blocos (fluxograma) são as seguintes:

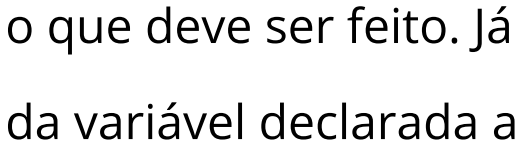
- Estar atento à sequência das instruções.
- Certificar-se de que o diagrama de blocos (fluxograma) comece de cima para baixo e da esquerda para direita.
- Ficar atento para não cruzar as linhas dos fluxos.

Vejamos, então, as representações de cada passo do diagrama:

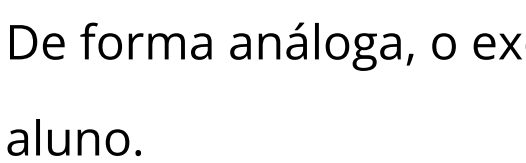
O símbolo terminal deu início ao algoritmo.

Ver anotações

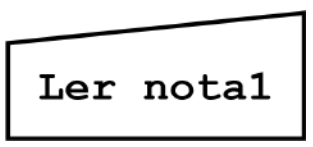
O símbolo de processamento definiu as variáveis.



O símbolo exibição mostra na tela o que o usuário deve fazer. Nesse caso, ele deve informar o valor da primeira nota do aluno, isto é, a nota do primeiro bimestre.



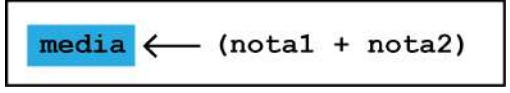
O símbolo de entrada manual libera o usuário para entrar com a primeira nota. Observe que há uma diferença entre “nota 1”, no símbolo anterior, e “nota1” descrita a seguir. No símbolo anterior, “nota 1” refere-se ao texto que será apresentado ao usuário, para que ele saiba o que deve ser feito. Já o símbolo que segue, “nota1”, refere-se ao nome da variável declarada anteriormente (no segundo símbolo utilizado nesse diagrama de blocos).



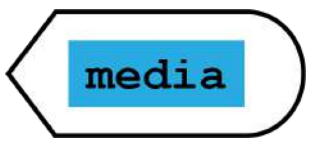
De forma análoga, o exemplo segue com a leitura da segunda nota do aluno.

O próximo símbolo de processamento realiza a atribuição do resultado do cálculo da média aritmética das duas notas lidas anteriormente à variável “media”.

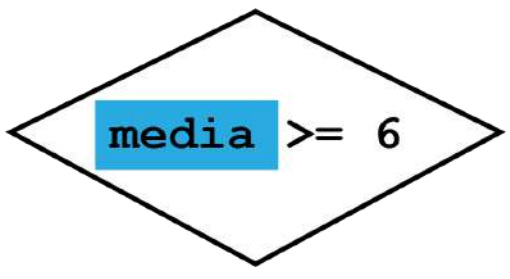
Neste momento, você pode se perguntar se não precisa adicionar a soma das notas em uma variável antes de calcular a média. A resposta é: depende! Se você for usar o valor da soma das notas para mostrá-lo na tela ou como entrada de outro cálculo, além da média, então sim, vale a pena armazená-lo em uma variável. Como esse não é o nosso caso, então não foi preciso declarar mais uma variável no algoritmo.



O símbolo de exibição mostra na tela o resultado da média calculada.

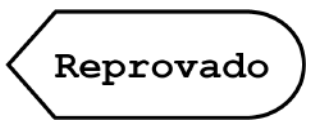


O símbolo de decisão define a condicional (verdadeiro ou falso) para a expressão “media >= 6”.

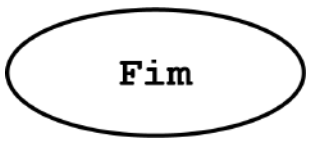


Se a condição for verdadeira, o texto impresso na tela do usuário será “Aprovado”.

Se a condição for falsa, o texto impresso na tela do usuário será “Reprovado”.

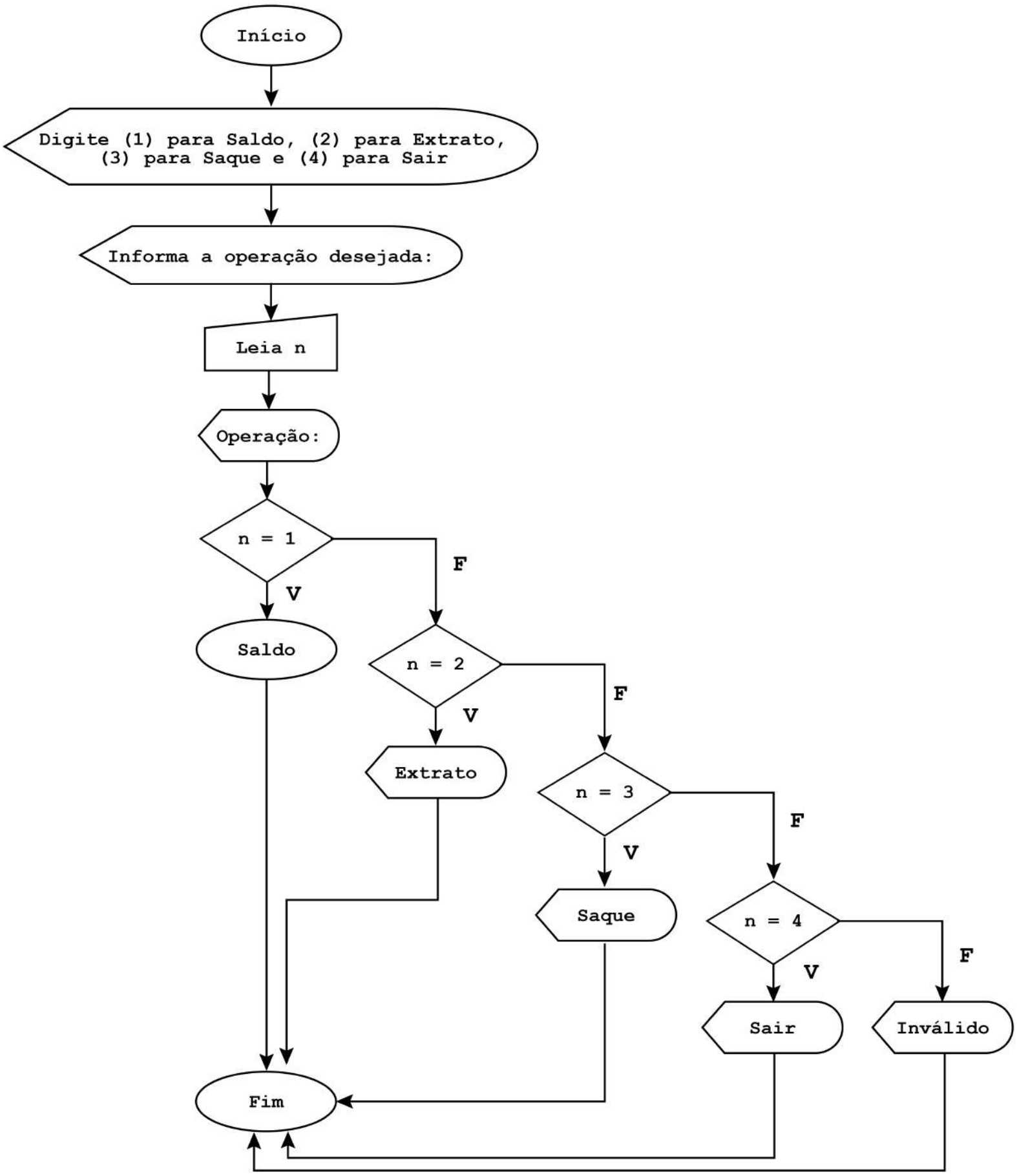


Para finalizar o algoritmo, o símbolo terminal é utilizado mais uma vez.



Outro exemplo que podemos destacar é a operação em um caixa eletrônico. Uma das primeiras atividades que o usuário deve realizar após ter se identificado é selecionar a operação a ser executada. Por exemplo: verificar saldo, emitir extrato, saque e sair. A Figura 1.4 ilustra o diagrama para realizar essa operação.

Figura 1.4 | Fluxograma de operação de caixa eletrônico



Fonte: elaborada pelo autor.

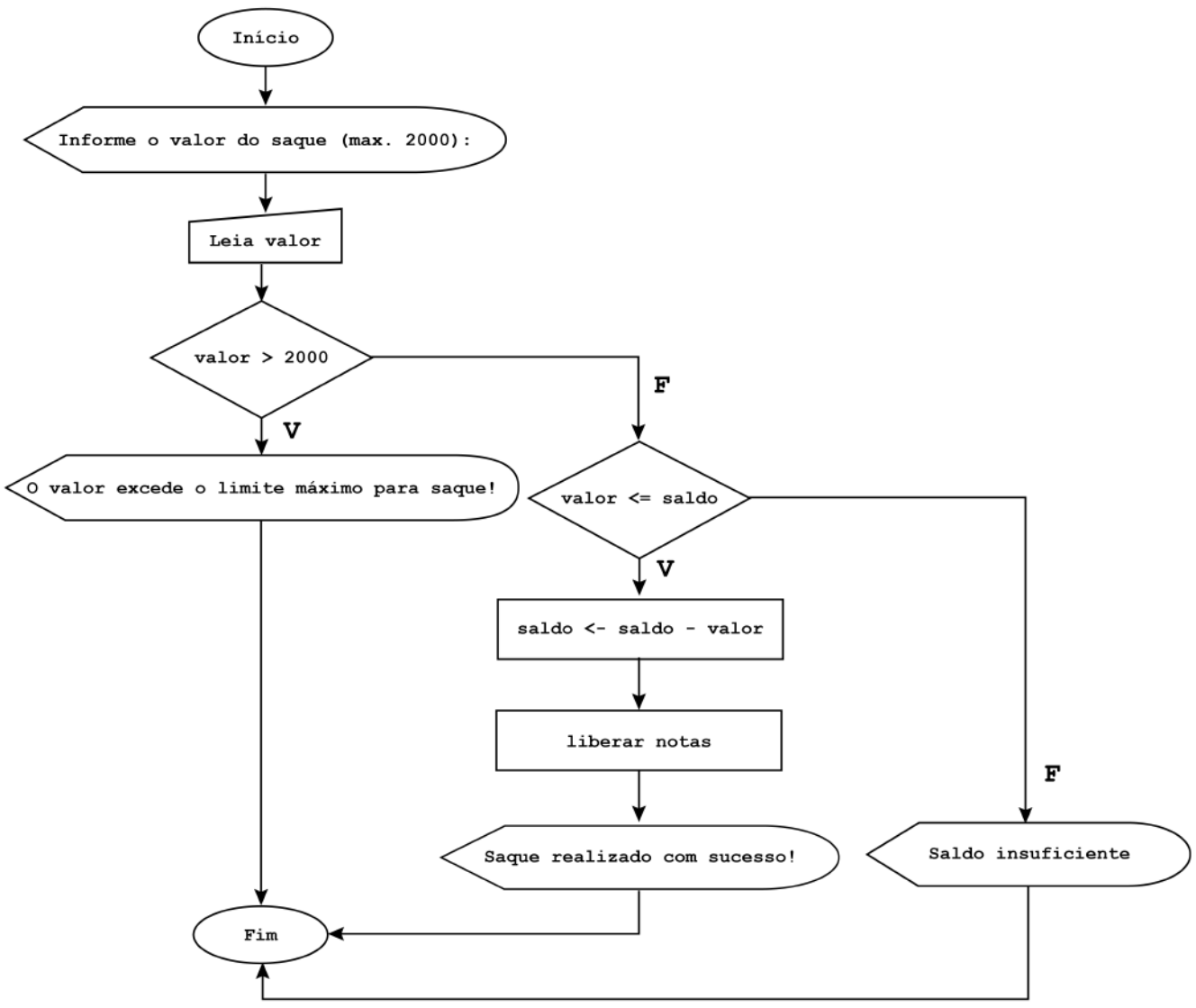
EXEMPLIFICANDO

Ainda a respeito do exemplo da Figura 1.4, para que o fluxograma não fique muito extenso, dificultando seu entendimento, você pode tratar cada operação bancária em um fluxograma à parte.

A Figura 1.5 ilustra o exemplo de fluxograma que trata da operação de saque em um sistema bancário.

Figura 1.5 | Fluxograma da operação de saque

Ver anotações



Fonte: elaborada pelo autor.

REFLITA

Observando o fluxograma que representa a operação de saque em um sistema bancário, apresentado na Figura 1.5, como você alteraria o fluxograma para que o usuário possa escolher informar ou não um novo valor, caso o valor atualmente informado por ele seja inválido (ou seja, exceda o limite máximo ou seja maior do que o saldo atual do cliente)?

PSEUDOCÓDIGO

Conforme apresentado, um algoritmo pode ser representado graficamente na forma de um fluxograma. Porém, existem outras maneiras de representar algoritmos, e uma delas é uma forma textual muito próxima da linguagem natural, denominada pseudocódigo. Segundo Aguilar (2011), o pseudocódigo corresponde à escrita de um algoritmo em palavras similares ao inglês ou ao português, com o intuito de facilitar a interpretação e o desenvolvimento de um programa. Nesse caso, para a criação de um pseudocódigo, deve-se

analisar o problema a ser resolvido e escrever o algoritmo, por meio de regras predefinidas, com a sequência de passos a ser seguida para a resolução.

Para a escrita do seu primeiro pseudocódigo, é necessário conhecer algumas estruturas importantes e comandos básicos. No Quadro 1.2 são descritos alguns desses comandos.

Quadro 1.2 | Comandos básicos utilizados para criar um pseudocódigo

Estruturas	**Descrição**
Escreva (“ “)	Imprime uma mensagem na tela.
Leia ( )	Lê valores digitados no teclado.
<-	Comando de atribuição.
inicio	Inicia o algoritmo/programa.
fimalgoritmo	Finaliza o algoritmo.
var	Realiza a declaração de variáveis
algoritmo	Indica o início do algoritmo.

Fonte: elaborado pelo autor.

ATENÇÃO

É importante estar atento para algumas regras básicas ao se utilizar pseudocódigos:

- Escolher um nome para identificar seu algoritmo.
- Avaliar as variáveis, dar atenção aos seus tipos e características.
- Descrever de forma clara o que será armazenado e se as variáveis destinadas a essa informação estão corretas.

- Verificar se as instruções fazem sentido e se têm uma sequência lógica.
- Avaliar o resultado e, quando pertinente, mostrá-lo na tela.
- Finalizar o algoritmo.

Ver anotações 0

Recordando o exemplo que calcula a média das notas de aluno, observe o pseudocódigo a seguir :

```
1   Algoritmo “Calcula média de notas”
2   Var
3           nota1, nota2, media: Real
4   Início
5     Escreva(“Informe a nota 1:”)
6     Leia(nota1)
7     Escreva(“Informe a nota 2:”)
8     Leia(nota2)
9     media <- (nota1 + nota2)/2
10    Escreva(media)
11    Se (media >=6) então
12        Escreva(“Aprovado”)
13   senão
14       Escreva(“Reprovado”)
15    fimse
16  Fimalgoritmo
17
```

Os detalhes dos pseudocódigos são explicados linha a linha a seguir.

Linha 1 – “Calcula média de notas”: esse é o nome reservado para identificar o algoritmo.

Linha 2 – “Var”: indica a declaração das variáveis.

Linha 3 – são os nomes dados para as variáveis (nota1, nota2, media). Nessa linha também é definido o tipo dessas variáveis (“Real”).

Linha 4 – inicia os procedimentos dos algoritmos (“Início”).

Linha 5 – “Escreva”: é um comando de saída que indica o que vai sair na tela do computador. Geralmente o conteúdo do texto a ser mostrado fica entre aspas (“Informe a nota 1:”).

Linha 6 – “Leia”: é comando de entrada, e o valor digitado é armazenado na variável (nota1).

Linha 9 – realiza o cálculo da média e atribui o valor encontrado à variável média: media ← (nota1 + nota2)/2.

Linha 11 – utiliza o resultado da média para criar uma condição verdadeira ou falsa: se (media >=6).

Linha 12 – se o resultado da média for maior ou igual a seis (condição verdadeira), o computador escreve na tela “Aprovado”.

Linha 14 – se o resultado da média for menor que seis (condição falsa), o computador escreve na tela “Reprovado”.

Linha 15 – encerra a condição (fimse).

Linha 16 – encerra o algoritmo com a palavra “Fimalgoritmo”.

EXEMPLIFICANDO

Para testar os pseudocódigos, você pode utilizar o Visualg. Trata-se de um software gratuito capaz de executar algoritmos escritos em pseudocódigo, utilizando o idioma português do Brasil. O download está disponível no site da ferramenta (VISUALG3, 2020b).

Veja a seguir o exemplo de algoritmo escrito em pseudocódigo e executado no Visualg:

```
algoritmo "Imprime o maior"
var
      num1, num2, maior: real
inicio
      Escreval("Digite o primeiro número: ")
      Leia (num1)
      Escreval("Digite o segundo número: ")
      Leia (num2)
      Se (num1 >= num2) então
          maior <- num1
      senão
          maior <- num2
      fimse
      Escreval("Maior número: ", maior)
Fimalgoritmo
```

CONCEITOS DE LINGUAGEM DE PROGRAMAÇÃO

Após compreendermos os conceitos, aplicações e tipos de algoritmos, vamos entender a importância da linguagem de programação e das suas famílias, assim como as projeções profissionais que a carreira de programador pode proporcionar.

Segundo Marçula (2013, p. 170):

> A Linguagem de Programação (LP) pode ser entendida como um conjunto de palavras (vocabulário) e um conjunto de regras gramaticais (para relacionar essas palavras) usados para instruir o sistema de computação a realizar tarefas específicas e, com isso, criar os programas. Cada linguagem tem o seu conjunto de palavras-chave e sintaxes.

Para Tucker (2010), da mesma forma que entendemos as linguagens naturais, utilizadas por todos no dia a dia, a linguagem de programação é a comunicação de ideias entre o computador e as pessoas. Ainda segundo o autor, as primeiras linguagens de computador utilizadas

foram as linguagens de máquina e a linguagem *assembly*, a partir da década de 1940. Desde então, muitas linguagens surgiram, bem como foram criados **paradigmas de linguagem de programação**.

De acordo com Houaiss, Franco e Villar (2001, p. 329), “paradigma significa modelo, padrão. No contexto da programação de computadores, um paradigma é um jeito, uma maneira, um estilo de se programar”. Segundo Tucker (2010), um paradigma de programação está relacionado a um padrão de soluções de problemas, os quais, por sua vez, estão relacionados a uma determinada linguagem de programação.

Segundo Tucker (2010), quatro paradigmas de programação tiveram sua evolução reconhecida nas últimas décadas:

1. **Programação imperativa**: considerado o paradigma mais antigo, pode armazenar o programa e suas variáveis juntamente, assim como a abstração procedural, as atribuições, as sequências, os laços, os comandos condicionais. Exemplo de linguagens de programação (LP) que utilizam programação imperativa: COBOL, Fortran, C, Ada e Perl.
2. **Programação orientada a objeto**: também conhecida na computação como POO, como o próprio nome sugere, é considerada uma coleção de objetos que se inter-relacionam. São exemplos de LP relacionados à POO: Smalltalk, C++, Java e C#.
3. **Programação funcional**: caraterizada por apresentar atuação matemática, cada uma com um espaço de entrada (domínio) e resultado (faixa). Exemplos de LP desse paradigma: Lisp, Scheme e Haskell.
4. **Programação lógica**: considerada uma programação declarativa, na qual um programa pode modelar uma situação-problema declarando qual resultado o programa deve obter em vez de como

ele deve ser obtido. Podemos citar como exemplo de LP lógica o Prolog.

Todas as linguagens de programação para a criação de um programa apresentam uma sintaxe, que nada mais é do que a forma como o programa é escrito. De acordo com Tucker (2010, p. 24), “[a] sintaxe de uma linguagem de programação é uma descrição precisa de todos os seus programas gramaticalmente corretos”.

Não há como negar: todas as áreas estão voltadas para a tecnologia e é por meio de diversas formas de pensamentos que os algoritmos são realizados. Por se tratar de algumas das profissões do futuro, é necessário ter em mente que as linguagens evoluem e que será preciso estar sempre atualizado, realizar certificações, estudar línguas e buscar novos caminhos na sua área de atuação. Você poderá ser um grande entusiasta em algoritmos e linguagens de programação.

Como já vimos os principais tipos de paradigmas e algumas linguagens de programação relacionadas a eles, vamos partir para o próximo passo e conhecer a linguagem com a qual trabalharemos ao longo deste livro, a saber, a linguagem C.

FAÇA VALER A PENA

Questão 1

Segundo Aguilar (2011), o pseudocódigo é considerado uma ferramenta que pode auxiliar na programação e pode ser escrito em palavras similares ao inglês ou português para facilitar a interpretação e o desenvolvimento de um programa.

Analise o algoritmo a seguir que calcula o bônus de 30% para os funcionários cadastrados e complete a linha de programação que está faltando:

```
1   Algoritmo "Calcula bônus"
2   Var
3      nome: Caractere
4      _____________________________
5
6   Inicio
7      Escreval("Informe o nome do funcionário: ")
8      Leia(nome)
9      Escreval("Informe o salário do funcionário: ")
10     Leia(salario)
11
12     _____________________________
13     salario_total <- salario + bonus
14
15     Escreval("Salário bonificado = R$", salario_total)
16
17  Fimalgoritmo
18
```

Assinale a alternativa correta:

a. salario, bonus: Real
bonus <- salario * (30/100)

b. salario, bonus, salario_total: Real
salario <- bonus * (30/100)

c. salario, bonus, salario_total: Real
bonus <- salario * (30/100)

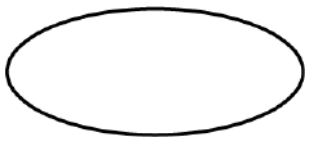
d. salario, bonus, salario_total: Real

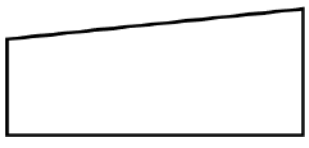
e. bonus <- salario * (30/100)

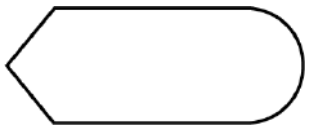
Questão 2

Segundo Manzano, Franco e Villar (2015), podemos caracterizar diagrama de blocos como um conjunto de símbolos gráficos, no qual cada um desses símbolos representa ações específicas a serem

executadas pelo computador. Vale lembrar que o diagrama de blocos determina a linha de raciocínio utilizada pelo programador para resolver um problema. Analise os símbolos a seguir:

I.  Esse símbolo significa um terminal, o qual pode representar o início ou o fim de um fluxo lógico. Em alguns casos, define as sub-rotinas.

II.  Esse símbolo significa exibição e é responsável por mostrar o resultado de uma ação, geralmente por meio da tela de um computador.

III.  Esse símbolo significa entrada manual e representa a entrada de dados para o programa, geralmente por meio de um teclado.

É correto o que se afirma em:

a. I, apenas.

b. II, apenas.

c. I e II, apenas.

d. II e III, apenas.

e. I, II e III.

Questão 3

De acordo com Houaiss, Franco e Villar (2001, p. 329), “paradigma significa modelo, padrão. No contexto da programação de computadores, um paradigma é um jeito, uma maneira, um estilo de se programar”. Para Tucker (2010), um paradigma de programação está relacionado a um padrão de soluções de problemas, os quais, por sua vez, estão relacionados a determinada linguagem de programação.

Com base no contexto apresentado, analise as afirmativas a seguir:

I. **Programação imperativa**: considerado o paradigma mais antigo, pode armazenar o programa e suas variáveis juntamente, assim como a abstração procedural, as atribuições, as sequências, os laços, os comandos condicionais.

II. **Programação orientada a objeto**: também conhecida na computação como POO, como o próprio nome sugere, é considerada uma programação somente por objetos, não podendo ter manipulações por código.

III. **Programação funcional**: caraterizada por apresentar atuação matemática, cada uma com um espaço de entrada (domínio) e resultado (faixa).

IV. **Programação lógica**: considerada uma programação declarativa, na qual um programa pode modelar uma situação-problema declarando qual resultado o programa deve obter, em vez de como ele deve ser obtido.

É correto o que se afirma apenas em:

a. I, II e III, apenas.

b. II, III e IV, apenas.

c. II e III, apenas.

d. III, apenas.

e. I, III e IV, apenas.

Ver anotações

REFERÊNCIAS

AGUILAR, L. J. **Fundamentos de programação**: algoritmos, estruturas de dados e objetos. 3. ed. Porto Alegre: AMGH, 2011.

BERG, A. C.; FIGUEIRÓ, J. P. **Lógica de programação**. 2 ed. Canoas: Ulbra, 1998.

BLOCKLY GAMES. Página inicial. 2020. Disponível em: https://bit.ly/3noPmSL. Acesso em: 28 set. 2020.

DAMAS, L. **Linguagem C**. Tradução: João Araújo Ribeiro, Orlando Bernardo Filho. 10. ed. Rio de Janeiro: LTC, 2016.

DIA DIAGRAM EDITOR. **Dia installer**. The Dia Developers, 2014. Disponível em: http://dia-installer.de/. Acesso em: 17 mar. 2018.

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron, 2000.

HOUAISS, A.; FRANCO, F. M. M.; VILLAR, M. S. **Dicionário Houaiss da Língua Portuguesa**. São Paulo: Objetiva, 2001.

LOPES, A.; GARCIA, G. **Introdução à programação**. 8. reimp. Rio de Janeiro: Elsevier, 2002.

LUCIDCHART. Página inicial. Lucid Software Inc., 2020. Disponível em: https://bit.ly/3kfaMzH. Acesso em: 17 mar. 2018.

MANZANO, J. A. N. G. **Estudo dirigido de Linguagem C**. 17. ed. rev. São Paulo: Érica, 2013.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. 2. ed. São Paulo: Érica, 2015.

MARÇULA, M. **Informática**: conceitos e aplicações. 4. ed. rev. São Paulo: Érica, 2013.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.

PEREIRA, A. P. **O que é algoritmo?** TecMundo, 12 maio 2009. Disponível em: https://bit.ly/35jE0Jk. Acesso em: 19 mar. 2018.

PIVA JUNIOR, D. *et al*. **Algoritmos e programação de computadores**. Rio de Janeiro: Elsevier, 2012.

PORTUGOL WEBSTUDIO. Página inicial. 2020. Disponível em: https://bit.ly/3lkNwSb. Acesso em: 16 out. 2020.

SALIBA, W. L. C. **Técnica de programação**: uma abordagem estruturada. São Paulo: Makron, 1993.

SANTOS, D. **Processamento da linguagem natural**: uma apresentação através das aplicações. Organização: Ranchhod. Lisboa: Caminho, 2001.

SOFFNER, R. **Algoritmos e programação em Linguagem C**. São Paulo: Saraiva, 2013.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 1994.

TUCKER, A. B. **Linguagens de programação**: princípios e paradigmas. Porto Alegre: AMGH, 2010.

VISUALG3. Página inicial. VisualG3, 2020a. Disponível em: http://visualg3.com.br/. Acesso em: 10 mar. 2020.

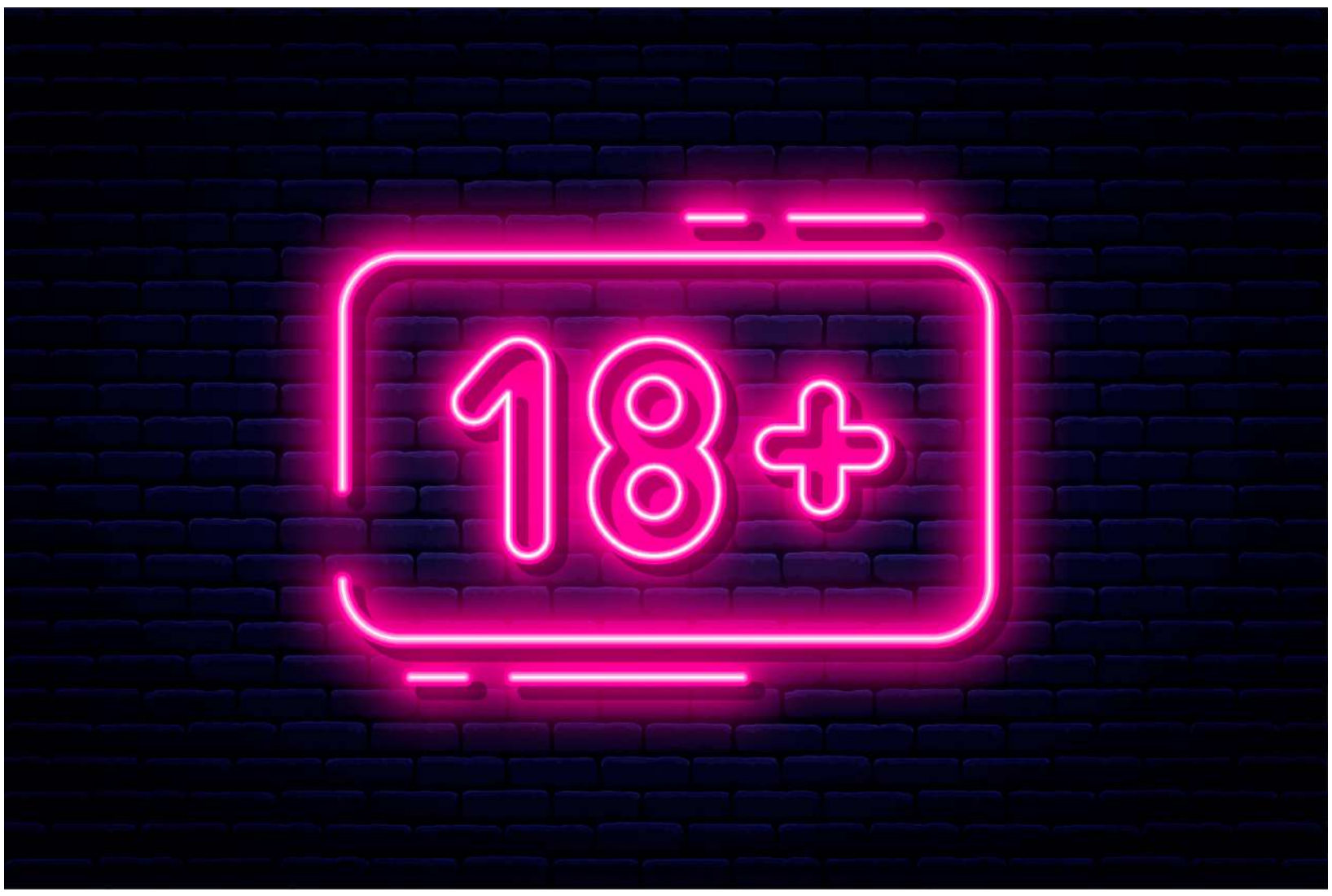
FOCO NO MERCADO DE TRABALHO

CONCEITOS DE ALGORITMOS E PROGRAMAÇÃO

Marcio Aparecido Artero

SOFTWARE DE LOCAÇÃO DE FILMES ON-LINE

Já pensou em criar uma funcionalidade para um software com base na classificação indicativa do filme e na idade do cliente? Esse será seu desafio!



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Caro aluno, chegou o momento de colocar em prática todo o conhecimento adquirido nesta seção. De acordo com a situação-problema, você foi incumbido de desenvolver uma nova funcionalidade para o software da locadora cliente da empresa para a qual você trabalha.

Para tal, como primeira proposta, você deverá elaborar um algoritmo utilizando a linguagem natural, diagramas de blocos (fluxogramas) e pseudocódigos.

Quando falamos em linguagem natural, devemos escrever a situação o mais próximo possível da linguagem convencional. Não se preocupe com os passos a serem realizados; foque primeiramente na solução do problema.

Para a realização do diagrama de blocos (fluxograma), concentre-se na descrição e nos significados dos símbolos.

Enfim, no pseudocódigo, o caminho é o seguinte:

- Escolher um nome.
- Avaliar as variáveis, dar atenção aos seus tipos e características (idade e classificação indicativa são variáveis do tipo “Inteiro”).
- Descrever de forma clara o que será armazenado e se as variáveis destinadas a essa informação estão corretas.
- Verificar se as instruções fazem sentido e se têm uma sequência lógica.
- Avaliar o resultado e mostrá-lo na tela.
- Finalizar o algoritmo.

Exemplo:

```
1  Algoritmo "Verifica classificação indicativa"
2  Var
3     class_indicativa, idade_cliente: Inteiro
4  Inicio
5     Escreval("Informe sua idade: ")
6     Leia(idade_cliente)
7     Escreval("Informe a classificação indicativa do filme: ")
8     Leia(class_indicativa)
9     se (idade_cliente <= class_indicativa) entao
10       Escreval("Este filme não é indicado para sua faixa
   etária")
11    senao
12       Escreval("Este filme é indicado para sua faixa etária")
13    fimse
14 Fimalgoritmo
15 }
```

São várias as maneiras que podem ser utilizadas para a conclusão de um algoritmo. Seguindo esse pensamento, resolva o seu algoritmo e, se possível, elabore outras formas de solução.

AVANÇANDO NA PRÁTICA

A TROCA

Um programador foi incumbido de realizar um algoritmo para coletar a quantidade de mulheres e de homens em determinado evento, porém algo deu errado e os valores ficaram invertidos. A variável que receberia o número de mulheres acabou recebendo o número de homens e vice-versa. Agora, você precisa ajustar rapidamente a situação. Para isso, elabore um algoritmo, em pseudocódigo, capaz de trocar o valor de duas variáveis do tipo inteiro.

RESOLUÇÃO

```
1  Algoritmo "Inverte valores"
2  Var
3     v1, v2, v_auxiliar: Inteiro
4  Inicio
5     Escreval("Informe o valor da variável 1: ")
6     Leia(v1)
7     Escreval("Informe o valor da variável 2: ")
8     Leia(v2)
9     Escreval("ANTES da troca: V1 =", v1, ", V2 =", v2)
10    v_auxiliar <- v1
11    v1 <- v2
12    v2 <- v_auxiliar
13    Escreval("DEPOIS da troca: V1 =", v1, ", V2 =", v2)
14 Fimalgoritmo
15
```

Como desafio, sugerimos que você realize o diagrama de blocos e a linguagem natural do algoritmo apresentado.

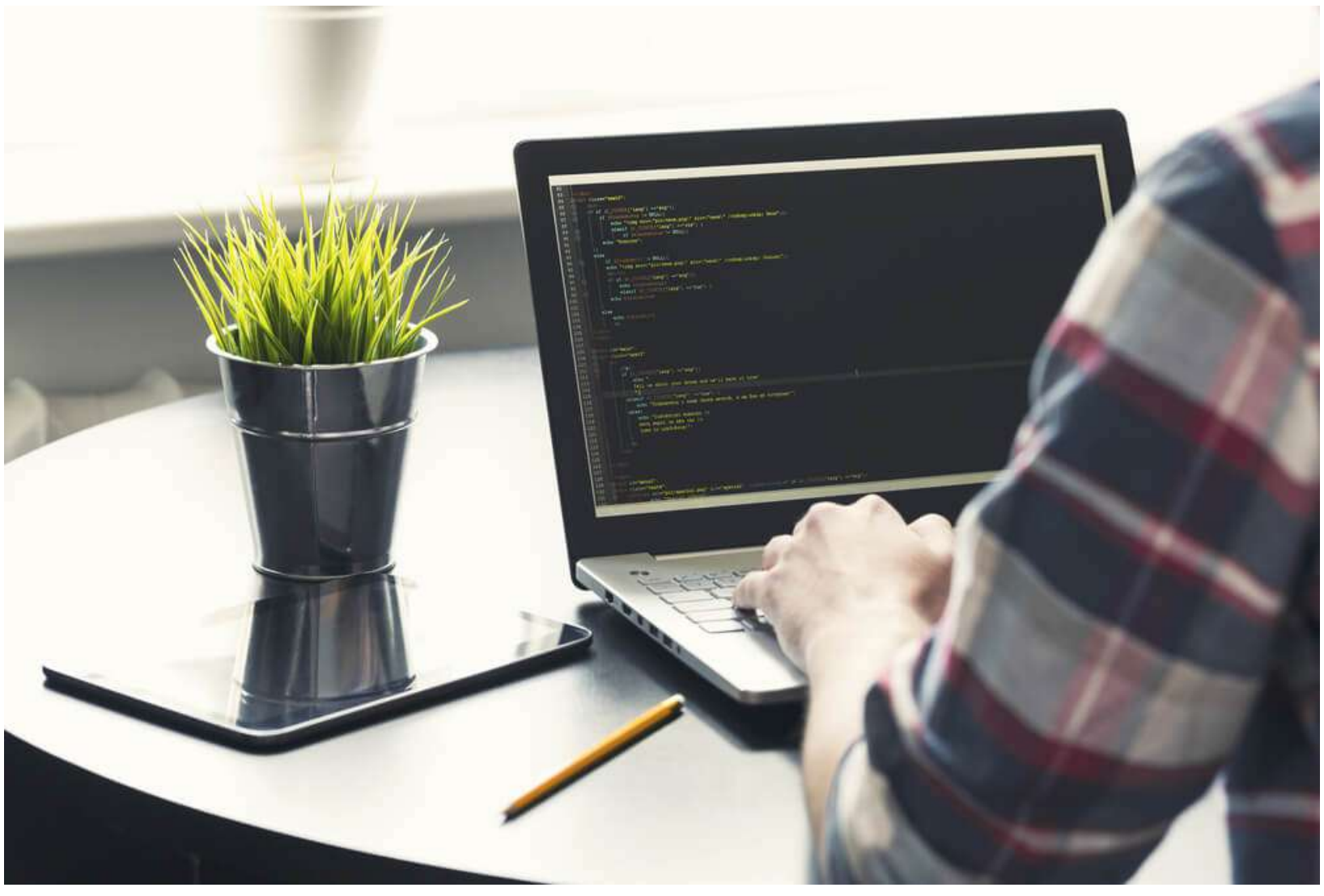
NÃO PODE FALTAR

COMPONENTES E ELEMENTOS DE LINGUAGEM DE PROGRAMAÇÃO

Vanessa Cadan Scheffer

COMO OS DADOS SÃO REPRESENTADOS E UTILIZADOS EM UMA LINGUAGEM DE PROGRAMAÇÃO

Você estudará os conceitos de variáveis e constantes, seus tipos, suas características e sua utilização em uma linguagem de programação.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro estudante, pegue um papel e uma caneta ou abra o bloco de notas. Olhe no relógio e anote a hora, o minuto e o segundo. Anotou? Novamente, olhe no relógio e faça a mesma anotação. Pronto? O valor foi o mesmo? E se anotasse novamente, seria igual? Certamente que não. Agora considere um computador que tem uma massa de 3 quilogramas, se você mudá-lo de mesa, sua massa altera? E se colocá-lo no chão? Se esses elementos (tempo e massa) fizessem parte de uma solução computacional, eles seriam representados da mesma forma em um algoritmo?

Nesta seção, veremos como os dados podem ser classificados em algoritmos implementados na linguagem C. Se existem diferentes tipos de dados, é natural que existam diferentes formas de representá-los em uma linguagem de programação.

A fim de colocarmos em prática os conhecimentos que serão adquiridos nesta seção, vamos analisar a seguinte situação-problema: você é um dos programadores de uma empresa responsável por criar um software de locação de filmes on-line, e foi incumbido de criar uma nova funcionalidade para o software, que consiste em detectar se um filme pode ou não ser locado pelo cliente com base na idade dele e na classificação indicativa do filme.

Até então, você já construiu um algoritmo capaz de receber como entradas a idade do cliente e a classificação indicativa do filme que ele pretende locar e, logo após, mostrar na tela um dos possíveis resultados: “Este filme não é indicado para sua faixa etária” ou “Este filme é indicado para sua faixa etária”.

Agora é hora de tirar essa ideia do papel e colocar para funcionar em um computador. Contudo, você ainda não dispõe de todos os conhecimentos necessários para implementar essa solução na linguagem C. Por isso, seu chefe lhe passou outra tarefa. Ele quer que seu programa seja capaz de ler a idade e o nome do cliente, bem como a classificação do filme que ele deseja locar.

Posteriormente, seu programa deve imprimir todas essas informações na tela, conforme o padrão a seguir:

```
Cliente: José das Couves
Idade: 18 anos
Classificação do filme: 12 anos
```

Seu chefe, que também é analista de sistemas, informou que você deve utilizar os conceitos de *`struct`*, variáveis e constantes para resolver esse problema.

CONCEITO-CHAVE

Durante nossa formação escolar, aprendemos a realizar diversos cálculos, por exemplo, a área de uma circunferência $A = \pi r^2$, em que π (pi) equivale a, aproximadamente, 3,14 e *r* é o raio. Pensando ainda nesse problema da área da circunferência, vamos analisar o valor da área para alguns raios. Observe a Tabela 1.1.

Tabela 1.1 | Alguns valores para área da circunferência

π(pi)	**Raio (r)**	**Área**
3,14	2 cm	12,56 cm^2
3,14	3 cm	28,26 cm^2
3,14	4 cm	50,24 cm^2

Fonte: elaborada pela autora.

Após aplicar a fórmula para alguns valores, percebemos que, conforme o valor do raio **varia**, a área também varia, mas o *pi* permanece **constante**. São inúmeros os exemplos que poderiam ser dados, nos quais se observam valores que variam ou que são fixos, isso porque nosso mundo é composto por situações que podem ou não variar, de acordo com certas condições. Sabendo que os sistemas computacionais são construídos para solucionar os mais diversos problemas, eles devem ser capazes de lidar com essa característica, ou seja, efetuar cálculo com valores que podem ou não variar. No mundo da programação, esses recursos são chamados de **variáveis** e **constantes**. A principal função desses elementos é armazenar temporariamente dados na memória de trabalho.

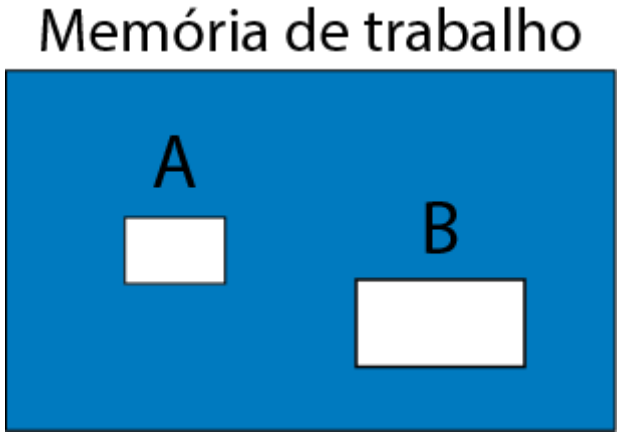
VARIÁVEIS

Deitel e Deitel (2011, p. 43) nos trazem a seguinte definição: “uma variável é uma posição na memória onde um valor pode ser armazenado para ser utilizado por um programa”. Soffner (2013, p. 33) incrementa dizendo que “variáveis são endereços de memória de trabalho que guardam, temporariamente, um valor utilizado pelo programa”. A associação que os autores fazem entre as variáveis e os endereços de memória nos ajuda a compreender que esse elemento, quando criado, existe de fato na memória do computador e, portanto, ocupa um espaço físico. O mesmo se aplica às constantes, porém, nesse caso, o valor armazenado nunca será alterado.

Na Figura 1.6 foi criada uma representação simbólica para a memória de um computador e dentro dessa memória foram alocadas duas variáveis, A e B. O espaço alocado para B é maior que o alocado para A, mas como é feita essa especificação? A quantidade de espaço que será alocada para uma variável pode ser especificada de duas maneiras: a primeira refere-se ao tipo de dado que será armazenado no espaço

reservado, caso em que o programador não consegue manipular o tamanho alocado; a segunda é feita de maneira manual pelo programador, por meio de funções específicas.

Figura 1.6 | Alocação de variáveis na memória de trabalho



Fonte: elaborada pela autora.

Todas as linguagens de programação têm **tipos primitivos** (ou básicos) e **compostos**. No grupo dos primitivos estão os seguintes tipos:

- **Numérico inteiro**: são valores inteiros que podem ser positivos, negativos ou zero. Alguns exemplos são as variáveis que armazenam idade, quantidade de produtos e código de identificação.
- **Numérico de ponto flutuante**: esse tipo armazena valores pertencentes ao conjunto dos números reais, ou seja, valores com casas decimais. Como exemplo, temos as variáveis que armazenam peso, altura e dinheiro, entre outras.
- **Caractere**: é o tipo usado para armazenar um caractere alfanumérico (letra, número, símbolo especial e outros). Como exemplo de uso, podemos citar o armazenamento do gênero de uma pessoa; caso seja feminino, armazena F, caso masculino, armazena M.
- **Booleano**: variáveis desse tipo só podem armazenar um dos dois valores: **verdadeiro** ou **falso**. Geralmente são usados para validações, por exemplo, para verificar se o usuário digitou um certo valor ou se ele selecionou uma determinada opção em uma lista, entre outros.

Para se usar uma variável na linguagem de programação C é preciso criá-la e, para isso, usa-se a seguinte sintaxe:

```
<tipo> <nome_da_variavel>;
```

Esse padrão é obrigatório e podemos usar os seguintes tipos primitivos: `int` (inteiro), `float` ou `double` (ponto flutuante) e `char` (caractere). O tipo booleano é representado pela palavra-chave `bool`, entretanto, para seu uso, é necessário incluir a biblioteca `<stdbool.h>`.

> Uma biblioteca é um conjunto de funções e tipos de dados que podem ser reutilizadas em nossos próprios programas.

Veja no Código 1.1 a criação de algumas variáveis na linguagem C.

Código 1.1 | Criação de variáveis na linguagem C

```c
#include <stdbool.h>
int main(){
    int idade;
    float salario = 1250.75;
    double porcentagem_desconto = 2.5;
    bool estaAprovado = false;
    char genero = 'M';
    return 0;
}
```

Fonte: elaborado pela autora.

É importante ressaltar que todo programa em C deve conter uma função específica, denominada "`main`". Ela representa o ponto inicial do seu programa, ou seja, o ponto

a partir do qual o computador começará a executá-lo. Não se preocupe, por enquanto, com o conceito de função, pois teremos toda uma seção para discutirmos sobre ele.

Voltando ao assunto das variáveis, destaca-se que, ao criar uma variável, o programador pode optar por já atribuir ou não um valor (por exemplo, para a variável "`idade`", nenhum valor foi atribuído a priori).

> **ATENÇÃO**
>
> **Inicialização de variáveis**
>
> É uma boa prática de programação sempre inicializar as variáveis com algum valor específico, evitando que recebam dados de processamentos anteriores e que estejam na memória (a esses dados nós damos o nome de lixo de memória). Portanto, quando a variável for numérica, sempre vamos iniciar com zero; quando booleana, com falso; quando do tipo caractere, usaremos ' ' para atribuir vazio.
>
> **Nome da variável**
>
> Outro ponto importante é o nome da variável, que deve ser escolhido de modo a identificar o dado a qual se refere. Por exemplo, em vez de usar "`ida`" para criar uma variável que armazenará a idade, opte pelo termo completo.
>
> Outro aspecto importante a ser ressaltado é que a maioria das linguagens de programação são `case sensitive`, o que significa que letras maiúsculas e minúsculas são tratadas com distinção. Então, a variável "`valor`" é diferente da variável "`Valor`". Além disso, no nome de uma variável não podem ser usados acentos nem caracteres especiais, como interrogação e espaços em brancos, entre outros. Nomes de variáveis também não podem iniciar com números. Por exemplo, "`1telefone`" **não** é um nome válido para variável, enquanto "`telefone1`" **é** um nome válido.

Ver anotações

Como já mencionado, a quantidade de espaço que será alocada para uma variável depende do tipo de variável. Por exemplo, para uma variável `int` serão alocados 4 bytes na memória (MANZANO; MATOS; LOURENÇO, 2010). O tamanho alocado na memória pelo tipo de variável limita o valor que pode ser guardado naquele espaço. Por exemplo, não seria possível guardar o valor 10 trilhões dentro da variável do tipo `int`. Vejamos, 4 bytes são 32 bits. Cada bit só pode armazenar zero ou um. Portanto, nós temos a seguinte equação: valor máximo de uma variável inteira = 2^{32}=4.294.967.296. Porém, esse valor precisa ser dividido por dois, pois um inteiro pode armazenar números negativos e positivos. Logo, uma variável `int` poderá ter um valor entre -2.147.423.648 e 2.147.423.648.

Para suprir parte da limitação dos valores que uma variável pode assumir pelo seu tipo, foram criados modificadores de tipos, os quais são palavras-chave usadas na declaração da variável que modifica sua capacidade-padrão. Os três principais modificadores são:

- **`Unsigned`**, usado para especificar que a variável armazenará somente a parte positiva do número.
- **`Short`**, que reduz o espaço reservado pela memória.
- **`Long`**, que aumenta a capacidade padrão.

A Tabela 1.2 mostra o tamanho e os possíveis valores de alguns tipos de dados na linguagem C, inclusive com os modificadores.

Tabela 1.2 | Tipos de variáveis e sua capacidade

Tipo	Tamanho (byte)	Valores
`int`	4	-2.147.423.648 até 2.147.423.648
`float`	4	$-3,4^{38}$ até $3,4^{38}$
`double`	8	$-1,7^{308}$ até $1,7^{308}$

Tipo	Tamanho (byte)	Valores
`char`	1	-128 até 127
`unsigned int`	4	4.294.967.296
`short int`	2	-32.768 até 32.767
`long double`	16	$-3,4^{4932}$ até $1,1^{4932}$

Fonte: adaptada de Manzano, Matos e Lourenço (2015, p. 35).

ESPECIFICADOR DE FORMATO

Na linguagem de programação C, cada tipo de variável usa um especificador de formato para fazer a impressão do valor que está guardado naquele espaço da memória (PEREIRA, 2017). Veja no Código 1.2, que foram criadas cinco variáveis (linhas 4 a 8) e para cada tipo, usou-se um especificador na hora de fazer a impressão com a função `printf()`. A função "`printf`" pertence à biblioteca `<stdio.h>`, por isso, é necessário inclui-la no início do programa. Na linha 10, para o tipo inteiro, foi usado `%d`. Nas linhas 11 e 12, para os pontos flutuantes, foi usado `%f`. Na linha 13, para o caractere, foi usado o `%c`, e, na linha 14, o especificador de ponto flutuante ganhou o adicional .3, o que significa que o resultado será impresso com três casas decimais. O símbolo `\n` é usado para gerar uma quebra de linha na saída do programa.

EXEMPLIFICANDO

Código 1.2 | Impressão de variáveis

```c
#include <stdio.h>

int main(){
    int idade = 18;
    float salario = 1250.75;
    double porcentagem_desconto = 2.5;
    char genero = 'F';
    float altura = 1.63;

    printf("\n Idade: %d",idade);
    printf("\n Salário: %f",salario);
    printf("\n Desconto (%): %f",
porcentagem_desconto);
    printf("\n Gênero: %c",genero);
    printf("\n Altura: %.3f",altura);
    return 0;
}

```

Fonte: elaborado pela autora.

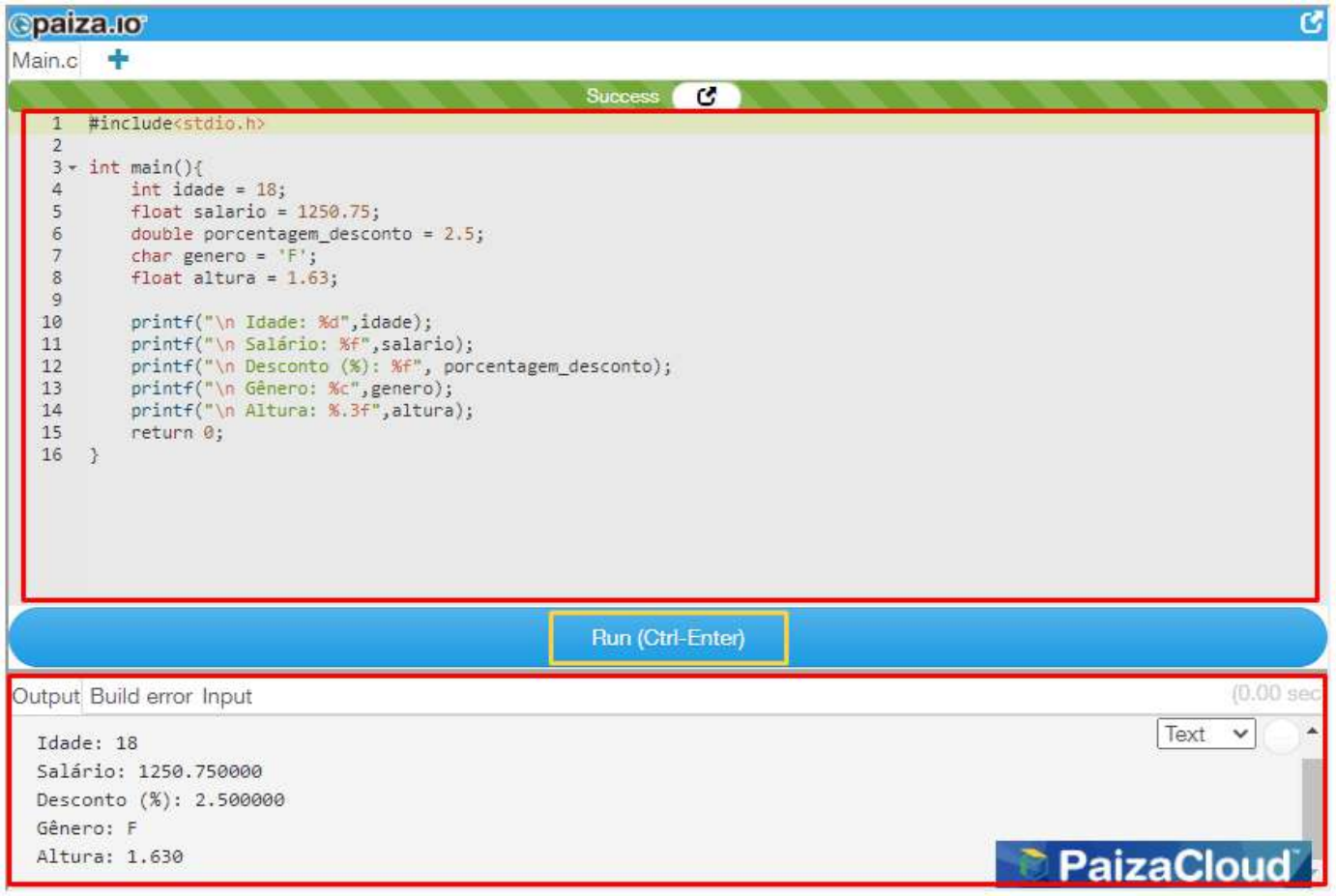
Vamos utilizar a ferramenta Paiza.io para testar as declarações das variáveis e a impressão de cada uma delas.

> O **Paiza.io** é um software gratuito e disponível para acesso via Web (PAIZA.IO, 2020). Ele permite a execução de códigos de várias linguagens de programação, dentre elas, a linguagem C.

Teste o Código 1.2 utilizando a ferramenta Paiza.io.

A Figura 1.7 mostra a captura de tela do Paiza dividida em 2 blocos.

Figura 1.7 |Orientação Paiza



Fonte: captura de tela do software Paiza.io elaborada pela autora.

No primeiro bloco, observe que temos os códigos que serão executados.

No segundo bloco, você poderá realizar a entrada de dados (**INPUT**) ou visualizar as saídas (**OUTPUT**) , ou seja, os resultados da execução do código. Mas para isso, precisamos utilizar o botão "**Run**" ou as teclas "**Ctrl-Enter**" para realizar a execução dos códigos.

O ENDEREÇO DE MEMÓRIA DE UMA VARIÁVEL

A memória de um computador é dividida em blocos de bytes (1 byte é um conjunto de 8 bits) e cada bloco tem um endereço que o identifica. Podemos fazer uma analogia com os endereços das casas, já que cada casa detém uma localização, e se existissem dois endereços iguais, certamente seria um grande problema. Já sabemos que as variáveis são usadas para reservar um espaço temporário na memória, que tem um endereço único que o identifica. Será que conseguimos saber o endereço de alocação de uma variável? A resposta é sim. Para sabermos o endereço de uma variável basta utilizarmos o **operador &** na hora de imprimir a variável.

ASSIMILE

O operador & é muito importante na linguagem C, justamente por permitir acessar diretamente os endereços de memória das variáveis.

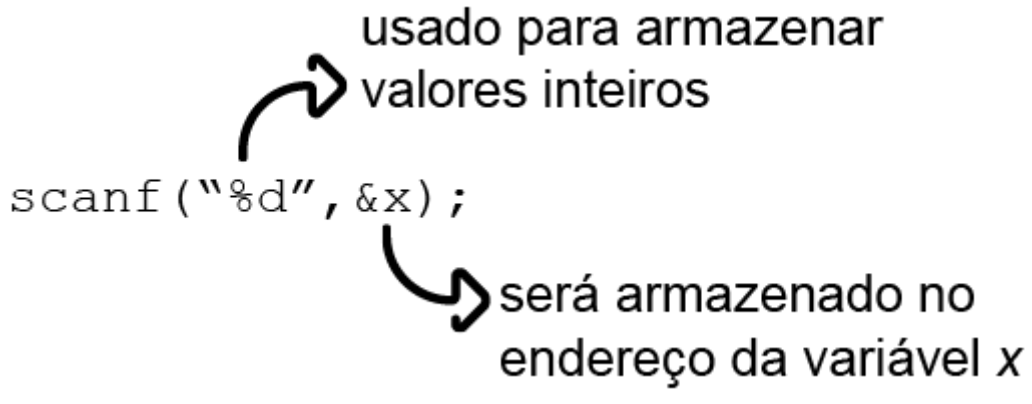
Conforme avançarmos em nossos estudos, usaremos esse operador & para atribuirmos valor a uma variável, a partir do teclado do computador.

Para armazenar valores digitados pelo usuário em uma variável, podemos usar a função `scanf()`, com a seguinte estrutura:

```
scanf(“especificador”,&variavel);
```

A Figura 1.8 apresenta um exemplo no qual se utilizou o especificador “%d” para indicar ao compilador que o valor a ser digitado será um inteiro, e esse valor será guardado no endereço de memória da variável x.

Figura 1.8 | Armazenamento em variáveis



Fonte: elaborada pela autora.

EXEMPLIFICANDO

Criando programas em C

1. Vamos criar um programa em C que armazena dois valores, um em cada variável. Para isso, o usuário terá que informar as entradas, que deverão ser armazenadas nas variáveis `valor1` e `valor2`. O Código 1.3 apresenta a solução. Observe que na linha 4, como todas as variáveis eram do mesmo tipo, foram declaradas na mesma linha, separadas por vírgula, e todas foram inicializadas com zero. Nas linhas 7 e 9, os valores digitados pelo usuário

serão armazenados nos endereços das variáveis `valor1` e `valor2`, respectivamente.

Código 1.3 | Impressão de valores armazenados

```c
#include <stdio.h>

int main(){
    float valor1=0, valor2=0;

    printf("\n Digite o primeiro valor:");
    scanf("%f",&valor1);
    printf("\n Digite o segundo valor:");
    scanf("%f",&valor2);
    printf("Variável 1 = %.2f",valor1);
    printf("Variável 2 = %.2f",valor2);
    return 0;
}
```

Fonte: elaborado pela autora.

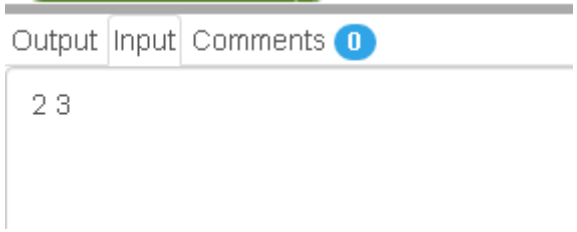
Vamos utilizar a ferramenta Paiza.io para testar impressão dos valores armazenados.

Veja como fica a saída do programa apresentado, após sua execução no Paiza.io.

No Paiza.io, quando o programa exige entrada de valores por meio do teclado, você deve especificá-los na aba “**Input**”.

Para o exemplo anterior (informar dois números reais), você poderia especificar os valores em uma mesma linha, com os valores separados por um espaço em branco, conforme ilustrado a seguir:

Figura 1.9 | Entrada de dados com valores em uma mesma linha e separados por espaço



Fonte: captura de tela do software Paiza.io elaborada pela autora.

Ou também poderia especificar cada valor em uma linha, conforme ilustrado a seguir:

Figura 1.10 | Entrada de dados com valores separados em diferentes linhas

Fonte: captura de tela do software Paiza.io elaborada pela autora.

2. Vamos criar um programa que armazena as coordenadas x, y com os seguintes valores (5,10). Em seguida, vamos pedir para imprimir o endereço dessas variáveis. Veja que, nas linhas 7 e 8, a impressão foi feita com a utilização de **&x**, **&y**, o que resulta na impressão dos endereços de memória de x e y em hexadecimal, pois usamos o especificador **%x**.

Veja como fica a saída do programa apresentado, utilizando a ferramenta Paiza.io.

CONSTANTES

Entende-se por constante um valor que nunca será alterado. Na linguagem C, existem duas formas de criar valores constantes. A primeira delas é usar a diretiva `#define`, logo após a inclusão das bibliotecas. Nesse caso, a sintaxe será da seguinte forma:

```
#define <nome_da_constante> <valor>
```

Veja que não há ponto e vírgula no final da declaração. Outro aspecto interessante é que a diretiva não utiliza espaço em memória, ela apenas cria um rótulo associado a um valor (MANZANO; MATOS; LOURENÇO,

2015). Assim, durante a compilação do programa, esse valor é substituído nos pontos em que o nome da constante é usado do código.

A outra forma de se criar valores constantes é similar à criação de variáveis, porém, antes do tipo, usa-se a palavra-chave `const`. Portanto, a sintaxe ficará como segue:

```
const <tipo> <nome_da_constante>;
```

Quando se utiliza a segunda forma de declaração, a alocação de espaço na memória segue o mesmo princípio das variáveis, ou seja, `int` alocará 4 bytes; `char`, 1 byte, entre outros. A principal diferença entre as constantes e as variáveis é que o valor de uma constante **nunca** poderá ser alterado. Caso você crie uma constante, por exemplo, `const int x = 10` e tente alterar o valor no decorrer do código, o compilador acusará um erro e não será gerado o arquivo executável.

No Código 1.4, há um exemplo das duas formas de sintaxes para constantes. Na linha 3 definimos uma constante (rótulo) chamada *pi* com valor 3.14. Na linha 6, criamos uma constante usando a palavra-chave `const`. Nas linhas 8 e 9 imprimimos o valor de cada constante. Observe que nada difere da impressão de variáveis.

Código 1.4 | Sintaxe para criação de constantes em C

```
1  #include <stdio.h>
2
3  #define PI 3.14
4
5  int main(){
6      const float G = 9.80;
7
8      printf("\n PI = %f", PI);
9      printf("\n G = %f", G);
10     return 0;
11 }
12
```

Ver anotações

Fonte: elaborado pela autora.

VARIÁVEIS COMPOSTAS

Como já explicitado, as variáveis são usadas para armazenar dados na memória do computador e esses dados podem ser de diferentes tipos (inteiro, decimal, caractere ou booleano), chamados de tipos primitivos. Vimos que podemos armazenar a idade de uma pessoa em uma variável do tipo `int` e a altura em um tipo `float`, por exemplo. Mas, e se fosse necessário armazenar quinze medidas da temperatura de um dispositivo, usaríamos quinze variáveis? Com nosso conhecimento até o momento teríamos que criar as quinze, porém muitas variáveis podem deixar o código maior, mais difícil de ler, entender e manter e, portanto, não é uma boa prática de programação. A melhor solução para armazenar diversos valores dentro de um mesmo contexto é utilizar variáveis compostas. Esse recurso permite armazenar diversos valores utilizando um mesmo nome de variável (MANZANO; MATOS; LOURENÇO, 2015).

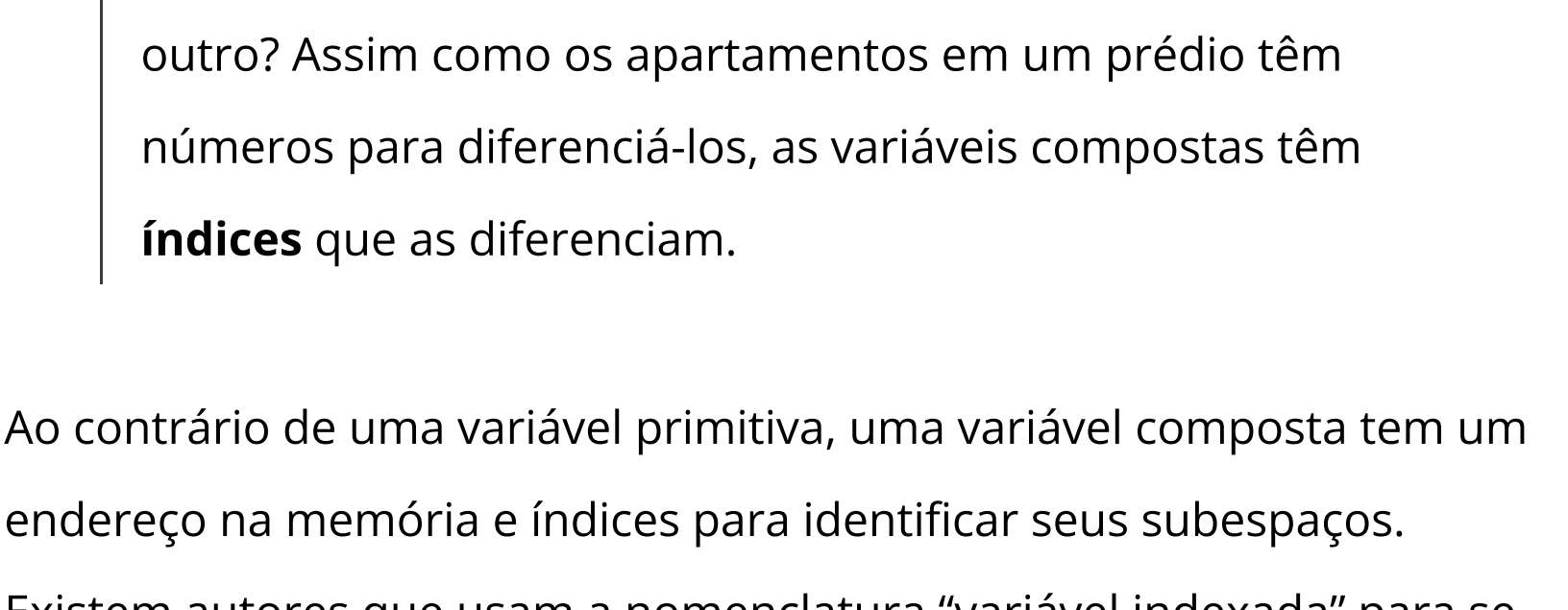
Quando alocamos uma variável primitiva, por exemplo um `int`, um espaço de 4 bytes é reservado na memória, ou seja, **um bloco** é reservado e seu endereço é usado para armazenamento e leitura dos dados. Quando alocamos uma variável composta do tipo `int`, **um conjunto de blocos** de 4 bytes será reservado. O tamanho desse conjunto (1, 2, 3 ...*N* blocos) é especificado pelo programador.

Para entendermos as variáveis compostas, vamos fazer uma analogia entre casas, prédios e as variáveis. Uma casa tem um endereço único que a identifica (que seria equivalente ao nome da variável em um programa), composto por rua, número, bairro, cidade, estado e CEP. Considerando que uma casa é construída para uma família morar, podemos compará-la a uma variável primitiva que armazena um único valor. Por outro lado, um prédio apresenta um endereço composto pelos mesmos parâmetros que a casa (ou seja, uma variável composta tem um nome único, assim como uma variável primitiva), porém nesse

mesmo endereço moram várias famílias. Assim são as variáveis compostas: em um mesmo endereço (nome de variável) são armazenados diversos valores. Esses conceitos estão ilustrados na Figura 1.11: veja que do lado esquerdo, a variável `casa1` armazena o valor 9 e a variável `casa2` armazena o valor -2,5; já no lado direito, a variável `predio1` armazena 3 valores inteiros e a variável `predio2` armazena dois valores decimais.

Ver anotações

Figura 1.11 | a) Variáveis primitivas b) Variáveis compostas



Fonte: elaborada pela autora.

REFLITA

Se, nas variáveis compostas, em um mesmo nome de variável, por exemplo `predio1` (veja o exemplo anterior), são guardados muitos valores, como diferenciar um valor do outro? Assim como os apartamentos em um prédio têm números para diferenciá-los, as variáveis compostas têm **índices** que as diferenciam.

Ao contrário de uma variável primitiva, uma variável composta tem um endereço na memória e índices para identificar seus subespaços. Existem autores que usam a nomenclatura “variável indexada” para se referir às variáveis compostas, pois sempre existirão índices (`index`) para os dados armazenados em cada espaço da variável composta (PIVA JUNIOR *et al.*, 2012).

As variáveis compostas são formadas a partir dos tipos primitivos e podem ser classificadas em homogêneas e heterogêneas. Além disso, podem ser unidimensionais ou multidimensionais, e a bidimensional é

mais comumente usada (MANZANO; MATOS; LOURENÇO, 2015). Quando armazenam valores do mesmo tipo primitivo, são homogêneas, mas quando armazenam valores de diferentes tipos, elas são heterogêneas.

VARIÁVEIS COMPOSTAS HOMOGÊNEAS UNIDIMENSIONAIS (VETORES)

As variáveis compostas `predio1` e `predio2`, ilustradas na Figura 1.11, são homogêneas, pois a primeira armazena apenas valores do tipo inteiro, e a segunda, somente do tipo decimal. Além disso, elas também são unidimensionais, e isso quer dizer que elas apresentam a estrutura de uma tabela contendo apenas 1 coluna e N linhas (o resultado não se altera se pensarmos em uma estrutura como uma tabela de 1 linha e N colunas). Esse tipo de estrutura de dados é chamado de **vetor** ou matriz unidimensional (MANZANO; MATOS; LOURENÇO, 2015).

Como o nome **vetor** é o mais utilizado entre os profissionais, adotaremos essa forma no restante do livro. Assim, vetores são conjuntos de dados de um único tipo dispostos de forma contígua, ou seja, um após o outro na memória. A criação de um vetor é similar a uma variável primitiva, tendo que acrescentar apenas um número entre colchetes indicando qual será o tamanho desse vetor (quantidade de blocos). Portanto, a sintaxe ficará da seguinte forma:

```
<tipo> <nome_do_vetor>[tamanho];
```

> **ASSIMILE**
>
> O vetor é uma estrutura de dados estática, ou seja, seu tamanho deve ser informado no momento da criação, pelo programador, e não é alterado por nenhum recurso em tempo de execução. Além disso, ao se criar um vetor com *N* posições, nem todas precisam ser utilizadas. Lembre-se, no entanto, de que quanto maior, mais espaço será reservado na memória de trabalho.

> **EXEMPLIFICANDO**

Vamos criar um vetor em C para armazenar a altura (em metros) de 3 pessoas. Veja no Código 1.5 que, na linha 3, foi criado o vetor *altura*, que foi inicializado com valores. Para armazenar valores no vetor no momento da criação, colocamos os elementos entre chaves ({ }) separados por vírgula. Da linha 4 a 6 é feita a impressão dos valores guardados no vetor *altura*.

Código 1.5 | Inicializando um vetor utilizando operador chaves ({ })

```
1  #include <stdio.h>
2  int main(){
3      float altura[3] = {1, 1.5, 1.7};
4      printf("\n Vetor altura[0] = %f",altura[0]);
5      printf("\n Vetor altura[1] = %f",altura[1]);
6      printf("\n Vetor altura[2] = %f",altura[2]);
7      return 0;
8  }
```

Fonte: elaborado pela autora.

Veja a execução do código utilizando a ferramenta Paiza.io. Observe os valores do vetor altura sendo impressos.

> Conforme pode ser visto no exemplo anterior, cada elemento no vetor é acessado por meio do seu índice, **que sempre começará pelo valor zero**. Por isso, no Código 1.5, na linha 4, usou-se `altura[0]` para imprimir o primeiro valor, `altura[1]` para imprimir o segundo e `altura[2]` para imprimir o terceiro.

Ainda a respeito do exemplo anterior, o índice do código apresentado é usado tanto para leitura como para escrita.

Outra forma de atribuir valores a um vetor altura é utilizando colchetes ([ ]), da seguinte maneira:

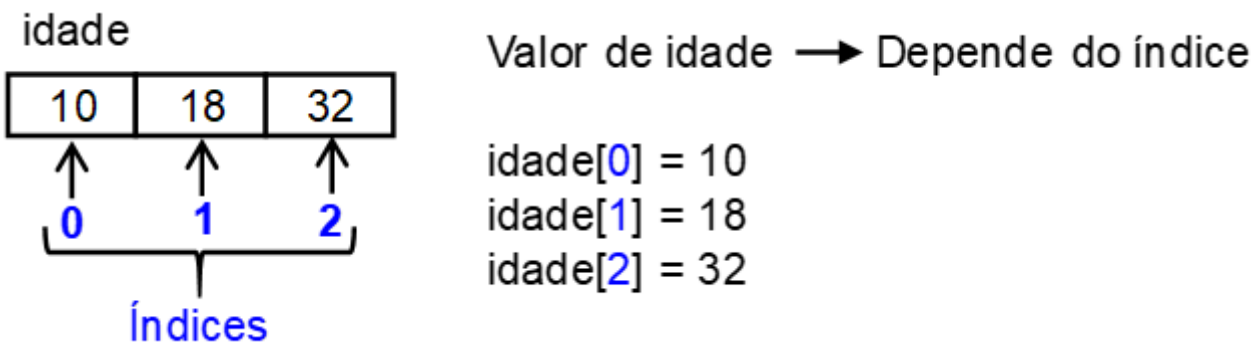
Código 1.6 | Atribuindo valores a um vetor com colchetes

```c
#include <stdio.h>
int main(){
    float altura[3];
    altura[0] = 1.50;
    altura[1] = 1.80;
    altura[2] = 2.02;
    printf("\n Vetor altura[0] = %f",altura[0]);
    printf("\n Vetor altura[1] = %f",altura[1]);
    printf("\n Vetor altura[2] = %f",altura[2]);
    return 0;
}
```

Fonte: elaborado pela autora.

Para ajudar a compreensão, observe a Figura 1.12, que representa um esquema para um vetor denominado *idade* na memória do computador. O valor do vetor depende da posição, ou seja, do índice. Ressalta-se que um vetor com N posições terá seus índices variando de 0 até *N-1*. Veja que o vetor `idade` tem capacidade de 3; com isso, seu índice varia de 0 até 2.

Figura 1.12 | Vetor idade



Fonte: elaborada pela autora.

Ver anotações

Na maioria dos programas, os dados a serem utilizados são digitados pelo usuário ou lidos de alguma base de dados. Para guardar um valor digitado pelo usuário em um vetor, usamos a função `scanf()`, porém, na variável, deverá ser especificado em qual posição do vetor se deseja guardar. Então, se quiséssemos guardar uma idade digitada pelo usuário na primeira posição do vetor, faríamos da seguinte forma:

```
printf("Digite uma idade: ");
scanf("%d",&idade[0]);
```

EXEMPLIFICANDO

Vamos criar um vetor para armazenar a idade de 3 pessoas, cujos valores são informados via teclado. Logo após a leitura das idades, o programa deve calcular e imprimir na tela a média de idade dessas pessoas. Veja no Código 1.7 que, na linha 3, foi criado o vetor do tipo inteiro, denominado *idade*. Ele foi inicializado com valores nulos (zero), o que é uma boa prática, para se evitar a realização de cálculos com lixo de memória. Logo após, na linha 4, declarou-se ainda uma variável primitiva, denominada `media`. Para armazenar valores no vetor, utilizou-se a função `scanf()` – linhas 6, 8 e 10. Nas linhas 11 a 12, são realizados, respectivamente, o cálculo da média das idades, bem como a impressão dela na tela.

Código 1.7 | Leitura de um vetor em C

```c
#include <stdio.h>
int main(void){
    int idade[3] = {0, 0, 0};
    float media = 0.0;
    printf("\n Informe a idade da pessoa 1: ");
    scanf("%d", &idade[0]);
    printf("\n Informe a idade da pessoa 2: ");
    scanf("%d", &idade[1]);
    printf("\n Informe a idade da pessoa 3: ");
    scanf("%d", &idade[2]);
    media = (idade[0] + idade[1] + idade[2]) / 3;
    printf("\n Média de idade = %.2f", media);
    return 0;
}
```

Fonte: elaborado pela autora.

Utilize o Paiza.io para executar o código acima.

VETOR DE CARACTERES (*STRING*)

Como é formada uma palavra? Por uma sequência de caracteres, correto? Vimos que o caractere é um tipo primitivo nas linguagens de programação. Sabendo que uma palavra é uma cadeia de caracteres, podemos concluir que esta é, na verdade, um vetor de caracteres (MANZANO; MATOS; LOURENÇO, 2015). No mundo da programação, um vetor de caracteres é chamado de *string*, portanto adotaremos essa nomenclatura.

A declaração de uma *string* em C é feita da seguinte forma:

```
char <nome_da_string>[tamanho];
```

Exemplos:

```
char nome[16];
char sobrenome[31];
char frase[101];
```

Ver anotações

Ao criarmos uma *string* em C, temos que nos atentar ao seu tamanho, pois a última posição da *string* é reservada pelo compilador, que atribui o valor "\0" para indicar o final da sequência. Portanto, a *string* `nome[16]` apresenta 15 "espaços" disponíveis para serem preenchidos.

A atribuição de valores à *string* pode ser feita no momento da declaração de duas formas: (i) entre chaves informando cada caractere, separados por vírgula (igual aos outros tipos de vetores); e (ii) atribuindo a palavra (ou frase) entre aspas duplas.

Exemplos:

```
char nome[16]={'J','o','a','o'};
char sobrenome[31] = "Alberto Gomes";
```

Existem várias funções que fazem a leitura e a impressão de *string* em C. Estudaremos duas delas. A primeira já é conhecida, a função `scanf()`, mas agora usando o especificador de formato `%s` para indicar que será armazenada uma *string*. Veja o código responsável por armazenar o nome digitado por um usuário:

```
char nome[16];
printf("\n Digite um nome:");
scanf("%s", nome);
printf("\n Nome digitado: %s", nome);
```

Uma observação importante é que nesse caso o operador & não é usado na função `scanf()`. Isso ocorre porque, ao usarmos o nome de um vetor sem a especificação de um índice para ele, o compilador já entende que se trata de um endereço de memória.

Essa forma de atribuição tem uma limitação: só é possível armazenar palavras simples; compostas, não. Isso acontece porque a função `scanf()` interrompe a atribuição quando encontra um espaço em branco. Para contornar essa limitação, uma opção é usar a função `fgets()`, que também faz parte do pacote padrão `<stdio.h>`. Essa função apresenta a seguinte sintaxe:

```
fgets(destino,tamanho,fluxo);
```

O *destino* especifica o nome da `string` que será usada para armazenar o valor lido do teclado. O *tamanho* deve ser o mesmo da declaração da `string`. O *fluxo* indica de onde está vindo a `string`, no nosso caso, sempre virá do teclado, portanto usaremos `stdin` (*standard input*).

Exemplo:

```
char frase[101];
printf("\n Digite uma frase:");
fflush(stdin);
fgets(frase, 101, stdin);
printf("\n Frase digitada: %s", frase);
```

Veja que antes de usar o `fgets()`, usamos a função `fflush(stdin)`. Apesar de não ser obrigatório, isso é uma boa prática, pois garante que a entrada padrão (`stdin`) seja limpa (sem informações de leituras anteriores) antes de armazenar.

> **REFLITA**
>
> O uso da função `fflush()`, conforme comentado anteriormente, tem suas limitações:
>
> - O programador pode esquecer de chamar essa função antes de realizar a leitura de uma string.
> - Para cada leitura com `fgets()`, o programador precisará fazer também uma chamada para `fflush()`.
>
> Você consegue imaginar alguma forma de minimizar essas limitações? Apesar de você não ter visto ainda como declarar novas funções, você já sabe para que elas servem e como utilizá-las. Então, a ideia seria criar uma função, por exemplo `lerString()`, a qual executaria a função `fflush()` e depois

> `fgets()`. Assim, o programador poderia usar sempre a função `lerString()`, sem ter que se preocupar com detalhes de limpeza da entrada padrão (`stdin`).
>
> Aguarde, pois voltaremos a esse assunto em uma seção apropriada.

VARIÁVEIS COMPOSTAS HOMOGÊNEAS BIDIMENSIONAIS (MATRIZES)

Observe a Tabela 1.3, que apresenta alguns dados fictícios sobre a temperatura da cidade de São Paulo em uma determinada semana. Essa tabela representa uma estrutura de dados matricial com 7 linhas e 2 colunas. Para armazenarmos esses valores em um programa de computador, precisamos de uma variável composta bidimensional ou, como é mais comumente conhecida, uma matriz bidimensional.

Tabela 1.3 | Temperatura máxima de São Paulo na última semana

Dia	**Temperatura (°C)**
1	26,1
2	27,7
3	30,0
4	32,3
5	27,6
6	29,5
7	29,9

Fonte: elaborada pela autora.

Para criarmos uma matriz em C usamos a seguinte sintaxe:

```
<tipo_dado> <nome_da_matriz>[numero_linhas][numero_colunas];
```

Observe o exemplo de declaração a seguir com uma matriz de inteiros e uma matriz de float.

```
1. int coordenadas[3][2];

2. float temperaturas[7][2];
```

Na linha 1 do código é criada uma estrutura com 3 linhas e 2 colunas. Na linha 2, é criada a estrutura da Tabela 1.3 (7 linhas e 2 colunas).

A criação e manipulação de matrizes bidimensionais exige a especificação de dois índices, portanto, a atribuição de valores deve ser feita da seguinte forma:

```
matriz[M][N] = valor;
```

M representa a linha que se pretende armazenar e **N**, a coluna. Assim como nos vetores, aqui os índices sempre iniciarão em zero.

Vamos criar uma matriz em C para armazenar as notas do primeiro e segundo bimestre de três alunos. Veja, na linha 3 do Código 1.8, que criamos uma matriz chamada notas, com 3 linhas e 2 colunas, o que significa que serão armazenados 6 valores (linhas x colunas). Nas linhas 6 e 7 do código, armazenamos as notas do primeiro aluno: veja que a linha não se altera (primeiro índice), já a coluna sim (segundo índice). O mesmo acontece para o segundo e terceiro alunos, que são armazenados respectivamente na segunda e terceira linhas da matriz.

No código, os textos que aparecem após o símbolo // são chamados de comentários. Eles não têm qualquer efeito sobre o comportamento do programa, pois são descartados durante o processo de compilação do código. Comentários servem, então, para melhorar a legibilidade do código para o programador que vai precisar, ler, entender e manter o programa futuramente.

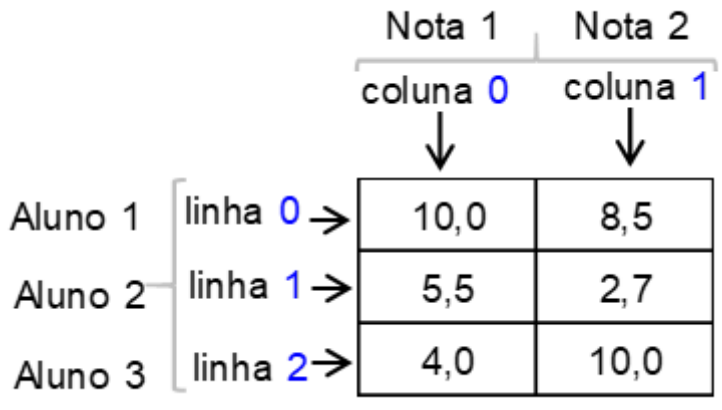
Código 1.8 | Matriz em C

```c
#include <stdio.h>
int main(){
    float notas[3][2];

    //aluno 1
    notas[0][0] = 10;
    notas[0][1] = 8.5;

    //aluno 2
    notas[1][0] = 5.5;
    notas[1][1] = 2.7;

    //aluno 3
    notas[2][0] = 4;
    notas[2][1] = 10;
    return 0;
}
```

Fonte: elaborado pela autora.

A Figura 1.13 ilustra a estrutura de dados que é criada na memória para o código do Código 1.8. Veja que as linhas foram usadas para representar os alunos e as colunas para as notas.

Figura 1.13 | Esquema de atribuição de notas



Fonte: elaborada pela autora.

Para armazenar valores digitados pelo usuário em uma matriz, usamos a função `scanf()`, indicando os dois índices para selecionar a posição que se deseja guardar. Para impressão de valores, também devemos

selecionar a posição por meio dos dois índices.

Exemplo:

```
float notas[3][2];
printf("Digite uma nota: ");
scanf("%f", as[1][0]);
printf("Nota digitada: %.2f", notas[1][0]);
```

Nesse exemplo, a nota digitada será guardada na linha 1, coluna 0, e será exibida na tela usando duas casas decimais.

VARIÁVEIS COMPOSTAS HETEROGÊNEAS (*STRUCTS*)

Já sabemos como otimizar o uso de variáveis usando as estruturas compostas (vetor e matriz). Porém, só podemos armazenar valores de um mesmo tipo. Além das estruturas homogêneas, a linguagem de programação C oferece variáveis compostas heterogêneas chamadas de **estruturas** (*`structs`*) ou, ainda, de registros por alguns autores (SOFFNER, 2013).

Assim como associamos os vetores e as matrizes a tabelas, podemos associar uma estrutura a uma ficha de cadastro com diversos campos. Por exemplo, o cadastro de um cliente poderia ser efetuado a partir da inserção do nome, idade, CPF e endereço em uma *`structs`*.

Na linguagem C, a criação de uma estrutura deve ser feita antes da função `main()` e deve apresentar a seguinte sintaxe:

```
struct < nome >{
  <tipo> <nome_da_variavel1>;
  <tipo> <nome_da_variavel2>;
  ...
};
```

`<nome>` é o nome da estrutura, por exemplo, cliente, carro, fornecedor. As variáveis internas são os campos nos quais se deseja guardar as informações dessa estrutura. Na prática, uma estrutura funciona como

um “tipo de dado” e seu uso sempre será atribuído a uma ou mais variáveis.

> **EXEMPLIFICANDO**
>
> Vamos criar uma estrutura para armazenar o modelo, o ano e o valor de um automóvel.
>
> No Código 1.9, “Struct em C”, a estrutura `automovel` foi criada entre as linhas 3 e 7. Mas, para utilizar essa estrutura, na linha 10 foi criada a variável `dadosAutomovel1`, que é do tipo `struct automóvel`. Somente após essas especificações é que podemos atribuir algum valor para as variáveis dessa estrutura de dados.
>
> Código 1.9 | Struct em C
>
> ```
> 1  #include<stdio.h>
> 2
> 3  struct automovel{
> 4      char modelo[20];
> 5      int ano;
> 6      float valor;
> 7  };
> 8
> 9  int main(){
> 10     struct automovel dadosAutomovel1;
> 11 }
> ```
>
> Fonte: elaborado pela autora.
>
> O acesso aos dados da *`struct`* (leitura e escrita) é feito com a utilização do nome da variável que recebeu como tipo a estrutura com um ponto (.) e o nome do campo da estrutura a ser acessado. Por exemplo, para acessar o campo ano da `struct automovel` do Código 1.9, “Struct em C”, devemos escrever: `dadosAutomovel1.ano`.

Veja no Paiza.io como realizar “Acesso aos dados de uma estrutura em C” a implementação completa para guardar valores digitados pelo usuário na estrutura `automovel` e depois imprimi-los.

VARIÁVEL DO TIPO PONTEIRO

Além das variáveis primitivas e compostas, existe um tipo de variável muito importante na linguagem C. São os chamados **ponteiros**, por meio dos quais podemos manipular outras variáveis e outros recursos por meio do acesso ao endereço de memória (SOFFNER, 2013). Existe uma relação direta entre um ponteiro e endereços de memória e é por esse motivo que esse tipo de variável é utilizado, principalmente para manipulação de memória, dando suporte às rotinas de alocação dinâmica (MANZANO; MATOS; LOURENÇO, 2015).

Variáveis do tipo ponteiro são usadas exclusivamente para armazenar endereços de memória. O acesso à memória é feito usando dois operadores: o asterisco (*), utilizado para declaração do ponteiro, e o “&”, que, como já vimos, é usado para acessar o endereço da memória, por isso é chamado de **operador de referência**. A sintaxe para criar um ponteiro tem como padrão:

```
<tipo> *<nome_do_ponteiro>;
```

Exemplo: `int *idade;`

Nesse exemplo, é criado um ponteiro do tipo inteiro, e isso significa que ele deverá “apontar” para o endereço de uma variável desse tipo. A criação de um ponteiro só faz sentido se for associada a algum endereço de memória. Para isso, usa-se a seguinte sintaxe:

```
1. int ano = 2018;

2. int *ponteiro_para_ano = &ano;
```

Na linha 1 criamos uma variável primitiva inteira, denominada ano, com valor 2018, e na linha 2 associamos um ponteiro chamado `ponteiro_para_ano` ao endereço da variável primitiva `ano`. Agora tudo que estiver atribuído à variável ano estará acessível ao ponteiro `ponteiro_para_ano`. A Figura 1.15 ilustra, de forma simplificada, o esquema de uma variável com um ponteiro na memória. O conteúdo do ponteiro é o endereço da variável a que ele aponta e que ele também ocupa espaço na memória.

Figura 1.15 | Ponteiro e variável primitiva na memória

Fonte: elaborada pela autora.

Para finalizar esta seção, vamos ver como imprimir as informações de um ponteiro. Podemos imprimir o conteúdo do ponteiro, que será o endereço da variável a que ele aponta. Utilizando o ponteiro criado anteriormente (`ponteiro_para_ano`), temos a seguinte sintaxe:

```
printf("\n Conteúdo do ponteiro: %p", ponteiro_para_ano);
```

O especificador de formato %p é usado para imprimir o endereço de memória armazenado em um ponteiro, em hexadecimal (o %x também poderia ser usado).

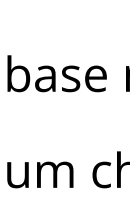
Também podemos acessar o conteúdo da variável que o ponteiro aponta, com a seguinte sintaxe:

```
printf("\n Conteúdo da variável pelo ponteiro: %d",
*ponteiro_para_ano);
```

A diferença do comando anterior é o asterisco antes do nome do ponteiro.

Podemos, também, imprimir o endereço do próprio ponteiro, por meio da seguinte sintaxe:

```
printf("\n Endereco do ponteiro: %p", &ponteiro_para_ano);
```

> **EXEMPLIFICANDO**
>
> O nome de um vetor nada mais é do que um ponteiro para o endereço de memória do seu primeiro elemento. Para isso, faça o seguinte:
>
> 
>
> Execute o código no Paiza.io e veja que a saída é exatamente a mesma, ou seja, pode-se utilizar o índice 0 do vetor ou o operador * para obter o valor armazenado no endereço de memória apontado por `num`.

Agora, imagine como o compilador sabe qual endereço de memória acessar para um índice X do vetor?

Vejamos: se sabemos o endereço de memória do primeiro elemento e também sabemos qual é o tamanho do bloco de memória alocado para cada elemento (isso é obtido com base no tipo do vetor, por exemplo, um inteiro ocupa 4 bytes, um char ocupa 1 byte etc.), então fica fácil saber qual o endereço de memória do índice X.

*Endereço de memória do índice X = endereço inicial + (tamanho do bloco de memória de um elemento * X)*

Assim, supondo que o endereço de memória inicial do vetor `num` seja 1080, que um inteiro ocupe 4 bytes de memória, então a posição de memória do elemento da posição 1 do vetor é dada por:

*Endereço do num[1] = 1080 + (4 * 1) = 1084*

Ver anotações

> Por isso é tão rápido para um computador acessar qualquer elemento de um vetor.

A compreensão do uso das variáveis é imprescindível, pois conheceremos métodos que podem ser usados para resolver problemas nos quais os dados são a matéria-prima. Continue seus estudos!

FAÇA VALER A PENA

Questão 1

Variáveis são usadas para guardar valores temporariamente na memória do computador. A linguagem C oferece recursos para que seja possível conhecer o endereço de memória que foi alocado. Durante a execução de um programa, uma variável pode assumir qualquer valor, desde que esteja de acordo com o tipo que foi especificado na sua criação.

A respeito dos tipos primitivos de variáveis, assinale a alternativa correta.

a. Todas as linguagens de programação têm os mesmos tipos primitivos de dados.

b. Para todos os tipos primitivos na linguagem C são alocados os mesmos espaços na memória.

c. Os valores numéricos podem ser armazenados em tipos primitivos inteiros ou de ponto flutuante.

d. O número 10 é inteiro e por isso não pode ser guardado em uma variável primitiva do tipo `float`.

e. O número 12.50 é decimal e por isso não pode ser guardado em uma variável primitiva do tipo `int`, pois gera um erro de compilação.

Questão 2

Constantes são usadas para guardar temporariamente valores fixos na memória de trabalho. O valor armazenado em uma constante não pode ser alterado em tempo de execução e essa é a principal diferença com relação às variáveis.

Ver anotações

A respeito das constantes na linguagem C, assinale a alternativa correta.

a. Valores constantes podem ser declarados usando o comando `#define <nome> <valor>.`

b. O comando `const int a = 10` alocará um espaço de 1 byte para a constante a.

c. O comando `const int a = 13.4` resultará em um erro de compilação.

d. O compilador trata o comando *const* da mesma forma que trata o comando *define*.

e. A constante definida pelo comando `#define g 9.8` ocupará 4 bytes na memória.

Ver anotações

Questão 3

A linguagem C de programação utiliza especificadores de formato para identificar o tipo de valor guardado nas variáveis e constantes. Eles devem ser usados tanto para leitura de um valor como para a impressão. Quando um programa é executado, o compilador usa esses elementos para fazer as devidas referências e conexões, por isso o uso correto é fundamental para os resultados.

Considerando o código apresentado, analise as asserções em seguida e, depois, escolha a opção correta.

```c
#include <stdio.h>

int main(){
    int idade = 0;
    float salario = 0;
    char a_letra = 'a';
    char A_letra = 'A';
}
```

I. O comando `scanf(“%f”,&idade)` guardará o valor digitado na variável idade.

II. O comando `printf(“%d”,a_letra)` imprimirá a letra *a* na tela.

III. O comando `printf(“%c”,A_letra)` imprimirá a letra *A* na tela.

Com base no contexto apresentado, é correto o que se afirma apenas em:

a. I.

b. II.

c. I e II.

d. III.

e. II e III.

Ver anotações

REFERÊNCIAS

AGUILAR, L. J. **Fundamentos de programação**: algoritmos, estruturas de dados e objetos. 3. ed. Porto Alegre: AMGH, 2011.

BARANIUK, C. As falhas numéricas que podem causar desastres. **BBC News Brasil**, 14 maio 2015. Disponível em: https://bbc.in/3llL7Xm. Acesso em: 20 out. 2020.

CORDEIRO, T. O que foi o Bug do Milênio? **Super Interessante**, São Paulo, 4 jul. 2018. Disponível em: https://bit.ly/35khiRl. Acesso em: 23 out. 2020.

DEITEL, P.; DEITEL, H. C. **Como programar**. 6. ed. São Paulo: Pearson, 2011.

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron, 2000.

GEEKS FOR GEEKS. **Errors in C/C++**. Geeks for Geeks, Noisa, Uttar Pradesh, 4 jan. 2019. Disponível em: https://bit.ly/3phFzPZ. Acesso em: 21 out. 2020.

LOPES, A.; GARCIA, G. **Introdução à programação**. 8. reimp. Rio de Janeiro: Elsevier, 2002.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. 2. ed. São Paulo: Érica, 2015.

MICROSOFT. **Math Constants**. Syntax. Visual Studio. Microsoft, 2016. Disponível em: https://bit.ly/32ADhSd. Acesso em: 11 mar. 2020.

ONLINE GDB BETA. Página inicial. GDB Online, 2020. Disponível em: https://www.onlinegdb.com/. Acesso em: 20 out. 2020.

PAIZA.IO. Página inicial. Paiza Inc., Tóquio, 2020. Disponível em: https://paiza.io/en. Acesso em: 29 out. 2020.

PEREIRA, S. do L. **Linguagem C**. Ime: USP, 2017. Disponível em: https://bit.ly/3naTKEA. Acesso em: 11 mar. 2020.

PEREIRA, A. P. **O que é algoritmo?** TecMundo, 12 maio 2009. Disponível em: https://bit.ly/2GVDsjC. Acesso em: 19 mar. 2018.

PIVA JUNIOR, D. *et al*. **Algoritmos e programação de computadores**. Rio de Janeiro: Elsevier, 2012.

SALIBA, W. L. C. **Técnica de programação**: uma abordagem estruturada. São Paulo: Makron, 1993.

SANTOS, D. **Processamento da linguagem natural**: uma apresentação através das aplicações. Organização: Ranchhod. Lisboa: Caminho, 2001.

SCHILDT, H. **C Completo e total**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 1994.

FOCO NO MERCADO DE TRABALHO

COMPONENTES E ELEMENTOS DE LINGUAGEM DE PROGRAMAÇÃO

Vanessa Cadan Scheffer

CRIAR A FUNCIONALIDADE PARA O SOFTWARE

O que é necessário saber para implementar a solução na linguagem C? Conceitos de variáveis, constantes, *structs* e *strings*? Vamos lá?

Fonte: Shutterstock.

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Para ampliar sua visão acerca das possibilidades de aplicação dos conhecimentos obtidos até o momento, vamos resolver a situação-problema proposta anteriormente: é hora de estruturar a solução para a nova funcionalidade do software da empresa.

Para tal, inicialmente precisamos declarar as variáveis de entrada do problema em questão. Como nome e idade do cliente pertencem a uma mesma entidade – a saber, um cliente –, podemos trabalhar com o conceito de `structs`. Podemos usar, também, o conceito de constante para armazenar o tamanho da `string` que armazenará o nome do cliente. Para armazenar a classificação do filme, basta uma variável primitiva do tipo inteiro.

Quanto ao processo de leitura e impressão dos valores dessas variáveis, deve-se estar atento ao uso do operador ponto (.) para acessar campos de um `struct`. Também é importante lembrar-se do uso das funções `fflush()` e `fgets()` para leitura de *strings*.

Código 1.10 | Classificação indicativa da locadora de filmes on-line

```c
#include <stdio.h>

#define TAM_NOME_CLIENTE 100

struct cliente {
    char nome[TAM_NOME_CLIENTE];
    int idade;
};

int main(){
    struct cliente cli;
    int classificacao_filme;

    printf("\n Informe o nome do cliente: ");
    fflush(stdin);
    fgets(cli.nome, TAM_NOME_CLIENTE, stdin);

    printf("\n Informe a idade do cliente: ");
    scanf("%d", &cli.idade);

    printf("\n Informe a classificação do filme: ");
    scanf("%d", &classificacao_filme);

    printf("\n Cliente: %s", cli.nome);
    printf("\n Idade: %d anos", cli.idade);
    printf("\n Classificação do filme: %d anos",
classificacao_filme);
    return 0;
}
```

Fonte: elaborado pela autora.

Utilize o Paiza.io para executar o código acima.

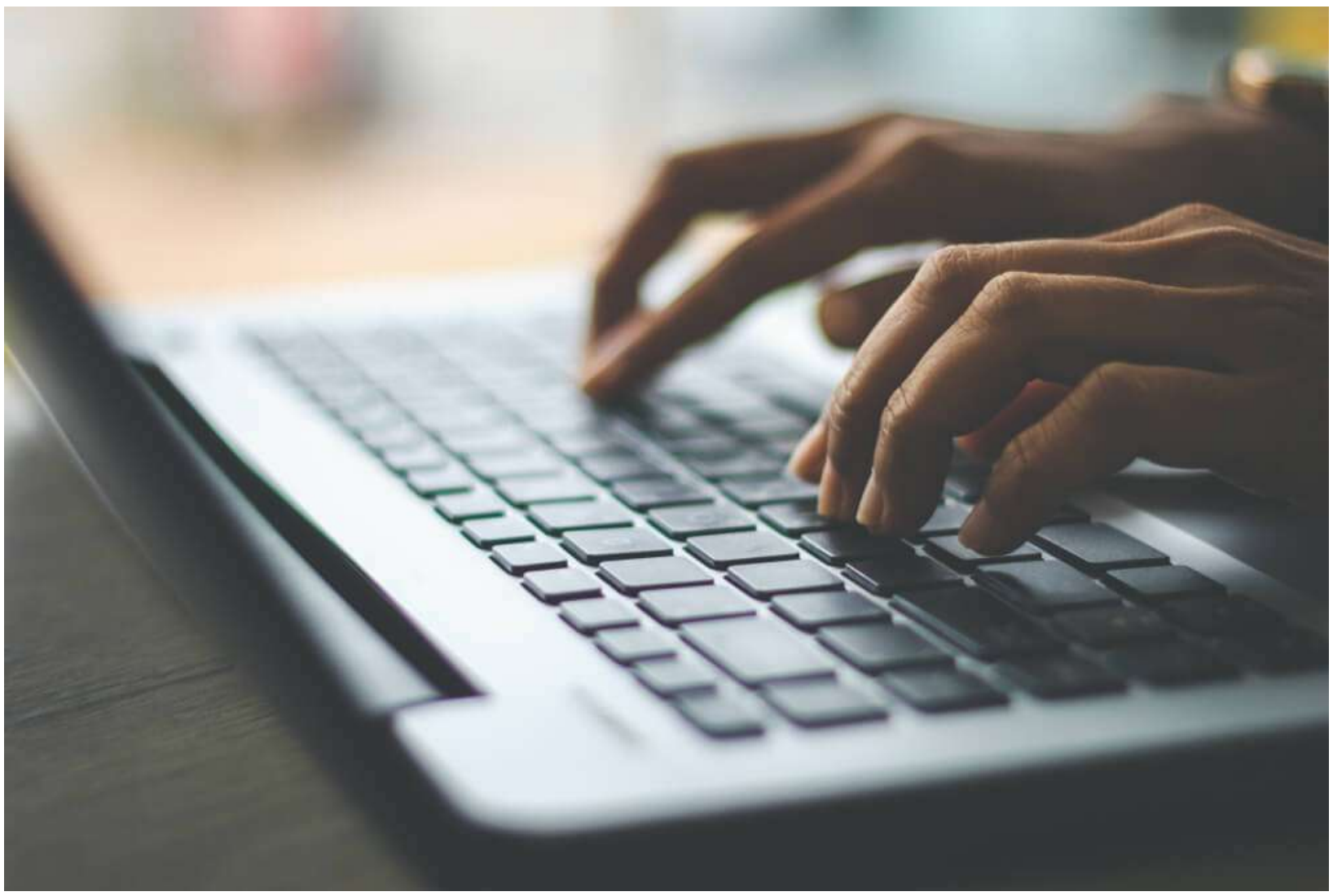
NÃO PODE FALTAR

OPERAÇÕES E EXPRESSÕES

Vanessa Cadan Scheffer

LEITURA DOS DADOS

A leitura dos dados é feita para processar e gerar informações. Essa etapa é construída com base na combinação de operações aritméticas, relacionais, lógicas e outras técnicas de programação.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Você já está dominando os tipos de dados que podem ser utilizados em um sistema computacional e já sabe como declará-los e guardá-los temporariamente em variáveis. Além disso, ao longo desta seção você aprenderá a processar os dados por meio de operações e expressões matemáticas e a combiná-las usando operadores lógicos.

A fim de colocarmos em prática os conhecimentos a serem aprendidos, vamos analisar a seguinte situação-problema: Seu chefe está muito contente, pois ele sabe que agora você será capaz de resolver problemas mais desafiadores na empresa. Sua última atividade foi escrever um programa na linguagem C que fosse capaz de ler a idade e o nome do cliente, bem como a classificação do filme que ele deseja locar. Posteriormente, seu programa deveria imprimir todas essas informações na tela, conforme o padrão a seguir:

```
Cliente: José das Couves
Idade: 18 anos
Classificação do filme: 12 anos
```

Agora, seu chefe pediu para você melhorar esse programa, de forma que ele seja capaz de mostrar ao cliente:

1. Se o filme desejado está disponível ou não.
2. Se o filme desejado pode ser locado pelo cliente, levando em consideração se o filme está disponível e se ele é indicado para sua faixa etária.
3. Quantos anos faltam para que a classificação do filme seja adequada à faixa etária do cliente.

CONCEITO-CHAVE

Desde o momento em que você liga um computador, tablet ou smartphone, centenas de processos são iniciados e passam a competir no processador para que possam ser executados e para que a “mágica” do mundo digital possa acontecer. Todos os resultados desses sistemas

são obtidos por meio do processamento de dados e, nesta seção, vamos estudar os recursos que lhe permitirão implementar soluções com processamento.

Sistemas computacionais são construídos para resolver os mais diversos problemas. Todos esses sistemas, independentemente da sua aplicação, são construídos em três partes: entrada, processamento e saída. Na entrada, os valores utilizados pelo processamento são lidos basicamente a partir de três fontes:

- **Digitados pelo usuário**: nesse caso, é a partir de uma interface textual ou gráfica que o usuário alimenta o sistema com dados.
- **Leitura de arquivos**: é possível implementar sistemas que fazem a leitura de dados com base em arquivos de texto e planilhas, entre outros.
- **Acesso a banco de dados**: nesse caso, são usados programas que fazem o gerenciamento da base de dados, à qual o sistema computacional tem acesso.

Nos três casos, a leitura dos dados é feita apenas para processar e gerar informações, e essa etapa é construída com base na combinação de operações aritméticas, relacionais, lógicas e outras técnicas de programação que você conhecerá no decorrer desta seção.

OPERADORES ARITMÉTICOS NA LINGUAGEM C

Vamos começar a aprimorar nossos algoritmos com as operações aritméticas. Observe o Quadro 1.3, que demonstra algumas operações disponíveis na linguagem C e seus respectivos exemplos.

Quadro 1.3 | Operações aritméticas básicas em linguagens de programação

Operador	Descrição	Exemplo	Resultado
+	Adição	z = x + y 4 + 2	z = 6

Operador	Descrição	Exemplo	Resultado
-	Subtração	z = x – y 4 – 2	z = 2
*	Multiplicação	z = x * y 4 * 2	z = 8
/	Divisão (Quociente)	z = x/y 4 / 2	z = 2
%	Módulo (resto de uma divisão)	z = x % y 4 % 2	z = 0

Fonte: adaptado de Manzano (2015, p. 43).

EXEMPLIFICANDO

Vamos criar um programa em C que some a idade de duas pessoas e imprima na tela o resultado. No Código 1.11 está o código que realiza tal processo. Veja que, primeiramente, as idades foram solicitadas e armazenadas em duas variáveis (linhas de 7 e 9 – entrada pelo usuário), para depois ser feito o processamento (linha 10) e, por fim, a exibição do resultado na tela (linha 11).

Código 1.11 | Soma da idade de duas pessoas

```c
#include <stdio.h>
int main(){
    int idade1 = 0;
    int idade2 = 0;
    int resultado=0;
    printf("\n Digite a primeira idade: ");
    scanf("%d", &idade1);
    printf("\n Digite a segunda idade: ");
    scanf("%d", &idade2);
    resultado = idade1 + idade2;
    printf("\n Resultado = %d", resultado);
}
```

Fonte: elaborado pela autora.

Agora é com você !

Para visualizar o objeto, acesse seu material digital.

Quando trabalhamos com operadores, a **ordem de precedência** é muito importante. Segundo Soffner (2013), os operadores aritméticos apresentam a seguinte ordem de execução:

1. Parênteses.
2. Potenciação e radiciação.
3. Multiplicação, divisão e módulo.
4. Soma e subtração.

REFLITA

Ao implementar uma solução em software, um dos maiores desafios é garantir que a lógica esteja correta e, tratando-se da parte de processamento, ao escrever uma expressão matemática, é preciso atentar à ordem de precedência dos operadores. Se, ao implementar uma solução que calcula a

média aritmética, você usar a expressão `resultado = a + b / c`, você terá o resultado correto? Se for um cálculo para o setor financeiro de uma empresa, seu cálculo mostraria lucro ou prejuízo?

Ver anotações

Das operações aritméticas apresentadas no Quadro 1.3, a operação módulo (%) talvez seja a que você ainda não tenha familiaridade. Essa operação faz a divisão de um número considerando somente a parte inteira do quociente e retorna o resto da divisão.

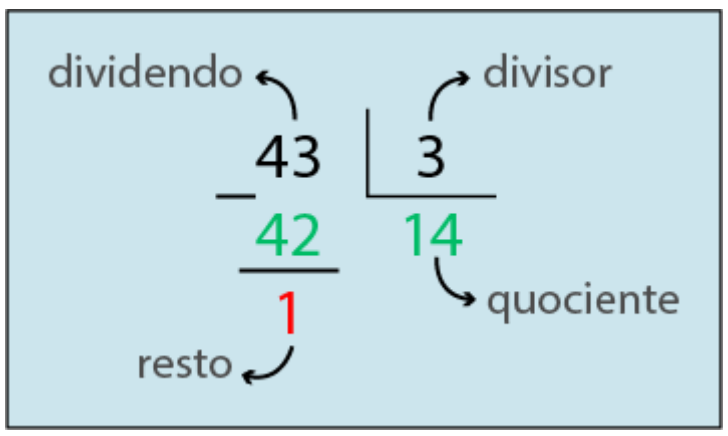
EXEMPLIFICANDO

Vamos aplicar o operador módulo para efetuar o processamento de 43 % 3.

```
1  #include <stdio.h>
2  int main(){
3      int resultado = 43 % 3;
4      printf("Operacao modulo 43 % 3 = %d",
   resultado);
5  }
```

Ao executar o código, obtém-se como resultado o resto da divisão, ou seja, nesse caso, o valor 1. Veja na Figura 1.16 o cálculo matemático que é efetuado e como o resultado é obtido., você terá o resultado correto? Se for um cálculo para o setor financeiro de uma empresa, seu cálculo mostraria lucro ou prejuízo?

Figura 1.16 | Operação aritmética módulo



Fonte: elaborada pela autora.

Agora é com você!

Para visualizar o objeto, acesse seu material digital.

REFLITA

Como podemos usar o operador aritmético módulo (%) para identificar se um número inteiro é ímpar ou par?

Ver anotações

Os operadores aritméticos podem ser classificados em **unário ou binário** (MANZANO, 2015). Os binários, que nós já conhecemos no Quadro 1.3, são operadores que usam dois componentes (operandos); já os operadores unários usam apenas um componente (operando). É o caso dos operadores aritméticos de incremento (++) e decremento (--). Esses operadores acrescentam ou diminuem “um” ao valor de uma variável e podem ser usados de duas formas:

- **Após a variável:**
  - Pós-incremento: x++; nesse caso, é adicionado 1 (um) após a primeira execução.
  - Pós-decremento: x- -; nesse caso, é decrementado 1 (um) após a primeira execução.
- **Antes da variável:**
  - Pré-incremento ++x; nesse caso, é adicionado 1 (um) antes da primeira execução.
  - Pré-decremento --x; nesse caso, é decrementado 1 (um) antes da primeira execução.

O Quadro 1.4 apresenta um resumo dos operadores unários.

Quadro 1.4 | Operadores aritméticos unários

Operador	Descrição	Exemplo	Resultado
++	Pós-incremento	x++	x+1

Operador	Descrição	Exemplo	Resultado
++	Pré-incremento	++x	x+1
--	Pós-decremento	y--	y-1
--	Pré-decremento	--y	y-1

Fonte: elaborado pela autora.

EXEMPLIFICANDO

Para entender a diferença entre as formas de uso dos operadores de incremento e decremento (pré e pós), vejamos os exemplos a seguir:

```
1  #include <stdio.h>
2  int main(void){
3          int x = 0;
4      int resultado = x++;
5      printf("\n Resultado = %d", resultado);
6      printf("\n X = %d", x );
7  }
```

Na linha 4, o operador pós-incremento é utilizado juntamente com o operador de atribuição (=). Isso significa que, primeiramente, o valor da variável x será atribuído à variável `resultado` e só depois (por isso se chama operador pós-incremento) o valor de x será incrementado de 1. Assim, o resultado da execução do programa é:

```
Resultado = 0
X = 1
```

Por outro lado, se tivéssemos utilizado o operador pré-incremento, o resultado seria diferente.

Faça o teste você mesmo. Altere a expressão da linha 4 do programa acima para: `int resultado = ++x;`

Nesse caso, a variável x seria, primeiramente, incrementada de 1 e somente depois seu valor seria atribuído à variável `resultado`. Portanto, a saída do programa seria:

```
Resultado = 1
X = 1
```

OPERADORES RELACIONAIS NA LINGUAGEM C

Faz parte do processamento de um programa fazer comparações entre valores, para, a partir do resultado, realizar novas ações. Por exemplo, podemos criar um programa que some a nota de dois bimestres de um aluno e efetue a média aritmética. Com base no resultado, se o aluno obtiver média superior a seis, ele estará aprovado; caso contrário, estará reprovado. Veja que é necessário fazer uma comparação da média obtida pelo aluno com a nota estabelecida como critério.

Em programação, para compararmos valores, usamos operadores relacionais. O Quadro 1.5 apresenta os operadores usados na linguagem de programação C (DEITEL; DEITEL, 2011).

Quadro 1.5 | Operadores relacionais em linguagens de programação

Operador	Descrição	Exemplo
==	igual a	x==y
!=	diferente de	x!=y
>	maior que	x>y
<	menor que	x<y
>=	maior ou igual que	x>=y
<=	menor ou igual que	x<=y

Fonte: adaptado de Manzano (2015, p. 82).

Os operadores relacionais são usados para construir expressões booleanas, ou seja, expressões que terão como resultado verdadeiro (valor 1) ou falso (valor 0). Quando fazemos alguma operação relacional na linguagem C, se a expressão for verdadeira, o resultado será 1; caso contrário, retornará zero, se a expressão for considerada falsa.

> **ATENÇÃO**
>
> Não confunda o operador de comparação "==" com o operador de atribuição "=". Lembre-se: esses operadores têm diferentes propósitos, portanto, se ocorrer esse equívoco – que é comum na linguagem C –, se você utilizar o operador de comparação em uma atribuição, o compilador acusará um erro.

Vamos criar um programa que solicita ao usuário dois números inteiros e faz algumas comparações com esses valores. Veja, no Código 1.12, que na linha 9 comparamos se os números são iguais; na linha 10, se o primeiro (n1) é maior que o segundo (n2) e, na linha 11, se o primeiro é menor ou igual ao segundo.

Código 1.12 | Comparações entre dois números

```
1  #include <stdio.h>
2  int main(){
3      int n1 = 0;
4      int n2 = 0;
5      printf("\n Digite o primeiro numero: ");
6      scanf("%d", &n1);
7      printf("\n Digite o segundo numero: ");
8      scanf("%d", &n2);
9      printf("\n n1 e n2 são iguais? %d", n1 == n2);
10     printf("\n n1 e maior que n2? %d", n1 > n2);
11     printf("\n n1 e menor ou igual a n2? %d", n1 <= n2);
12 }
```

Fonte: elaborado pela autora .

Agora é com você!

OPERADORES LÓGICOS NA LINGUAGEM C

Além dos operadores relacionais, outro importante recurso para o processamento é a utilização de operadores lógicos, que têm como fundamento a lógica matemática clássica e a lógica booleana (GERSTING, 2017). O Quadro 1.6 apresenta os operadores lógicos que podem ser usados na linguagem C.

Quadro 1.6 | Operadores lógicos em linguagens de programação

Operador	Descrição	Exemplo
!	negação (*NOT* - NÃO)	!(x==y)
&&	conjunção (*AND* - E)	(x>y)&&(a==b)
||	disjunção (*OR* - OU)	(x>y)||(a==b)

Fonte: adaptado de Soffner (2013, p. 35).

Os operadores lógicos podem ser utilizados juntamente aos relacionais, unindo duas ou mais expressões relacionais simples e criando expressões mais complexas.

ASSIMILE

O operador de negação é usado para inverter o resultado da expressão. O operador de conjunção é usado para criar condições em que todas as alternativas devem ser verdadeiras para que o resultado seja verdadeiro. O operador de disjunção é usado para criar condições em que basta uma condição ser verdadeira para que o resultado também o seja.

Veja, no Código 1.13, o uso dos operadores relacionais e lógicos aplicados à comparação dos valores de três variáveis. Na linha 4, a condição criada será verdadeira caso o valor de a seja igual ao de b **e** o valor de a também seja igual a c. Nesse caso, como a primeira condição não é verdadeira, o resultado da expressão será 0. Na linha 5, a condição criada será verdadeira caso uma das condições seja satisfeita (isto é, seja verdadeira). Nesse caso, como a é igual a c, o resultado será 1. Por fim, na linha 6, invertemos esse resultado da expressão anterior com o operador de negação. Ou seja, o resultado será 0.

Ver anotações

Código 1.13 | Operadores relacionais e lógicos

```
1  #include <stdio.h>
2  int main(){
3      int a = 5, b = 10, c = 5;
4      printf("\n (a == b) && (a == c) = %d", ((a == b) && (a ==
   c)));
5      printf("\n (a == b) || (a == c) = %d", ((a == b) || (a ==
   c)));
6      printf("\n !(a == b) || (a == c) = %d", !((a == b) || (a
   == c)));
7  }
```

Fonte: elaborado pela autora.

FUNÇÕES PREDEFINIDAS PARA A LINGUAGEM C

Para facilitar o desenvolvimento de soluções em software, cada linguagem de programação oferece um conjunto de funções predefinidas, que ficam à disposição dos programadores. Entende-se por função “um conjunto de instruções que efetuam uma tarefa específica” (MANZANO, 2015, p. 153).

PESQUISE MAIS

> Existe uma série de bibliotecas e funções disponíveis na linguagem C que podem facilitar o desenvolvimento de soluções. No conteúdo, a seguir, você encontra uma vasta referência a esses elementos:
>
> - TUTORIALS POINT. **C Standard Library Reference Tutorial**. Tutorials point, 2020.

Nesta disciplina, você já usou algumas das funções da linguagem C, por exemplo, para imprimir uma mensagem na tela: usa-se a função `printf()`, que pertence à biblioteca `stdio.h`. Uma biblioteca é caracterizada por um conjunto de funções divididas por contexto (MANZANO, 2015). Vamos apresentar algumas funções que costumam aparecer com frequência nos programas implementados na linguagem C (Quadro 1.7).

Quadro 1.7 | Algumas bibliotecas e funções na linguagem C

Biblioteca	**Função**	**Descrição**

`<stdio.h>`	`printf()`
	`scanf()`
	`fgets(variavel, tamanho, fluxo)`
		Imprime na tela.
		Faz leitura de um dado digitado.
		Faz a leitura de uma linha digitada.

`<math.h>`	`pow(base,potencia)`
	`sqrt(numero)`
	`sin(angulo)`
	`cos(angulo)`
		Operação de potenciação.
		Calcula a raiz quadrada.
		Calcula o seno de um ângulo.
		Calcula o cosseno de um ângulo.

Biblioteca	Função	Descrição
`<string.h>`	`strcmp(string1, string2)` `strcpy(destino, origem)`	Verifica se duas *strings* são iguais. Copia uma *string* da origem para o destino.

`<stdlib.h>`	`malloc(tamanho)`
	`realloc(local,tamanho)`
	`free(local)`
		Aloca dinamicamente espaço na memória.
		Modifica um espaço já alocado dinamicamente.
		Libera um espaço alocado dinamicamente.

Fonte: adaptado de Tutorials point (2020).

A função `strcmp(string1, string2)` compara o conteúdo de duas *strings* e pode retornar três resultados, o valor nulo (zero), positivo ou negativo, conforme as seguintes regras:

- Quando as *strings* forem iguais, a função retorna 0.
- Quando as *strings* forem diferentes e o primeiro caractere não coincidir entre elas, sendo “maior” na primeira, a função retorna um valor positivo. Entende-se por “maior” o caractere com maior código ASCII, que é atribuído em ordem alfabética, ou seja, o caractere b é maior que a.
- Quando as *strings* forem diferentes e a primeira apresentar o caractere, não coincidente e “menor” que a segunda, então o valor resultante é negativo. Por exemplo, o caractere d é menor que h.
- Caso o primeiro caractere de ambas as *strings* seja igual, as duas regras anteriores são aplicadas para o próximo caractere, e assim sucessivamente.

ASSIMILE

Ver anotações

Cada função, independentemente da linguagem de programação, precisa ter um tipo de retorno, por exemplo, retornar um inteiro, um real, um booleano, entre outros. A função de comparação de strings, por exemplo, tem como tipo de retorno um número inteiro.

EXEMPLIFICANDO

Observe o uso da função `strcmp` no código a seguir.

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(void){
4      printf("\n ARARA == ARARA? %d", strcmp("ARARA",
   "ARARA"));
5      printf("\n ARARA == BANANA? %d",
   strcmp("ARARA", "BANANA"));
6      printf("\n BANANA == ARARA? %d",
   strcmp("BANANA", "ARARA"));
7  }
```

Teste o código na ferramenta Paiza.io. Clique em **Run** executar e visualizar as saídas do código.

Na primeira chamada da função, compara-se ARARA com ARARA. Como as *strings* são iguais, o resultado será 0. Na segunda chamada, a comparação é entre ARARA e BANANA. Como ARARA é “menor” que BANANA, alfabeticamente falando, então, o resultado será negativo (nesse caso, o valor impresso foi -1). Na última chamada da função, compara-se BANANA com ARARA. Como BANANA é “maior” que ARARA, alfabeticamente falando, então, o resultado será positivo (nesse caso, o valor impresso foi 1).

Note que, para a função `strcmp`, a ordem em que os parâmetros são passados é importante.

Finalizamos esta unidade, na qual exploramos as formas de armazenamento temporário de dados em diversos tipos de variáveis e aprendemos a utilizar os operadores para realizar o processamento dos dados.

FAÇA VALER A PENA

Questão 1

Todo sistema computacional é construído para se obter alguma solução automatizada. Uma das áreas promissoras da computação é a mineração de dados, que, como o nome sugere, refere-se a um determinado montante de dados e ao modo como eles podem ser minerados para gerar informações de valor. Dentro do processamento de informações, os operadores matemáticos, relacionais e lógicos são essenciais, pois são a base do processo.

Considerando a expressão `resultado = a + b * (c – b) / a` e os valores a=2, b=3 e c=5, escolha a opção correta.

a. O valor em resultado será 5.

b. O valor em resultado será 10.

c. O valor em resultado será 6.

d. O valor em resultado será 7.

e. O valor em resultado será 8.

Questão 2

Considerando o comando `printf("%d",((a > b) || (b < c) && (c < b)))`, é correto afirmar que:

I. O resultado será 1 (um) para a = 30, b = 20, c = 10

PORQUE

II. Para a expressão lógica proposta, basta que uma das condições seja verdadeira.

A respeito dessas asserções, assinale a alternativa correta.

Ver anotações

a. As asserções I e II são proposições verdadeiras e a II justifica a I.

b. As asserções I e II são proposições verdadeiras e a II não justifica a I.

c. A asserção I é uma proposição verdadeira e a II, falsa.

d. A asserção I é uma proposição falsa e a II, verdadeira.

e. As asserções I e II são proposições falsas.

Ver anotações 0

Questão 3

Algumas soluções em software necessitam de ordenação de *strings*; por exemplo, em um relatório de folha de pagamento dos funcionários de uma empresa. Para comparação entre *strings*, a fim de se obter a ordenação desejada, pode-se usar a função `strcmp`, da biblioteca `<string.h>`.

```c
#include <stdio.h>
#include <string.h>

#define TAM_NOME 100

struct funcionario {
    char nome[TAM_NOME];
    float salario;
};

int main(){
    struct funcionario f1, f2;
    strcpy(f1.nome, "Pedro");
    strcpy(f2.nome, "Maria");
    printf("\n %d", strcmp(f1.nome, f2.nome));
}
```

Considerando o código apresentado, escolha a alternativa correta:

a. Esse código não pode ser executado, pois apresenta erro de programação.

b. O resultado da execução do programa é um valor negativo.

c. O resultado da execução do programa é 0 (zero).

d. O resultado da execução do programa é um valor positivo.

e. O resultado da execução do programa é indeterminado.

REFERÊNCIAS

BACKES, A. **Linguagem C – Completa e Descomplicada**. São Paulo: Grupo GEN, 2018. Disponível em: https://bit.ly/3ku61Cx. Acesso em: 21 out. 2020.

DEITEL, P.; DEITEL, H. C. **Como programar**. 6. ed. São Paulo: Pearson, 2011.

DOMINGUES, P.; TÁVORA, V. Programação (in)segura – transbordo de memória. **Revista Programar**, ed. 52, mar. 2016. Disponível em: https://bit.ly/3niSpLR. Acesso em: 1 nov. 2020.

GERSTING, J. L. **Fundamentos matemáticos para a ciência da computação**: matemática discreta e suas aplicações. Rio de Janeiro: LTC, 2017.

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA (IBGE). **Frota de veículos**. Pesquisas. 2014. Disponível em: https://bit.ly/36l8EBq. Acesso em: 10 mar. 2020.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015. 480 p.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. 2. ed. São Paulo: Érica, 2015.

PEREIRA, S. do L. **Linguagem C**. Ime: USP, 2017. Disponível em: https://bit.ly/3llZIBT. Acesso em: 11 mar. 2020.

PIVA JUNIOR, D. *et al*. **Algoritmos e programação de computadores**. Rio de Janeiro: Elsevier, 2012.

SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.

TARIFA, A. Big Data: descubra o que é e como usar na sua empresa. **Endeavor**, [s.d.]. Disponível em: https://bit.ly/3eVuyPx. Acesso em: 23 jul. 2018.

TUTORIALS POINT. **C Standard Library Reference Tutorial**. Tutorials point, 2020. Disponível em: https://bit.ly/3ls9BOQ. Acesso em: 23 jul. 2018.

FOCO NO MERCADO DE TRABALHO

OPERAÇÕES E EXPRESSÕES

Vanessa Cadan Scheffer

QUE TAL APRIMORAR O SOFTWARE DE LOCAÇÃO DE FILMES ON-LINE?

A melhoria é informar se o filme desejado está disponível para ser locado e quantos anos faltam para que a classificação do filme seja adequada à faixa etária do cliente.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Para ampliar sua visão acerca das possibilidades de aplicação dos conhecimentos obtidos até o momento, vamos retomar a situação-problema: seu chefe solicitou que você aprimore um programa que vem sendo desenvolvido na empresa onde trabalha. Vamos então recordar o que já sabemos?

Ver anotações

```c
#include <stdio.h>

#define TAM_NOME_CLIENTE 100

struct cliente {
    char nome[TAM_NOME_CLIENTE];
    int idade;
};

int main(void){
    struct cliente cli;
    int classificacao_filme;

    printf("\n Informe o nome do cliente: ");
    fflush(stdin);
    fgets(cli.nome, TAM_NOME_CLIENTE, stdin);

    printf("\n Informe a idade do cliente: ");
    scanf("%d", &cli.idade);

    printf("\n Informe a classificação do filme: ");
    scanf("%d", &classificacao_filme);

    printf("\n Cliente: %s", cli.nome);
    printf("\n Idade: %d anos", cli.idade);
    printf("\n Classificação do filme: %d anos",
classificacao_filme);

}
```

A versão atual do seu programa é capaz de ler a idade e o nome do cliente, bem como a classificação do filme que ele deseja locar e, posteriormente, imprimir todas essas informações na tela.

Uma das novas funcionalidades propostas por seu chefe é que o programa informe ao cliente se o filme desejado está disponível para ser locado ou não. Nós já temos uma informação a respeito do filme, que é sua classificação. Então, para que nosso código fique organizado, vamos criar uma *struct* denominada Filme, assim como fizemos com o cliente.

```
struct filme {
  int classificacao_filme;
  int esta_disponivel;
};
```

Você também precisa de um trecho de código capaz de ler a informação se o filme está disponível ou não do teclado e imprimi-la na tela. Com os conhecimentos aprendidos, você consegue realizar essa etapa.

A próxima melhoria a ser feita no programa é informar se o filme desejado pode ser locado pelo cliente, levando em consideração se o filme está disponível e se é indicado para a sua faixa etária. Nós já temos as informações necessárias para efetuar esse processamento. Então, vamos criar uma expressão lógica que resultará em 0 (falso) ou 1 (verdadeiro), dependendo dos valores das variáveis de entrada. Para que o resultado seja 1 (verdadeiro), a idade do cliente deve ser maior ou igual à classificação do filme **e** o filme deve estar disponível. Com base na versão atual do código do programa, a expressão lógica a ser utilizada é:

```
(fi.esta_disponivel) && (cli.idade >=
fi.classificacao_filme))
```

Por fim, o programa deve informar quantos anos faltam para que a classificação do filme seja adequada à faixa etária do cliente. Nesse caso, precisamos pensar um pouco mais para resolver esse problema com os recursos que aprendemos até o momento. Você se lembra de que o resultado de uma expressão lógica é 0 ou 1. Então, se pensarmos na expressão (`cli.idade < fi.classificacao_filme`), seu resultado será 0 quando a idade do cliente for maior ou igual à classificação do filme ou 1, quando a idade do cliente for menor do que a classificação do filme.

```
(cli.idade < fi.classificacao_filme) *
(fi.classificacao_filme - cli.idade))
```

O resultado dessa expressão será um número inteiro, que responde à seguinte pergunta: “quantos anos faltam para que a classificação do filme seja adequada à faixa etária do cliente?”.

O código completo do programa que resolve a situação-problema encontra-se a seguir:

```c
#include <stdio.h>

#define TAM_NOME_CLIENTE 100

struct cliente {
    char nome[TAM_NOME_CLIENTE];
    int idade;
};

struct filme {
    int classificacao_filme;
    int esta_disponivel;
};

int main(void){
    struct cliente cli;
    struct filme fi;


    printf("\n Informe o nome do cliente: ");
    fflush(stdin);
    fgets(cli.nome, TAM_NOME_CLIENTE, stdin);

    printf("\n Informe a idade do cliente: ");
    scanf("%d", &cli.idade);

    printf("\n Informe a classificação do filme: ");
    scanf("%d", &fi.classificacao_filme);

    printf("\n Informe (0) se o filme não está disponível e (1) caso contrário: ");
    scanf("%d", &fi.esta_disponivel);

    printf("\n Cliente: %s", cli.nome);
    printf("\n Idade: %d anos", cli.idade);
    printf("\n Classificação do filme: %d anos", fi.classificacao_filme);
```

Ver anotações

```
36      printf("\n Está disponível: %d", fi.esta_disponivel);
37      printf("\n Filme pode ser locado pelo cliente: %d",
(fi.esta_disponivel) && (cli.idade >=
fi.classificacao_filme));
38      printf("\n Anos restantes: %d", (cli.idade <
fi.classificacao_filme) * (fi.classificacao_filme - cli.idade
));
```

Ver anotações 0

Agora é com você!

AVANÇANDO NA PRÁTICA

CÁLCULO DE DESCONTO

Uma pizzaria o procurou, pois gostaria de automatizar seu caixa. A princípio, foi-lhe solicitado apenas implementar um cálculo simples, em que, dado o valor total da conta de uma mesa, o programa calcula o desconto concedido e divide esse valor pela quantidade de integrantes da mesa. O programa deve receber como dados de entrada o valor da conta, a quantidade de pessoas e o percentual de desconto (%). Com os dados no programa, como deverá ser feito o cálculo do valor total da conta com o desconto e o valor que cada pessoa deverá pagar?

RESOLUÇÃO

O código, no Código 1.14, apresenta o resultado do problema. Um ponto importante é o cálculo do desconto feito na linha 15, para cuja montagem utilizamos uma regra de três simples. Outro ponto é o cálculo do valor por pessoa, feito na linha 18 diretamente dentro do comando de impressão. Esse recurso pode ser usado quando não é preciso armazenar o valor.

Código 1.14 | Resolução do problema

```c
#include <stdio.h>
int main(){
    float valor_bruto = 0;
    float valor_liquido = 0;
    float desconto = 0;
    int qtd_pessoas = 0;

    printf("\n Digite o valor total da conta: ");
    scanf("%f", &valor_bruto);
    printf("\n Digite a quantidade de pessoas: ");
    scanf("%d", &qtd_pessoas);
    printf("\n Digite o desconto (em porcentagem): ");
    scanf("%f", &desconto);

    valor_liquido = valor_bruto - (valor_bruto * desconto/100);

    printf("\n Valor da conta com desconto = %f", valor_liquido);
    printf("\n Valor a ser pago por pessoa = %f", valor_liquido/qtd_pessoas);
}
```

Fonte: elaborado pela autora .

Agora é com você!

Ver anotações 0

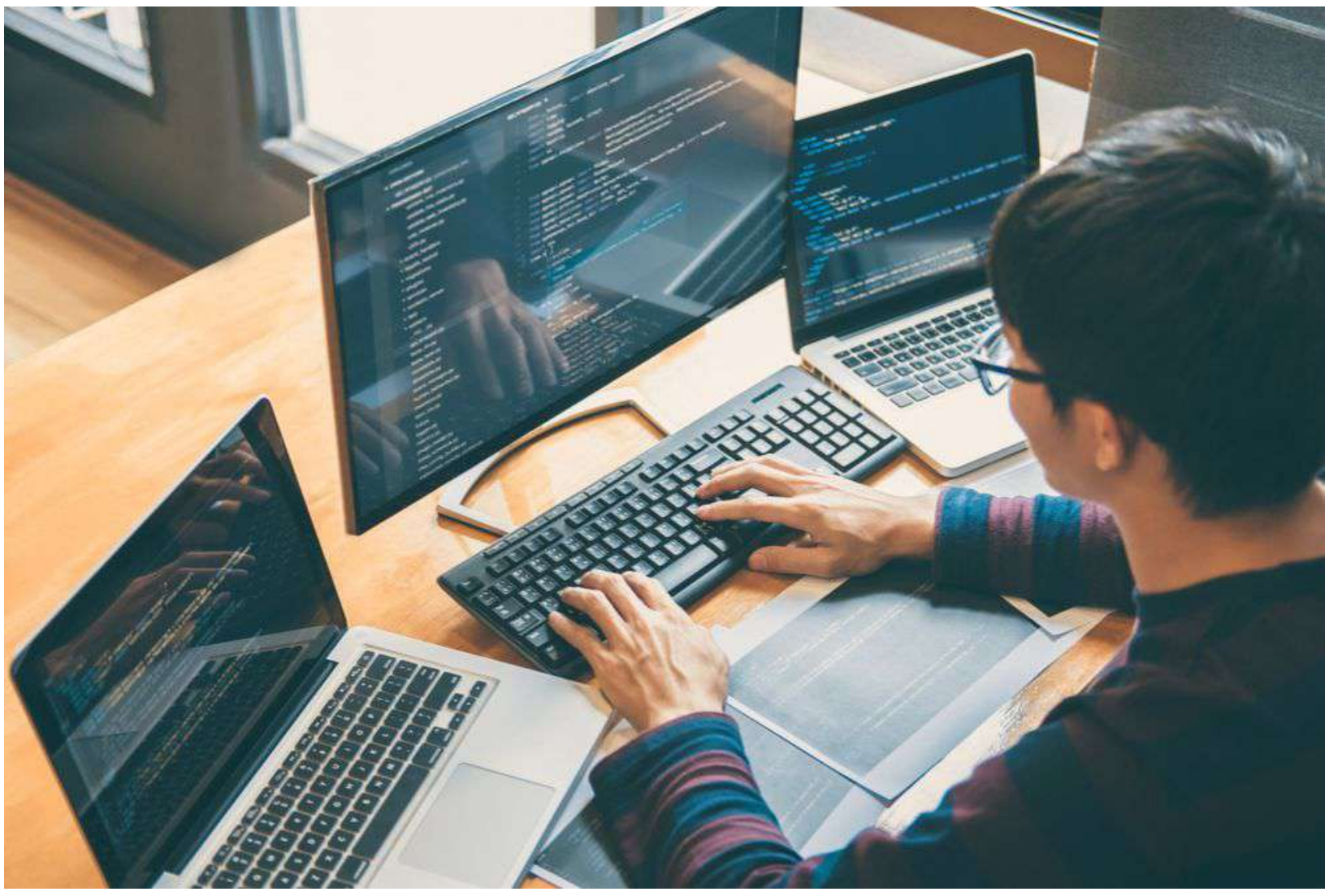
NÃO PODE FALTAR

ESTRUTURAS DE DECISÃO CONDICIONAIS

Marcio Aparecido Artero

TOMADA DE DECISÕES

As condicionais estão por toda parte, praticamente tudo que realizamos em nossa vida envolve uma ou mais condições, portanto, é necessário tomar decisões, para resolver as situações.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Podemos observar que tudo que realizamos em nossa vida envolve uma condição, certo? Se, para uma pergunta, temos várias possibilidades de resposta e se a pergunta contém condições, teremos que tomar decisões. Se você decidiu estudar programação é porque, de alguma forma, isso vai contribuir para sua formação profissional; caso contrário, faria outro curso ou seguiria outros rumos. Compreende? As condicionais estão por toda parte, seja nos fornecendo uma solução ou várias possibilidades de solução.

Nesta unidade, a fim de colocarmos em prática os conhecimentos a serem aprendidos, consideraremos o seguinte contexto de aprendizagem: vamos tratar de uma famosa instituição de ensino, referência no mercado educacional, para a qual você presta serviço por meio da empresa de software onde trabalha. Você foi eleito o funcionário do mês; seu diferencial foi o treinamento que realizou com os estagiários. Em reconhecimento ao seu trabalho, foi concedido a você o direito de efetivar um dos estagiários. Agora é o momento de dar sequência ao treinamento e se empenhar ao máximo para garantir o aprendizado do seu novo funcionário, que até então era estagiário. Ele já conhece os conceitos de programação, o trabalho com algoritmos, variáveis e tipos de dados.

O próximo passo será começar a trabalhar com a linguagem de programação C e, para isso, nada melhor do que começar pelas estruturas de decisões condicionais (Seção 3.1), avançando para as estruturas de repetição condicionais (Seção 3.2) e finalizando com as estruturas de repetições determinísticas (Seção 3.3).

É certo que a programação condicionada passará a ser uma oportunidade de otimização e melhoramento das rotinas da instituição de ensino para a qual você e seu estagiário prestarão serviço. Assim como um grande maratonista se prepara para uma corrida, você deverá realizar vários exercícios de programação para chegar à frente.

Bons estudos!

Caro aluno, certamente você está preparado para o próximo passo na programação de computadores e já estudou os conceitos de algoritmos, linguagem de programação, tipos de dados, constantes, variáveis, operadores e expressões, portanto, agora é o momento de avançar. Nesta seção, você vai estudar as estruturas de decisão condicionais, começando com uma analogia.

Um grande evento será realizado na sua cidade e seu cantor favorito fará uma apresentação. Você, sortudo que é, ganhou um convite para uma visita ao camarote do cantor e, em seguida, assistir ao show junto à banda do cantor. Porém, nem tudo são flores! Você foi escalado pela empresa em que trabalha para a implantação de um software em um dos seus clientes. Para que o tempo seja suficiente e você possa usufruir o seu presente, foram oferecidas as seguintes condições: se você instalar o software até as 12h, poderá realizar o treinamento no período da tarde e, então, ir ao show; se não, terá que agendar o treinamento para o final do dia e comprometer a sua ida ao show.

Veja que é uma analogia simples de estrutura de decisão condicional e que poderíamos, ainda, criar alguns casos que proporcionariam a sua ida ao show. Diante de tal situação, voltaremos a nossa atenção à instituição de ensino em que você e seu funcionário (ex-estagiário) prestarão serviço por meio da empresa de software onde vocês trabalham. A instituição de ensino está passando por um processo de otimização nas suas atividades e colocou o seguinte desafio para vocês: realizar um programa em linguagem C que calcule o valor do salário líquido, levando em consideração os descontos de INSS e Imposto de Renda (Tabelas 2.1 e 2.2).

Tabela 2.1 | Descontos INSS

SALÁRIO BRUTO (R$)	ALÍQUOTA / INSS
Até 1.045,00	7,5%

SALÁRIO BRUTO (R$)	ALÍQUOTA / INSS
De 1.045,01 até 2.089,60	9%
De 2.089,61 até 3.134,40	12%
Acima de 3.134,40	14%

Fonte: elaborada pelo autor.

Tabela 2.2 | Descontos IR

SALÁRIO BRUTO (R$)	ALÍQUOTA / IR
Até 1.903,98	-
De 1.903,99 até 2.826,65	7,5%
De 2.826,66 até 3.751,05	15,0%
De 3.751,06 até 4.664,68	22,5%
Acima de 4.664,68	27,5%

Fonte: elaborada pelo autor.

Bons estudos!

CONCEITO-CHAVE

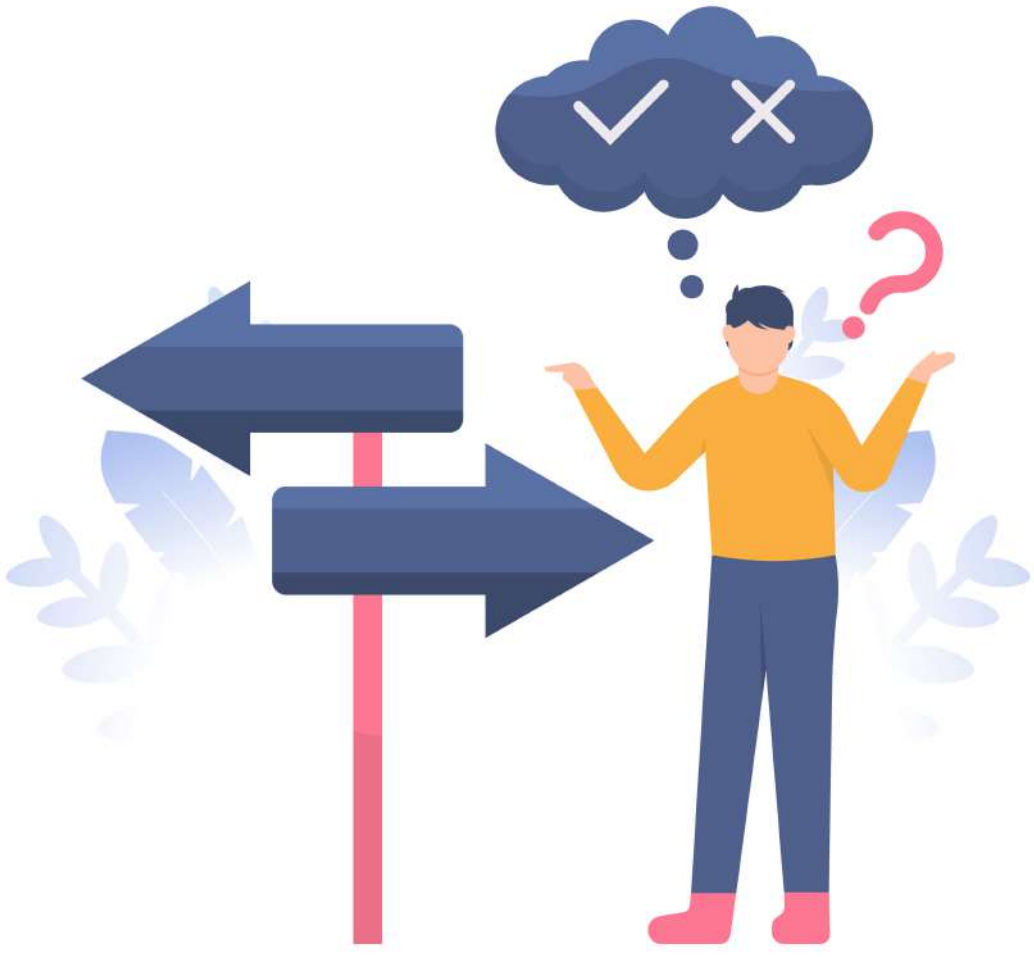
Caro aluno, agora que você já estudou os tipos de variáveis, os tipos de dados, as operações e as expressões para composição de um programa de computador, chegou o momento de trabalhar as estruturas condicionais. Você já se deparou com situações ou questões em que temos caminhos distintos a serem seguidos, certo? Por exemplo, na sentença “domingo irei ao cinema se estiver chovendo, caso contrário, irei à praia” temos duas alternativas baseada em condições; no caso, se chover (for verdadeira) irei ao cinema, se não chover (falsa), irei à praia. Geralmente, em programação trabalhamos com esse tipo de estrutura,

que são as ações, também chamadas de instruções. Muitas vezes essas instruções são escolhidas com base em algum critério ou condição que deve ser satisfeita.

ESTRUTURA DE DECISÃO CONDICIONAL *IF/ELSE*

Assim, vamos iniciar a seção com a estrutura de decisão condicional *if/else* (se/então).

Figura 2.1 | Condições



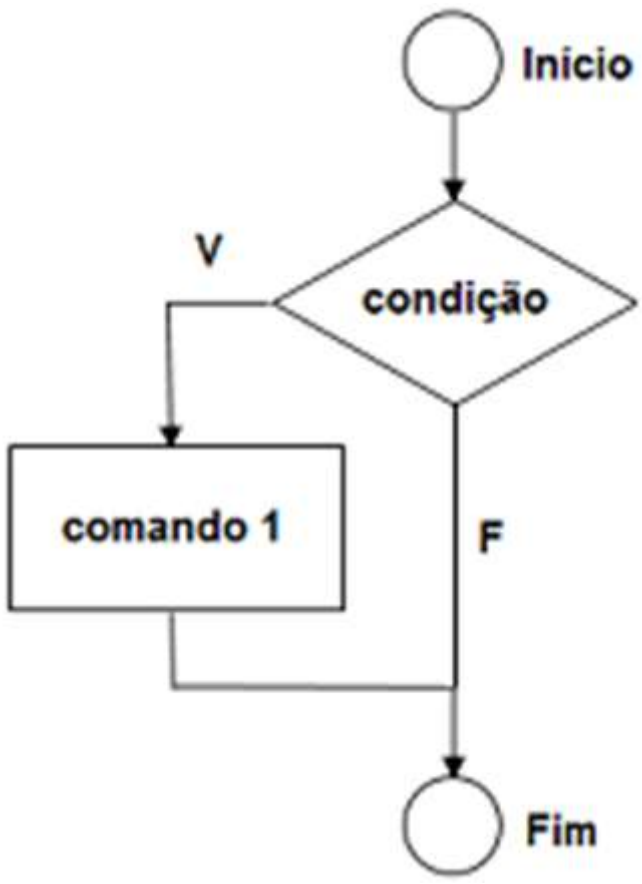
Fonte: Shutterstock.

Segundo Manzano (2013), para a solução de um problema envolvendo situações, podemos utilizar a instrução *if* (em português “se”). A estrutura tem como objetivo testar um conjunto de instruções e tomar uma decisão, criando um desvio dentro do programa para que possamos chegar a uma condição verdadeira ou falsa. Lembre-se de que a instrução pode receber valores em ambos os casos.

Na linguagem de programação C, utilizamos chaves ("{" e "}") para determinar o início e o fim de um bloco de instruções. Já a condição deve estar entre parênteses “()”. Observe na Figura 2.2 como fica a

estrutura condicional simples utilizando fluxograma e, na sequência, a sintaxe da instrução *if* utilizada na linguagem C.

Figura 2.2 | Fluxograma representando a função *if*



Fonte: elaborada pelo autor.

Agora, veja a representação da sintaxe:

```
if <(condição)> {

  <conjunto de instruções>;
```

ESTRUTURA CONDICIONAL SIMPLES

No exemplo a seguir, usaremos uma aplicação de condicional simples, lembrando que será executado um teste lógico em que, se o resultado for verdadeiro, ela trará uma resposta; caso contrário, retornará nada. Veja no exemplo que segue a situação de um jovem que verifica se poderá ou não tirar a carteira de habilitação:

Código 2.1 | Instrução *if* - idade

```c
#include <stdio.h>
int main() {
    int idade;
    printf("\n Digite sua idade: ");
    scanf("%d", &idade);
    if (idade >= 18) {
            printf("\n Você já pode tirar sua carteira deHabilitação, você é maior de 18");
    }
    return 0;
}
```

Ver anotações

Fonte: elaborado pelo autor.

Teste o Código 2.1 utilizando a ferramenta Paiza.io.

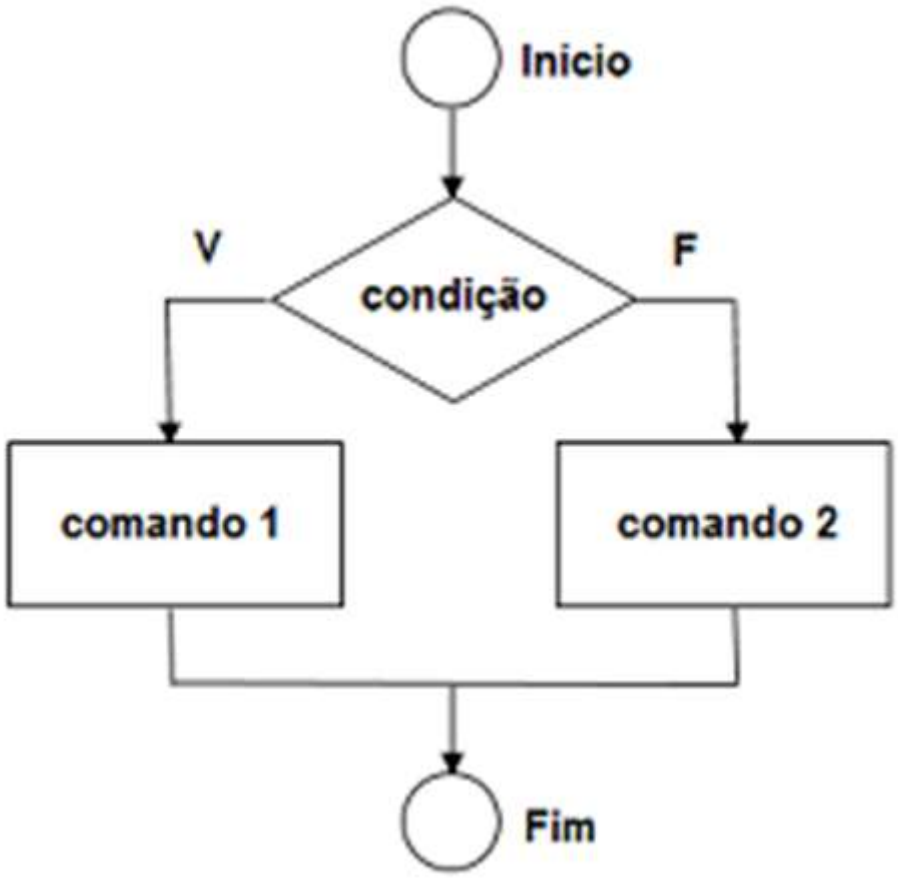
Você deve inserir a idade na aba *input* na ferramenta; após executar, vá até a aba *input*, entre com os dados, execute novamente e observe a saída

> Nesse exemplo, não é considerado o “se não” (***else***).
> Simplesmente, se a condição não for verdadeira, ela exibirá nada como resposta.

ESTRUTURA CONDICIONAL COMPOSTA

Agora, veremos a seguir, a **estrutura condicional composta**, que completa a nossa condição inicial com o comando else, que significa “se não”. Vejamos como fica a estrutura no fluxograma da Figura 2.3:

Figura 2.3 | Fluxograma representando as funções if e else



Fonte: elaborada pelo autor.

Agora, veja a representação da sintaxe:

```
if <(condição)> {

   <primeiro conjunto de instruções>;

} else {

   <segundo conjunto de instruções>;

}
```

Vamos testar o código que verifica a obrigatoriedade do voto de acordo com a idade do eleitor. Assim, definimos duas variáveis que correspondem às idades de dois eleitores e faremos o teste e análise do fluxo de execução do código.

Código 2.2 | Instrução *if-else* - eleitores

```c
#include <stdio.h>
int main() {
// Your code here!
        int eleitor1 = 18;
     int eleitor2 = 60;

        if (eleitor1 && eleitor2 >= 18 ) {
        printf ("\n Eleitores");
    } else {
        printf ("\n Não eleitores");
    }
        return 0;
}
```

Ver anotações

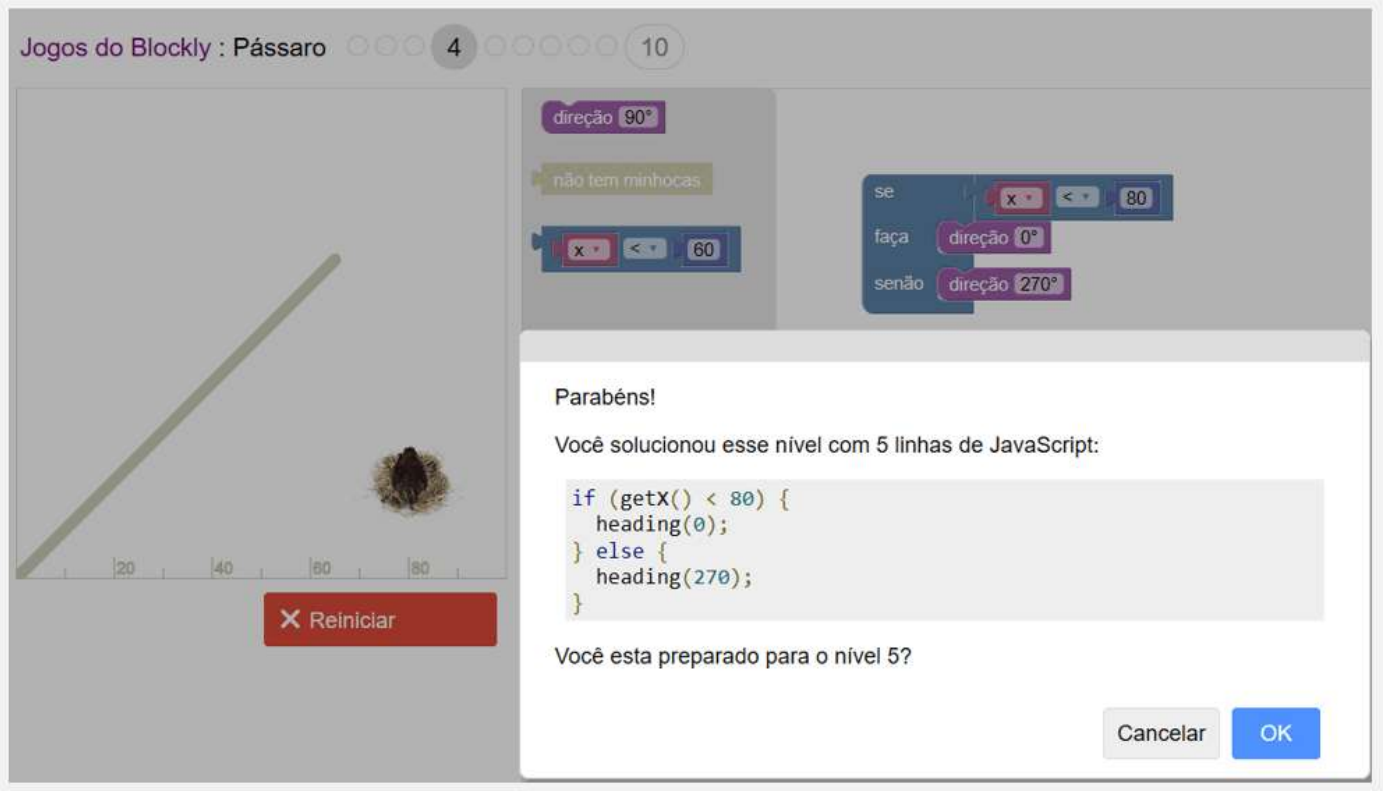
Fonte: elaborado pelo autor.

EXEMPLIFICANDO

Vamos treinar um pouco a estrutura condicional *if-else*. Um pássaro quer chegar ao seu ninho, porém ele precisa seguir alguns caminhos, e você deve incluir as posições com ângulos corretos de forma que ele chegue ao seu destino. Se você não o posicionar corretamente ele não chegará ao seu destino, e você deverá reiniciar o jogo.

Observe o exemplo na plataforma Blockly (BLOCKLY GAMES, [s. d.]. Iniciamos o jogo no nível 4. Observe na Figura 2.4 que temos um eixo x que marca a posição do pássaro, que está enumerada. No caso, completamos o desafio com 5 linhas de código envolvendo as condições *if* e *else*. Vamos seguir com os próximos.

Figura 2.4 | Jogo Pássaro – Blockly



Fonte: captura de tela do Jogo Pássaro – Blockly elaborada pelo autor.

> Vamos para um desafio mais difícil? X e Y marcam os posicionamentos do pássaro, e devemos ir até a minhoca e, depois, até o ninho. Tente você mesmo e verifique em quantas linhas de código consegue resolver o problema. Observe que agora temos dois eixos, x e y, para modificar o posicionamento do pássaro.

Vamos, agora, criar outra situação para a estrutura condicional composta em linguagem C: Maria e João estão se preparando para uma viagem. Se o orçamento final deles for igual ou maior a R$ 10.000 eles farão uma viagem internacional; caso contrário, deverão fazer uma viagem nacional.

Código 2.3 | Estrutura condicional composta - orçamento

```c
#include <stdio.h>
int main() {
    float orcamento;
    printf("\n Digite o valor do orcamento para viagem \n");
    scanf("%f", &orcamento);
    if (orcamento >= 10000) {
            printf("\n João e Maria possuem orçamento para uma viagem internacional");
    } else {
        printf("\n João e Maria irão optar por uma viagem nacional");
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.3 utilizando a ferramenta Paiza.io.

Melhorou o entendimento?

Ficou bem mais estruturado, e o comando *else* possibilitou um retorno à condicional *if*.

EXEMPLIFICANDO

Para reforçar o seu conhecimento, vamos ver o exemplo que segue em linguagem de programação C, que retorna se o valor de um número digitado é par ou ímpar, representando uma estrutura condicional composta. Além disso, aplicaremos o operador aritmético módulo (%).

Código 2.4 | Estrutura condicional composta – par ou ímpar

```c
#include <stdio.h>
int main() {
        int num;
        printf ("\n Digite um número inteiro: ");
        scanf ("%d", &num);
        if (num % 2 == 0) {
        printf ("\n O número é par");
    } else {
        printf ("\n O numero é ímpar");
    }
        return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.4 utilizando a ferramenta Paiza.io.

Sabemos que todo número par é divisível por 2; isso significa que ao dividirmos esse número por 2, o resto será zero. Sabemos, ainda, que o operador módulo (%) nos devolve o resto da divisão inteira entre dois números. Agora ficou fácil: basta verificarmos se a expressão `num % 2`, onde `num` é a variável inteira que armazena o número informado, é igual a zero. Caso sim, num é par; caso contrário, é ímpar.

ESTRUTURA CONDICIONAL DE SELEÇÃO DE CASOS

Dando sequência aos estudos, vamos conhecer a **estrutura condicional de seleção de casos**, *switch-case* que, segundo Schildt (1997, p. 35), "testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere". Quando os valores são avaliados o comando é executado.

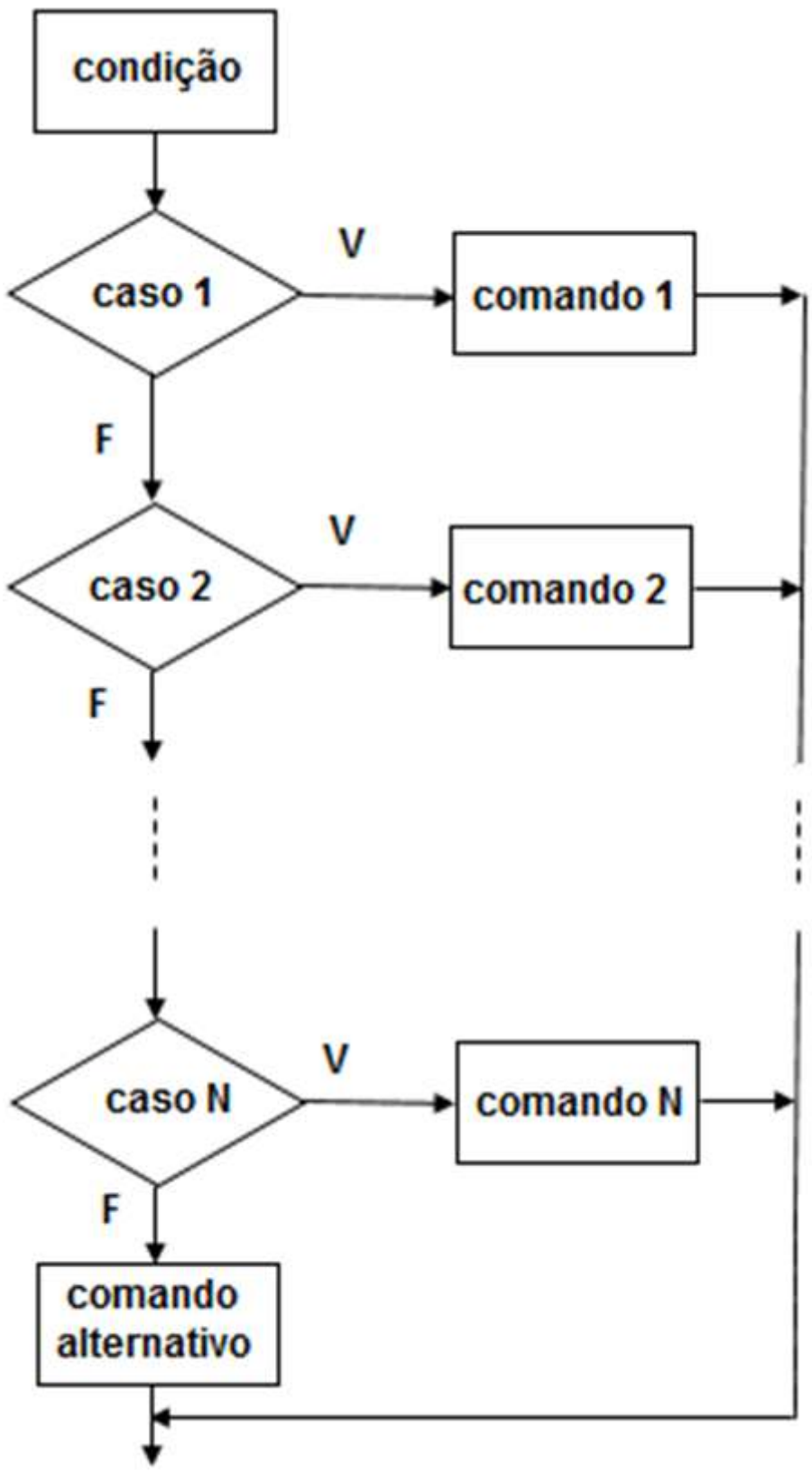
Devemos estar atentos a algumas particularidades para o comando *switch-case*:

Ver anotações

- Caso nenhum dos valores seja encontrado, o comando *default* será executado.
- Os comandos são executados até o ponto em que o comando *break* for localizado.

Veja na Figura 2.5 o fluxograma representando a estrutura condicional de seleção de casos:

Figura 2.5 | Fluxograma de estrutura condicional de seleção de casos



Fonte: elaborada pelo autor.

Vamos ver como fica a sintaxe em linguagem C:

```
switch (variável){

case constante1:

     <comandos>
```

```
     break;
case constante2:
     <comandos>
     break;
default: <comandos>
}
```

REFLITA

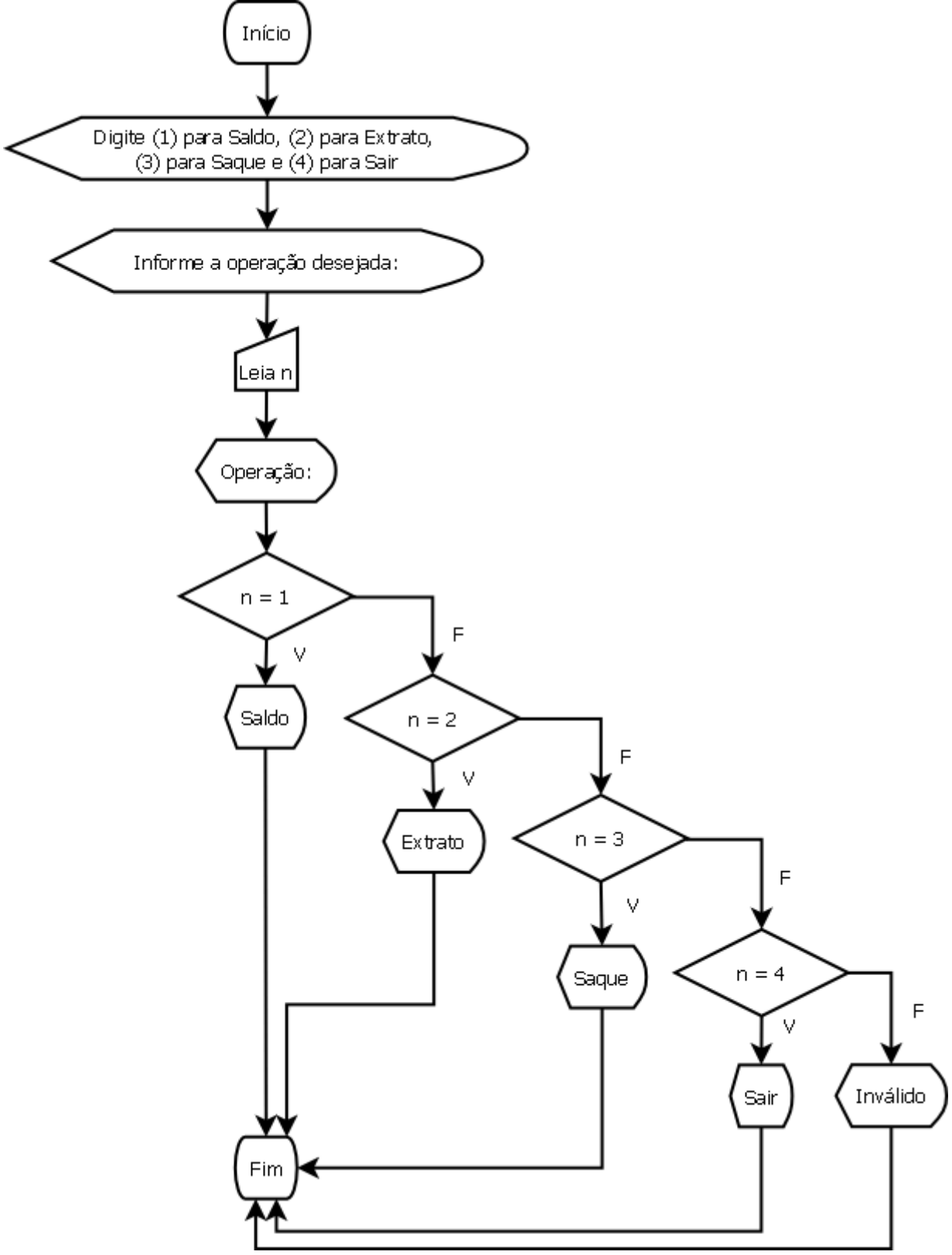
Por que, ao contrário dos outros casos, o caso *default* não precisa conter um comando *break* após os comandos?

Veja que o comando *break* é utilizado para forçar a saída da estrutura condicional, ou seja, ele sai do comando sem executar as próximas instruções. Caso não seja colocado o comando *break*, o programa continua e averigua o próximo caso até o fim do *switch* ou até encontrar um *break*.

EXEMPLIFICANDO

A Figura 2.6 mostra um exemplo de operação em um caixa eletrônico, utilizando fluxograma. Uma das primeiras atividades que o usuário deve realizar, após ter se identificado, é selecionar a operação a ser executada, por exemplo: verificar saldo, emitir extrato, saque e sair.

Figura. 2.6 | Fluxograma de uma operação em um caixa eletrônico



Fonte: elaborada pelo autor.

Que tal colocar em prática o uso da estrutura condicional switch-case com esse exemplo? O código a seguir traduz o fluxograma anterior para a linguagem de programação C.

Código 2.5 | Instrução *switch-case* – operações bancárias

```c
#include <stdio.h>
int main(){
    int n;

    printf("\n(1) para Saldo\n(2) para
Extrato\n(3) para Saque\n(4) para Sair");
    printf("\nInforme a operação desejada: ");

    scanf("%d", &n);

    switch(n) {
        case 1:
            printf("\nSaldo");
            break;
        case 2:
            printf("\nExtrato");
            break;
        case 3:
            printf("\nSaque");
            break;
        case 4:
            printf("\nSair");
            break;
        default:
            printf("\nInválido");
    }
return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.5 utilizando a ferramenta Paiza.io.

Para fixar o que está sendo estudado, vamos aplicar, a seguir, um exemplo cuja finalidade é descobrir o desconto que um cliente terá, de acordo com a escolha de uma cor específica marcada em cada produto:

EXEMPLIFICANDO

Código 2.6 | Instrução switch-case – desconto por cor

```c
#include <stdio.h>
int main() {
        char x;
        float valor, desc, total;
        printf("\n Digite o valor da compra: ");
        scanf("%f", &valor);
        printf("\n Digite a letra que representa o seu desconto de acordo com a cor: ");
        printf("\n a. azul");
        printf("\n v. vermelho");
      printf("\n b. branco");
        printf("\n Digite sua opcao:");
        scanf("%s", &x);
        switch(x) {
        case 'a':
                printf("\n Você escolheu a cor azul, seu desconto será de 30 por cento");
                desc = 30;
                break;

        case 'v':
                printf("\n Você escolheu a cor vermelha, seu desconto será de 20 por cento");
                desc = 20;
             break;
     case 'b':
                printf("\n Você escolheu a cor branca, seu desconto será de 10 por cento");
                desc = 10;
                break;
     default:
                printf("\n Opcão inválida, não será concedido desconto\n");
                desc = 0;
         }
    total = valor - (valor * desc / 100);
```

Ver anotações

```
32      printf("\n O valor da sua compra é R$ %.2f",
    total);
33
34  return 0;
35  }
```

Fonte: elaborado pelo autor.

Teste o Código 2.6 utilizando a ferramenta Paiza.io.

ASSIMILE

Para não perder o ritmo dos estudos, vamos relembrar os operadores lógicos e a tabela verdade:

Quadro 2.1 | Operadores lógicos

Operadores	**Função**
!	Negação – NOT
&&	Conjunção – AND
||	Disjunção – OR

Fonte: elaborado pelo autor.

Quadro 2.2 | Tabela Verdade

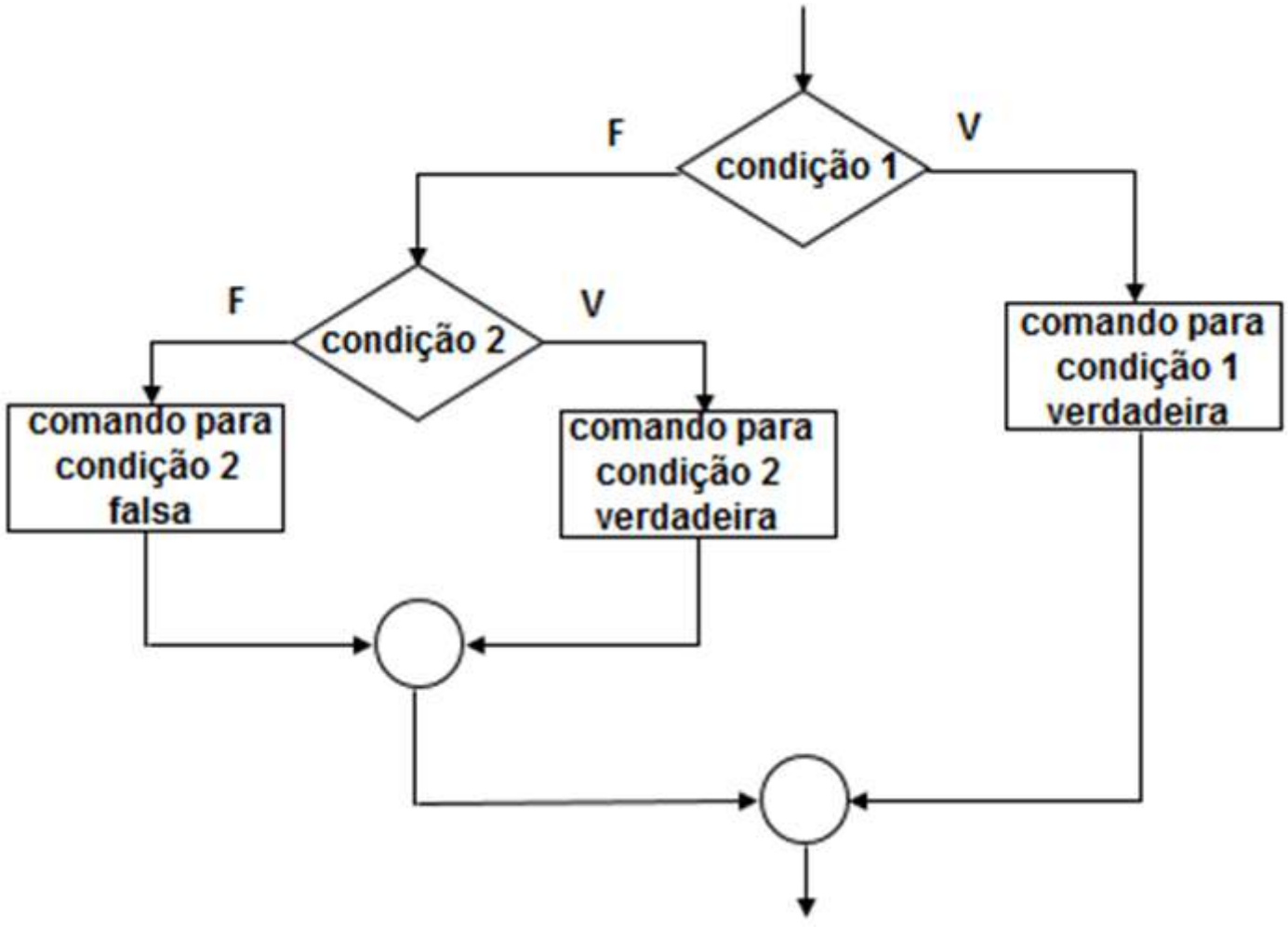
A	**B**	**A && B**	**A || B**	**! A**	**! B**
Verdade	Verdade	Verdade	Verdade	Falso	Falso
Verdade	Falso	Falso	Verdade	Falso	Verdad
Falso	Verdade	Falso	Verdade	Verdade	Falso
Falso	Falso	Falso	Falso	Verdade	Verdad

Fonte: elaborado pelo autor.

ESTRUTURA CONDICIONAL ENCADEADA

Para finalizar a seção que trata das estruturas de decisão condicionais, vamos entender o funcionamento da **estrutura condicional encadeada**, também conhecida como ***ifs* aninhados**. Segundo Schildt (1997), essa estrutura é um comando *if* que é o objeto de outros *if* e *else*. Em resumo, um comando *else* sempre estará ligado ao comando *if* de seu nível de aninhamento. Veja na Figura 2.7 um dos tipos de fluxogramas que representa uma estrutura condicional encadeada.

Figura 2.7 | Fluxograma de estrutura condicional encadeada



Fonte: elaborada pelo autor.

Podemos caracterizar a sintaxe de uma estrutura condicional encadeada da seguinte forma:

```
if (condição) {
   instrução;
} else {
   if (condição) {
      instrução;
   } else(condição) {
```

```
        instrução;
    }
}
```

Agora, veja no programa a seguir uma estrutura condicional encadeada, em que são analisadas as possíveis situações de um aluno, de acordo com sua nota final:

Código 2.7 | Estrutura condicional encadeada – nota final

```
1  #include <stdio.h>
2  int main(){
3      float nota_final;
4      printf("\n Informe a nota final do aluno: ");
5      scanf("%f", ¬a_final);
6
7      if (nota_final >= 60) {
8          printf("\n Aprovado");
9      } else {
10         if (nota_final >= 50) {
11             printf("\n Em recuperação");
12         } else {
13             printf("\n Reprovado");
14         }
15     }
16     return 0;
17 }
```

Fonte: elaborado pelo autor.

Teste o Código 2.7 utilizando a ferramenta Paiza.io.

De acordo com o programa anterior, se a nota final do aluno for igual ou superior a 60, ele é considerado *aprovado*. Caso contrário, há um comando *if* aninhado (linhas 10 a 14), para verificarmos se o aluno está *em recuperação*, isto é, se a nota final dele está entre 50 e 60 ou se está *reprovado*, caso a nota final seja menor do que 50 pontos.

Ver anotações

Uma forma alternativa e mais apropriada de se programar o código apresentado é usar a estrutura *else if*. Com essa estrutura, você evita encadeamentos muito profundos, tornando o código mais fácil de ler, entender e manter.

Podemos caracterizar a sintaxe de uma estrutura condicional *else if* da seguinte forma:

```
if (condição) {
  instrução;
} else if (condição) {
  instrução;
} else {
  instrução;
}
```

Veja como ficaria o código do programa que analisa a situação de um aluno, de acordo com sua nota final, usando a estrutura *else if*.

Código 2.8 | Estrutura *else-if* – nota final

```
#include <stdio.h>
int main(){
    float nota_final;
    printf(“\n Informe a nota final do aluno: “);
    scanf(“%f”, ¬a_final);

    if (nota_final >= 60) {
        printf(“\n Aprovado”);
    } else if(nota_final >= 50) {
        printf(“\n Em recuperação”);
    } else {
        printf(“\n Reprovado”);
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.8 utilizando a ferramenta Paiza.io.

Bons estudos e até a próxima seção!

FAÇA VALER A PENA

Questão 1

Podemos dizer que o comando *else* é uma forma de negar o que foi colocado em uma situação do comando *if*. Sendo assim, *else* é o caso contrário do comando *if*.

Funciona da seguinte forma:

```
if <(condição)> {
  <conjunto de comandos>;
} else {
  <conjunto de comandos>;
}
```

Assinale a alternativa que melhor se compõe à contextualização apresentada.

a. Para cada *else* é necessário um *if* anterior, no entanto, nem todos os *ifs* precisam de um *else*.

b. Para cada *else* é necessário um *if* posterior e todos os *ifs* precisam de um *else*.

c. Vários *ifs* precisam de um único *else* dentro de uma condição.

d. Para cada *if* é necessário um *else* para completar uma condição.

e. Podemos dizer que o comando *else* impõe condições.

Questão 2

A estrutura condicional encadeada também é conhecida como *ifs* aninhados, ou seja, é um comando *if* que é o objeto de outros *if* e *else*.

Assinale a alternativa que corresponde à sintaxe da estrutura condicional encadeada.

a.
```
if (condição) {
  comando;
}
  if (condição) {
```

Ver anotações

```
    comando;
  }
```

```
b.  if (condição) {
  comando;
} else {
    if (condição) {
      comando;
    } else {
      comando;
    }
  }
```

```
c.  if (condição) {
  comando;
  } else {
    comando;
  }
```

```
d.  else (condição) {
  comando;
  } if (condição) {
    comando;
  }
```

```
e.  if (condição) {
  comando;
} else {
    else (condição) {
      comando;
    }
  }
```

Questão 3

A estrutura condicional de seleção de casos, switch-case, segundo Schildt (1997, p. 35) “testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere”, ou seja, quando os valores são avaliados o comando é executado.

Levando em consideração a estrutura condicional de seleção utilizando casos, qual a principal função dos comandos `default` e `break`. Assinale a alternativa correta:

a. O comando *default* é executado quando nenhum dos valores é executado. Já o comando *break* determina o fim de uma das opções de comando. O comando *break* é obrigatório.

b. O comando *default* é executado quando nenhum dos valores é executado, já o comando *break* determina o início de uma das opções de comando.

c. O comando *default* é executado para iniciar um conjunto de comandos, já o comando *break* determina o fim de uma das opções de comando.

d. O comando *default* é executado no início das condições de valores, já o comando *break* determina o início de uma das opções de comando.

e. O comando *default* é executado quando nenhum dos valores é executado, já o comando *break* determina o fim de uma das opções de comando.

Ver anotações 0

REFERÊNCIAS

02 - EXERCÍCIO - Estruturas de repetição em C. 10 mar. 2017. 1 vídeo (10 min 38s). Publicado pelo canal Hélio Esperidião. Disponível em: https://cutt.ly/ejWMzh7. Acesso em: 20 nov. 2020.

13 - PROGRAMAÇÃO em Linguagem C - Desvio Condicional Aninhado - if / else if. 27 fev. 2015. 1 vídeo (11 min 12s). Publicado pelo canal Bóson Treinamentos. Disponível em: https://cutt.ly/bjWMcF7. Acesso em: 20 nov. 2020.

BLOCKLY GAMES. **Jogos do Blockly**: Pássaro. Blockly Games, [s. l.], [s. d.]. Disponível em: https://cutt.ly/LjWMboM. Acesso em: 20 nov. 2020.

DAMAS, L. **Linguagem C**. 10. ed. Rio de Janeiro: LTC, 2016.

EDELWEISS, N. **Algoritmos e programação com exemplos em Pascal e C**. Porto Alegre: Bookman, 2014.

MANZANO, J. A. N. G. **Estudo Dirigido de Linguagem C**. 17. ed. rev. São Paulo: Érica, 2013.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.

PROGRAMAÇÃO C - Aula 07 - while, do-while, for – Estruturas de repetição. 10 abr. 2012. 1 vídeo (15 min 56s). Publicado pelo canal Peterson Lobato. Disponível em: https://cutt.ly/QjWMnMk. Acesso em: 20 nov. 2020.

SCHILDT, H. **C Completo e total**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

SOFFNER, R. **Algoritmos e Programação em Linguagem C**. São Paulo: Saraiva, 2013.

FOCO NO MERCADO DE TRABALHO

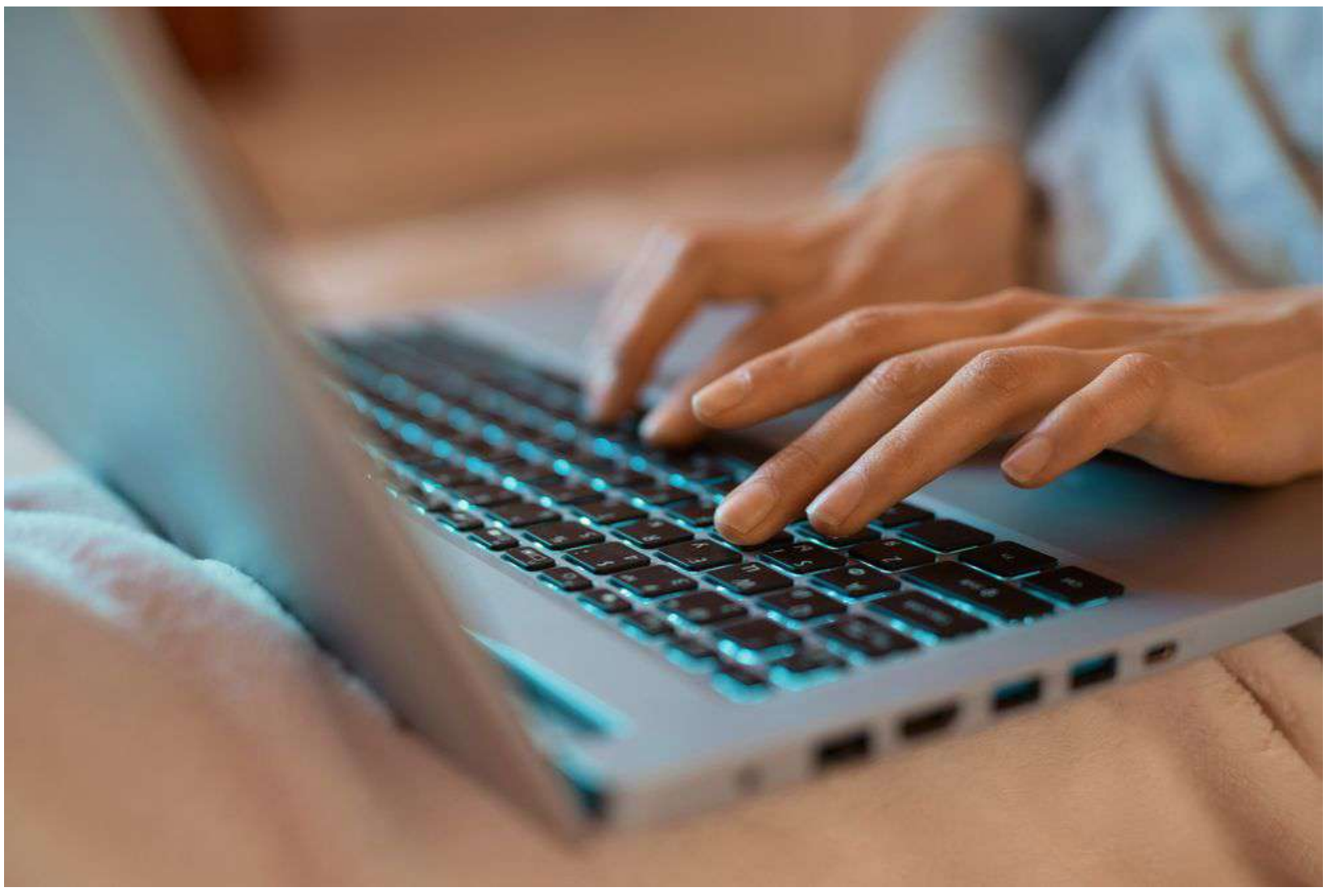
ESTRUTURAS DE DECISÃO CONDICIONAIS

Marcio Aparecido Artero

Ver anotações

PROGRAMA PARA CÁLCULO DE SALÁRIO

Criação de um programa em linguagem C para calcular o valor do salário líquido, levando em consideração os descontos de INSS e Imposto de Renda, utilizando estruturas de decisão condicional.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

Agora que você já conhece as estruturas de decisão condicional, chegou o momento de verificar se realmente o seu funcionário (ex-estagiário) conseguiu resolver o programa em linguagem C, que calcula o valor do salário líquido, levando em consideração os descontos de INSS e Imposto de Renda.

O código a seguir é uma das formas de se chegar ao resultado, mas você ainda pode acrescentar mais variáveis ao programa e, assim, calcular os dependentes, convênios e outros descontos, até mesmo os benefícios na sua folha de pagamento.

Código 2.9 | Programa em linguagem C – cálculos de salário, desconto e imposto

```c
#include <stdio.h>

int main() {
    float salario_bruto, desc_inss, desc_ir, salario_liquido;

    printf("\n\n Cálculo de Salário Líquido com desconto do IR e INSS");
    printf("\n Digite seu salário bruto: ");
    scanf("%f", &salario_bruto);

    // Cálculo do desconto do INSS
    if (salario_bruto < = 1045.0) {
        desc_inss = salario_bruto * 0.075;
    } else if (salario_bruto < = 2089.60) {
        desc_inss = salario_bruto * 0.09;
    } else if (salario_bruto <= 3134.40) {
        desc_inss = salario_bruto * 0.12;
    } else {
        desc_inss = salario_bruto * 0.14;
    }

    // Cálculo do desconto do IR
    if (salario_bruto <= 1903.98) {
        desc_ir = 0;
    } else if (salario_bruto <= 2826.65) {
        desc_ir = salario_bruto * 0.075;
    } else if (salario_bruto <= 3751.05) {
        desc_ir = salario_bruto * 0.15;
    } else if (salario_bruto <= 4664.68) {
        desc_ir = salario_bruto * 0.225;
    } else {
        desc_ir = salario_bruto * 0.275;
    }

    salario_liquido = salario_bruto - desc_inss - desc_ir;

    printf( "\n Desconto INSS: R$ %.2f", desc_inss);
```

Ver anotações

```
37      printf( "\n Desconto IR: R$ %.2f", desc_ir);
38      printf( "\n Salário líquido: R$ %.2f", salario_liquido);
39
40      return 0;
41  }
```

Fonte: elaborado pelo autor.

Teste o Código 2.9 utilizando a ferramenta Paiza.io.

AVANÇANDO NA PRÁTICA

SEMANA DO DESCONTO

Na dinâmica do dia a dia de uma pizzaria, você resolveu implementar um programa em linguagem C para que em cada dia da semana fosse ofertado um desconto aos seus clientes. Seria mais ou menos assim: na segunda-feira, o desconto seria de 30% no valor da pizza; na terça, 40%; na quarta, a pizza é em dobro; na quinta, 20% de desconto; na sexta, 10%; no sábado não haverá desconto; no domingo, ganha-se o refrigerante. Existem várias formas de criar esse programa em linguagem C, certo? Qual maneira você escolheria par criar esse programa?

RESOLUÇÃO

Para resolver o caso dos descontos da pizzaria, veja uma das opções que você pode adotar. É claro que existem outras formas de chegar no mesmo resultado, até mesmo com uma melhor otimização.

Código 2.10 | Programa linguagem C - Pizzaria

```c
#include <stdio.h>
int main () {
    int dia_da_semana;
    printf ("\n Digite o número correspondente ao dia da semana: ");
    printf("\n Digite 1 para Domingo");
    printf("\n Digite 2 para Segunda");
    printf("\n Digite 3 para Terça");
    printf("\n Digite 4 para Quarta");
    printf("\n Digite 5 para Quinta");
    printf("\n Digite 6 para Sexta");
    printf("\n Digite 7 para Sábado");
    scanf("%d", &dia_da_semana);
    switch (dia_da_semana) {
    case 1:
        printf ("\n Domingo é dia de refri grátis");
        break;
    case 2:
        printf ("\n Segunda o desconto será de 40 por cento no valor da pizza");
        break;
    case 3:
        printf ("\n Terça o desconto será de 30 por cento no valor da pizza");
        break;
    case 4:
        printf ("\n Quarta é dia de pizza em dobro");
        break;
    case 5:
        printf ("\n Quinta o desconto será de 20 por cento no valor da pizza");
        break;
    case 6:
        printf ("\n Sexta o desconto será de 10 por cento no valor da pizza");
        break;
    case 7:
```

Ver anotações

```
33          printf ("\n Sinto muito. Sábado não tem
desconto");
34          break;
35       default :
36          printf ("\n Opção inválida!");
37    }
38    return 0;
39  }
```

Teste o Código 2.10 utilizando a ferramenta Paiza.io.

NÃO PODE FALTAR

ESTRUTURAS DE REPETIÇÃO CONDICIONAL

Marcio Aparecido Artero

OTIMIZAÇÃO DAS SOLUÇÕES DE PROBLEMAS

As estruturas de repetição são utilizadas para realização de tarefas repetitivas vinculadas a uma condição, envolvendo variáveis de controle, contadores e acumuladores.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro aluno, chegamos a mais um desafio do nosso curso, no qual você terá a oportunidade de estudar as estruturas de repetição condicionais *while* e *do/while*, seus comparativos e aplicações.

Assim como as estruturas de decisão, as estruturas de repetição têm a função de otimizar as soluções de problemas. Considere que você decidiu distribuir cinco livros de computação ao final de um evento; a estrutura de repetição, por exemplo, ficaria assim: enquanto o número de pessoas for menor que cinco, você entregará um livro; depois, a distribuição será encerrada. Veja que a expressão “enquanto” foi utilizada no início da frase.

Pois bem, para colocarmos os conhecimentos a serem aprendidos em prática, vamos analisar a seguinte situação: você deverá criar um programa em linguagem C para ajudar a instituição de ensino na qual você se graduou. Foi solicitada a elaboração de um programa que receberá as notas finais dos alunos de determinada disciplina. O professor poderá entrar com quantas notas ele desejar e, por fim, o programa deverá apresentar a média final dessa disciplina.

Pense nas soluções e execute o código em um compilador de linguagem C. Apresente o código livre de erros em um documento de texto. Boa aula!

CONCEITO-CHAVE

Você já observou a quantidade de tarefas repetitivas que realizamos no dia a dia? Indo ao trabalho todos os dias, por exemplo, podemos nos deparar com situações rotineiras no caminho: paramos em semáforos, viramos à esquerda, à direita e seguimos em frente algumas vezes para chegar ao nosso destino. Nesse caso, vivemos circunstâncias que se repetem um determinado número de vezes. Podemos associar esse tipo de ação também ao contexto computacional, por exemplo, ao desenvolver um programa para contar o número de caixas de produtos em uma esteira de produção ou o número de clientes que realizou a

compra de um produto específico, também para contar quantos produtos ainda há no estoque de uma loja. Essas situações correspondem à ideia de contagem, repetindo até que o último produto seja vendido e até que o último cliente realize a compra do último produto – são tarefas repetitivas. A Figura 2.8 ilustra um exemplo de aplicação de repetição: a simples contagem de caixas em uma esteira. Nessa situação, seriam necessários outros aparatos, como sensores, mas é possível realizar a contagem a partir de uma estrutura de repetição.

Figura 2.8 | Caixas em uma esteira de produção

Fonte: Shutterstock.

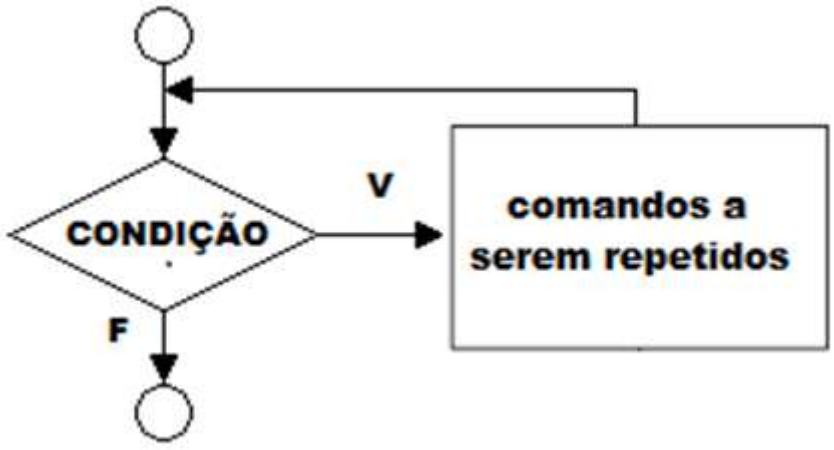
ESTRUTURA DE REPETIÇÃO COM TESTE NO INÍCIO - *WHILE*

Assim, chegou o momento de encarar o desafio de estudar as estruturas de repetição. Segundo Manzano (2013), para a solução de um problema, é possível utilizar a instrução *if* para tomada de decisão e para criar desvios dentro de um programa para uma condição verdadeira ou falsa. Seguindo essa premissa, vamos iniciar nossos estudos com as **repetições com teste no início – *while***. É preciso estar ciente de que algo será repetidamente executado enquanto uma condição verdadeira for verificada e de que somente após a sua negativa essa condição será interrompida.

Segundo Soffner (2013, p. 64), o programa “não executará qualquer repetição (e as ações que ali dentro estiverem programadas) sem antes testar uma condição”. Na realização dessa condição, vamos fazer uso do comando iterativo while, que significa “enquanto” em português. Veja na Figura 2.9 a forma simplificada do fluxograma do comando while direcionado para o teste no início.

Ver anotações

Figura 2.9 | Fluxograma do comando while



Fonte: elaborada pelo autor.

Como o programa será elaborado em linguagem C, veja a sintaxe com a repetição com teste no início:

```
while (<condição>) {
  Comando 1;
  Comando 2;
  Comando n;
}
```

Em alguns casos, quando utilizamos um teste no início, pode ocorrer o famoso loop (laço) infinito (quando um processo é executado repetidamente). Para que isso não aconteça, você poderá utilizar os seguintes recursos:

- **Contador**: é utilizado para controlar as repetições, quando são determinadas.
- **Incremento e decremento**: trabalham o número do contador, seja aumentando, seja diminuindo.

- **Acumulador**: segundo Soffner (2013), somará as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.
- **Condição de parada**: utilizada para determinar o momento de parar, quando não se tem um valor exato dessa repetição.

> **REFLITA**
>
> Perceba que quando aplicamos um laço várias instruções podem ser aplicadas, até mesmo a existência de um laço dentro de outro laço. Que nome damos a esse laço?

Observe o exemplo a seguir, uma aplicação do comando *while* em um teste no início, que deverá mostrar a palavra “PROGRAMA” dez vezes:

Código 2.11 | Comando *while* – teste no início

```
#include <stdio.h>
 int main() {
     int cont = 0;

     // Será executado enquanto cont for menor que 10
     while (cont < 10) {
          printf("\n PROGRAMA");

          // incrementa cont, para que não entre em loop infinito
          cont++;
     }
     return 0;
 }
```

Fonte: elaborado pelo autor.

Teste o Código 2.11 utilizando a ferramenta Paiza.io.

No exemplo do código, podemos observar alguns dos recursos comentados, tais como contador, incremento e condição de parada. Na linha 3, uma variável denominada *cont* é inicializada com o valor 0. Essa variável servirá como **contador** do programa, isto é, ela vai indicar quantas repetições do laço já foram executadas. Na linha 6 nós temos o que chamamos de **condição de parada**, ou seja, uma condição lógica que, ao se tornar falsa, determinará a parada do laço de repetição. Nesse caso, estamos utilizando o contador que foi criado na condição de parada. Quando ele chegar ao valor 10 (ou maior), a condição de parada se tornará falsa e o laço será encerrado. Por fim, na linha 10 nós temos o **incremento** do contador. Sem isso, o programa entraria em loop infinito, uma vez que o valor da variável *cont* não seria atualizado, portanto, a condição de parada nunca se tornaria falsa. Duas outras formas de se representar o incremento do contador são: `cont = cont + 1` ou `cont += 1`. Esse tipo de instrução é conhecido como atribuição composta.

O próximo exemplo é para checar se a nota final de um aluno está entre 0 e 10.

Código 2.12 | Comando *while* – validação de entrada de dados

```c
#include <stdio.h>
int main(void) {
   int parar = 0;
   float nota;
   printf("\nDigite a nota final do aluno: ");
   scanf("%f", ¬nota);
   while (parar != 1) {
      if(nota < 0 || nota > 10)
      {
        printf("\nNota inválida! Digite a nota final do aluno: ");
        scanf("%f", ¬nota);
      }
      else
      {
         printf("\nNota válida, encerrando...");
         parar = 1;
      }
    }
    return 0;
   }
```

Fonte: elaborado pelo autor.

Teste o Código 2.12 utilizando a ferramenta Paiza.io.

> Lembre-se de que você deve inserir as notas na aba *input* na ferramenta. Neste caso, insira mais de um valor, todos separados por espaços para que o paiza.io leia e opere as notas através do laço de repetição.

Esse é um bom exemplo de validação de entrada de dados, um recurso muito utilizado em diversos softwares atuais. A validação consiste em garantir que os dados de entrada informados pelo usuário do sistema estejam dentro do esperado para o bom funcionamento do software.

No caso do código apresentado, é um software educacional, e não queremos que uma nota negativa ou maior do que o limite máximo (10) seja inserida no software. Por isso, na linha 6, a nota do aluno é lida uma primeira vez. Logo após (linha 7), temos um laço *while*, cuja condição de parada é verdadeira (1) quando a nota do aluno estiver entre 0 e 10. Isto é, o laço só será encerrado quando a nota do aluno for válida, atribuindo o valor 1 à variável parar. Lendo de outra forma, enquanto a condição (linha 8) em que nota do aluno for menor do que 0 ou maior do que 10, o programa exibirá uma mensagem informando ao usuário que a nota é inválida e solicitando a entrada de uma nova nota (linha 10). Após a mensagem, a nova nota é lida na linha 11.

Caso não haja valores que atendem a condição da linha 8, o programa nunca será encerrado e o laço permanece em loop infinito.

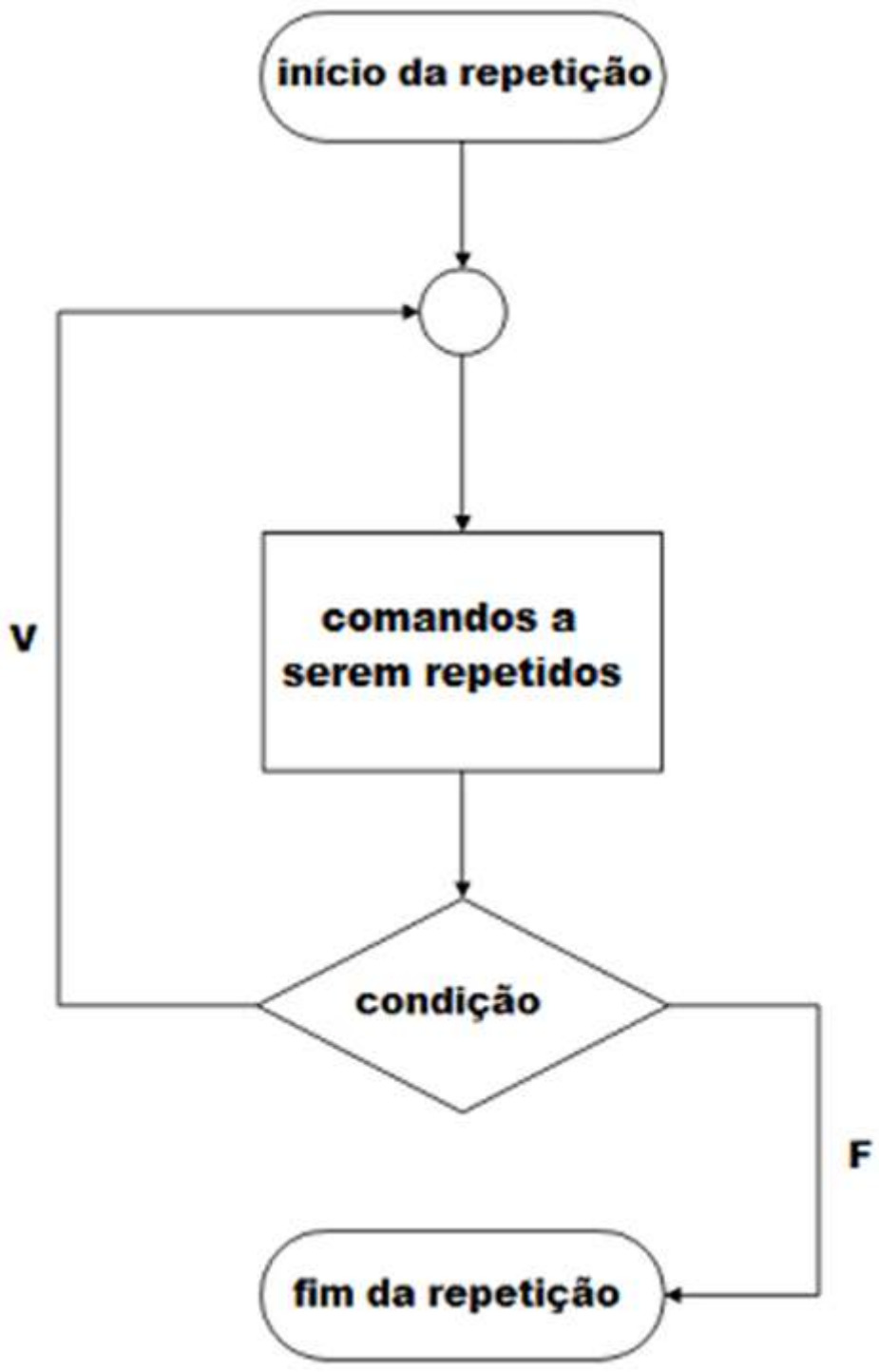
> O paiza.io notificará o erro *"File size limit exceeded (core dumped)"*, indicando que o limite de leitura do input foi excedido, pois não foi encontrado uma nota válida.

ESTRUTURA DE REPETIÇÃO COM TESTE NO FIM – *DO-WHILE*

Agora, vamos aplicar as **repetições com testes no final (*do-while*)**. Segundo Schildt (1997), o laço *do-while* analisa a condição ao final do laço, ou seja, os comandos são executados antes do teste de condição. Nesse caso específico, ao contrário do *while*, os comandos são executados pelos menos uma vez.

A Figura 2.10 ilustra o fluxograma utilizando o teste de repetição no final:

Figura 2.10 | Fluxograma com teste de repetição no final



Fonte: elaborada pelo autor.

Observe a sintaxe para realização da repetição com teste no final:

```
do {
  comandos;
} while (condição);
```

Observe como ficaria o exemplo anterior, com a utilização do comando *do-while*:

Código 2.13 | Comando do-while – teste no final

Ver anotações 0

```c
1   #include <stdio.h>
2   int main(void) {
3      int parar = 0;
4      float nota;
5       do {
6          printf("\nDigite a nota final do aluno (min: 0, max:
    10): ");
7          scanf("%f", ¬nota);
8          if(nota < 0 || nota > 10)
9          {
10           printf("\nNota inválida!");
11         }
12         else
13         {
14            printf("\nNota válida, encerrando...");
15            parar = 1;
16         }
17      } while (parar != 1);
18    return 0;
19  }
```

Fonte: elaborado pelo autor.

Agora é a sua vez de testar o código utilizando a ferramenta Paiza.io.

A validação de dados de entrada do usuário, como explicamos anteriormente, é um ótimo exemplo para utilização do laço *do-while*. Isso porque a entrada precisa ser lida pelo menos uma vez para (linha 7), para então decidirmos se podemos encerrar o laço ou se devemos solicitar a entrada novamente (condição de parada – linha 17).

Outro potencial uso do laço *do-while* é na implementação de menus de opções, conforme apresentado no exemplo a seguir. Você verá um programa que calcula a metragem quadrada de um terreno, usando o teste no final para criar a opção de digitar novos valores sem sair do programa:

Código 2.14 | Laço *do-while* – metragem terreno

```c
#include <stdio.h>
int main() {
    float metragem1, metragem2, resultado;
    int resp;
    metragem1 = 0;
    metragem2 = 0;
    resultado = 0;
    printf("\nC Á L C U L O   D E   M E T R O S   Q U A D R A D O S");
    do {
        printf("\n\nDigite a primeira metragem do terreno: ");
        scanf("%f", &metragem1);
        printf("\nDigite a segunda metragem do terreno: ");
        scanf("%f", &metragem2);
        resultado = metragem1 * metragem2;
        printf("\n\nO Terreno tem = %.2f m2", resultado);
        printf("\n\nDigite 1 para continuar ou 2 para sair: ");
        scanf("%d", &resp);
    } while (resp == 1);
    return 0;
}
```

Ver anotações

Fonte: elaborado pelo autor.

Teste o Código 2.14 utilizando a ferramenta Paiza.io.

Na sequência dos nossos estudos, vamos trabalhar com outra aplicação das estruturas de repetição condicionais. Nesse caso, realizando um programa que simula uma conta bancária (tela de opções das transações), adaptado do livro do Soffner (2013).

Código 2.15 | Comando *do-while* – conta bancária

```c
#include <stdio.h>
 int main() {
     float soma=0;
     float valor;
     int opcao;
     do {
         printf("\n M E N U   D E  O P Ç Õ E S");
         printf("\n 1. Depósito");
         printf("\n 2. Saque");
         printf("\n 3. Saldo");
         printf("\n 4. Sair");
         printf("\n Informe uma opção: ");
         scanf("%d", &opcao);
         switch(opcao) {
             case 1:
                 printf("\n Informe o valor: ");
                 scanf("%f", &valor);
                 soma += valor;
                 break;
             case 2:
                 printf("\n Informe o valor: ");
                 scanf("%f", &valor);
                 soma -= valor;
                 break;
             case 3:
                 printf("\n Saldo atual = R$ %.2f", soma);
                 break;
             case 4:
                 printf("\n Saindo...");
                 break;
             default:
                 printf("\n Opção inválida!");
         }
     } while (opcao != 4);
     printf("\n\n Fim das operações!");
          return 0;
```

Ver anotações

```
37      }
```

Fonte: elaborado pelo autor.

Teste o Código 2.15 utilizando a ferramenta Paiza.io.

Nas linhas 3 a 5, temos a declaração das variáveis iniciais do programa. Na linha 6, temos o início do bloco do comando *do-while*. Isso significa que tudo o que está inserido entre as linhas 6 e 34 pode executar uma ou mais vezes. Nas linhas 7 a 12 temos a impressão do menu de opções do programa na tela do usuário. Já na linha 13, há a leitura da opção desejada do usuário. Com base nessa opção, o comando *switch-case* (linha 14) redirecionará o fluxo de execução do programa para o bloco mais adequado. Caso nenhum caso (*case*) seja encontrado, então o bloco de instruções do caso padrão (*default*), que está linha 32 será executado. Ao final do comando switch-case, temos a condição de parada do bloco *do-while* (linha 34). Nessa condição, é verificado se a opção selecionada pelo usuário é diferente de 4, que corresponde à opção “Sair”. Caso isso seja verdadeiro, o fluxo do programa volta para o início do bloco *do-while*, e todo processo comentado anteriormente se repete. Caso contrário, o programa é encerrado. Quanto às operações de saque e depósito, trata-se de instruções simples, já estudadas neste livro. A única diferença está na utilização do comando de atribuição composta duas vezes, uma para a operação de depósito e outra para a de saque.

ASSIMILE

Algumas variáveis podem sofrer alterações baseadas nos seus valores anteriores. Para facilitar, você pode utilizar o que chamamos de atribuição composta, que indica qual operação será realizada. Nesse caso, coloca-se o operador à esquerda do sinal de atribuição. Por exemplo: `y += 1`, que tem o

mesmo efeito que `y = y + 1`, nesse caso, evitando colocar a variável à direita da atribuição. O mesmo pode ser feito com as operações de subtração, multiplicação e de divisão.

Chegamos ao final de mais uma seção e agora é o momento de aplicar o conhecimento adquirido. Bons estudos!

FAÇA VALER A PENA

Questão 1

Sobre o comando *while*, podemos afirmar que:

I. O programa não executará nenhuma repetição (as ações que ali dentro estiverem programadas) sem antes testar uma condição.

II. Pode ocorrer o famoso *loop* infinito.

III. Geralmente usamos o comando *while* quando não sabemos quantas vezes o laço da condição deve ser repetido.

É correto o que se afirma em:

a. I, apenas.

b. I e II, apenas.

c. II, apenas.

d. I, II e III.

e. III, apenas.

Questão 2

Levando em consideração que precisamos estar atentos para que não ocorra um loop infinito, analise as afirmações a seguir:

I. Contador – é utilizado para controlar as repetições, quando são determinadas.

II. Incremento e decremento – trabalham o número do contador, seja aumentando, seja diminuindo.

III. Acumulador – segundo Soffner (2013), somará as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.

IV. Condição de parada – utilizada para determinar o momento de parar quando não se tem um valor exato dessa repetição.

É correto o que se afirma em:

a. I, apenas.

b. I, III e IV, apenas.

c. II e III, apenas.

d. IV, apenas.

e. I, II, III e IV.

Questão 3

Segundo Soffner (2013), quando aplicado o comando do/while, as ações são executadas pelo menos uma vez antes do teste condicional. Analise o programa a seguir, que realiza a soma dos números positivos usando repetição com teste no final, e complete as partes que estão faltando.

```
#include<stdio.h>
int main() {
  int n;
  int soma = 0;
  _____ {
    printf("Digite um número positivo para ser somado ou
negativo
        para sair: ");
    scanf("%d", &n);
    ___ (n >= 0) {
      soma = soma + n;
    }
  } while(       );
```

Ver anotações

```
  printf("A soma é %d\n", soma);
  return 0;
}
```

Assinale a alternativa que completa as lacunas no programa.

a. “do”; “if”; “n >= 0”.

b. “if”; “n=”; “n=0”.

c. “if”; “else”; “n >= 0”.

d. “if”; “else”; “n <= 0”.

e. “do”; “if”; “n = 0”.

Ver anotações

REFERÊNCIAS

02 - EXERCÍCIO - Estruturas de repetição em C. 10 mar. 2017. 1 vídeo (10 min 38s). Publicado pelo canal Hélio Esperidião. Disponível em: https://cutt.ly/rjW4fQp. Acesso em: 20 nov. 2020.

13 - PROGRAMAÇÃO em Linguagem C - Desvio Condicional Aninhado - if / else if. 27 fev. 2015. 1 vídeo (11 min 12s). Publicado pelo canal Bóson Treinamentos. Disponível em: https://cutt.ly/bjW4g9G. Acesso em: 20 nov. 2020.

DAMAS, L. **Linguagem C**. 10. ed. Rio de Janeiro: LTC, 2016.

EDELWEISS, N. **Algoritmos e programação com exemplos em Pascal e C**. Porto Alegre: Bookman, 2014.

MANZANO, J. A. N. G. **Estudo Dirigido de Linguagem C**. 17. ed. rev. São Paulo: Érica, 2013.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.

PAIZA.IO. Página inicial. Paiza Inc., Tóquio, 2020. Disponível em: https://paiza.io/en. Acesso em: 29 out. 2020.

PROGRAMAÇÃO C - Aula 07 - while, do-while, for – Estruturas de repetição. 10 abr. 2012. 1 vídeo (15 min 56s). Publicado pelo canal Peterson Lobato. Disponível em: https://cutt.ly/jjW4kik. Acesso em: 20 nov. 2020.

PROGRAMAR EM C - Como Utilizar "do while" - Aula 13. 24 out. 2012. 1 vídeo (7 min 3s). Publicado no canal De aluno para aluno. Disponível em: https://cutt.ly/9jW4lDN. Acesso em: 20 nov. 2020.

SCHILDT, H. **C Completo e total**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

SOFFNER, R. **Algoritmos e Programação em Linguagem C**. São Paulo: Saraiva, 2013.

FOCO NO MERCADO DE TRABALHO

ESTRUTURAS DE REPETIÇÃO CONDICIONAL

Marcio Aparecido Artero

PROGRAMA PARA CÁLCULO DA MÉDIAS DE NOTAS

Criação de um programa em linguagem C para calcular a média de notas de uma disciplina, de acordo com as notas dos alunos, informadas pelo professor, utilizando estruturas de repetição condicionais.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

Para ampliar sua visão acerca das possibilidades de aplicação dos conhecimentos obtidos até o momento, vamos retomar a situação-problema apresentada anteriormente: você deverá criar um programa em linguagem C que calcule a média de notas de uma disciplina, de acordo com as notas dos alunos, informadas pelo professor.

Para resolver essa situação, é sugerida uma das possíveis soluções:

- Criar uma variável para entrada das notas, outra para acumular o valor das notas e outra para contabilizar quantas notas foram informadas.
- Após o lançamento de cada nota, solicitar ao usuário que informe se deseja continuar digitando outras notas ou não.
- Ao final, calcula-se a média e a apresenta na tela.

Código 2.16 | Comando *do-while* – média de notas

```c
#include <stdio.h>
 int main() {
     int qtde_notas = 0, opcao;
     float nota, media, soma_notas = 0.0;

     do {
         printf("\nDigite a nota do aluno %d: ", qtde_notas + 1);
         scanf("%f", ¬a);
         qtde_notas += 1;
         soma_notas += nota;
         printf("\nDigite 1 para informar outra nota ou 2 para encerrar: ");
         scanf("%d", &opcao);
     } while (opcao != 2);

     printf("\n\nQuantidade de alunos = %d", qtde_notas);
     media = soma_notas / (float) qtde_notas;
     printf("\nMédia das notas = %.2f", media);
     return 0;
  }
```

Fonte: elaborado pelo autor.

Teste o Código 2.16 utilizando a ferramenta Paiza.io.

Essa solução apresenta algumas similaridades com o exemplo bancário apresentado nesta seção. Nesse exemplo, nas linhas 3 e 4 são declaradas as variáveis usadas ao longo do programa. Das linhas 6 a 13 temos o comando *do-while*, cujas instruções serão repetidas enquanto a opção 2 (encerrar) não for informada pelo usuário na leitura da variável "opção" (linha 12). Nas linhas 8 a 10 temos a leitura da nota do aluno, o incremento da variável "qtde_notas" e do acumulador "soma_notas". Uma vez que o usuário optar pelo encerramento da entrada de notas, o

programa vai calcular e imprimir a média das notas da turma, juntamente com a quantidade de notas informadas pelo usuário (linhas 15 a 17).

Você pode realizar outros testes e criar situações para a solução desse problema.

> **PESQUISE MAIS**
>
> O comando *do-while* pode ter várias aplicações. Veja o vídeo disponível no YouTube e indicado a seguir a respeito desse tema.
>
> PROGRAMAR EM C - Como Utilizar "do while" - Aula 13. 24 out. 2012. 1 vídeo (7 min 3s). Publicado no canal De aluno para aluno.

AVANÇANDO NA PRÁTICA

VALOR NUTRICIONAL DO SORVETE

Acreditamos que a maioria das pessoas aprecia um bom sorvete, mas não tem o conhecimento do seu valor nutricional. Para resolver tal situação, você foi contratado por uma sorveteria para elaborar um programa em que os clientes consigam ver os valores nutricionais de cada sorvete que pretendem consumir.

RESOLUÇÃO

Com o intuito de resolver essa situação, vamos trabalhar a estrutura de repetição com teste no final, mas você deverá analisar e aperfeiçoar o programa sugerido com uma pesquisa das informações nutricionais de cada sorvete e deverá, ainda, melhorar as rotinas do programa.

Código 2.17 | Comando *do-while* – informações nutricionais

```c
#include <stdio.h>
int main () {
        int i;
        do {
            printf ("\n\nINFORMAÇÃO NUTRICIONAL DE SORVETES");
        printf ("\n\n(1)...flocos");
        printf ("\n(2)...morango");
        printf ("\n(3)...leite condensado");
                printf ("\nDigite um número que corresponde ao sabor desejado: ");
        scanf("%d", &i);
    } while ( i < 1 || i > 3);

        switch (i) {
        case 1:
            printf ("\n\nVocê escolheu flocos.");
            break;
        case 2:
                printf ("\n\nVocê escolheu morango.");
                break;
        case 3:
                printf ("\n\nVocê escolheu leite condensado.");
        break;
    }
            return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.17 utilizando a ferramenta Paiza.io.

Lembre-se: o treinamento deixará você cada vez melhor e mais confiante na programação.

Ver anotações 0

NÃO PODE FALTAR

ESTRUTURAS DE REPETIÇÃO DETERMINÍSTICAS

Márcio Aparecido Artero

LOOPS OU LAÇOS DE REPETIÇÃO PARA USO EM DIVERSAS SITUAÇÕES

As estruturas de repetição também são utilizadas para repetir uma série de operações semelhantes que são executadas para todos os elementos de uma lista de dados, armazenados em tabelas, vetores ou matrizes.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Você, novamente, recebeu uma missão da instituição de ensino na qual se formou. Foi solicitado um programa em linguagem C para transformar o nome digitado dos alunos em letras maiúsculas, assim, caso uma pessoa digite o nome do aluno em letras minúsculas, o programa transformará automaticamente em maiúsculas. Essa é uma prática muito comum no dia a dia do desenvolvimento de software. Por exemplo, ao fazer a busca por um aluno da instituição usando seu nome, antes, é necessário transformar o nome informado, bem como os nomes dos alunos cadastrados no software para caixa alta (maiúsculas) ou baixa (minúsculas). Isso evitará que o fato de uma letra ser informada maiúscula ou minúscula prejudique a busca, ou seja, tanto faz se o usuário informou “josé” ou “José” ou “JOSÉ”; o resultado da busca deve ser sempre o mesmo. Outra aplicação prática desse exercício é quando se quer gerar uma lista de presença dos alunos de determinada turma. É comum sempre imprimir os nomes em caixa alta (maiúsculas), mantendo um padrão entre os nomes e facilitando a leitura do professor. Após a criação do programa, entregue o código no formato TXT para documentação.

Qual função em linguagem C devemos usar para converter maiúsculas em minúsculas e vice-versa? Caro aluno, são tantas as possibilidades de aplicação de estrutura de repetição que não podemos deixar passar esse novo desafio. Então, vamos buscá-lo? Pratique bastante e bons estudos!

CONCEITO-CHAVE

Estruturas de repetição possibilitam a repetição de um comando ou de uma sequência de comandos várias vezes. Essas estruturas, também chamadas de laços de repetição ou loops, são utilizadas em diversas aplicações. Considere um sistema de compras de passagens aéreas: um

cliente quer viajar para um destino em determinado dia e horário; para cada horário de disponibilidade do voo haverá uma quantidade fixa de assentos disponíveis. Como a aeronave tem uma quantidade fixa de poltronas, pode-se utilizar o comando for, assim, caso as poltronas tenham se esgotado, o usuário não conseguirá comprar a passagem. Podemos comparar esse loop executado a um comando for: pense que a cada cliente que compra uma poltrona é incrementado em 1 do total de poltronas – logo, em comandos de compra – e o pagamento da passagem será concluída.

Figura 2.11 | Reservas de assentos em um voo

Fonte: Shutterstock.

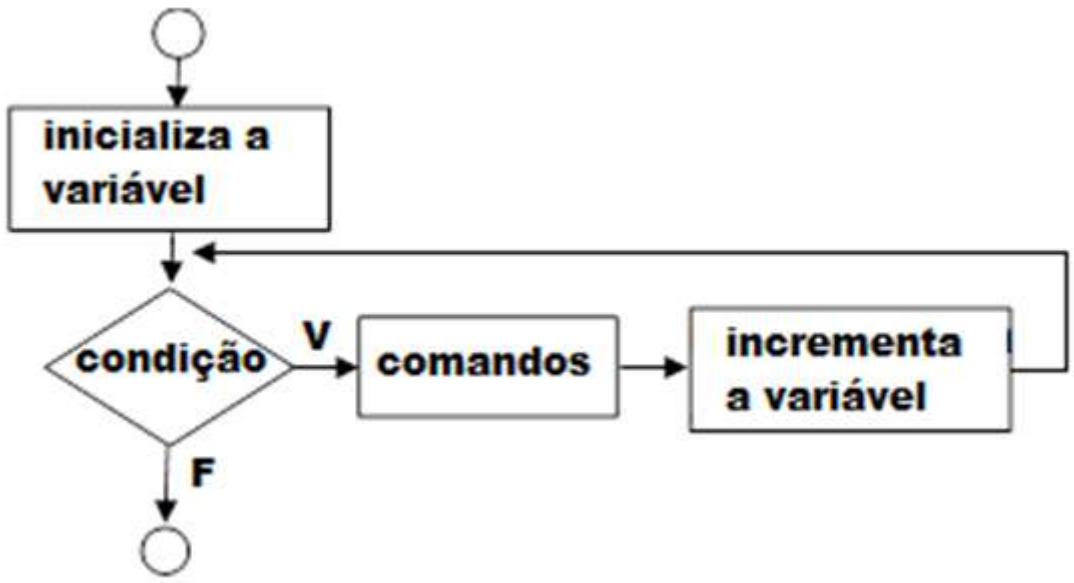
ESTRUTURA DE REPETIÇÃO COM VARIÁVEIS DE CONTROLE - *FOR*

Nesta seção, aprenderemos a estrutura de repetição usando *for* e suas aplicações, bem como os comparativos com as outras estruturas de repetição.

Para tal, vamos iniciar falando da **repetição com variáveis de controle**, ou seja, como aplicaremos o laço *for*. Esse comando – que em português significa “para”, segundo Mizrahi (2008) –, é geralmente usado para repetir um comando ou uma sequência de comandos um

determinado número de vezes, isto é, podemos determinar quantas vezes acontecerá a repetição. Veja na Figura 2.12 como é representada a estrutura de repetição usando o comando for.

Figura 2.12 | Fluxograma de repetição com variáveis de controle



Fonte: elaborada pelo autor.

A sintaxe usando a linguagem de programação em C fica da seguinte forma:

```
for(inicialização; condição de parada; incremento) {
  comando ou sequência de comandos;
}
```

Na aplicação do comando *for* você encontra três expressões separadas por ponto e vírgula. O significado de cada uma delas é apresentado a seguir:

- **Inicialização**: neste momento, coloca-se a instrução de atribuição para inicializar o contador do laço. A inicialização é executada uma única vez antes de começar o laço.
- **Condição de parada**: realiza um teste que determina se a condição é verdadeira ou falsa; se for verdadeira, permanece no laço e, se for falsa, encerra o laço e passa para a próxima instrução.
- **Incremento**: parte das nossas explicações anteriores, em que é possível incrementar uma repetição de acordo com um contador específico, lembrando que o incremento é executado depois dos comandos.

Para facilitar ainda mais, veja a seguinte representação:

for (y = 0; y <= 10; y++)

y = 0	**y <= 10**	**y++**
Declaração e atribuição de valor inicial para y, assim, tem o seu valor iniciado em 0.	O laço se repetirá enquanto y for menor ou igual a 10.	Ao final da execução dos comandos do laço de repetição, y será incrementado em 1.

Ver anotações

Agora, vamos aplicar essa representação em um programa que mostra uma sequência de números, em que y vai de 0 a 10. Observe a saída do código utilizando a ferramenta Paiza.io.

Código 2.18 | Comando *for* – sequência de números

```c
#include <stdio.h>
int main() {
    int x, y;
    for(y = 0; y <= 10; y++) {
        printf("\ny=%d", y);
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.18 utilizando a ferramenta Paiza.io.

ASSIMILE

O incremento em um comando *for* não precisar ser unitário (de um em um). Você pode utilizar incrementos maiores para, por exemplo, imprimir todos os números pares de 0 a 10:

Código 2.19 | Comando *for* – incremento

```c
#include <stdio.h>
int main() {
    int x, y;
    for(y = 0; y <= 10; y += 2) {
        printf("\ny=%d", y);
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

No código a seguir, vamos criar uma contagem regressiva de um número qualquer, digitado pelo usuário:

Código 2.20 | Comando *for* – contagem regressiva

```c
#include <stdio.h>
int main() {
    int contador;
    printf("\nDigite um número para contagem regressiva: ");
    scanf("%d", &contador);
    for (contador; contador >= 1; contador--) {
        printf("%d ", contador);
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

EXEMPLIFICANDO

Você pode usar o comando *break* dentro de um laço for para uma determinada condição, forçando, assim, o término do laço. Veja o exemplo a seguir:

VETORES

Agora, vamos trabalhar algumas aplicações utilizando vetores. Segundo Manzano (2013; 2015), vetor (*array*) é um tipo especial de variável, capaz de armazenar diversos valores “ao mesmo tempo”, usando um mesmo nome de variável. Por armazenar diversos valores, também é chamado de variável composta. Veja a sintaxe a seguir para utilização de vetores:

```
tipo variavel[n]
```

em que “n” representa a quantidade de elementos do vetor.

Vamos a outro exemplo em que armazenamos valores em um vetor. Lembre-se de que os elementos de um vetor são acessados por meio de índices, que vão de 0 à n – 1, em que n é a quantidade de elementos.

Código 2.21 | Comando *for* – vetor

```
#include <stdio.h>

#define VET_TAM 5

int main () {
    int num[VET_TAM];

    for (int i = 0; i < VET_TAM; i++) {
        printf("\nEntre com um número: ");
        scanf("%d", &num[i]);
        printf("\nO valor digitado foi: %d", num[i]);
    }

    return 0;
}

```

Fonte: elaborado pelo autor.

EXEMPLIFICANDO

Que tal usarmos o comando *for* e *while* no mesmo código?

O programa a seguir encontra a primeira posição para um determinado número inserido pelo usuário.

Vejamos, então, como fica a programação:

Código 2.22 | Comandos for e *while*

```c
#include <stdio.h>

#define VET_TAM 5

int main () {
    int vetor[VET_TAM];
    int num, i = 0, achou = 0;

    // Preenche vetor
    for (int i = 0; i < VET_TAM; i++) {
        printf("\nEntre com um número: ");
        scanf("%d", &vetor[i]);
    }

    printf("\n\nInforme o número a ser encontrado: ");
    scanf("%d", &num);

    while(i < VET_TAM) {
        if (vetor[i] == num) {
            achou = 1;
            break;
        }
        i++;
    }

    if (achou == 1) {
        printf("\nO número %d foi encontrado na posição %d do vetor", num, i);
    } else {
        printf("\nO número %d não foi encontrado no vetor", num);
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.22 utilizando a ferramenta Paiza.io.

COMANDO *CONTINUE*

Dando prosseguimento aos nossos estudos, vamos conhecer o comando *continue*. Segundo Damas (2016), um comando *continue*, dentro de um laço, possibilita que a execução de comandos corrente seja terminada, passando à próxima iteração do laço, ou seja, quando usamos o *continue* dentro de um laço, este é passado para a próxima iteração.

Vejamos, agora, um programa que percorrerá os números de 1 a 30, imprimindo na tela apenas os números ímpares. Nas iterações com números pares, o comando *continue* é usado para levar o programa para a próxima iteração. As linhas 3 a 8 declaram um comando for para iterar (percorrer) entre os números 1 a 30, imprimindo seu valor da tela, conforme mostra a linha 7. Contudo, na linha 4 é feito um teste condicional para verificar se o número da iteração corrente, representado pela variável i, é par. Caso seja, então o comando continue é executado, fazendo com que o laço for vá para a próxima iteração, sem executar as linhas que estão abaixo do comando *continue*. Nesse sentido, pode-se afirmar que o programa não imprime o valor dos números pares na tela.

Código 2.23 | Comando *for* – continue

```
1  #include <stdio.h>
2  int main() {
3      for (int i = 1; i <= 30; i++) {
4          if (i % 2 == 0) {
5              continue;
6          }
7          printf("%d ", i);
8      }
9  }
```

Fonte: elaborado pelo autor.

Agora, você pode testar o código utilizando a ferramenta Paiza.io.

> **REFLITA**
>
> Segundo Damas (2016), o comando *continue* poderá apenas ser utilizado dentro de laços. No entanto, o comando *break* pode ser utilizado em laços ou nas instruções *switch*. Existem outras formas de continuar uma instrução dentro do laço?

Ver anotações 0

MATRIZES

Dando sequência aos nossos estudos, vamos entender como são aplicadas as **matrizes**. “Matrizes são arranjos de duas ou mais dimensões. Assim como nos vetores, todos os elementos de uma matriz são do mesmo tipo, armazenando informações semanticamente semelhantes” (EDELWEISS, 2014, p. 196).

Veja como fica a sintaxe de matrizes de duas dimensões:

```
tipo variável[M][N]
```

M representa a quantidade de linhas e N a quantidade de colunas.

Importante lembrar que:

- Em qualquer variável composta, o índice começa por zero, então, em uma matriz, o primeiro espaço para armazenamento é sempre (0,0), ou seja, índice 0, tanto para linha como para coluna.
- Não é obrigatório que todas as posições sejam ocupadas, sendo possível declarar uma matriz com 10 linhas (ou colunas) e usar somente uma.

Observe o programa a seguir: temos uma matriz 3x3 em que os valores são lançados de acordo com a linha e coluna e que é montada no formato da matriz. Nesse código, na linha 5 é declarada uma matriz 3x3. Das linhas 6 a 11, essa matriz é percorrida, célula por célula, por meio de um comando *for*, e os valores são lidos e adicionados na matriz

(linhas 8 e 9). É possível notar que há um comando *for* (linhas 7 a 10) dentro de outro comando *for* (linhas 6 a 11). Isso é bastante comum de se ver quando se trabalha com matrizes bidimensionais, pois é preciso percorrer a matriz em duas dimensões (linhas e colunas), então, há um comando for para lidar com as linhas (o mais externo) e um para lidar com as colunas (o mais interno). Continuando a explicação do código, as linhas 12 a 18 tratam da impressão dos dados da matriz na tela. O processo é basicamente o mesmo das linhas 6 a 11, com a diferença que agora o comando *printf* é usando, em vez do *scanf*.

Veja o resultado da programação na Figura 2.13.

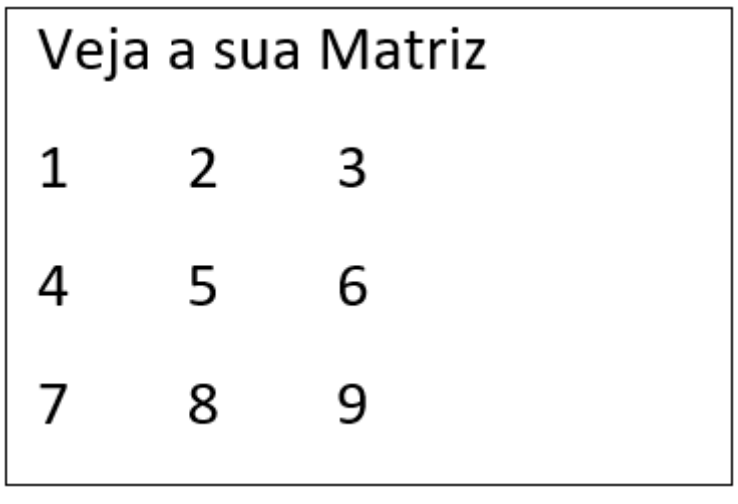
Código 2.24 | Comando *for* – matriz

```
1  #include <stdio.h>
2
3  int main() {
4      int linha, coluna;
5      int matriz[3][3];
6      for (linha = 0; linha < 3; linha++) {
7          for (coluna = 0; coluna < 3; coluna++) {
8              printf("\nDigitar os valores da matriz para
   [linha %d, coluna %d]: ", linha + 1, coluna + 1);
9              scanf("%d", &matriz[linha][coluna]);
10         }
11     }
12     printf("\n\nVeja a sua Matriz\n");
13     for (linha = 0; linha <= 2; linha++) {
14         for (coluna=0;coluna<3;coluna++) {
15             printf("%d\t", matriz[linha][coluna]);
16         }
17         printf("\n");
18     }
19     return 0;
20 }
```

Fonte: elaborado pelo autor.

Teste o Código 2.24 utilizando a ferramenta Paiza.io.

Figura 2.13 | Resultado da impressão de uma matriz 3x3



Fonte: captura de tela da saída do programa Paiza.io elaborada pelo autor.

Para encerrar a seção, vamos trabalhar em um problema sobre matrizes: a ideia é descobrir se determinada matriz é diagonal ou não. Uma matriz diagonal é definida como uma matriz em que todos os elementos cujo $i \neq j$ são nulos. Esse problema utiliza estruturas de decisão e repetição e uma das possíveis soluções para o problema apresentado no código a seguir:

Código 2.25 | Comando *for* – matriz diagonal

```c
#include <stdio.h>

int main() {
    int linha, coluna;
    int matriz[3][3];
    int eh_diagonal = 1;

    for (linha = 0; linha < 3; linha++) {
        for (coluna = 0; coluna < 3; coluna++) {
            printf("\nDigitar os valores da matriz para [linha %d, coluna %d]: ", linha + 1, coluna + 1);
            scanf("%d", &matriz[linha][coluna]);
        }
    }
    for (linha = 0; linha < 3; linha++) {
        for (coluna = 0; coluna < 3; coluna++) {
            if (coluna != linha && matriz[linha][coluna] != 0) {
                eh_diagonal = 0;
            }
        }
    }

    if (eh_diagonal == 1) {
        printf("\n\nSua matriz é diagonal!");
    } else {
        printf("\n\nSua matriz não é diagonal!");
    }
    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 2.25 utilizando a ferramenta Paiza.io.

As linhas 8 a 13 do código apresentado são responsáveis por preencher a matriz 3x3 declarada na linha 5 com números lidos do teclado, assim como vimos no exemplo anterior. As linhas 14 a 20 verificam se a matriz informada é ou não uma matriz diagonal. Para isso, no comando *if* da linha 16 é verificado se o valor das variáveis “linha” e “coluna” são diferentes (essas variáveis representam os índices da matriz). Caso sejam, então verifica-se se o valor contido na posição matriz[linha] [coluna] é diferente de 0 (zero). Isso é feito, pois como é dito no enunciado do problema, para uma matriz ser considerada diagonal, todos os elementos fora da diagonal principal (ou seja, elementos cuja posição da linha e coluna são iguais) devem ser nulos. Caso o resultado da condição da linha 16 seja verdadeira, então o valor da variável “eh_diagonal” passa a ser igual a 0 (zero), representando que aquela matriz não é uma matriz diagonal. Caso contrário, o valor da variável “eh_diagonal” continua sendo igual a 1 (um), conforme declarado na linha 6 do programa. Ao finalizar a execução do laço *for*, nas linhas 22 a 26 é feita a verificação da variável “eh_diagonal”. Caso seu valor seja igual a 1, então é impresso na tela a mensagem “Sua matriz é diagonal!” (linha 23), caso contrário, será impresso “Sua matriz não é diagonal!” (linha 25).

EXEMPLIFICANDO

Retomando o exemplo que demos no início desta seção, suponha um sistema de compras de passagens aéreas. Um cliente quer viajar para um destino em determinado dia e horário e, para cada horário de disponibilidade do voo, haverá uma quantidade fixa de poltronas disponíveis.

Para armazenar essas informações usaremos uma matriz 3x5, em que cada linha representa um horário de voo e cada coluna representa uma das seguintes situações: poltrona livre (que será representado pelo número inteiro 0), poltrona ocupada (que será representado pelo número inteiro 1) e

poltrona indisponível (que será representado pelo número inteiro -1). Por exemplo, considerando a matriz a seguir, temos o seguinte:

i. O primeiro horário de voo disponibiliza 4 poltronas, sendo que as três primeiras estão ocupadas e a 4ª poltrona está livre.

ii. O segundo horário conta com 5 poltronas, contudo, todas já estão ocupadas.

iii. E o terceiro e último horário tem apenas 2 poltronas e todas encontram-se livres.

```
0   1  1  0 -1
0   1  1  1  1
0   0 -1 -1 -1
```

Considerando a matriz anterior como entrada, leia o horário do voo solicitado pelo usuário (1, 2 ou 3) e informe uma das seguintes opções: “Seu voo foi reservado com sucesso!” ou “Não há poltronas disponíveis nesse voo!”. Caso o voo seja agendado, a matriz de informações do voo deve ser atualizada.

Para ficar mais interessante, crie um menu de opções bem simples, contendo “Reservar voo” e “Sair”.

Uma possível solução para o problema é apresentada a seguir:

Código 2.26 | Matriz - menu

```c
#include <stdio.h>
int main() {
    // Matriz inicial
    int mat[3][5] = {
        1, 1, 1, 0, -1,
        1, 1, 1, 1, 1,
        0, 0, -1, -1, -1
    };

    int opcao;
    int numVoo;
    int reservou;

    do  {
        printf("\n1 - Reservar voo\n2 - Sair\nInforme a opção desejada: ");
        scanf("%d", &opcao);
        switch(opcao) {
            case 1:
                printf("Informe o número do voo (1-3): ");
                scanf("%d", &numVoo);
                reservou = 0;
                if (numVoo >= 1 && numVoo <= 3) {
                    for(int i = 0; i < 5 && reservou == 0; i++) {
                        if (mat[numVoo - 1][i] == 0) {
                            reservou = 1;
                            mat[numVoo - 1][i] = 1;
                        }
                    }
                    if (reservou == 1) printf("\nSeu voo foi reservado com sucesso!");
                    else printf("\nNão há poltronas disponíveis nesse voo!");
```

Ver anotações

```
                    } else {
                        printf("Opção inválida!");
                    }
                    break;
                case 2:
                    printf("\nSaindo...");
                    break;
                default: printf("Opção inválida!");
            }
        } while (opcao != 2);
        return 0;
    }
```

Fonte: elaborado pelo autor

Teste o Código 2.26 utilizando a ferramenta Paiza.io.

PESQUISE MAIS

O vídeo referido a seguir traz uma dinâmica muito interessante na aplicação de vetores e matrizes. Realizado de “aluno para aluno”, apresenta uma revisão bem minuciosa da programação em linguagem C:

PROGRAMAR em C - Revisão Vetores/Matrizes - Aula 27. [*S. l.*], 21 nov. 2012. (22 min. 28 s.). Publicado pelo canal De aluno para aluno.

Chegamos ao final desta seção. Procure praticar e testar os programas apresentados. Lembre-se: sempre existe uma forma diferente para resolver problemas computacionais. Sucesso!

FAÇA VALER A PENA

Questão 1

Análise o código apresentado a seguir:

Ver anotações 0

```c
#include <stdio.h>
int main() {
    int A[5];
    int I, SOMA = 0;
    printf("\nSomatorio de elementos\n\n");
    for (I = 0; I <= 4; I++) {
        printf("\nInforme um valor para o elemento nr. %2d:
", I);
        scanf("%d", &A[I]);
    }
    for (I = 0; I <= 4; I++) {
        if (A[I] % 2 != 0) {
            SOMA += A[I];
        }
    }
    printf("\n\nA soma dos elementos equivale a: %d", SOMA);
    return 0;
}
```

Segundo Manzano (2013), podemos dizer que o código a seguir executa as seguintes instruções:

I. Inicia o contador de índice, variável I, como 0 (zero).

II. Lê os cinco valores, um a um.

III. Verifica se o elemento é par; se sim, efetua a soma dos elementos.

Apresenta o total de todos os elementos pares da matriz.

Após a análise do código, é correto o que se afirma em:

a. I, apenas.

b. I e II, apenas.

c. I, II e III.

d. II, apenas.

e. III, apenas.

0

Ver anotações

Questão 2

Quando trabalhamos com o comando *for*, podemos encontrar três expressões separadas por ponto e vírgula. A primeira expressão é a inicialização, que é executada uma única vez, antes de começar o laço. A segunda é a condição de parada, em que é realizado um teste que determina se a condição é verdadeira ou falsa e, caso seja verdadeira, permanece no laço, caso falsa, encerra o laço e passa para a próxima instrução. A última expressão é executada depois dos comandos.

O nome dado para essa última expressão é:

a. Somatório.

b. Finalização.

c. Processamento.

d. Incremento.

e. Substituição.

Questão 3

Analise o código do programa a seguir, em que foi utilizada a estrutura de repetição com variável de controle:

```
1  #include <stdio.h>
2  int main() {
3     int contador; //variável de controle do loop
4     for(contador = 1; contador <= 10; contador++) {
5         printf("%d ", contador);
6     }
7     return 0;
8  }
```

Analisando o programa apresentado, assinale a alternativa que corresponde à leitura que podemos fazer da linha 4.

a. Na primeira expressão, “contador” tem o seu valor iniciado em “1”. Na segunda expressão, “contador” está condicionado a ser igual a “10”. Na terceira expressão, “contador” será realizado o incrementado de 1 para ao seu valor.

Ver anotações

b. Na primeira expressão, “contador” tem o seu valor iniciado em “1”. Na segunda expressão, “contador” está condicionado a ser menor que “10”. Na terceira expressão, “contador” será realizado o incrementado de 1 para ao seu valor.

c. Na primeira expressão, “contador” tem o seu valor iniciado em “1”. Na segunda expressão, “contador” está condicionado a ser maior ou igual a “10”. Na terceira expressão, “contador” será realizado o decremento de 1 para ao seu valor.

d. Na primeira expressão, “contador” tem o seu valor iniciado em “0”. Na segunda expressão, “contador” está condicionado a ser menor ou igual a “10”. Na terceira expressão, “contador” será realizado o incrementado de 2 para ao seu valor.

e. Na primeira expressão, “contador” tem o seu valor iniciado em “1”. Na segunda expressão, “contador” está condicionado a ser menor ou igual a “10”. Na terceira expressão, “contador” será realizado o incrementado de 1 para ao seu valor.

REFERÊNCIAS

02 - EXERCÍCIO - Estruturas de repetição em C. [S. l.], 10 mar. 2017. 1 vídeo (10 min 38s). Publicado pelo canal Hélio Espiridião. Disponível em: https://cutt.ly/njEroNN. Acesso em: 10 jan. 2021.

13 - PROGRAMAÇÃO em Linguagem C - Desvio Condicional Aninhado - if / else if. [S. l.], 27 fev. 2015. 1 vídeo (11 min. 12 s.). Publicado pelo canal Bóson Treinamentos. Disponível em: https://cutt.ly/rjErauV. Acesso em: 10 jan. 2021.

DAMAS, L. **Linguagem C**. 10. ed. Rio de Janeiro: LTC, 2016.

EDELWEISS, N. **Algoritmos e programação com exemplos em Pascal e C**. Porto Alegre: Bookman, 2014.

MANZANO, J. A. N. G. **Estudo dirigido de linguagem C**. 17. ed. rev. São Paulo: Érica, 2013.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo : Érica, 2015.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.

PROGRAMAÇÃO C - Aula 07 - while, do-while, for - Estruturas de repetição. [*S. l.*], 10 abr. 2012. 1 vídeo (15 min. 56 s.). Publicado pelo canal Peterson Lobato. Disponível em: https://cutt.ly/ajErsxs. Acesso em:

10 jan. 2021.

PROGRAMAR em C - Revisão Vetores/Matrizes - Aula 27. [*S. l.*], 21 nov. 2012. 1 vídeo (22 min. 28 s.). Publicado pelo canal De aluno para aluno. Disponível em: https://cutt.ly/2jErdMl. Acesso em: 10 jan. 2021.

SCHILDT, H. **C Completo e total**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

SOFFNER, R. **Algoritmos e Programação em Linguagem C**. São Paulo: Saraiva, 2013.

FOCO NO MERCADO DE TRABALHO

ESTRUTURAS DE REPETIÇÃO DETERMINÍSTICAS

Márcio Aparecido Artero

PROGRAMA PARA CONVERSÃO DE TEXTO EM LETRAS MAIÚSCULAS

Criação de um programa em linguagem C para transformar o nome digitado dos alunos em letras maiúsculas, para não prejudicar a busca , manter o padrão e facilitar a leitura.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Acreditamos que você já esteja preparado para solucionar o desafio dado pela instituição de ensino. Foi solicitado um programa em linguagem C para transformar o nome digitado dos alunos em letras maiúsculas.

Para resolver essa questão, é importante que você esteja atento às seguintes condições antes de iniciar a programação:

Utilizar a biblioteca `ctype.h` dentro da linguagem C, que proporcionará o uso de funções e macros para trabalhar com caracteres.

No nosso desafio, especificamente, podemos utilizar a função `toupper`, que converte os caracteres minúsculos em maiúsculos.

Agora sim, vamos à programação:

Código 2.27 | Caracteres

```c
#include <stdio.h>
#include <ctype.h>

#define NOME_TAM 30

int main() {
    char nome[NOME_TAM];
    int i;
    printf("\nDigite o nome do(a) aluno(a):");
    fgets(nome, NOME_TAM, stdin);
    printf("\nNome ANTES da mudança: %s", nome);

    for(int i = 0; i < NOME_TAM; i++) {
        nome[i] = toupper(nome[i]);
    }

    printf("\nNome DEPOIS da mudança: %s", nome);
    return 0;
}
```

Fonte: elaborado pelo autor.

Teste o Código 2.27 utilizando a ferramenta Paiza.io.

Boa sorte e ótimos estudos!

AVANÇANDO NA PRÁTICA

FORMATAR CPF

Você foi contratado por uma empresa de comunicação para resolver um problema na digitação dos números de CPF dos clientes. A questão é que, quando o usuário digita o CPF com pontos e traço, a indexação e busca são dificultadas, ou seja, pode acontecer um erro de autenticidade. Para resolver esse impasse, você deverá desenvolver um programa para padronizar o formato do CPF, eliminando os pontos e traços digitados pelos usuários. Como resolver essa situação?

Ver anotações

RESOLUÇÃO

Para resolver a questão, você poderá usar vetores com laços de repetições para eliminar os pontos e traço e o comando continue dentro da estrutura de repetição. Veja no código uma das possíveis soluções:

Código 2.28 | CPF

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
        char cpf_entrada[15];
        char cpf_corrigido[15];
    int n = 0;

        printf("\nInforme seu CPF (XXX.XXX.XXX-XX):");
        scanf("%s", cpf_entrada);

        for(int i = 0; i < 14; i++) {
                if(cpf_entrada[i] == '.' || cpf_entrada[i] == '-'){
                        continue;
                }
                cpf_corrigido[n] = cpf_entrada[i];
                n++;
        }
        cpf_corrigido[n] = '\0'; // caractere terminador da string

        printf("\nCPF corrigido = %s", cpf_corrigido);
        return 0;
}
```

Fonte: elaborado pelo autor.

Teste o código 2.28 utilizando a ferramenta Paiza.io.

Ver anotações

Muito bem, agora é com você! Modifique e tente otimizar ao máximo os seus programas.

Ver anotações 0

NÃO PODE FALTAR

PROCEDIMENTOS E FUNÇÕES

Vanessa Cadan Scheffer

O QUE SÃO FUNÇÕES OU PROCEDIMENTOS?

São blocos escritos tanto para dividir a complexidade de um problema maior quanto para evitar a repetição de códigos. Essa técnica também pode ser chamada de modularização, ou seja, um problema será resolvido em diferentes módulos (MANZANO, 2015).

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Caro estudante, bem-vindo à unidade sobre funções e recursividade. Ao longo do livro, você teve a oportunidade de conhecer diversas técnicas que possibilitam resolver os mais variados problemas. Esse conhecimento pode lhe abrir muitas oportunidades profissionais, uma vez que você passa a olhar e a pensar a respeito da solução de um problema de maneira mais estruturada.

Todos os computadores (e agora os smartphones e tablets) precisam de um sistema que faça o gerenciamento dos recursos, o qual é chamado de sistema operacional (SO). Existem três SO que se destacam no mercado: o MacOS, o Windows e o Linux. Segundo Counter (2018), somente o núcleo (*Kernel*) do código fonte do Linux apresenta mais de 20 milhões de linhas de código. Você consegue imaginar como os programadores conseguem manter tudo isso organizado? Como conseguem encontrar certas funcionalidades? Certamente não é procurando linha por linha. Tanto o desenvolvimento quanto a manutenção de um sistema só são realizáveis porque as funcionalidades são divididas em "blocos", que são chamados de funções ou procedimentos, assunto central desta unidade de estudo, o qual o habilitará a criar soluções computacionais com tais recursos.

A carreira de desenvolvedor não se limita a criar soluções comerciais, como sistemas de vendas, sistemas de controle de estoques etc. O universo desse profissional é amplo e o que não falta são áreas com possibilidades de atuação. Você foi contratado por um laboratório de pesquisa que conta com a atuação de engenheiros, físicos, químicos e matemáticos, os quais trabalham pesquisando e desenvolvendo soluções para empresas que os contratam em regime terceirizado. Muitas vezes as soluções desenvolvidas precisam ser implementadas em um software, e você foi o escolhido para essa missão.

Nesta primeira seção você aprenderá a criar funções que retornam valores. Na segunda seção avançaremos com as funções que, além de retornar, também recebem valores. Por fim, na terceira seção, veremos uma classe especial de funções, chamadas de recursivas.

Bons estudos!

PRATICAR PARA APRENDER

Você foi contratado por um laboratório de pesquisa que presta serviço terceirizado para atuar juntamente com diversos profissionais. Seu primeiro trabalho será com o núcleo de nutricionistas. Eles receberam uma demanda da construtora local para calcular o IMC (Índice de Massa Corporal) de um indivíduo e dizer qual é a sua situação. Há três tipos de situações: abaixo do peso, peso ideal e sobrepeso. A escolha de qual situação retornar depende do IMC do indivíduo, que é calculado pela fórmula **IMC = Peso / Altura²**. A regra para escolher a situação está especificada no Quadro 3.1.

Quadro 3.1 | Regras para escolha da situação do indivíduo com base em seu IMC

Regra	**Situação**
IMC < 18.5	Abaixo do peso
18.5 <= IMC < 24.9	Peso ideal
IMC >= 24.9	Sobrepeso

Fonte: elaborado pela autora.

Como você poderá automatizar o processo, com base na entrada das medidas (peso e altura)? Como o programa escolherá e informará a situação do indivíduo?

Bons estudos!

CONCEITO-CHAVE

Em uma empresa, quando precisa resolver um problema na folha de pagamento, você se dirige até o setor financeiro. Quando vai sair de férias, precisa ir até o RH para assinar o recibo de férias. Quando precisa de um determinado suprimento, o setor responsável o atende. Mas e se

não houvesse essa divisão, como você saberia a quem recorrer em determinada situação? Seria mais difícil e desorganizado, certo? Um software deve ser construído seguindo o mesmo princípio de organização: cada funcionalidade deve ser colocada em um “local” com uma respectiva identificação, para que o requisitante a possa encontrar. Uma das técnicas de programação utilizada para construir programas dessa forma é a construção de funções, que você aprenderá nesta seção.

FUNÇÕES

Até esse momento, você já implementou diversos algoritmos na linguagem C. Nesses programas, mesmo sem ainda conhecer formalmente, você já utilizou funções que fazem parte das bibliotecas da linguagem C, como *printf()* e *scanf()*. Na verdade, você utilizou uma função desde a primeira implementação, mesmo sem conhecê-la, pois é uma exigência da linguagem. Observe o programa no Código 3.1 que imprime uma mensagem na tela. Veja o comando na linha 2, ***int main()***. Esse comando especifica uma função que se chama “*main*” e que vai devolver para quem a requisitou um valor inteiro, nesse caso, zero. Vamos entender como construir e como funciona esse importante recurso.

Código 3.1 | Programa *Hello World*

```
1  #include<stdio.h>
2  int main(){
3      printf("Hello World!");
4      return 0;
5  }
```

Fonte: elaborado pela autora.

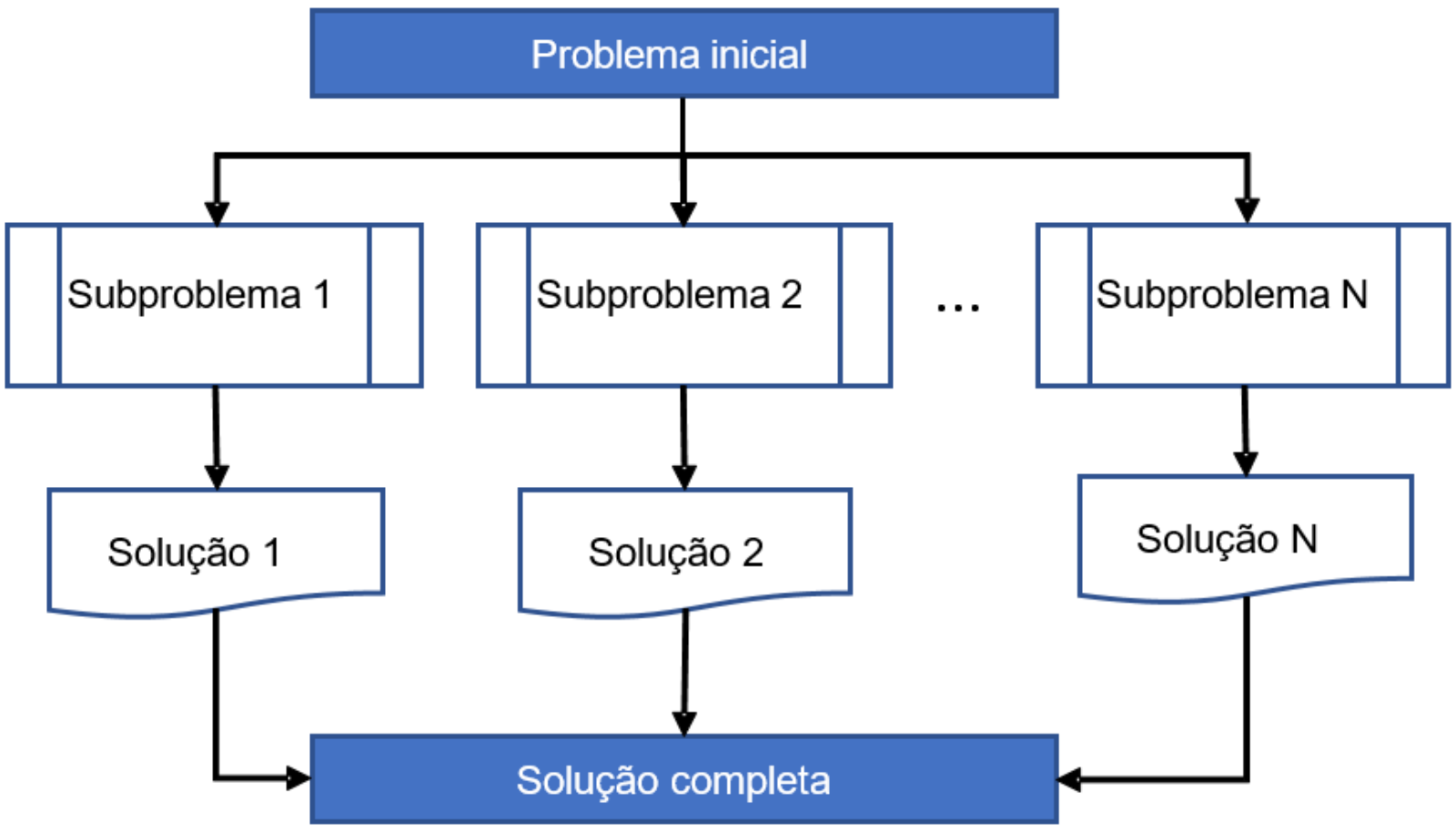
A ideia de criar programas com blocos de funcionalidades vem de uma técnica de projeto de algoritmos chamada *dividir para conquistar* (MANZANO; MATOS; LOURENÇO, 2015). A ideia é simples: dado um problema, este deve ser dividido em problemas menores, o que facilita a resolução e organização. A técnica consiste em três passos (Figura 3.1):

1. **Dividir**: quebrar um problema em outros subproblemas menores. “Solucionar pequenos problemas, em vez de um grande problema, é, do ponto de vista computacional, supostamente mais fácil” (MANZANO; MATOS; LOURENÇO, 2015, p. 102).

2. **Conquistar**: usar uma sequência de instruções separada, para resolver cada subproblema.

3. **Combinar**: juntar a solução de cada subproblema para alcançar a solução completa do problema original.

Figura 3.1 | Esquema da técnica *dividir* para conquistar



Fonte: adaptada de Manzano, Matos e Lourenço (2015, p. 102).

Agora que foi apresentada a ideia de subdividir um problema, podemos avançar com os conceitos sobre as funções.

Uma função é definida como um trecho de código escrito para solucionar um subproblema (SOFFNER, 2013). Esses blocos são escritos tanto para dividir a complexidade de um problema maior quanto para

evitar a repetição de códigos. Essa técnica também pode ser chamada de modularização, ou seja, um problema será resolvido em diferentes módulos (MANZANO, 2015).

> **ASSIMILE**
>
> A modularização é uma técnica de programação que permite a divisão da solução de um problema a fim de diminuir a complexidade, tornar o código mais organizado e evitar a repetição de códigos.

SINTAXE PARA CRIAR FUNÇÕES

Para criar uma função utiliza-se a seguinte sintaxe:

```
<tipo de retorno> <nome> (<parâmetros>){
   <Comandos da função>
   <Retorno> (não obrigatório)
}
```

Em cada declaração da função, alguns parâmetros são obrigatórios e outros opcionais. Veja cada parâmetro.

- **<tipo de retorno>** – Obrigatório. Esse parâmetro indica qual o tipo de valor que a função vai retornar. Pode ser um valor inteiro (*int*), decimal (*float* ou *double*), caractere (*char*) etc. Quando a sub-rotina faz um processamento e não retorna qualquer valor, usa-se o tipo *void* e, nesse caso, é chamado de **procedimento** (MANZANO, 2015).
- **<nome>** – Obrigatório. Parâmetro que especifica o nome que identificará a função. É como o nome de uma pessoa; para convidá-la para sair você precisa “chamá-la pelo nome”. O nome não pode ter acento nem caractere especial (mesmas regras para nomes de variáveis).
- **<parênteses depois do nome>** – Obrigatório. Toda função ou procedimento sempre terá o nome acompanhado de parênteses. Por exemplo: *main(), printf(), somar()* etc.

- **<comandos da função>** – Opcional. Veremos na próxima seção.
- **<retorno>** – Quando o tipo de retorno for *void* esse parâmetro não precisa ser usado, porém, quando não for *void* é obrigatório. O valor a ser retornado tem que ser compatível com o tipo de retorno, senão o problema resultará em erro de compilação em algumas linguagens, e em outras retornará um valor errôneo. Na linguagem C, vai ser retornado um valor de acordo com o tipo.

LOCAL DA FUNÇÃO

Em qual parte do código a função deve ser programada?

A função *main()*, que traduzindo significa principal, é de uso obrigatório em várias linguagens de programação, por exemplo, em C, em Java e em C#. Ela é usada para identificar qual é a rotina principal do programa e por onde a execução deve começar.

Na linguagem C, vamos adotar sempre a criação das funções (sub-rotinas) antes da função *main()*, por uma questão de praticidade e conveniência.

> **EXEMPLIFICANDO**
>
> Veja no Código 3.2 um programa que utiliza uma função para calcular a soma entre dois números e, a seguir, sua explicação detalhada.

Código 3.2 | Programa com a função *somar()*

```c
#include<stdio.h>
int somar(){
    return 2 + 3;
}
int main(){
    int resultado = 0;
    resultado = somar();
    printf("O resultado da funcao e =
%d",resultado);
    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.2 utilizando a ferramenta Paiza.io.

A função *somar()* no Código 3.2 inicia-se na linha 2 e termina na linha 4, com o fechamento das chaves. Vamos analisar os parâmetros para esse caso.

<tipo de retorno> – Foi especificado que a função vai retornar um valor inteiro (*int*).

<nome> – somar.

<comandos da função> – Essa função foi criada de modo a retornar a soma de dois valores, tudo em um único comando: `return 2 + 3`.

Vale ressaltar que cada programador criará suas funções da maneira que julgar mais adequado. Por exemplo, se os comandos tivessem sido escritos na forma:

```c
int somar(){
 int x = 0;
 x = 2 + 3;
 return x;
}
```

A função teria o mesmo efeito. Mas veja, da forma que está no Código 3.2, escrevemos duas linhas a menos (imagine essa diferença sendo acumulada nas mais de 20 milhões linhas do Linux). Veja que no Código 3.2 foi usada uma variável a menos, o que significa economia de memória do computador e do processamento para sua alocação.

> **ASSIMILE**
>
> Os comandos que serão usados em cada função dependem da escolha de cada desenvolvedor. Tenha em mente que dado um problema, pode haver 1, 2, ..., *N* maneiras diferentes de resolver; o importante é tentar fazer escolhas que otimizem o uso dos recursos computacionais

Outra característica da utilização de funções é que elas “quebram” a linearidade de execução, pois a execução pode “dar saltos” quando uma função é invocada (SOFFNER, 2013).

Para entender melhor como funciona esse mecanismo, vamos criar uma função que solicita um número para o usuário, calcula o quadrado desse número e retorna o resultado. Veja no Código 3.3 essa solução. Todo programa sempre começa pela função *main()*, ou seja, esse programa começará a ser executado na linha 10. Na linha 12 a função *calcular()* é chamada, ou seja, a execução “pula” para a linha 3. Na função *calcular()* é solicitado um valor ao usuário (linha 5), armazenado em uma variável (linha 6) e retornado o valor multiplicado por ele mesmo (linha 7). Após retornar o valor, a execução do programa “volta” para a linha 10, pois nesse ponto a função foi chamada. O resultado da função é armazenado na variável *resultado* e impresso na linha 13.

Código 3.3 | Função para calcular o quadrado de um número

```c
#include<stdio.h>

float calcular() {
    float num;
    printf("\nDigite um número: ");
    scanf("%f", &num);
    return num * num;
}

int main(){
    float resultado = 0;
    resultado = calcular();
    printf("\nO quadrado do número digitado é %.2f ",
resultado);
    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.3 utilizando a ferramenta Paiza.io.

> **REFLITA**
>
> A utilização de funções permite que a execução de um programa “pule” entre as linhas, ou seja, que a execução não aconteça de modo sequencial da primeira até a última linha. Essa característica pode deixar a execução mais lenta? Esses “saltos” podem, em algum momento, causar um erro de execução? Os valores das variáveis podem se perder ou se sobrepor?

O USO DE FUNÇÕES COM PONTEIROS NA LINGUAGEM C

Você viu que uma função pode retornar um número inteiro, um real e um caractere; mas e um vetor? Será possível retornar? A resposta é sim! Para isso, devemos utilizar **ponteiros** (ou apontador, como alguns programadores o chamam). Não é possível criar funções como *int[10]*

Ver anotações 0

calcular(), onde *int[10]* quer dizer que a função retorna um vetor com 10 posições. A única forma de retornar um vetor é por meio de um ponteiro (MANZANO, 2015).

Um ponteiro é um tipo especial de variável, que armazena um endereço de memória, podemos utilizar esse recurso para retornar o endereço de um vetor; assim, a função que "chamou" terá acesso ao vetor que foi calculado dentro da função (MANZANO; MATOS; LOURENÇO, 2015).

Para exemplificar o uso desse recurso vamos implementar uma função, que cria um vetor de dez posições e os preenche com valores aleatórios, imprime os valores e posteriormente passa esse vetor para "quem" chamar a função. Veja no Código 3.4 a implementação dessa função. O programa começa sua execução pela linha 15, na função *main()*. Na linha 17 é criado um ponteiro do tipo inteiro, ou seja, este deverá apontar para um local que tenha número inteiro. Na linha 18 é criada uma variável para controle do laço de repetição. Na linha 20 a função *gerarRandomico()* é invocada; nesse momento a execução "pula" para a linha 3, a qual indica que a função retornará um endereço para valores inteiros (*int**). Na linha 5 é criado um vetor de números inteiros com 10 posições e este é estático (*static*), ou seja, o valor desse vetor não será alterado entre diferentes chamadas dessa função. Na linha 8 é criada uma estrutura de repetição para percorrer as 10 posições do vetor. Na linha 9, para cada posição do vetor é gerado um valor aleatório por meio da função *rand() % 100* (gera valores entre 0 e 99), e na linha 10 o valor gerado é impresso para que possamos comparar posteriormente. Na linha 12 a função retorna o vetor, agora preenchido, por meio do comando *return r*; com isso, a execução volta para a linha 20, armazenando o endereço obtido pela função no ponteiro *p*. Na linha 22 é criada uma estrutura de repetição para percorrer o vetor.

Código 3.4 | Função que retorna vetor

```c
#include<stdio.h>

int* gerarRandomico() {

    static int r[10];
    int a;

    for(a = 0; a < 10; a++) {
        r[a] = rand() % 100;
        printf("\nr[%d] = %d", a, r[a]);
    }
    return r;
}

int main(){

   int *p;
   int i;

   p = gerarRandomico();

   for ( i = 0; i < 10; i++ ) {
      printf("\np[%d] = %d", i, p[i]);
   }
   return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.4 utilizando a ferramenta Paiza.io.

O resultado do Código 3.4 pode ser conferido na Figura 3.2.

Figura 3.2 | Resultado do Código 3.4

```
r[0] = 83
r[1] = 86
r[2] = 77
r[3] = 15
r[4] = 93
r[5] = 35
r[6] = 86
r[7] = 92
r[8] = 49
r[9] = 21
```

```
p[0] = 83
p[1] = 86
p[2] = 77
p[3] = 15
p[4] = 93
p[5] = 35
p[6] = 86
p[7] = 92
p[8] = 49
p[9] = 21
```

Fonte: elaborada pela autora.

Outro uso importante de ponteiros com funções é na **alocação de memória dinâmica**. A função *malloc()*, pertencente à biblioteca <*stdio.h*>, é usada para alocar memória dinamicamente. Entender o tipo de retorno dessa função é muito importante, principalmente para seu avanço, quando você começar a estudar estruturas de dados. Veja esse comando:

```
int* memoria = (int*) malloc(sizeof(int));
```

A função *malloc()* pode retornar dois valores: NULL ou um ponteiro genérico (ponteiro genérico é do tipo *void**) (MANZANO, 2015). Quando houver um problema na tentativa de alocar memória, a função retornará NULL, e quando tudo der certo, a função retornará o ponteiro genérico para a primeira posição de memória alocada (SOFFNER, 2013). Nesse caso é preciso converter esse ponteiro genérico para um tipo específico, de acordo com o tipo de dado que será guardado nesse espaço reservado. Para isso é preciso fazer uma conversão, chamada de *cast*, que consiste em colocar entre parênteses, antes da chamada da função, o tipo de valor para o qual se deseja converter.

O número passado entre parênteses equivale à quantidade de bytes a serem alocados pelo computador. Neste caso, o operador `sizeof` foi usado para calcular a quantidade de bytes necessários para se alocar um número inteiro. Usar o operador `sizeof` é uma boa prática, pois

libera o programador da responsabilidade de determinar a quantidade de memória necessária para cada tipo de dado, na arquitetura específica em que o programa executará.

Agora que vimos o tipo de retorno da função *malloc()*, vamos entender como usar essa função dentro de uma função criada por nós.

Veja no Código 3.5: a função *alocar()* foi criada da linha 4 até 7 e seu tipo de retorno foi especificado como *int**; isso significa que o espaço será alocado para guardar valores inteiros. Veja que na linha 8 foi criado um ponteiro inteiro, chamado *num*, e a função *alocar()* foi chamada, tendo seu resultado armazenado no ponteiro *num*. Em seguida, usamos uma estrutura condicional para saber se alocação foi feita com êxito. Caso o valor do ponteiro *memoria* seja diferente de NULL (linha 10), então sabemos que a alocação foi feita com sucesso, pedimos que o usuário informe um número inteiro e imprimimos seu resultado na tela.

Código 3.5 | Função para alocar memória dinamicamente

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int* alocar() {
5      int *memoria = (int*) malloc(sizeof(int));
6      return memoria;
7  }
8  int main(){
9     int *num = alocar();
10    if (num != NULL){
11         printf("\nInforme um número inteiro: ");
12         scanf("%d", num);
13         printf("\nNúmero informado: %d", *num);
14    }
15
16    return 0;
17 }
```

Ver anotações

Fonte: elaborado pela autora.

Nesta seção você aprendeu a criar funções que, após um determinado conjunto de instruções, retorna um valor para “quem” chamou a sub-rotina. Esse conhecimento permitirá a você criar programas mais organizados, além de evitar repetição de códigos.

FAÇA VALER A PENA

Questão 1

Dado um certo problema para ser resolvido por meio de um programa, a solução pode ser implementada em blocos de funcionalidades, técnica essa conhecida como *dividir para conquistar*. A aplicação dessa técnica em uma linguagem de programação pode ser feita por meio de funções ou procedimentos

A respeito de funções e procedimentos, analise as afirmativas a seguir:

I. Funções e procedimentos têm o mesmo objetivo, ou seja, resolver parte de um problema maior. Ambas as técnicas farão o processamento de uma funcionalidade e retornarão um valor para “quem” chamou a sub-rotina.

II. Em uma função criada para retornar um valor inteiro, o comando *return* não pode retornar outro tipo de valor.

III. Uma função pode ser invocada quantas vezes for necessário em um programa.

É correto o que se afirma apenas em:

a. I.

b. II.

c. III.

d. I e II.

e. II e III.

Questão 2

Funções são usadas para organizar o código, evitando a repetição de linhas de comandos. Uma boa prática de programação é avaliar se um determinado trecho precisa ser escrito mais de uma vez. Se a resposta for sim, então esse trecho deve ser transformado em uma sub-rotina.

Avalie o código a seguir.

```c
#include<stdio.h>

int somar(){
  return 2 + 3.23;
}
int main(){
  int resultado = 0;
  resultado = somar();
  printf("\nO resultado da função é = %d", resultado);
  return 0;
}
```

Com base no contexto apresentado, é correto afirmar que:

a. Será impresso na tela *"O resultado da função é = 5.23"*.

b. Será impresso na tela *"O resultado da função é = 2"*.

c. Será impresso na tela *"O resultado da função é = 3"*.

d. Será impresso na tela *"O resultado da função é = 5"*.

e. Será dado um erro de execução, pois a função espera retornar um *int*, e está sendo retornado um número real.

Questão 3

Vetores são estruturas de dados estáticas, ou seja, não são redimensionadas em tempo de execução. Uma vez criadas com tamanho N, esse tamanho se mantém fixo. Para criar uma função que retorna um vetor é preciso recorrer ao uso de ponteiros.

```c
#include<stdio.h>
#include<stdlib.h>

int* retornarVetor(){
  int* v = (int*) malloc(sizeof(int) * 10);
  int a;
  for(a = 0; a < 10; ++a) {
    v[a] = 2 * a;
  }
  return v;
}

int main(){
  int *p;
  p = retornarVetor();
  printf("\nValor = %d", p[2]);
  return 0;
}
```

Com base no contexto apresentado, assinale a alternativa correta.

a. Será impresso na tela “Valor = 0”.

b. Será impresso na tela “Valor = 2”.

c. Será impresso na tela “Valor = 4”.

d. Será impresso na tela “Valor = 6”.

e. Será impresso na tela “Valor = 8”.

REFERÊNCIAS

[C] AULA 60 - Alocação Dinâmica - Parte 1 – Introdução. [*S. l.*], 5 nov. 2012. 1 vídeo (5 min. 25 s.). Publicado pelo canal Linguagem C Programação Descomplicada. Disponível em: https://goo.gl/ps7VPa. Acesso em: 30 jun. 2016.

ALMEIDA, R.; ZANLORENSSI, G. A evolução do número de linhas de telefone fixo e celular no mundo. **Nexo**, 2 maio 2018. Disponível em: https://cutt.ly/5jSOpbP. Acesso em: 7 jul. 2018.

COUNTER, L. **Lines of code of the Linux Kernel Versions**. Disponível em: https://cutt.ly/zjSOt0G. Acesso em: 30 jun. 2018.

FEOFILOFF, P. **Recursão e algoritmos recursivos**. Instituto de Matemática e Estatística da Universidade de São Paulo, 21 maio 2018. Disponível em: https://cutt.ly/OjSOwOH. Acesso em: 2 jan. 2018.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. São Paulo: Érica, 2015.

OLIVEIRA, R. **Algoritmos e programação de computadores**. Notas de aula. Instituto de Computação da Universidade Estadual de Campinas – UNICAMP, [s. d.]. Disponível em: https://cutt.ly/mjSI75P. Acesso em: 16 jul. 2018.

PIRES, A. A. **Cálculo numérico**: prática com algoritmos e planilhas. São Paulo: Atlas, 2015.

SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.

STACK EXCHANGE INC. **Problema no FGTES no C [duplicada]**. Stack Overflow, c2021. Disponível em: https://cutt.ly/PjSI38E. Acesso em: 10 jan. 2021.

FOCO NO MERCADO DE TRABALHO

PROCEDIMENTOS E FUNÇÕES

Vanessa Cadan Scheffer

FUNÇÃO PARA CALCULAR IMC (ÍNDICE DE MASSA CORPORAL)

Criação da função que calcula o IMC (Índice de Massa Corporal) através de uma fórmula com base no peso e altura do indivíduo e uso de estruturas condicionais.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

Você foi contratado por um laboratório de pesquisas multidisciplinar que atende a várias empresas. Seu primeiro trabalho é desenvolver uma solução para a equipe de nutrição. Um cliente deseja uma calculadora de IMC (Índice de Massa Corporal). Os nutricionistas lhe passaram a fórmula para calcular o IMC com base no peso e na altura do indivíduo.

Para resolver esse problema você deve ter a fórmula para o cálculo do IMC anotada: **IMC = Peso / Altura²**. Também será preciso criar condicionais para verificar a situação do indivíduo (abaixo do peso, peso ideal ou sobrepeso), com base em seu IMC.

Eis algumas dicas para resolver o problema:

1. Crie duas variáveis para armazenar valores reais, pois o peso e a altura certamente não serão inteiros.
2. Leia o valor dessas variáveis dentro da função que calcula o IMC.
3. Retorne o valor do IMC.
4. Na função principal, chame a função que calcula o IMC e guarde seu retorno em uma variável inteira.
5. Use uma estrutura condicional (if-else if-else) para identificar e imprimir na tela a situação do indivíduo.

Eis uma possível solução para a situação problema:

Código 3.6 | Cálculo do IMC

```c
#include<stdio.h>

float calcularIMC() {
    float peso, altura, imc;

    printf("\nInforme seu peso (em Kg): ");
    scanf("%f", &peso);
    printf("\nInforme sua altura (em metros): ");
    scanf("%f", &altura);

    imc = peso / (altura * altura);

    return imc;
}

int main(){
    float imc = calcularIMC();

    if (imc < 18.5) printf("\nIMC = %.2f, Situação: Abaixo do peso!", imc);
    else if (imc < 24.9) printf("\nIMC = %.2f, Situação: Peso ideal!", imc);
    else  printf("\nIMC = %.2f, Situação: Sobrepeso!", imc);
}
```

Fonte: elaborado pela autora.

Agora, você pode testar o código utilizando a ferramenta Paiza.io.

AVANÇANDO NA PRÁTICA

FUNÇÃO PARA GERAR SENHAS

Uma empresa está ingressando no mercado de segurança de dados e você foi contratado para implementar um algoritmo que cria senhas automaticamente para os clientes. A ideia é pedir para o usuário digitar

uma frase e, a partir das letras na frase, o algoritmo gerará uma senha aleatória.

RESOLUÇÃO

O programa para geração de senhas pode ser construído utilizando uma função; além disso, a função *rand()* também pode ser usada para escolher, aleatoriamente, as letras da frase que serão usadas na senha. Veja no Código 3.7 uma possível solução para o problema.

Código 3.7 | Função para gerar senhas aleatoriamente

```c
#include<stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAM_FRASE 100
#define TAM_SENHA 6

char* gerarSenha() {
    int maxTamFrase = TAM_FRASE + 1; // precisa considerar o caractere de encerramento da string
    int tamSenha = TAM_SENHA + 1; // precisa considerar o caractere de encerramento da string

    char strFrase[maxTamFrase];
    char* strSenha = (char*) malloc(sizeof(char) * tamSenha);

    printf("\nEntre uma frase sem espaços em branco entre as palavras: ");
    fflush(stdin);
    fgets(strFrase, maxTamFrase, stdin);

    int tamFrase = strlen(strFrase); // quantidade de caracteres da frase informada
    srand(time(NULL)); // usa uma nova semente para o gerador de números aleatórios (rand)

    for(int i = 0; i < TAM_SENHA; i++) {
        int nAleatorio = rand() % tamFrase;
        strSenha[i] = strFrase[nAleatorio];
    }
    strSenha[tamSenha] = '\0';

    return strSenha;
}

int main(){
```

Ver anotações

```
32      char* senha = gerarSenha();
33      printf("\nSenha gerada: %s", senha);
34  }
```

Fonte: elaborado pela autora.

Agora, você pode testar o código utilizando a ferramenta Paiza.io.

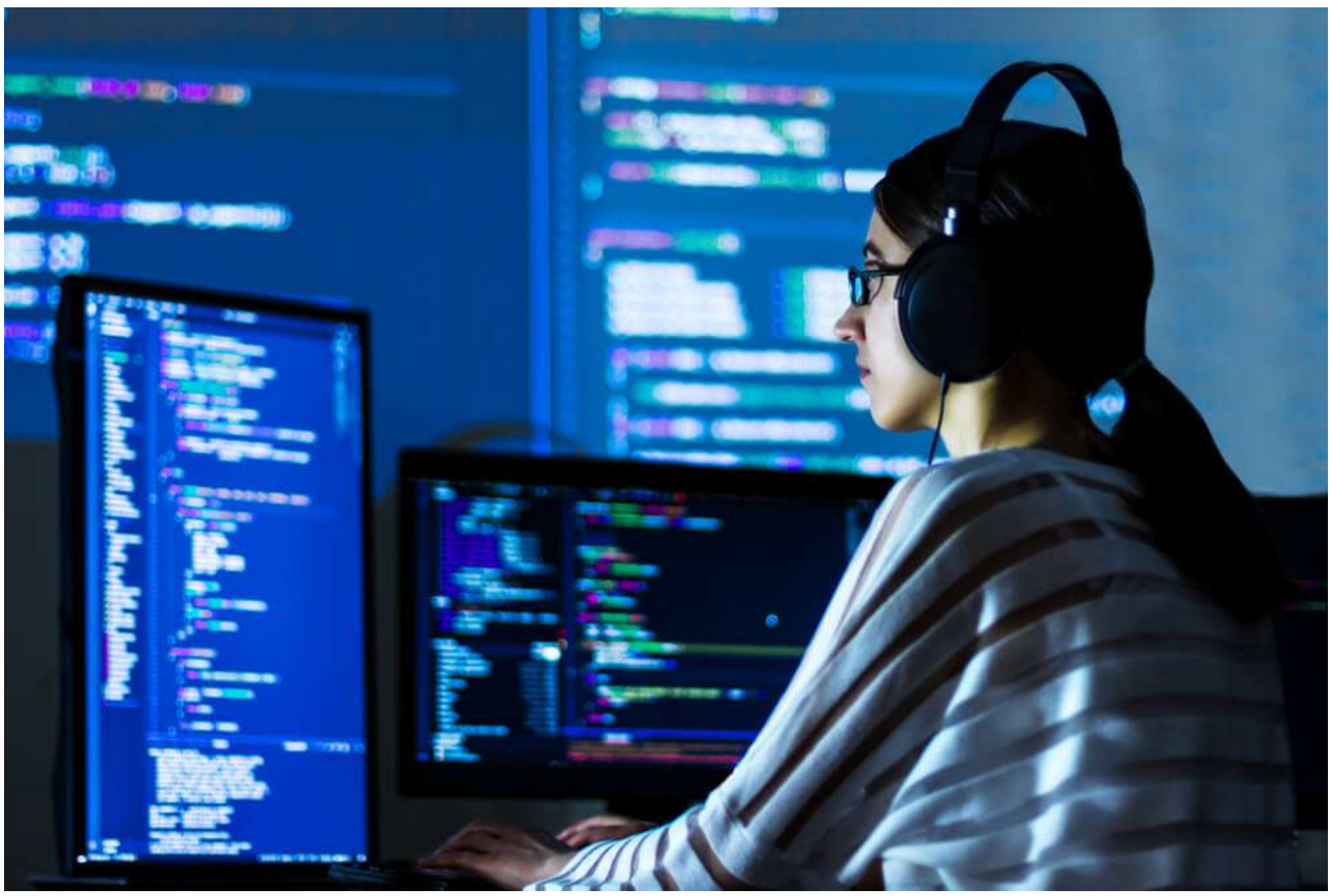
NÃO PODE FALTAR

ESCOPO E PASSAGEM DE PARÂMETROS

Vanessa Cadan Scheffer

QUAL A DIFERENÇA ENTRE VARIÁVEIS LOCAIS E GLOBAIS?

A diferença entre elas é o local em que são definidas, que determina seu escopo e visibilidade dentro do programa.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Dando continuidade ao seu trabalho no laboratório de pesquisa multidisciplinar, após construir com êxito o programa para a equipe de nutrição, agora você trabalhará com o núcleo de estatística. Você precisa criar funções que facilitem alguns dos cálculos mais comuns usados por estatísticos: a média e a mediana. Com base nessas funções tão importantes, podemos, por exemplo, calcular a média e a mediana dos IMCs de uma determinada população. A ideia é que, a partir de um conjunto de números reais (vetor), seu programa calcule a média e a mediana desses conjuntos.

> Relembrando: a média é o resultado da soma de todos os números do conjunto dividida pela quantidade de números que foram somados. Já a mediana será o número que ocupa a posição central do conjunto, considerando que o conjunto esteja crescentemente ordenado.

Para o cálculo da mediana, há uma particularidade que deve ser observada. Se a quantidade de números do conjunto for ímpar, a mediana será o elemento do meio do vetor. Por exemplo, para o vetor [21, 24, 31, 32, 32, **32**, 33, 44, 65], a mediana será o número 32, que está bem no meio do vetor. Porém, se a quantidade de número do conjunto for par, então a mediana será a média aritmética dos dois valores centrais. Por exemplo, para o vetor [18, 19, 19, 22, **44**, **45**, 46, 46, 47, 48], a mediana será (44 + 45) / 2 ➔ 89 /2, que é igual a 44,5.

Além do cálculo, seu programa deverá exibir os resultados na tela.

Com esse conhecimento seus programas ficarão mais organizados e otimizados. Vamos lá!

CONCEITO-CHAVE

Caro aluno, você sabe quantos números de linhas telefônicas ativas existem no mundo hoje? Segundo Almeida e Zanlorenssi (2018), o número de linhas ativas ultrapassa a marca de 8 bilhões. No Brasil, um

número de telefone fixo é dividido em três partes: código da região, código da cidade e o número. Imagine um usuário desatento que deseja ligar para uma cidade diferente da que ele reside e ele digita 3441-1111. Essa ligação iria para qual cidade? Existem várias cidades com o código 3441 e o que as diferencia é o código da região, delimitando o escopo daquele número. O mesmo acontece com variáveis em um programa: o seu escopo definirá o limite de utilização desse recurso.

ESCOPO DE VARIÁVEIS

Variáveis são usadas para armazenar dados temporariamente na memória, porém o local em que esse recurso é definido no código de um programa determina seu escopo e sua visibilidade. Observe no Código 3.8.

Código 3.8 | Exemplo de variáveis em funções

```
#include<stdio.h>

int testar(){
    int x = 10;
    return x;
}
int main(){
    int x = 20;
    printf("\n Valor de x na função main() = %d", x);
    printf("\n Valor de x na função testar() = %d",
testar());

    return 0;
}
```

Fonte: elaborado pela autora.

Na implementação do Código 3.8 temos duas variáveis chamadas “x”. Isso acarretará algum erro? A resposta, nesse caso, é não, pois mesmo as variáveis tendo o mesmo nome, elas são definidas em lugares diferentes: uma está dentro da função *main()* e outra dentro da *testar()*, e cada função terá seu espaço na memória de forma independente. Na memória, as variáveis são localizadas pelo seu endereço, portanto, mesmo sendo declaradas com o mesmo nome, seus endereços são distintos.

A partir desse exemplo, pode-se definir o escopo de uma variável como “a relação de alcance que se tem com o local onde certo recurso se encontra definido, de modo que possa ser ‘enxergado’ pelas várias partes do código de um programa” (MANZANO, 2015, p. 288). O escopo é dividido em duas categorias, local ou global (MANZANO; MATOS; LOURENÇO, 2015).

VARIÁVEIS LOCAL E GLOBAL

No exemplo do Código 3.8, ambas variáveis são **locais**, ou seja, elas existem e podem ser vistas somente dentro do corpo da função onde foram definidas. Para definir uma variável **global**, é preciso criá-la fora de qualquer função, assim ela será visível por todas as funções do programa. No contexto apresentado nessa seção, elas serão criadas logo após a inclusão das bibliotecas.

No Código 3.9 é apresentado um exemplo de declaração de uma variável global, na linha 3, logo após a inclusão da biblioteca de entrada e saída padrão. Veja que na função principal não foi definida qualquer variável com o nome de “x” e mesmo assim seu valor pode ser impresso na linha 10, pois é acessado o valor da variável global. Já na linha 12 é impresso o valor da variável global modificado pela função *testar()*, que retorna o dobro do valor.

Código 3.9 | Exemplo de variável global

```c
#include<stdio.h>

int x = 10;

void testar(){
    x = 2 * x;
}

int main(){
    printf("\nValor de x global = %d", x);
    testar();
    printf("\nValor de x global alterado em testar() = %d",
x);

    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.9 utilizando a ferramenta Paiza.io.

> **ASSIMILE**
>
> A utilização de variáveis globais permite otimizar a alocação de memória, pois em vários casos o desenvolvedor não precisará criar variáveis locais. Por outro lado, essa técnica de programação deve ser usada com cautela, pois variáveis locais são criadas e destruídas ao fim da função, enquanto as globais permanecem na memória durante todo o tempo de execução.

A seguir será apresentado um exemplo da utilização do escopo global de uma variável. Vamos criar um programa que calcula a média entre duas temperaturas distintas. Observe no Código 3.10, na linha 3, em que foram declaradas duas variáveis. É importante destacar que o programa sempre começa pela função principal e a execução inicia na

linha 8. Na linha 9, é solicitado ao usuário digitar duas temperaturas (em que é pedido conforme a linha 10), as quais são armazenadas dentro das variáveis globais criadas. Na linha 11, a função *calcularMedia()* é invocada para fazer o cálculo da média usando os valores das variáveis globais. Nesse exemplo, fica clara a utilidade dessa técnica de programação, pois as variáveis são usadas em diferentes funções, otimizando o uso da memória, pois não foi preciso criar mais variáveis locais.

Código 3.10 | Calcular média de temperatura com variável global

```
1  #include<stdio.h>
2
3  float t1, t2;
4
5  float calcularMedia(){
6      return (t1 + t2) / 2;
7  }
8  int main() {
9      printf("\nDigite as duas temperaturas: ");
10     scanf("%f %f", &t1, &t2);
11     printf("\nA temperatura média é %.2f", calcularMedia());
12
13     return 0;
14 }
```

Fonte: elaborado pela autora.

Teste o Código 3.10 utilizando a ferramenta Paiza.io.

EXEMPLIFICANDO

É possível criar variáveis com mesmo nome em diferentes funções, pois o escopo delas é local. Entretanto, se existir uma variável global e uma local com mesmo nome, por exemplo:

Código 3.11 | Variáveis com mesmo nome

```
1  #include<stdio.h>
2  int x = 10;
3  int main(){
4    int x = -1;
5    printf("\nValor de x = %d", x);
6  }
```

Fonte: elaborado pela autora.

Qual valor será impresso na variável x? A variável local sempre sobrescreverá o valor da global, portanto, nesse caso, será impresso o valor -1 na função principal.

Teste o código utilizando a ferramenta Paiza.io.

Na linguagem C, para conseguirmos acessar o valor de uma variável global, dentro de uma função que apresenta uma variável local com mesmo nome, devemos usar a instrução ***extern*** (MANZANO, 2015). No Código 3.12, veja como utilizar variáveis globais e locais com mesmo nome na linguagem C. Observe que foi necessário criar uma variável chamada “b”, com um bloco de instruções (linhas 8 – 11), que atribui à nova variável o valor “externo” de x.

Código 3.12 | Variável global e local com mesmo nome

```c
#include<stdio.h>

int x = 10;

int main(){
  int x = -1;
  int b;
  {
    extern int x;
    b = x;
  }
  printf("\n Valor de x = %d",x);
  printf("\n Valor de b (x global) = %d",b);
  return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.12 utilizando a ferramenta Paiza.io.

PASSAGEM DE PARÂMETROS PARA FUNÇÕES

Para criar funções usamos a seguinte sintaxe:

```
<tipo de retorno> <nome> (<parâmetros>) {
   <Comandos da função>
   <Retorno> (não obrigatório)
}
```

Com exceção dos parâmetros que uma função pode receber, todos os demais já foram apresentados, portanto, nos dedicaremos a entender esse importante recurso.

Ao definir uma função, podemos também estabelecer que ela receberá informações “de quem” a invocou. Por exemplo, ao criar uma função que calcula a média, podemos definir que “quem chamar” deve informar os valores sobre os quais o cálculo será efetuado.

PASSAGEM POR VALOR

Na sintaxe, para criar uma função que recebe parâmetros é preciso especificar qual o tipo de valor que será recebido. Uma função pode receber parâmetros na forma de valor ou de referência (SOFFNER, 2013). Na passagem parâmetros por valores, a função cria variáveis locais automaticamente para armazenar esses valores e, após a execução da função, essas variáveis são liberadas. Veja, no Código 3.13, um exemplo de definição e chamada de função com passagem de valores. Observe que, na linha 2, a função *somar()* foi definida para receber dois valores inteiros. Internamente serão criadas as variáveis "a" e "b" locais, para guardar **uma cópia** desses valores até o final da função. Na linha 8, a função *somar()* foi invocada, passando os dois valores inteiros que a função espera receber, e o resultado do cálculo será guardado na variável "result".

Código 3.13 | Chamada de função com passagem de valores

```
1  #include<stdio.h>
2  int somar(int a, int b){
3      return a + b;
4  }
5
6  int main(){
7      int result;
8      result = somar(10, 15);
9      printf("\nResultado da soma = %d", result);
10
11     return 0;
12 }
```

Fonte: elaborado pela autora.

Teste o Código 3.13 utilizando a ferramenta Paiza.io.

ASSIMILE

A técnica de passagem de parâmetros para uma função é extremamente usada em todas as linguagens de programação. Ao chamar uma função que espera receber parâmetros, caso os argumentos esperados não sejam informados, será gerado um erro de compilação.

Ao utilizar variáveis como argumentos na passagem de parâmetros por valores, essas variáveis não são alteradas, pois é fornecida uma cópia dos valores armazenados para a função (SOFFNER, 2013).

Para ficar clara essa importante definição, veja no Código 3.14. A execução do programa começa na linha 10, pela função principal, na qual são criadas duas variáveis “n1” e “n2”. Na linha 13, o comando determina a impressão dos valores das variáveis, na linha 16, a função *testar()* é invocada passando como parâmetro as duas variáveis. Nesse momento, é criada uma cópia de cada variável na memória para utilização da função. Veja que dentro da função o valor das variáveis é alterado e impresso, mas essa alteração é local, ou seja, é feita na cópia dos valores e não afetará o valor inicial das variáveis criadas na função principal. Na linha 19, imprimimos novamente os valores após a chamada da função. A Figura 3.3 apresenta o resultado desse programa.

Código 3.14 | Variáveis em chamada de função com passagem de valores

```c
#include<stdio.h>

void testar(int n1, int n2){
    n1 = -1;
    n2 = -2;
    printf("\n\nValores dentro da função testar(): ");
    printf("\nn1 = %d e n2 = %d", n1, n2);
}

int main(){
    int n1 = 10;
    int n2 = 20;
    printf("\nValores antes de chamar a função testar(): ");
    printf("\nn1 = %d e n2 = %d", n1, n2);

    testar(n1, n2);

    printf("\n\nValores depois de chamar a função testar():
");
    printf("\nn1 = %d e n2 = %d", n1, n2);

    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.14 utilizando a ferramenta Paiza.io.

Figura 3.3 | Resultado do Código 3.14

```
Valores antes de chamar a função testar():
n1 = 10 e n2 = 20

Valores dentro da função testar():
n1 = -1 e n2 = -2

Valores depois de chamar a função testar():
n1 = 10 e n2 = 20
```

Fonte: elaborada pela autora.

PASSAGEM POR REFERÊNCIA

A utilização de passagem de parâmetros por referência está diretamente ligada aos conceitos de ponteiro e endereço de memória. A ideia da técnica é análoga à passagem por valores, ou seja, a função será definida de modo a receber certos parâmetros e “quem” faz a chamada do método deve informar esses argumentos. Entretanto, o comportamento e o resultado são diferentes.

Na passagem por referência não será criada uma cópia dos argumentos passados; na verdade, será passado o endereço da variável e a função trabalhará diretamente com os valores ali armazenados (SOFFNER, 2013).

Como a função utiliza o endereço (ponteiros), na sintaxe, serão usados os operadores * e & (MANZANO, 2015). Na definição da função os parâmetros a serem recebidos devem ser declarados com *, por exemplo: *int testar(int* parametro1, int* parametro2)*. Na chamada da função, os parâmetros devem ser passados com o &, por exemplo: *resultado = testar(&n1, &n2)*. Veja no Código 3.15 um exemplo de uma função que passa variáveis como referência. Observe que a única diferença desse código para o do Código 3.14 é a utilização dos operadores * e &, porém, o resultado é completamente diferente (Figura 3.4). Com a passagem por referência os valores das variáveis são alterados. Nas linhas 4 e 5, usamos asterisco para acessar o conteúdo guardado dentro do endereço que ele aponta, pois se usássemos somente n1 e n2 acessaríamos o endereço que eles apontam.

Código 3.15 | Variáveis em chamada de função com passagem de referência

```c
#include<stdio.h>

void testar(int* n1, int* n2){
    *n1 = -1;
    *n2 = -2;
    printf("\n\nValores dentro da função testar(): ");
    printf("\nn1 = %d e n2 = %d", *n1, *n2);
}

int main(){
    int n1 = 10;
    int n2 = 20;
    printf("\nValores antes de chamar a função testar(): ");
    printf("\nn1 = %d e n2 = %d", n1, n2);

    testar(&n1, &n2);

    printf("\n\nValores depois de chamar a função testar():
");
    printf("\nn1 = %d e n2 = %d", n1, n2);

    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.15 utilizando a ferramenta Paiza.io.

Figura 3.4 | Resultado do Código 3.15

```
Valores antes de chamar a função testar():
n1 = 10 e n2 = 20

Valores dentro da função testar():
n1 = -1 e n2 = -2

Valores depois de chamar a função testar():
n1 = -1 e n2 = -2
```

Fonte: elaborada pela autora.

PASSAGEM DE VETOR

Para finalizar esta seção, vamos ver como passar um vetor para uma função. Esse recurso pode ser utilizado para criar funções que preenchem e imprimem o conteúdo armazenado em um vetor, evitando a repetição de trechos de código. Na definição da função, os parâmetros a serem recebidos devem ser declarados com colchetes sem especificar o tamanho, por exemplo: *int testar(int v1[], int v2[])*. Na chamada da função, os parâmetros devem ser passados como se fossem variáveis simples, por exemplo: *resultado = testar(n1,n2).*

Na linguagem C, a passagem de um vetor é feita implicitamente por referência. Isso significa que mesmo não utilizando os operadores “*” e “&”, quando uma função que recebe um vetor é invocada, o que é realmente passado é o endereço da primeira posição do vetor.

Para entender esse conceito, vamos criar um programa que, por meio de uma função, preencha um vetor de três posições e em outra função percorra o vetor imprimindo o dobro de cada valor do vetor. Veja, no Código 3.16, o código que começa a ser executado pela linha 17. Na linha 20, a função inserir() é invocada, passando como parâmetro o vetor “numeros”. Propositalmente criamos a função na linha 2, recebendo como argumento um vetor de nome “a”, para que você entenda que o nome da variável que a função recebe não precisa ser o mesmo nome usado na chamada. Na maioria das vezes, utilizamos o mesmo nome como boa prática de programação. Na função *inserir()*, será solicitado ao usuário digitar os valores que serão armazenados no vetor “*numeros*”, pois foi passado como referência implicitamente. Após essa função finalizar, a execução vai para a linha 21 e depois para a 22, na qual é invocada a função *imprimir()*, novamente passando o vetor “numeros” como argumento. Mais uma vez, criamos a função na linha 10 com o nome do vetor diferente para reforçar que não precisam ser os mesmos. Nessa função o vetor é percorrido, imprimindo o dobro de cada valor. Veja na Figura 3.5 o resultado desse programa.

Código 3.16 | Função com passagem de vetor

```c
#include<stdio.h>
void inserir(int a[]) {
    int i = 0;
    for(i = 0; i < 3; i++){
        printf("\nDigite o valor %d: ", i);
        scanf("%d", &a[i]);
    }
}

void imprimir(int b[]) {
    int i = 0;
    for(i = 0; i < 3; i++){
        printf("\nNúmero[%d] = %d", i, 2 * b[i]);
    }
}

int main(){
    int numeros[3];
    printf("\nPreenchendo o vetor... \n ");
    inserir(numeros);
    printf("\n\nDobro dos valores informados:");
    imprimir(numeros);
    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.16 utilizando a ferramenta Paiza.io.

Figura 3.5 | Resultado do Código 3.16.

```
Preenchendo o vetor...
Digite o valor 0:
Digite o valor 1:
Digite o valor 2:

Dobro dos valores informados:
Número[0] = 2
Número[1] = 4
Número[2] = 6
```

Fonte: elaborada pela autora.

Com esta seção demos um passo importante no desenvolvimento de softwares, pois os conceitos apresentados são utilizados em todas as linguagens de programação. Continue estudando e se aprofundando no tema.

FAÇA VALER A PENA

Questão 1

O uso de funções permite criar programas mais organizados, sem repetição de códigos e ainda com possibilidade de reutilização, pois caso você implemente uma função de uso comum, poderá compartilhá-la com outros desenvolvedores. Em linguagens do paradigma orientado a objetos, as funções são chamadas de métodos, mas o princípio de construção e funcionamento é o mesmo.

A respeito das funções, analise cada uma das afirmativas e determine se é verdadeira ou falsa.

I. ( ) Funções que retornam um valor do tipo *float* só podem receber como parâmetros valores do mesmo tipo, ou seja, *float*.

II. ( ) Funções que trabalham com passagem de parâmetros por referência não criam cópias das variáveis recebidas na memória.

III. ( ) Funções que trabalham com passagem de parâmetros por valor criam cópias das variáveis recebidas na memória.

Assinale a alternativa correta.

a. V – V – V.

b. V – F – V.

c. F – V – V.

d. F – F – V.

e. F – V – F.

Questão 2

Além de retornar valores, as funções também podem receber parâmetros de “quem” a chamou. Essa técnica é muito utilizada e pode economizar na criação de variáveis ao longo da função principal. Os tipos de parâmetros que uma função/procedimento pode receber são classificados em passagem por valor e passagem por referência.

Análise o código a seguir.

```
1  #include<stdio.h>
2
3  void pensar(int a, int b) {
4      a = 11;
5      b = 12;
6  }
7  int main() {
8      int a = -11;
9      int b = -12;
10     pensar(a, b);
11     printf("\n a = %d e b = %d", a, b);
12     return 0;
13 }
```

De acordo com o código apresentado, será impresso na linha 11:

a. a = -11 e b = -12.

b. a = 11 e b = 12.

c. a = -11 e b = 12.

d. a = 11 e b = -12.

e. Será dado um erro de compilação.

Ver anotações 0

Questão 3

Uma função pode receber parâmetros por valor ou por referência. No primeiro caso, são criadas cópias das variáveis na memória, e então o valor original não é alterado. Para trabalhar com passagem por referência é preciso recorrer ao uso de ponteiros, pois são variáveis especiais que armazenam endereços de memória.

Análise o código a seguir:

```
1  #include<stdio.h>
2
3  void pensar(int* a, int* b) {
4      *a = 10;
5      *b = 20;
6  }
7  int main(){
8      int a = -30;
9      int b = -40;
10     pensar(a, b);
11     printf("\n a = %d e b = %d", a, b);
12     return 0;
13 }
```

De acordo com o código apresentado, será impresso na linha 11:

a. a = -30 e b = -40.

b. a = 10 e b = 20.

c. a = -30 e b = 20.

d. a = 10 e b = -40.

e. Será dado um erro de compilação.

REFERÊNCIAS

Ver anotações

[C] AULA 60 - Alocação Dinâmica - Parte 1 – Introdução. [*S. l.*], 5 nov. 2012. 1 vídeo (5 min. 25 s.). Publicado pelo canal Linguagem C Programação Descomplicada. Disponível em: https://goo.gl/ps7VPa. Acesso em: 30 jun. 2016.

ALMEIDA, R.; ZANLORENSSI, G. A evolução do número de linhas de telefone fixo e celular no mundo. **Nexo**, 2 maio 2018. Disponível em: https://cutt.ly/5jSOpbP. Acesso em: 7 jul. 2018.

COUNTER, L. **Lines of code of the Linux Kernel Versions**. Disponível em: https://cutt.ly/zjSOt0G. Acesso em: 30 jun. 2018.

CPPREFERENCE. **Implicit conversions**. Cppreference, [s. l.], [s. d.]. Disponível em: https://cutt.ly/8jS1KWD. Acesso em: 10 jan. 2021.

FEOFILOFF, P. **Recursão e algoritmos recursivos**. Instituto de Matemática e Estatística da Universidade de São Paulo, 21 maio 2018. Disponível em: https://cutt.ly/OjSOwOH. Acesso em: 2 jan. 2018.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. São Paulo: Érica, 2015.

OLIVEIRA, R. **Algoritmos e programação de computadores**. Notas de aula. Instituto de Computação da Universidade Estadual de Campinas – UNICAMP, [s. d.]. Disponível em: https://cutt.ly/mjSI75P. Acesso em: 16 jul. 2018.

PIRES, A. A. **Cálculo numérico**: prática com algoritmos e planilhas. São Paulo: Atlas, 2015.

SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.

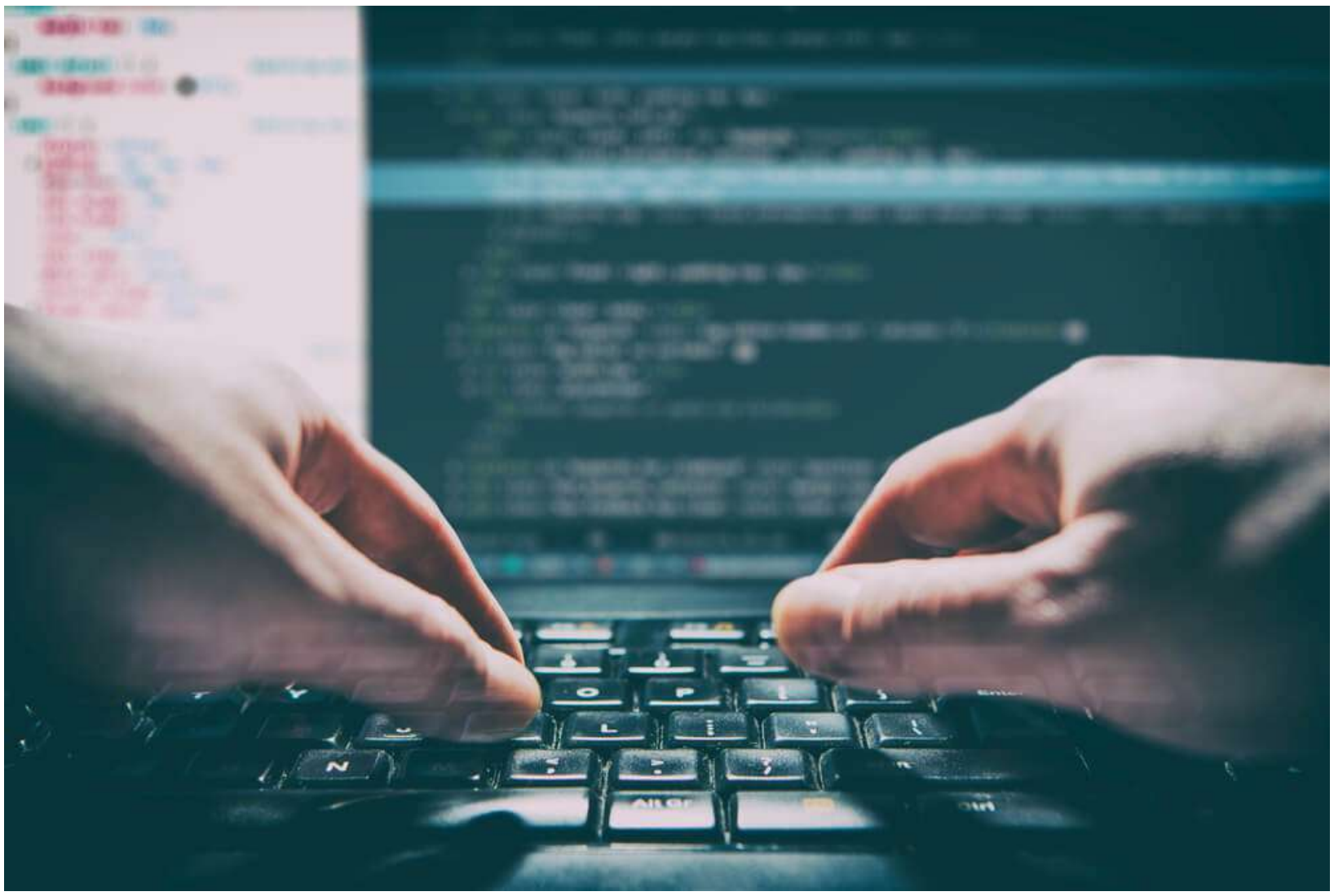
FOCO NO MERCADO DE TRABALHO

ESCOPO E PASSAGEM DE PARÂMETROS

Vanessa Cadan Scheffer

FUNÇÕES PARA AUTOMATIZAR O CÁLCULO DA MÉDIA E DA MEDIANA

Criação de duas funções para calcular a média aritmética e a mediana do conjunto de números passado como parâmetro de um vetor de números reais.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

Você foi contratado por um laboratório de pesquisa que presta serviço para diversas empresas, e agora precisa fazer um programa para a equipe de estatísticos. Foi solicitado a você automatizar o cálculo da média e da mediana e de um conjunto de números reais. Seu programa, além de calcular os resultados, deverá imprimi-los na tela.

Para implementar essa solução você pode criar duas funções que recebem como parâmetro um vetor de números reais, juntamente com seu tamanho (quantidade de elementos).

O primeiro passo é criar as funções que farão os cálculos da média e da mediana. Você pode criá-las da seguinte forma:

- `float calcularMedia(float conjunto[], int tam)`
- `float calcularMediana(float conjunto[], int tam)`

Nessas funções, a média aritmética e a mediana do conjunto de números passado como parâmetro deverão ser calculadas e retornadas como resultado.

Em seguida, é preciso criar a função principal, na qual será declarado e populado o conjunto de dados. Em posse dos valores, as funções `calcularMedia()` e `calcularMediana()` deverão ser invocadas, e os valores, passados. Os resultados dessas funções devem ser guardados dentro de variáveis, da seguinte forma:

- `media = calcularMedia(vet, tam)`
- `mediana = calcularMedia(vet, tam)`

Como último passo, deverão ser impressos os resultados. Para conferir uma opção de implementação do problema, veja o código a seguir:

Código 3.17 | Cálculos da média e da mediana

```c
#include<stdio.h>
#define VET_TAM 6

float calcularMedia(float conjunto[], int tam) {
    float soma = 0.0, resultado = 0.0;
    for(int i = 0; i < tam; i++) {
        soma += conjunto[i];
    }
    resultado = soma / (float) tam;
    return resultado;
}

float calcularMediana(float conjunto[], int tam) {
    float resultado = 0.0;
    if (tam % 2 != 0) { // tam é impar
        resultado = conjunto[tam / 2];
    } else {            // tam é par
        resultado = (conjunto[tam / 2] + conjunto[(tam / 2) - 1]) / 2;
    }
    return resultado;
}

int main(void){
    float vet[VET_TAM] = {1, 2, 3, 4, 5, 6};
    float media = calcularMedia(vet, VET_TAM);
    float mediana = calcularMediana(vet, VET_TAM);
    printf("\nMédia = %.2f", media);
    printf("\nMediana = %.2f", mediana);
}
```

Fonte: elaborado pela autora.

Agora, você pode testar o código utilizando a ferramenta Paiza.io.

AVANÇANDO NA PRÁTICA

Ver anotações

RENDIMENTO DE INVESTIMENTOS

Você foi contratado por uma agência de créditos pessoais para implementar um programa que calcula o total de rendimentos (usando juros simples) que um cliente terá em determinado investimento. O cliente informará o valor que pretende investir, qual o plano e quanto tempo pretende deixar o dinheiro investido. No “plano A”, o rendimento é de 2%, porém, o cliente não pode solicitar o resgate antes de 24 meses. Já no “plano B” o rendimento é de 0,8% e o tempo mínimo para resgate é de 12 meses. Faça um programa que peça as informações para o usuário e, a partir de uma função, calcule o rendimento que o cliente terá.

RESOLUÇÃO

Para implementar a solução, primeiro você deve saber a fórmula de juros simples $j=C.i.t$, na qual j é o juro, C é o capital inicial, i é a taxa e t é o tempo. Veja no Código 3.18 uma possível implementação para o problema.

Código 3.18 | Programa para calcular juros simples

```c
#include<stdio.h>

float calcularInvestimento(float valor, char plano, int meses) {
    if ((plano =='A' || plano =='a') && meses >= 24){
        return valor * 0.02 * meses;
    } else {
        printf("\nDados inválidos!");
        return 0;
    }

    if ((plano =='B' || plano =='b') && meses >= 12){
        return valor * 0.008 * meses;
    } else {
        printf("\nDados inválidos!");
        return 0;
    }
}

int main(){
    float valorInv = 0;
    float rendimento = 0;
    char plano;
    int tempo = 0;

    printf("\nDigite o plano: ");
    scanf("%c", &plano);

    printf("\nDigite o valor a ser investido: ");
    scanf("%f", &valorInv);

    printf("\nDigite o tempo para resgate: ");
    scanf("%d", &tempo);

    rendimento = calcularInvestimento(valorInv, plano, tempo);
    printf("\nSeu rendimento será de %.2f", rendimento);
```

Ver anotações 0

```
36
37      return 0;
38  }
```

Fonte: elaborado pela autora.

Agora, você pode testar o código utilizando a ferramenta Paiza.io.

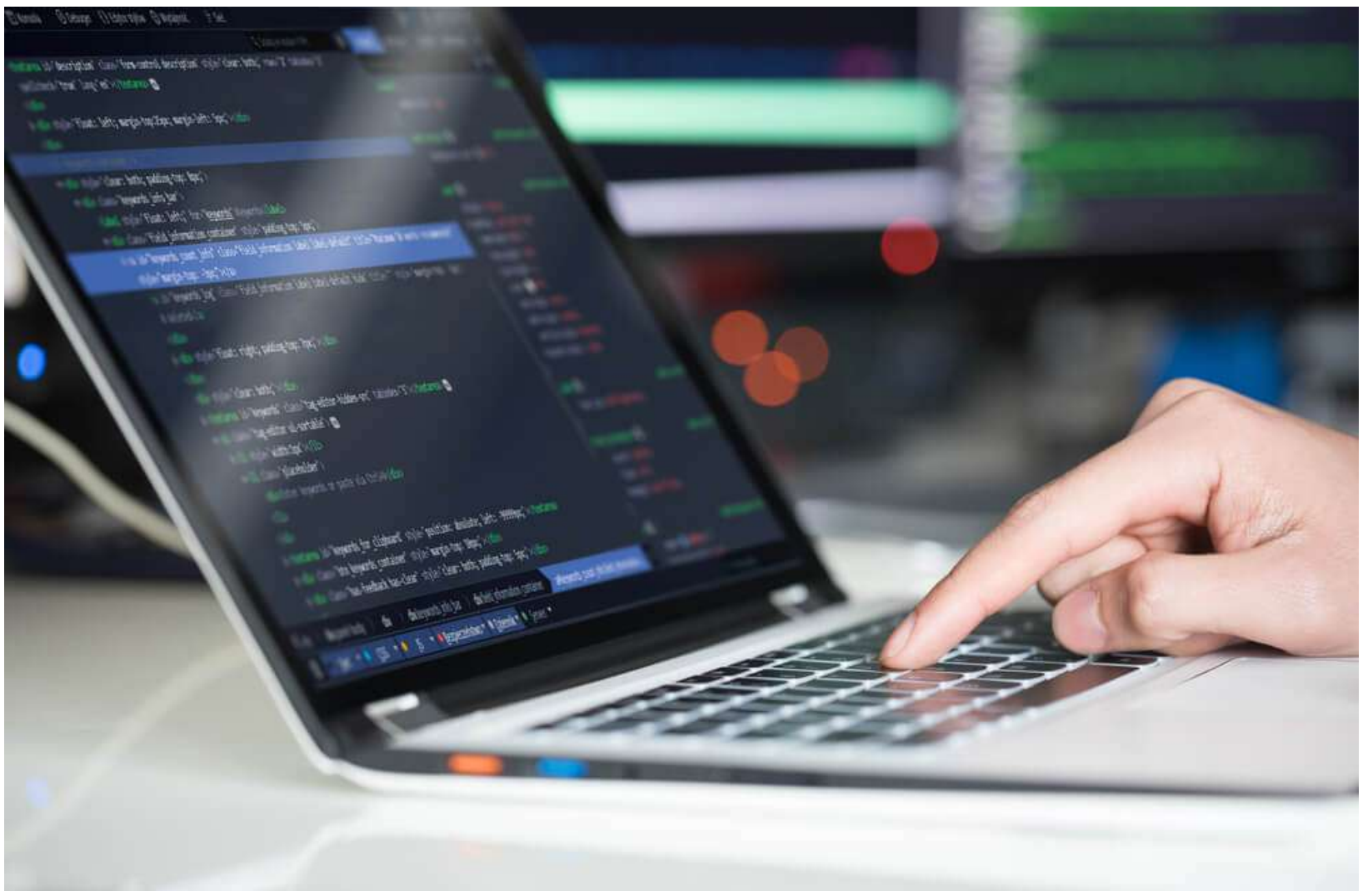
NÃO PODE FALTAR

RECURSIVIDADE

Vanessa Cadan Scheffer

O QUE É FUNÇÃO RECURSIVA?

É uma técnica de programação onde uma função chama a si própria.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Após trabalhar com a equipe de nutricionistas e estatísticos, agora é hora de enfrentar um novo desafio com a equipe de matemáticos do laboratório em que atua. Foi solicitado a você implementar um

programa que calcula se um número é primo ou não, ou seja, se esse número é divisível apenas por 1 e por ele mesmo. Dessa forma, você deverá entregar o arquivo compilado para que eles possam utilizar. Após conhecer sua nova tarefa, seu gerente, que também é programador, pediu para que utilizasse a técnica de recursividade de cauda na implementação, portanto o usuário precisa informar o número que deseja verificar se é primo ou não. Pense um pouco e responda: o que determinará o término do cálculo; em outras palavras, qual será o caso base dessa recursão?

Aproveite mais essa oportunidade de aprimoramento e bons estudos!

CONCEITO-CHAVE

Caro aluno, bem-vindo à última seção no estudo de funções e recursividade. A caminhada foi longa e muito produtiva. Você teve a oportunidade de conhecer diversas técnicas de programação e agora continuaremos avançando no estudo das funções. Você se lembra do algoritmo que calcula o fatorial de um número, o qual vimos na seção sobre estruturas de repetição? Para implementar computacionalmente uma solução desse tipo, você já conhece as estruturas de repetição, como o for, entretanto essa técnica não é a única opção, já que você também pode utilizar a recursividade, assunto central desta seção.

Nesta unidade, apresentamos as funções. Vimos como criar uma função, qual sua importância dentro de uma implementação e estudamos a saída de dados de uma função, bem como a entrada, feita por meio dos parâmetros.

FUNÇÃO RECURSIVA

Entre as funções existe uma categoria especial chamada de **funções recursivas**. Para começarmos a nos apropriar dessa nova técnica de programação, primeiro vamos entender o significado da palavra recursão. Ao consultar diversos dicionários, como o Houaiss ou o Aulete, temos como resultado que a palavra recursão (ou recursividade) está

associada à ideia de recorrência de uma determinada situação. Quando trazemos esse conceito para o universo da programação, deparamo-nos com as funções recursivas.

> Nessa técnica, uma função é dita recursiva quando ela chama a si própria ou, nas palavras de Soffner: “Recursividade é a possibilidade de uma função chamar a si mesma” (SOFFNER, 2013, p. 107).

A sintaxe para implementação de uma função recursiva nada difere das funções gerais, ou seja, deverá ter um tipo de retorno, o nome da função, os parênteses e os parâmetros quando necessário. A diferença estará no corpo da função, pois a função será invocada dentro dela mesma.

Observe a Figura 3.6. Nela ilustramos a construção da função, bem como a chamada dela, primeiro na função principal e depois dentro dela mesma.

Figura 3.6 | Algoritmo para função recursiva

Fonte: elaborada pela autora.

> **ASSIMILE**
>
> Em termos de sintaxe, uma função recursiva difere de outras funções simplesmente pelo fato de apresentar, em seu conjunto de comandos, uma chamada a si própria.

Embora a sintaxe da função recursiva seja similar à das não recursivas, o funcionamento de ambas é bastante distinto e o mau uso dessa técnica pode acarretar uso indevido de memória, muitas vezes chegando a travar a aplicação e o sistema (MANZANO, 2015). Para entender o processo, vamos estabelecer alguns pontos de atenção:

- A função recursiva chama a si própria até que um ponto de parada seja estabelecido. O ponto de parada poderá ser alcançado por meio de uma estrutura condicional ou por meio de um valor informado pelo usuário.

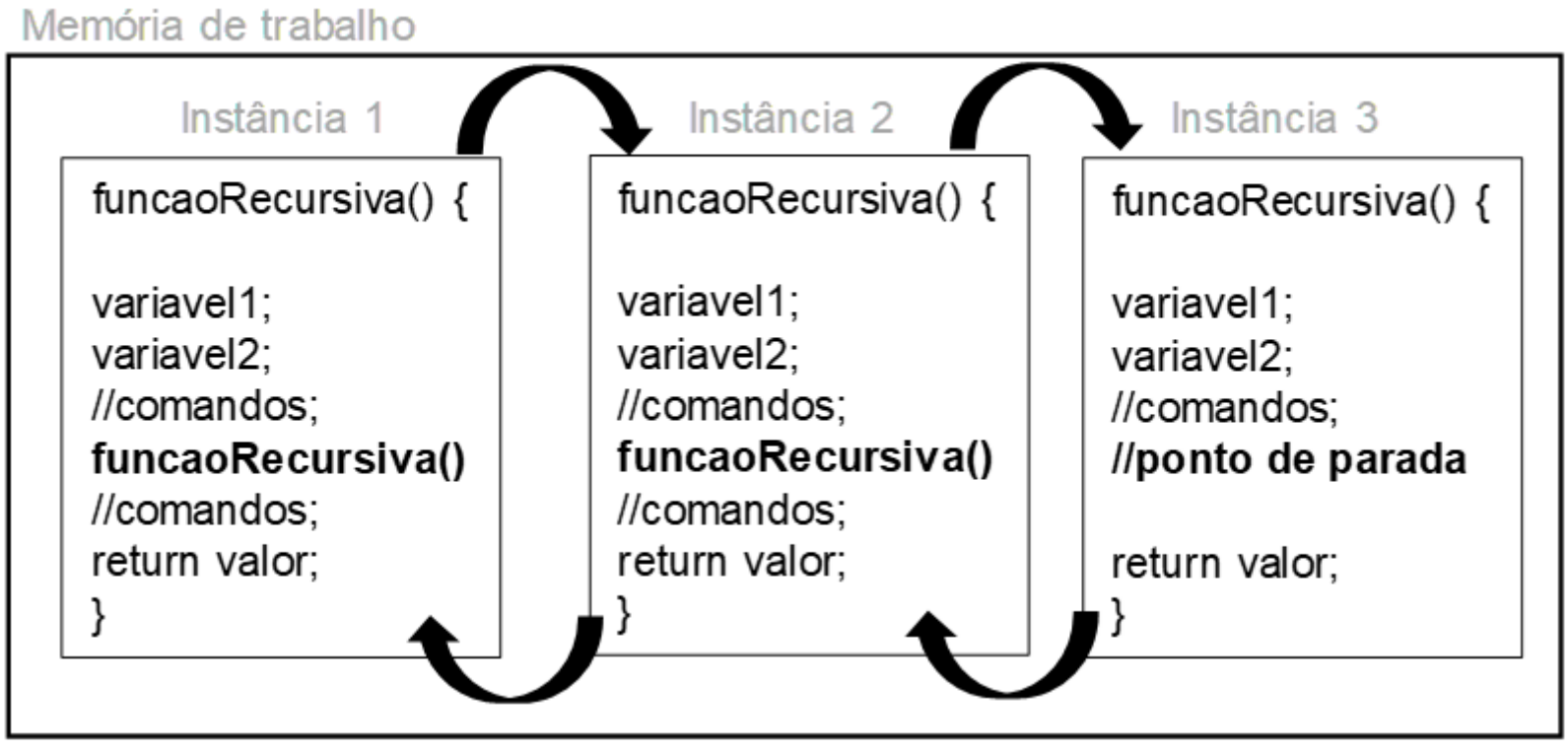
- Uma função apresenta em seu corpo variáveis e comandos, os quais são alocados na memória de trabalho. No uso de uma função recursiva, os recursos (variáveis e comandos) são alocados (instanciados) em outro local da memória, ou seja, para cada chamada da função, novos espaços são destinados à execução do programa. E é justamente por esse motivo que o ponto de parada é crucial.

- As variáveis criadas em cada instância da função na memória são independentes, ou seja, mesmo as variáveis tendo nomes iguais, cada uma tem seu próprio endereço de memória e a alteração do valor em uma não afetará a outra.

Para auxiliá-lo na compreensão desse mecanismo observe a Figura 3.7.

A instância 1 representa a primeira chamada à função *funcaoRecursiva()*, esta, por sua vez, tem em seu corpo um comando que invoca a si mesma; nesse momento é criada a segunda instância dessa função na memória de trabalho. Veja que um novo espaço é alocado, com variáveis e comandos, e, como a função é recursiva, novamente ela chama a si mesma, criando, então, a terceira instância da função. Dentro da terceira instância, uma determinada condição de parada é satisfeita, nesse caso, a função deixa de ser instanciada e passa a retornar valores.

Figura 3.7 | Mecanismo da função recursiva



Fonte: elaborada pela autora.

- A partir do momento que a função recursiva alcança o ponto de parada, cada instância da função passa a retornar seus resultados para a instância anterior (a que fez a chamada). No caso da Figura 3.7, a instância três retornará para a dois, e a dois retornará para a um. Veja que, se o ponto de parada não fosse especificado na última chamada, a função seria instanciada até haver um estouro de memória.

Toda função recursiva deve ter uma instância com um caso que interromperá a chamada a novas instâncias. Essa instância é chamada de **caso base**, pois representa o caso mais simples, que resultará na interrupção.

> **REFLITA**
>
> Toda função recursiva tem que dispor de um critério de parada. A instância da função que atenderá a esse critério é chamada de caso base. Um programador que implementa de maneira equivocada o critério de parada, acarretará um erro somente na sua aplicação, ou tal erro poderá afetar outras partes do sistema?

Vamos implementar uma função recursiva que faz a somatória dos antecessores de um número inteiro positivo, informado pelo usuário, ou seja, se o usuário digitar 5, o programa deverá retornar o resultado da soma 5 + 4 + 3 + 2 + 1 + 0. Com base nesse exemplo, você consegue determinar quando a função deverá parar de executar? Veja, a função deverá somar até o valor zero, portanto esse será o critério de parada. Veja a implementação da função no Código 3.19 e a seguir sua explicação.

Código 3.19 | Função recursiva para soma

```
1  #include<stdio.h>
2  int somar(int valor) {
3      if(valor == 0) {
4          //critério de parada
5          return valor;
6      } else {
7          //chamada recursiva
8          return valor + somar(valor - 1);
9      }
10 }
11 int main() {
12     int n, resultado;
13     printf("\nDigite um número inteiro positivo : ");
14     scanf("%d", &n);
15     resultado = somar(n); // primeira chamada da função
16     printf("\nResultado = %d",resultado);
17     return 0;
18 }
```

Fonte: elaborado pela autora.

Teste o Código 3.19 utilizando a ferramenta Paiza.io.

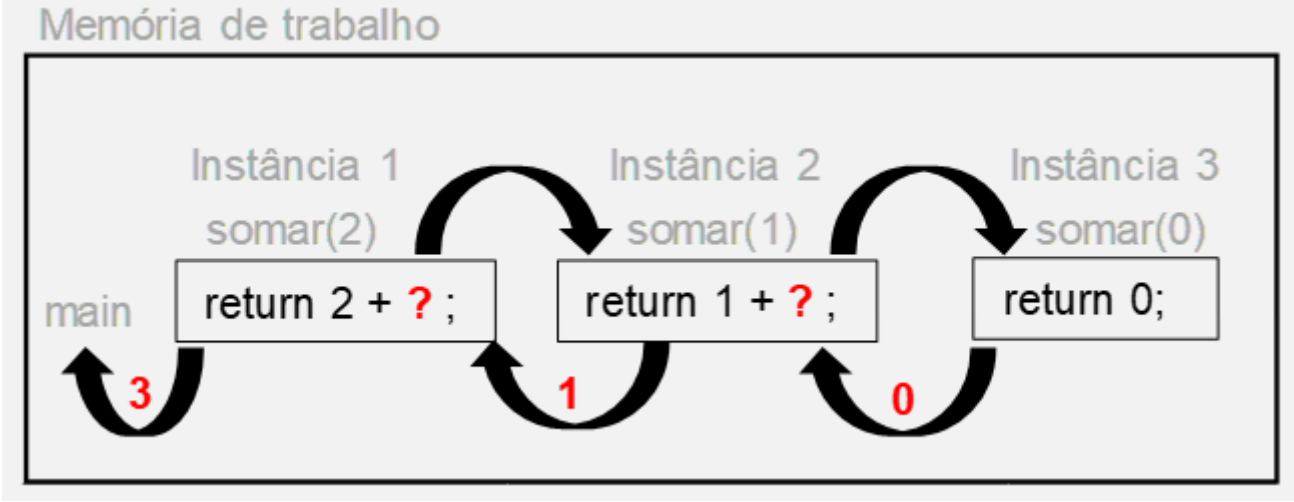
A execução do programa no Código 3.19 começará na linha 11, pela função principal (*main*). Na linha 15, a função *somar()* é invocada, passando como parâmetro um número inteiro digitado pelo usuário.

Nesse momento, a execução “salta” para a linha 2, onde a função é criada. Observe que ela foi criada para retornar e receber um valor inteiro. Na linha 3, o condicional foi usado como critério de parada; veja que, se o valor for diferente (!=) de zero, a execução passa para a linha 8, na qual a função é invocada novamente, mas dessa vez passando como parâmetro o valor menos 1. Quando o valor for zero, retorna-se o próprio valor, isto é, 0.

EXEMPLIFICANDO

A Figura 3.8 exemplifica o funcionamento na memória do computador da função *somar()*. Nessa ilustração, o usuário digitou o valor *2*, então a função *main()* invocará a função *somar(2)*, criando a primeira instância e passando esse valor. O valor 2 é diferente de zero na primeira instância, então, o critério de parada não é satisfeito e a função chama a si própria, criando a segunda instância, mas, agora, passando o valor 1 como parâmetro *somar(1)*. Veja que, na primeira instância, o valor a ser retornado é *2 + ?*, pois ainda não se conhece o resultado da função. Na segunda instância o valor também é diferente de zero, portanto a função chama a si mesma novamente, agora passando como parâmetro zero (valor – 1). Veja que o retorno fica como *1 + ?*, pois também não se conhece, ainda, o resultado da função. Na terceira instância, o critério de parada é satisfeito, nesse caso a função retorna zero. Esse valor será usado pela instância anterior que, após somar 1 + 0, retornará seu resultado para a instância 1, que somará 2 + 1 e retornará o valor para a função principal, fechando o ciclo de recursividade.

Figura 3.8 | Exemplo função *somar()*



Fonte: elaborada pela autora.

Uma das grandes dúvidas dos programadores é quando utilizar a recursividade em vez de uma estrutura de repetição. A função *somar()*, criada no Código 3.19, poderia ser substituída por uma estrutura de repetição usando *for*?

No exemplo dado, poderia ser escrito algo como:

```
for(int i = 0; i <= 2; i++) {
   resultado = resultado + i;
}
```

A verdade é que poderia, sim, ser substituída. A técnica de recursividade pode substituir o uso de estruturas de repetição, tornando o código mais elegante do ponto de vista das boas práticas de programação. Entretanto, como você viu, funções recursivas podem consumir mais memória que as estruturas iterativas. Para ajudar a elucidar quando optar por essa técnica, veja o que diz um professor da Universidade de São Paulo (USP):

> Muitos problemas têm a seguinte propriedade: cada instância do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm estrutura recursiva. Para resolver um tal problema, podemos aplicar o seguinte método:
> • se a instância em questão for pequena,
> resolva-a diretamente (use força bruta se necessário);
> • senão
> reduza-a a uma instância menor do mesmo problema,
> aplique o método à instância menor,
> volte à instância original.
> [...] A aplicação desse método produz um algoritmo recursivo. (FEOFILOFF, 2017, p. 1, grifo do original)

Portanto, a recursividade usada para indicar quando um problema maior pode ser dividido em instâncias menores do mesmo problema, porém considerando a utilização dos recursos computacionais que cada método empregará.

Não poderíamos falar de funções recursivas sem apresentar o exemplo do cálculo do fatorial, um clássico no estudo dessa técnica. O fatorial de um número qualquer N consiste em multiplicações sucessivas até que N seja igual ao valor unitário, ou seja, , que resulta em 120. Observe o Código 3.20, que implementa essa solução usando uma função recursiva. Em seguida, observe a explicação.

Código 3.20 | Função recursiva para fatorial

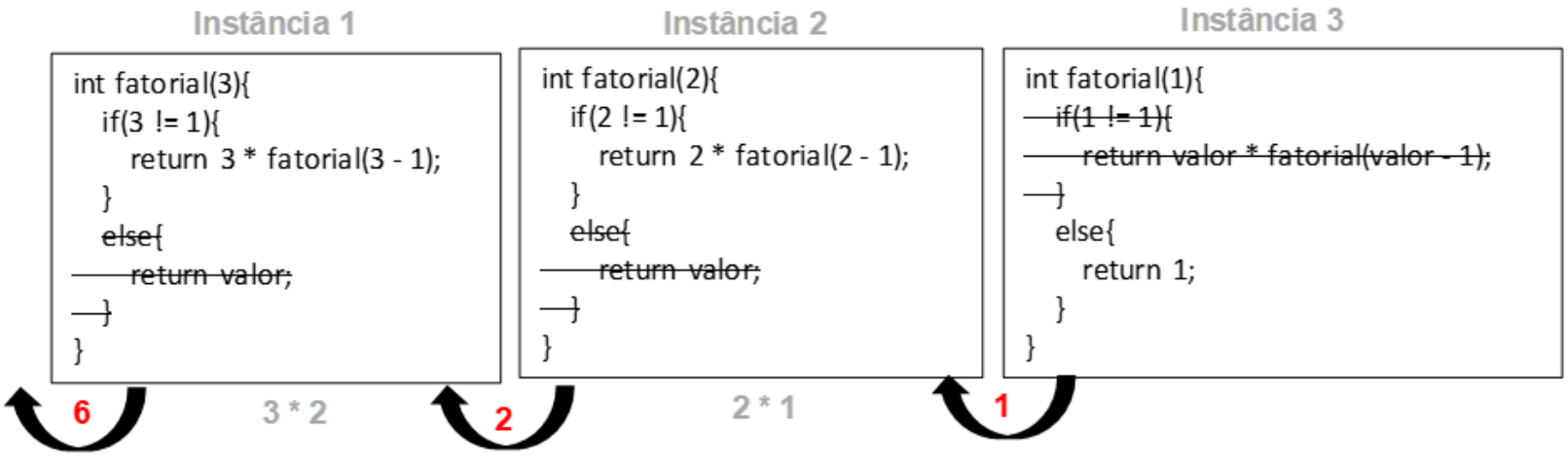
```c
#include<stdio.h>
int fatorial(int valor) {
    if(valor != 1) {
        //chamada recursiva
        return valor * fatorial(valor - 1);
    } else {
        //critério de parada
        return valor;
    }
}
int main() {
    int n, resultado;
    printf("\nDigite um número inteiro positivo: ");
    scanf("%d", &n);
    resultado = fatorial(n);
    printf("\nResultado = %d", resultado);
    return 0;
}
```

Fonte: elaborado pela autora.

Teste o Código 3.20 utilizando a ferramenta Paiza.io.

A execução do Código 3.20, inicia pela função principal, a qual solicita um número ao usuário e, na linha 15, invoca a função *fatorial()*, passando o valor digitado como parâmetro. Dentro da função *fatorial()*, enquanto o valor for diferente de 1, a função chamará a si própria, criando instâncias na memória, passando a cada vez como parâmetro o valor decrementado de 1. Quando o valor chegar a 1, a função retorna o próprio valor, permitindo assim o retorno da multiplicação dos valores encontrados em cada instância. É importante entender bem o funcionamento. Observe a Figura 3.9, que ilustra as instâncias quando o usuário digita o número 3. Veja que os resultados só são obtidos quando a função chega no caso base e, então, começa a "devolver" o resultado para a instância anterior.

Figura 3.9 | Exemplo de função *fatorial()*



Fonte: elaborada pela autora.

RECURSIVIDADE EM CAUDA

Esse mecanismo é custoso para o computador, pois tem que alocar recursos para as variáveis e comandos da função, procedimento chamado de **empilhamento**, além de ter que armazenar o local onde foi feita a chamada da função (OLIVEIRA, 2018). Para usar a memória de forma mais otimizada, existe uma alternativa chamada **recursividade em cauda**. Nesse tipo de técnica, a recursividade funcionará como uma função iterativa (OLIVEIRA, 2018).

Uma função é caracterizada como recursiva em cauda quando a chamada a si mesma é a última operação a ser feita no corpo da função. Nesse tipo de função, o caso base costuma ser passado como

parâmetro, o que resultará em um comportamento diferente.

Para entender, vamos implementar o cálculo do fatorial usando essa técnica. Veja o Código 3.21. Observe, na linha 8, que a função recursiva em cauda retorna o fatorial, sem nenhuma outra operação matemática, e que passa o número a ser calculado e o critério de parada junto. Foi necessário criar uma função auxiliar para que o usuário possa calcular o fatorial passando apenas um número inteiro.

Código 3.21 | Recursividade em cauda

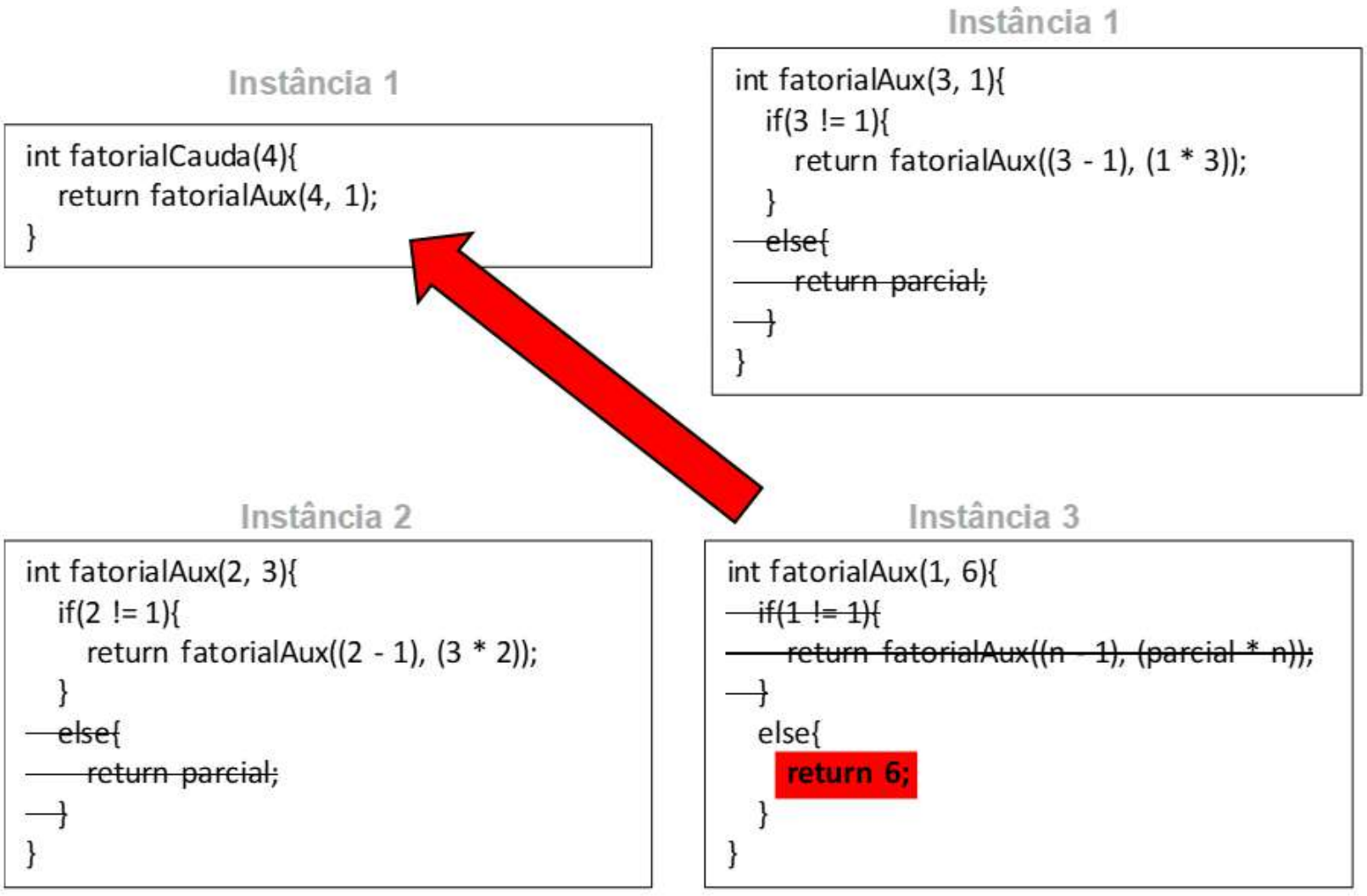
```
1  #include<stdio.h>
2  int fatorialCauda(int n) {
3      return fatorialAux(n, 1);
4  }
5
6  int fatorialAux(int n, int parcial) {
7      if (n != 1) {
8          return fatorialAux(n - 1, parcial * n);
9      } else {
10         return parcial;
11     }
12 }
13 int main() {
14     int n, resultado;
15     printf("\nDigite um número inteiro positivo: ");
16     scanf("%d", &n);
17     resultado = fatorialCauda(n);
18     printf("\nResultado do fatorial = %d", resultado);
19     return 0;
20 }
```

Fonte: elaborado pela autora.

Teste o Código 3.21 utilizando a ferramenta Paiza.io.

Observe, na Figura 3.21, que o mecanismo de funcionamento da recursividade em cauda é diferente. A função *main()* invoca a função *fatorialCauda()*, passando como parâmetro o valor a ser calculado. Esta, por sua vez, invoca a função *fatorialAux*, passando o valor a ser calculado e o critério de parada. A partir desse ponto, inicia a maior diferença entre as técnicas. Veja que as instâncias vão sendo criadas, porém, quando chega na última (nesse caso a instância 3), o resultado já é obtido e as funções não precisam retornar o valor para "quem" invocou, pois o comando que invoca a função era o último comando da função. Isso gera uma otimização na memória, pois não precisa armazenar nenhum ponto para devolução de valores.

Figura 3.10 | Exemplo de recursividade em cauda



Fonte: elaborada pela autora.

Chegamos, então, a mais um final de seção. Não se esqueça de que quanto mais você exercitar, mais algoritmos implementar, mais desafios solucionar, mais sua lógica se desenvolverá e você terá maior facilidade para desenvolver novos programas. Agora você já tem conhecimento para solucionar seu próximo desafio. Boa sorte!

FAÇA VALER A PENA

Questão 1

A recursividade é uma técnica de programação usada para tornar o código mais elegante e organizado, o que pode facilitar a manutenção. Essa técnica, em muitos casos, pode ser usada para substituir uma estrutura de repetição iterativa, como aquelas que usam o for.

Com base no contexto apresentado, avalie as seguintes asserções e a relação proposta entre elas.

I. As estruturas de repetição sempre podem ser substituídas por funções recursivas.

PORQUE

II. Uma função recursiva funciona como um laço de repetição, o qual será interrompido somente quando o caso base for satisfeito.

A respeito dessas asserções e da relação entre elas, assinale a alternativa correta.

a. As asserções I e II são proposições verdadeiras, e a II é uma justificativa correta da I.

b. As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa correta da I.

c. A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.

d. A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.

e. As asserções I e II são proposições falsas.

Questão 2

Para criar uma função recursiva, a sintaxe nada difere das funções gerais, portanto é necessário informar o tipo de retorno, o nome e se recebe ou não parâmetros. O grande diferencial das funções recursivas e tradicionais é um comando no corpo da função, que invoca a si própria. Analise o código a seguir e escolha a opção que representa o que será impresso na linha 10.

```c
#include<stdio.h>
int somar(int valor) {
    if (valor != 0){
        return valor + somar(valor - 1);
    } else {
        return valor;
    }
}
int main() {
    printf("\nResultado  = %d", somar(6));
    return 0;
}
```

Ver anotações

Com base no contexto apresentado, assinale a alternativa correta.

a. Resultado = 21.

b. Resultado = 0.

c. Resultado = 12.

d. Resultado = 6.

e. Resultado = 5.

Questão 3

A recursividade é uma técnica de programação que deve ser usada com cautela, pois a cada chamada à função, novos recursos são alocados na memória, em um processo chamado de empilhamento, que cresce rapidamente com as chamadas, podendo acarretar um estouro de memória.

A respeito de funções recursivas, analise as afirmativas a seguir.

I. Existe uma classe específica de funções recursivas chamada de recursividade em cauda que, embora tenha a mesma sintaxe no corpo da função, apresenta comportamento diferente das demais funções.

II. Uma função é caracterizada como recursiva em cauda quando a chamada a si mesma é a última operação a ser feita no corpo da função.

III. Em uma função que implementa a recursividade em cauda, a instância que fez a chamada recursiva depende do resultado da próxima.

IV. O uso da recursividade em cauda torna opcional o uso do caso base, pois a última instância retornará o valor final esperado.

A respeito das afirmativas apresentadas, é correto o que se afirma apenas em:

a. I, II e III.

b. II, III e IV.

c. II e IV.

d. II.

e. IV

Ver anotações

REFERÊNCIAS

ALMEIDA, R.; ZANLORENSSI, G. A evolução do número de linhas de telefone fixo e celular no mundo. **Nexo Jornal**, 2 maio 2018. Disponível em: https://cutt.ly/7jS47V9. Acesso em: 7 jul. 2018.

COUNTER, L. **Lines of code of the Linux Kernel Versions**. Disponível em: https://cutt.ly/zjSOt0G. Acesso em: 30 jun. 2018.

FEOFILOFF, P. **Recursão e algoritmos recursivos**. Instituto de Matemática e Estatística da Universidade de São Paulo, 21 maio 2018. Disponível em: https://cutt.ly/OjSOwOH. Acesso em: 2 jan. 2018.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. São Paulo: Érica, 2015.

OLIVEIRA, R. **Algoritmos e programação de computadores**. Notas de aula. Instituto de Computação da Universidade Estadual de Campinas – UNICAMP, [s. d.]. Disponível em: https://cutt.ly/mjSI75P. Acesso em: 16 jul. 2018.

PIRES, A. A. **Cálculo numérico**: prática com algoritmos e planilhas. São Paulo: Atlas, 2015.

SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.

TORRES, F. **Criptografia**: o método RSA. Apostila MA553 A – Teoria Aritmética dos Números. IMECC, Unicamp, [s. d.]. Disponível em: https://cutt.ly/JjS7rKs. Acesso em: 10 jan. 2021.

FOCO NO MERCADO DE TRABALHO

RECURSIVIDADE

Vanessa Cadan Scheffer

FUNÇÕES RECURSIVAS DE CAUDA

Implementação de um programa que resolve se um número é primo ou não, usando funções recursivas de cauda.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Após conhecer a técnica de recursividade, chegou o momento de implementar o programa para os matemáticos do laboratório onde trabalha. Lembre-se: foi solicitado a você implementar um programa

que resolve se um número é primo ou não, porém, usando funções recursivas de cauda.

Para implementar essa solução, você precisa de um programa que solicite ao usuário um número.

Você pode seguir os seguintes passos para a implementação:

1. Crie a função *ehPrimo()* de modo que ela receba um parâmetro, o valor a ser verificado e retorne um número inteiro:
   int ehPrimo(int n)

2. Crie uma função auxiliar *ehPrimoAux()*, que recebe, além do valor a ser verificado, um inteiro x, inicialmente 2, que será utilizado para calcular os divisores de 2 até n.

3. Caso o número n seja divisível por x, então, a função deve retornar o valor 0, informando que o número não é um primo. Caso isso não aconteça até que x seja igual a n, então retorna-se 1.

Veja, no Código 3.22, uma possível implementação para essa solução.

Código 3.22 | Verificação se um número é primo ou não

```
#include<stdio.h>

int ehPrimo(int n) {
    return ehPrimoAux(n, 2);
}

int ehPrimoAux(int n, int x) {
    if (x == n) {
        return 1;
    } else if (n % x == 0) {
        return 0;
    } else {
        return ehPrimoAux(n, x + 1);
    }
}
```

Fonte: elaborado pela autora.

Agora, você pode testar o código da solução completa utilizando a ferramenta Paiza.io.

AVANÇANDO NA PRÁTICA

MÁXIMO DIVISOR COMUM

Você foi contratado como professor de programação e, em conversa com o professor de matemática instrumental, ficou sabendo que os alunos têm dúvidas quanto ao mecanismo para calcular o Máximo Divisor Comum (MDC). Seu novo colega sugeriu que você implementasse essa solução com os alunos, para que eles possam, de fato, compreender o algoritmo. Como os alunos já têm conhecimento em programação, como você implementaria a solução?

RESOLUÇÃO

Para implementar essa solução, você deve optar por funções recursivas a fim de desenvolver ainda mais o raciocínio lógico. Para isso, você terá que recorrer ao método numérico de divisões sucessivas (PIRES, 2015). Para entender o mecanismo, considere como exemplo encontrar o MDC entre 16 e 24.

1. É preciso dividir o primeiro número pelo segundo e guardar o resto: $\frac{16}{24} = 1$, com resto 16.
2. O divisor da operação anterior deve ser dividido pelo resto da divisão. $\frac{24}{16} = 1$, com resto 8.
3. O segundo passo deve ser repetido, até que o resto seja nulo e, então, o divisor será o MDC. $\frac{16}{8} = 2$, com resto 0.

Veja, no Código 3.23, uma possível implementação para o MDC.

Código 3.23 | MDC com recursividade

```c
#include<stdio.h>
int calcularMDC(int a, int b) {
    int r = a % b;
    if (r == 0) {
        return b;
    } else {
        return calcularMDC(b,r);
    }
}
int main(){
    int n1, n2, resultado;
    printf("\nDigite dois números inteiros positivos: ");
    scanf("%d %d", &n1, &n2);
    resultado = calcularMDC(n1,n2);
    printf("\nMDC = %d",resultado);
    return 0;
}
```

Fonte: elaborado pela autora.

Agora, você pode testar o código utilizando a ferramenta Paiza.io.

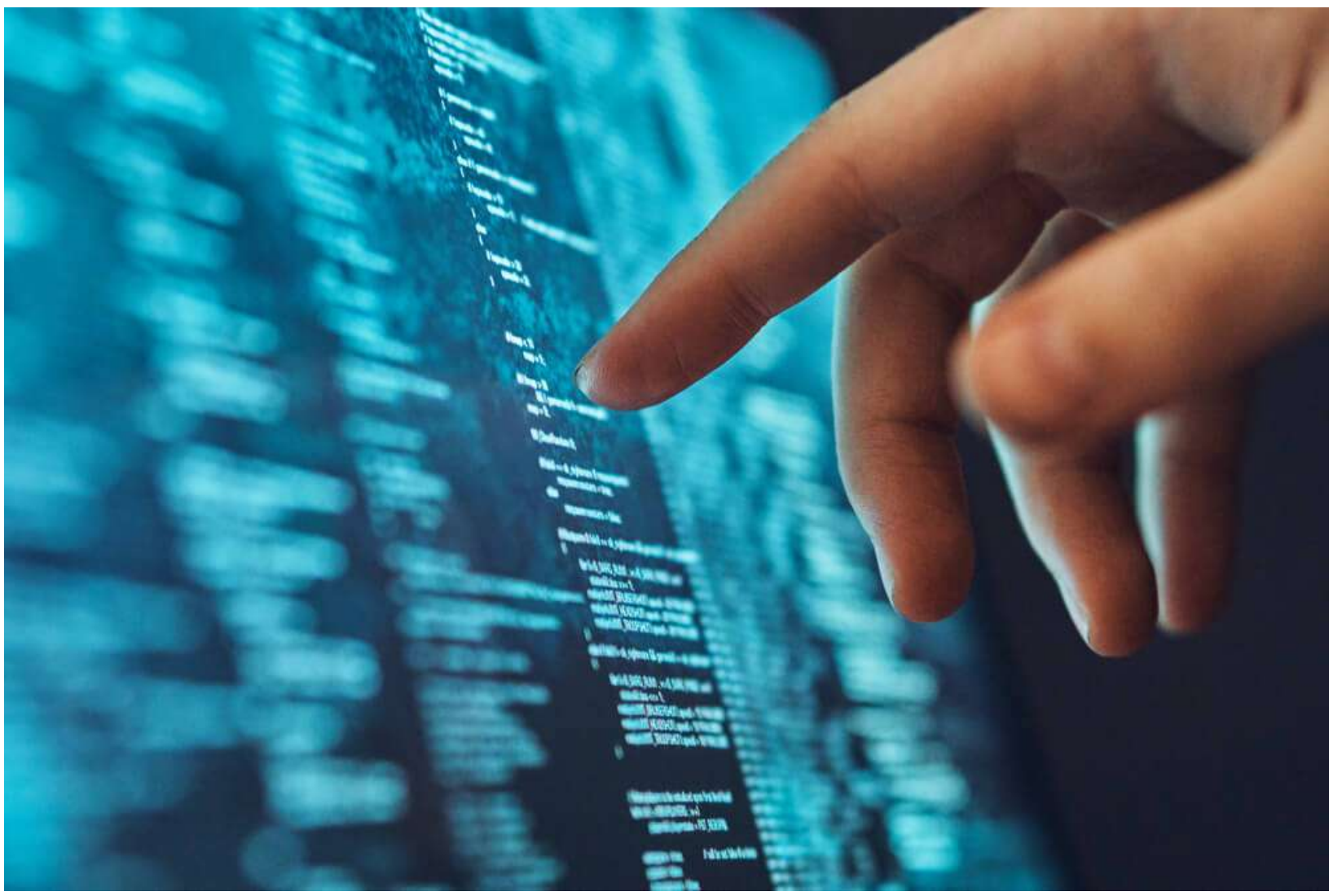
NÃO PODE FALTAR

LISTAS

Paulo Afonso Parreira Júnior

O QUE SÃO AS ESTRUTURAS DE DADOS DO TIPO LISTA?

As listas representam um conjunto dinâmico cujos elementos podem ser inseridos e retirados de qualquer parte da estrutura, assim como uma lista de compras.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Caro aluno, a maior parte dos sistemas de software existentes hoje em dia utiliza Estrutura de dados (ED) em seu código fonte; desde um sistema bancário até aquele seu jogo de última geração preferido, passando pelos aplicativos móveis que estão no seu smartphone. Por exemplo, ao fazer a compra de uma passagem aérea, você entra em uma **lista** de passageiros; ao fazer o pedido em uma lanchonete, seu pedido entra em uma **fila** de espera para ser preparado; por último, ao utilizar a função desfazer do seu editor de textos, você está pedindo a ele que desfaça o último comando de uma **pilha** de comandos realizados por você no editor. Todas as palavras destacadas anteriormente são exemplos de estruturas de dados que você aprenderá a desenvolver e usar nesta unidade. E não é apenas isso: sempre quando você utiliza um tipo de dado em uma linguagem como C, em última análise, você está utilizando uma estrutura de dados, que nada mais é do que um tipo de dado associado a um conjunto de operações que podem ser realizadas sobre ele. Enfim, já deu para perceber a importância desse conceito de estrutura de dados, certo?

Até aqui, você aprendeu a respeito das ferramentas básicas que quase todo tipo de linguagem de programação apresenta, tais como atribuições, funções, variáveis e constantes, entre outros. Como você verá, elas serão fundamentais para o que vamos aprender nesta seção.

A partir de agora, você conhecerá o conceito de listas ligadas, um tipo de ED, sua construção e uso adequados, bem como sua aplicação em programas de computador (Seção 1). Logo após, nas Seções 2 e 3, você verá como implementar outros tipos de ED que também são muito importantes, tais como pilhas e filas.

Bons estudos!

PRATICAR PARA APRENDER

Você foi contratado como desenvolvedor de uma empresa do ramo de jogos digitais. Como primeira tarefa, você foi encarregado de desenvolver uma Estrutura de Dados (ED) do tipo lista para ser usada

em vários jogos desenvolvidos pela empresa. Felizmente, você tinha acabado de aprender a implementar esse tipo de ED na faculdade. Então você fez um ótimo trabalho, implementando as funções básicas da lista, e sua estrutura de dados está sendo usada em vários jogos na empresa, o que deixou seu chefe muito contente.

Contudo, novas demandas surgiram e você deve implementar novas funções para sua ED, visando atender às necessidades da empresa. Seu chefe confiou a você a implementação das seguintes funções:

- *posicao_menor(li)*: retorna a posição menor elemento da lista "*li*"; e
- *existe(li, item)*: verifica se o "item" existe ou não na lista "*li*".

Para demonstrar seu trabalho, ele pediu que você usasse a função "*posicao_menor*", desenvolvida por você para apresentar o nome do inimigo que está mais próximo do jogador em determinado momento do jogo. O prazo para resolver esse desafio não é tão longo, então, bom trabalho!

CONCEITO-CHAVE

Conjuntos são fundamentais para a computação, assim como o são para a matemática. Enquanto os conjuntos matemáticos são invariáveis, os conjuntos manipulados por algoritmos podem crescer, diminuir ou sofrer mudanças em seus elementos ao longo da execução de um programa (CORMEN *et al.*, 2012).

Os conjuntos que apresentam tais características são conhecidos como conjuntos dinâmicos e são indispensáveis para se resolver uma série de problemas em computação. Algoritmos podem exigir a execução de vários tipos de operações em conjuntos, como testar a pertinência de um elemento no conjunto, inserir ou eliminar um determinado elemento e outros.

Um mecanismo utilizado para organizar nossa informação e prover operações convenientes e eficientes para acessá-la e manipulá-la é conhecido como **estrutura de dados**.

Diversos tipos de estruturas de dados têm sido propostos e o conhecimento das características dessas estruturas é um fator importante para a escolha da estrutura que melhor se encaixa na solução de um determinado problema. Nesta unidade, examinaremos a representação de conjuntos dinâmicos por meio de três estruturas de dados elementares amplamente conhecidas como lista, pilha e fila.

Vamos começar, então, com as listas.

LISTA

A lista representa um conjunto dinâmico cujos elementos podem ser inseridos e retirados de qualquer parte da estrutura. Trata-se da estrutura mais flexível, em termos de liberdade para a manipulação de elementos, se comparada com as outras estruturas que apresentaremos.

A analogia que pode ser feita para esta estrutura de dados é uma lista de compras de um supermercado. Uma lista de compras pode ser consultada em qualquer ordem e os itens dessa lista podem ser encontrados à medida que a pessoa caminha pelo supermercado; assim, um item no meio da lista pode ser removido antes do item do início da lista.

Devido à sua flexibilidade, a lista é uma das estruturas mais utilizadas para o desenvolvimento de softwares, em geral. As principais operações que devem estar presentes em uma ED do tipo lista são:

- ***criar()***: cria e retorna uma lista vazia.
- ***inserir(li, pos, item)***: adiciona o elemento "item" na posição "*pos*" da lista "*li*".
- ***remover(li, pos)***: remove e retorna o elemento da posição "*pos*" da lista "*li*".
- ***obter(li, pos)***: retorna (sem remover) o elemento da posição "*pos*" da lista "*li*".

- ***tamanho(li)***: retorna a quantidade de elementos existentes na lista "*li*".
- ***vazia(li)***: retorna se a lista "*li*" está vazia ou não.
- ***liberar(li)***: desaloca a lista "*li*" da memória do computador.

O Quadro 4.1 apresenta uma série de operações e seus efeitos sobre uma ED lista. Considere que o elemento mais à esquerda da lista consiste em seu início, cujo índice é igual a 0 (zero) e o elemento mais à direita, o seu fim, cujo índice é igual ao tamanho da lista menos um.

ASSIMILE

Assim como nos vetores, na ED lista, os elementos também são indexados de 0 a n – 1, onde n corresponde ao tamanho da lista.

Quadro 4.1 | Sequência de operações em uma ED lista

Operação	Saída	Conteúdo da Lista
criar()	li	()
inserir(li, 0, 7)	-	(7)
inserir(li, 0, 4)	-	(47)
obter(li, 1)	7	(4 7)
inserir (li, 2, 2)	-	(4 7 2)
obter(li, 3)	Erro	(4 7 2)
remover(li, 1)	7	(4 2)
inserir(li, 1, 5)	-	(4 5 2)
inserir(li, 1, 3)	-	(4 3 5 2)
inserir(li, 4, 9)	-	(4 3 5 2 9)

Operação	Saída	Conteúdo da Lista
obter(li, 2)	5	(4 3 5 2 9)
tamanho(li)	5	(4 3 5 2 9)
liberar(li)	-	-

Fonte: elaborado pelo autor.

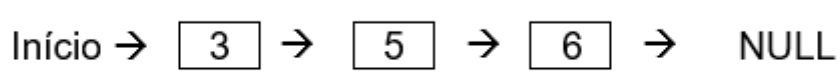
> **REFLITA**
>
> No Quadro 4.1 nós vimos várias operações que podem ser realizadas sobre uma ED lista. Contudo, entre elas não existe uma função para verificar se a lista está “cheia” ou não?
>
> Reflita: é possível que uma lista ligada esteja “cheia” a ponto de não poder mais receber novos elementos?

LISTA SIMPLESMENTE LIGADA

Podemos implementar ED na linguagem C de pelo menos duas formas: utilizando vetores ou utilizando estruturas ligadas (ou encadeadas). Estruturas de dados ligadas alocam espaços na memória do computador à medida que novos elementos precisam ser adicionados e os desalocam à medida que não precisamos mais deles. Por isso, é dito que as estruturas de dados ligadas proporcionam uma melhor utilização da memória.

Observe a ilustração de uma lista simplesmente ligada na Figura 4.1.

Figura 4.1 | Ilustração de uma lista ligada



Fonte: elaborada pelo autor.

Uma lista simplesmente ligada consiste em uma sequência de elementos, denominados **nós** (representados por retângulos com uma seta em uma de suas extremidades na Figura 4.1). Cada nó possui uma

informação (valor que aparece dentro do retângulo) e um ponteiro para o próximo nó da lista (seta na extremidade direita do retângulo). O último nó da lista aponta para **NULL**, representando que não existe um próximo nó na lista. A lista propriamente dita é representada apenas por um ponteiro para o primeiro nó dessa lista, chamado de **Início** na Figura 4.1.

Uma lista simplesmente ligada tem esse nome porque cada nó da estrutura tem apenas um ponteiro para o próximo nó da lista. Existe também o conceito de lista duplamente ligada, que veremos mais à frente nesta seção. Em uma lista duplamente ligada, cada nó apresenta dois ponteiros, um que aponta para o próximo nó e outro que aponta para o nó anterior.

> **ASSIMILE**
>
> Observando a Figura 4.1, não é difícil entender que, em uma lista vazia, o ponteiro “Início” apontará para NULL, uma vez que não há elementos na lista. Assim, já podemos imaginar que a operação “vazia”, que retorna se a lista está vazia ou não, deve comparar se o ponteiro “Início” é igual a NULL ou não.

Vamos, neste momento, passar à parte prática e implementar nossa primeira ED, uma **lista simplesmente ligada**.

Começaremos criando algumas *structs*. O conceito já foi apresentado, mas, para relembrar, *structs* permitem ao programador criar variáveis compostas heterogêneas na linguagem C. Com *structs* você consegue criar variáveis que podem armazenar mais de uma informação, assim como os vetores. Mas, ao contrário dos vetores, que só permitem armazenar valores de um mesmo tipo, *structs* oferecem a possibilidade de armazenarmos valores de tipos diferentes, tais como números e caracteres, entre outros (por isso são chamadas de variáveis compostas heterogêneas).

Na linguagem C, a criação de uma estrutura deve ser feita fora de qualquer função e deve apresentar a seguinte sintaxe:

```
struct <nome>{
  <tipo> <nome_da_variavel1>;
  <tipo> <nome_da_variavel2>;
  ...
};
```

<nome> é o nome da estrutura, por exemplo, cliente, carro, fornecedor, nó, lista e outros. As variáveis internas são os campos da *struct*, nos quais se deseja guardar as informações.

Agora que revisamos rapidamente o conceito de *struct*, vamos criar uma *struct* para representar cada nó da nossa lista simplesmente ligada.

Como vimos anteriormente, cada nó contém uma informação e um ponteiro para o próximo nó. Assim, a *struct* para o nó de lista ficará assim:

```
struct No {
  int info;
  struct No* proximo;
};
```

Por questão de simplicidade, estamos considerando que a informação presente em um nó (definida pela variável "*info*") é um número inteiro. Contudo, você pode armazenar qualquer tipo de informação que desejar, como *strings*, número reais, ou até mesmo uma variável de outro tipo *struct*.

É possível ver que o campo "*proximo*" é um ponteiro que aponta para outro nó da estrutura. Para declarar um ponteiro, utilizamos o operador * após o tipo da variável.

Uma vez criada a *struct* que representa um nó da lista, precisamos de uma *struct* para representar a lista propriamente dita. Essa *struct* deve conter, como campos, um ponteiro para o primeiro nó da lista e a

quantidade de elementos presentes nessa lista. O código dessa *struct* é o seguinte:

```
struct Lista {
  struct No* inicio;
  int tamanho;
};
```

Partiremos agora para a implementação do comportamento que “dará vida” a nossa ED, isto é, as funções que nos permitirão criar uma lista, inserir, remover e consultar elementos nela, entre outras coisas.

A primeira e uma das mais importantes funções que vamos implementar é a função que cria a própria lista. Vejamos o código desta função (Código 4.1) e, depois, a explicaremos passo a passo.

Código 4.1 | Função para criar uma lista vazia

```
1  struct Lista* criar() {
2      struct Lista* nova_lista = (struct Lista*)
   malloc(sizeof(struct Lista));
3      if (nova_lista != NULL) {
4          nova_lista->inicio = NULL;
5          nova_lista->tamanho = 0;
6      }
7      return nova_lista;
8  }
```

Fonte: elaborado pelo autor.

Basicamente, o que essa função faz é instanciar dinamicamente uma variável do tipo *struct* “Lista” na memória (linha 2) e configurar os valores de seus campos. O campo “inicio”, que é um ponteiro, deve apontar para NULL, uma vez que estamos criando uma lista vazia (linha 4). NULL é uma palavra reservada da linguagem C, muito usada com ponteiros, indicando que ele não está apontando para qualquer posição de memória válida do computador. Analogamente, o campo “tamanho”

deve ser inicializado com o valor igual a 0 (zero) – linha 5. Feito isso, deve-se retornar o endereço da memória alocado para a variável do tipo "Lista" (linha 7). Um ponto importante a destacar é que, antes de configurar os valores dos campos da *struct* "*Lista*", é preciso testar se a memória foi corretamente alocada para a lista (linha 3). Isso é importante, pois caso a função "*malloc*" não consiga alocar o espaço necessário para a lista, ela retornará o valor NULL.

Ao utilizar a função "*malloc*" quando criamos um ponteiro para uma variável do tipo de *struct*, a forma de acesso aos campos da *struct* muda. Em vez de usarmos o operador. (ponto), passamos a utilizar o operador ➔ (seta), conforme pode ser visto nas linhas 4 e 5 no Código 4.1.

Já podemos criar e utilizar nossa primeira lista simplesmente ligada. Mas, antes disso, vamos implementar mais uma função, a saber, a função que verifica se a lista está vazia ou não. Para sabermos se uma lista está vazia ou não, basta verificar se o ponteiro "inicio" da lista está apontando para NULL. Para isso, é preciso ter acesso à lista, que será passada por referência para a função. Veja, no Código 4.2, o código da função que verifica se a lista está vazia.

Código 4.2 | Função para verificar se uma lista está vazia

```
1  #include <stdbool.h>
2  #include <assert.h>
3  bool vazia(struct Lista* li) {
4      assert(li != NULL);
5      if (li->inicio == NULL) {
6          return true;
7      } else {
8          return false;
9      }
10 }
```

Fonte: elaborado pelo autor

Dois destaques para essa função são necessários. Primeiramente, estamos utilizando o tipo de dados *bool*, que não existe, por padrão, na linguagem C. Então, para utilizá-lo, é preciso importar a biblioteca `<stdbool.h>`, como pode ser visto no Código 4.2 (linha 1). Outro ponto importante é o uso da instrução "*assert()*" (linha 4). Essa instrução é implementada pela biblioteca `<assert.h>` (linha 2) e seu objetivo é verificar o resultado de uma expressão lógica passada por parâmetro para ela. Caso o resultado da expressão seja verdadeiro, o fluxo do programa segue normalmente, caso contrário, o programa é abortado (terminado) e um erro é apresentado ao usuário. Isso é necessário porque, antes de realizar qualquer operação sobre a lista, é importante verificar se o endereço de memória apontado por ela não é NULL.

EXEMPLIFICANDO

Há outra forma de se implementar a função "*vazia()*". Pense um pouco e tente implementar uma versão alternativa para essa função sem usar o ponteiro "*inicio*" da lista.

A solução consiste em verificar o tamanho da lista. Se ele for igual a 0 (zero), é porque a lista está vazia; caso contrário, a lista não está vazia.

```
bool vazia(struct Lista* li) {
  assert(li != NULL);
  if (li->tamanho == 0) {
    return true;
  } else {
    return false;
  }
}
```

Agora podemos usar nossa lista. Vamos criar uma lista vazia e testar se ela realmente está vazia. Para isso, utilizamos a função *main()*, como pode ser visto no Código 4.3.

Código 4.3 | Função para criar uma lista e testar se ela está vazia

```
1  int main() {
2     struct Lista* minha_lista = criar();
3     if (vazia(minha_lista) == true) {
4         printf("\nOK, lista vazia!");
5     } else {
6         printf("\nOps... algo deu errado!");
7     }
8     return 0;
9  }
```

Fonte: elaborado pelo autor.

Ver anotações

Na linha 2 criamos uma lista vazia, usando a função “criar()”, e armazenamos o endereço dessa lista no ponteiro “minha_lista”. Nós usaremos esse ponteiro sempre que quisermos realizar alguma operação em nossa lista. Logo após, nas linhas 3 a 7, estamos testando se a lista está vazia, por meio da função “vazia()”, e imprimindo uma mensagem apropriada na tela. Visualmente falando, o resultado da execução do código apresentado no Código 4.3 é o que pode ser visto na Figura 4.2.

Figura 4.2 | Ilustração de uma lista ligada vazia

Início → NULL

Fonte: elaborada pelo autor.

O código completo da nossa ED lista simplesmente ligada até o momento pode ser visto a seguir, utilizando a ferramenta Paiza.io.

Trataremos, agora, de algumas funções mais desafiadoras. Vamos pensar na função que insere um novo elemento na lista. Como dissemos anteriormente, a ED lista é a estrutura mais flexível de todas que estudaremos nesta unidade, permitindo a inserção de elementos em quaisquer posições da lista.

Vamos iniciar com a implementação da função inserir, conforme pode ser visto no Código 4.4.

Código 4.4 | Função para inserir um elemento em uma lista

```
1  void inserir (struct Lista* li, int pos, int item) {
2      assert(li != NULL);
3      assert(pos >= 0 && pos <= li->tamanho);
4      struct No* novo_no = (struct No*) malloc(sizeof(struct
   No));
5      novo_no->info = item;
6      if (pos == 0) {
7          novo_no->proximo = li->inicio;
8          li->inicio = novo_no;
9      } else {
10         struct No* aux = li->inicio;
11         for(int i = 0; i < pos - 1; i++) {
12             aux = aux->proximo;
13         }
14         novo_no->proximo = aux->proximo;
15         aux->proximo = novo_no;
16     }
17     li->tamanho++;
18 }
```

Fonte: elaborado pelo autor.

O primeiro passo dessa função é verificar se o ponteiro que representa a lista, "*li*", não é nulo (linha 2). Logo após, precisamos verificar se a posição em que o elemento será inserido é válida (linha 3). No caso da função "inserir", podemos passar como posição qualquer número entre 0 e *n*, onde *n* corresponde à quantidade de elementos da lista.

Feito isso, precisamos instanciar um novo **nó**, que armazenará o novo elemento a ser inserido na lista (linha 4). Feito isso, devemos atualizar a variável "*info*" do novo nó com a informação recebida no parâmetro "item" da função (linha 5). Antes de continuar, precisamos fazemos um

teste para verificar se se trata de uma inserção no início da lista ou não. Mas por que isso? Porque, como veremos, o processo de inserção de um elemento no início da lista é diferente do processo de inserção nas demais posições da lista.

Caso a inserção seja no início da lista, isto é, o valor do parâmetro "*pos*" seja igual a 0 (zero), devemos fazer com que o ponteiro "*próximo*" desse nó aponte para o endereço que está sendo apontado pelo ponteiro "*inicio*" da lista (linha 7). Isso ocorre porque, como estamos inserindo um nó no início da lista, o novo nó passará a ser o primeiro elemento da lista e o nó que era o primeiro, passará a ser o segundo. Contudo, se paramos aqui, teremos um problema, pois o ponteiro "inicio" da lista continuará apontando para o mesmo lugar. Devemos, então, atualizá-lo para que ele aponte para o novo nó (linha 8).

> **REFLITA**
>
> O que aconteceria se invertêssemos a ordem das instruções das linhas 7 e 8 no código que insere um elemento na lista (Código 4.4)?

Se a inserção era para ser realizada no início da lista, o procedimento está completo. É importante incrementarmos o valor da variável "*tamanho*" da lista (linha 17), uma vez que um novo elemento foi inserido nela.

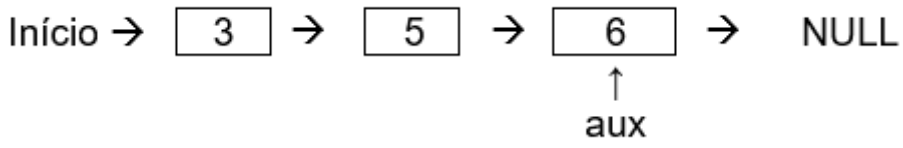
E se quisermos inserir um elemento em qualquer outra posição da lista, entre o início (definido pela posição 0) e o fim (definido pela posição *n*, onde *n* corresponde ao tamanho da lista)? A função "*inserir*" trata esse caso no seu bloco "*else*", localizado entre as linhas 9 e 16.

Mas por que há um tratamento diferente para os casos de inserção em outras posições da lista? A resposta é que, ao contrário do que acontece com o primeiro elemento da lista, nós não temos acesso direto a cada nó da lista. Assim, precisaremos percorrer essa lista até encontrar a posição em que gostaríamos de adicionar um novo elemento. Isso é

feito nas linhas 10 a 13 do código da função "*inserir*". Inicialmente é criado um ponteiro auxiliar, denominado "*aux*" (você pode dar o nome que quiser para ele) e fazê-lo apontar para o primeiro elemento da lista (linha 10). Logo depois, utilizamos uma estrutura de repetição for para percorrer a lista, atualizando o valor do ponteiro "*aux*" a cada iteração.

Supondo que desejamos inserir um novo elemento na posição 3 da lista apresentada na Figura 4.3, o ponteiro "*aux*" deve estar posicionado exatamente sobre o elemento 6, quando o trecho de código das linhas 10 a 13 for executado.

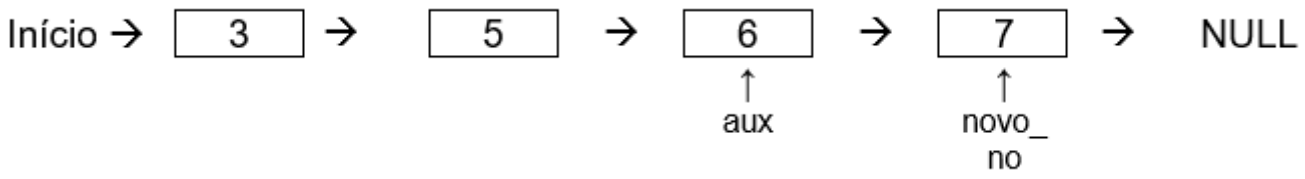
Figura 4.3 | Ilustração da busca por um nó da lista simplesmente ligada



Fonte: elaborada pelo autor.

Conforme pode ser percebido, o ponteiro "*aux*" apontará para o nó anterior à posição em que o novo elemento será inserido. E para que? Isso é importante porque precisamos atualizar o ponteiro próximo do novo nó, fazendo-o apontar para o próximo do nó "*aux*" (linha 14), bem como atualizar o ponteiro "*proximo*" do nó "*aux*", fazendo-o apontar para o nosso novo nó. O resultado visual dessa inserção pode ser visto na Figura 4.4.

Figura 4.4 | Inserção de um elemento no final da lista simplesmente ligada



Fonte: elaborada pelo autor.

EXEMPLIFICANDO

Com base no que aprendemos até o momento, escreva um programa que seja capaz de criar uma lista ligada conforme apresentado na Figura 4.4.

Uma possível solução para esse problema é:

```
int main() {
  struct Lista* minha_lista = criar(); // Lista = []
  inserir(minha_lista, 0, 5);     // Lista = [5]
  inserir(minha_lista, 0, 3);     // Lista = [3, 5]
  inserir(minha_lista, 2, 6);     // Lista = [3, 5,
6]
  inserir(minha_lista, 3, 7);     // Lista = [3, 5,
6, 7]
  return 0;
}
```

Ver anotações

Até o momento, criamos funções que nos permitem inserir elementos em uma lista simplesmente ligada. Antes de passarmos para as demais funções da lista, vamos criar uma função que, por padrão, não existe nesta ED, mas que será útil para verificar o correto funcionamento da lista. Trata-se da função "*imprimir*", que será responsável por imprimir todos os elementos da lista, do primeiro até o último. Vejamos como fica essa função no Código 4.5.

Código 4.5 | Função para imprimir os elementos de uma lista

```
1  void imprimir(struct Lista* li) {
2      assert(li != NULL);
3      printf("\nLista: ");
4      struct No* aux = li->inicio;
5      for(int i = 0; i < li->tamanho; i++) {
6          printf("%d ", aux->info);
7          aux = aux->proximo;
8      }
9  }
```

Fonte: elaborado pelo autor.

Veja que o código é bem parecido com o da função "*inserir*", com a diferença que estamos percorrendo a lista apenas para imprimir, não para inserir qualquer elemento nela.

Com isso em mãos, tente reproduzir, no papel, a execução do código da função "*main()*" no Código 4.6. Depois, use a função "imprimir" para confirmar sua resposta.

Código 4.6 | Função que insere vários elementos em uma lista

```
1  int main() {
2       struct Lista* minha_lista = criar();
3       inserir(minha_lista, 0, 5);
4       inserir(minha_lista, 0, 3);
5       inserir(minha_lista, 2, 6);
6       inserir(minha_lista, 1, 0);
7       inserir(minha_lista, 4, 7);
8       inserir(minha_lista, 1, 2);
9       inserir(minha_lista, 5, 6);
10      inserir(minha_lista, 3, 4);
11      return 0;
12 }
```

Fonte: elaborado pelo autor.

O código completo juntamente com a resposta pode ser visto a seguir, utilizando a ferramenta Paiza.io.

Falta-nos, agora, implementar as seguintes funções: "*tamanho*", que retorna a quantidade de elementos da lista, "*obter*", que retorna o elemento de determinada posição da lista, "*remover*", que remove e retorna o elemento de determina posição da lista e "*liberar*", que desaloca a memória usada pela lista. Dessas, a mais simples de todas é a função "*tamanho*", cujo código é apresentado no Código 4.7.

Código 4.7 | Função para retornar a quantidade de elementos de uma lista

```c
int tamanho(struct Lista* li) {
   assert(li != NULL);
   return li->tamanho;
}
```

Fonte: elaborado pelo autor.

Essa função simplesmente acessa e retorna o valor da variável "*tamanho*", armazenada na *struct* "*Lista*". Você pode testar a função com o código do Código 4.8, cujo resultado esperado é 2.

Código 4.8 | Função que imprime a quantidade de elementos de uma lista na tela

```c
int main() {
   struct Lista* minha_lista = criar();
   inserir(minha_lista, 0, 5);
   inserir(minha_lista, 0, 3);
   printf("\n%d", tamanho(minha_lista));
   return 0;
}
```

Fonte: elaborado pela autora.

O código completo juntamente com a resposta pode ser visto a seguir, utilizando a ferramenta Paiza.io.

A próxima função que vamos implementar é a função "obter", cujo código fonte encontra-se no Código 4.9.

Código 4.9 | Função para obter o elemento de determinada posição da lista

```c
int obter(struct Lista* li, int pos) {
   assert(li != NULL);
   assert(pos >= 0 && pos < li->tamanho);
   struct No* aux = li->inicio;
   for(int i = 0; i < pos; i++) {
      aux = aux->proximo;
   }
   return aux->info;
}
```

Ver anotações

Fonte: elaborado pelo autor.

Essa função tem instruções similares às da função "*inserir*", certo? Por exemplo, inicialmente, devemos verificar se a posição informada é válida, antes de retornar o elemento da lista (linha 3). A diferença, aqui, é que as posições válidas são de 0 à *n – 1*, onde *n* é o tamanho da lista. Ou seja, nós podemos inserir um novo elemento na posição n, que é a última posição da lista, mas não podemos obter um elemento da posição *n*, pois essa posição está "vazia". Analogamente, nós precisamos utilizar uma estrutura de repetição for (linhas 5 a 7) para acessar a posição desejada e só depois retornar o valor do elemento (linha 8).

Vamos, agora, implementar a função "*remover*" (Código 4.10). Faremos da mesma forma que fizemos com a função "inserir", apresentando inicialmente o código e comentando-o passo a passo.

Código 4.10 | Função para remove o elemento de determinada posição de uma lista

```c
1  int remover(struct Lista* li, int pos) {
2     assert(vazia(li) == false);
3     assert(pos >= 0 && pos < li->tamanho);
4
5     struct No* ant = NULL;
6     struct No* aux = li->inicio;
7     for(int i = 0; i < pos; i++) {
8        ant = aux;
9        aux = aux->proximo;
10    }
11
12    if (ant == NULL) {
13        li->inicio = aux->proximo;
14    } else {
15        ant->proximo = aux->proximo;
16    }
17
18    int elemento = aux->info;
19    li->tamanho--;
20    free(aux);
21    return elemento;
22 }
```

Fonte: elaborado pelo autor.

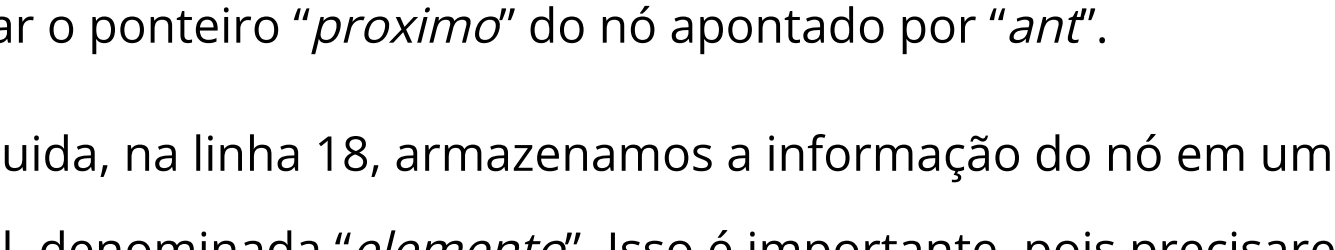
A função "*remover*" inicia verificando se a lista não está vazia (linha 2). Isso é importante, porque é impossível remover qualquer elemento de uma lista vazia. Posteriormente, verifica-se se a posição a ser removida é válida (linha 3). Assim, como na função "*obter*", nós só podemos remover elementos que estão entre as posições 0 e *n – 1*, onde *n* é o tamanho da lista.

Atenção: diferentemente do que fizemos na função "*inserir*", para a função "*remover*" precisamos de dois ponteiros auxiliares, denominados "*ant*" e "*aux*" (linhas 5 e 6). O ponteiro "*ant*" apontará para o nó anterior à posição que desejamos remover. Isso é necessário, pois precisaremos

atualizar o ponteiro "*proximo*" desse nó, uma vez que o nó a sua frente será removido. O ponteiro "*aux*" apontará exatamente para o nó que desejamos remover. É por meio dele que teremos a referência para a posição de memória a ser liberada.

O laço *for* (linhas 7 a 10) percorre a lista até encontrar o nó da posição que desejamos remover. Veja como ficariam os ponteiros "*ant*" e "*aux*" caso desejássemos remover o nó da posição 3 da lista da Figura 4.5.

Figura 4.5 | Ilustração do processo de busca de um nó a ser removido da lista



Fonte: elaborada pelo autor.

As linhas 12 a 16 atualizam alguns ponteiros, antes de realizar a remoção do nó apontado por "*aux*". Quando o nó "*ant*" aponta para NULL, significa que estamos tentando remover o elemento da primeira posição. Assim, temos que atualizar o ponteiro "*inicio*" da lista para que ele aponte para o segundo elemento da lista (linha 13). Caso o ponteiro "*ant*" não aponte para NULL, então queremos remover algum nó que esteja entre o segundo e último nó da lista. Nesse caso, devemos atualizar o ponteiro "*proximo*" do nó apontado por "*ant*".

Em seguida, na linha 18, armazenamos a informação do nó em uma variável, denominada "*elemento*". Isso é importante, pois precisaremos retornar o valor armazenado no nó ao final da execução da função. A linha 19 simplesmente decrementa o valor da variável "*tamanho*" da lista. Após isso, é necessário desalocar a memória alocada para nó da lista, usando a função "*free*" (linha 20).

Todas as vezes em que alocamos memória com a função "*malloc*" devemos usar a função "*free*" para desalocá-la. Caso contrário, a memória continuará reservada para o programa até o final de sua execução. Por fim, na linha 21, retornamos à informação armazenada no nó que foi removido.

Agora só precisamos implementar a função "*liberar*" e teremos a nossa lista simplesmente encadeada completa e funcional. A função liberar é bem simples, pois ela utiliza duas funções já implementadas anteriormente. Essa é uma das grandes vantagens do uso de funções: a reutilização de código. Observe o código da função liberar no Código 4.11.

Código 4.11 | Função para liberar uma lista da memória

```
1  void liberar(struct Lista* li) {
2     while(vazia(li) == false) {
3        remover(li, 0);
4     }
5     free(li);
6  }
```

Fonte: elaborado pelo autor.

Nessa função não precisamos usar a instrução "*assert*", pois a função "vazia" já faz essa verificação por nós. O funcionamento da função "liberar" consiste em remover elementos do início da lista enquanto a lista não estiver vazia (linhas 2 a 4). Por fim, a memória alocada para a *struct* "*Lista*" também é liberada na linha 5, por meio da função "*free*".

REFLITA

Por que a função "*liberar*" sempre invoca a função "*remover*" passando como parâmetro a posição 0 (zero)?

O código completo da ED lista simplesmente encadeada pode ser visto a seguir, utilizando a ferramenta Paiza.io.

LISTA DUPLAMENTE LIGADA

Uma limitação da ED lista simplesmente ligada é que qualquer inserção/remoção/leitura feita no final da lista tem um custo computacional alto, pois é preciso percorrer todos os nós da lista até

encontrar a última posição. Esse custo é ainda maior se quisermos, por exemplo, percorrer a lista de trás para frente, ou seja, do último elemento para o primeiro.

Para tentar mitigar essas limitações, surgiu o conceito de **lista duplamente ligada**. Há duas diferenças importantes da lista duplamente ligada para a lista simplesmente ligada.

i. Cada nó possui, além do ponteiro para o próximo nó, um ponteiro para o nó anterior. Isso permite que consigamos navegar na lista de trás para frente.

ii. A lista apresenta, além do ponteiro para o primeiro nó, um ponteiro para o último nó da lista. Isso permite que consigamos ler, inserir e remover elementos no final da lista, sem precisar percorrê-la toda.

Veja as alterações necessárias no código das *structs* "*Lista*" e "*No*" de uma lista duplamente ligada:

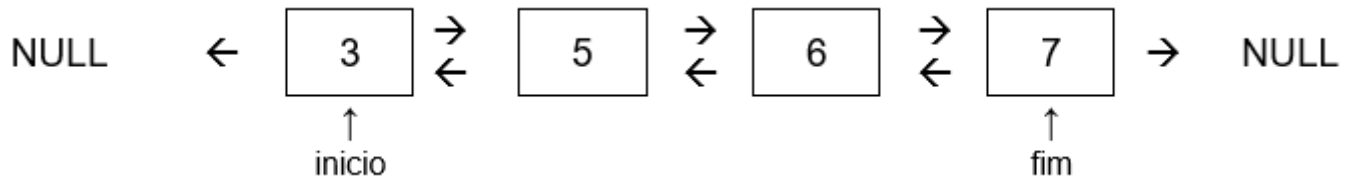
```
struct No {
  int info;
  struct No* anterior;
  struct No* proximo;
};
```

```
struct ListaDupla {
  struct No* inicio;
  struct No* fim;
  int tamanho;
};
```

Para diferenciarmos da lista simplesmente ligada, nós mudamos o nome da *struct* "*Lista*" para "*ListaDupla*".

A ilustração de uma lista duplamente ligada pode ser visualizada na Figura 4.6. A seta apontada para a esquerda representa o ponteiro "*anterior*" de um nó.

Figura 4.6 | Ilustração do processo de busca de um nó a ser removido da lista



Fonte: elaborada pelo autor.

A partir de agora, apresentaremos as mesmas operações da lista simplesmente ligada, comentando a diferença de implementação para o caso da lista duplamente ligada.

Comecemos pela função “criar” (Código 4.12).

Código 4.12 | Função para criar uma lista duplamente encadeada

```
1  struct ListaDupla* criar() {
2      struct ListaDupla* nova_lista = (struct ListaDupla*)
   malloc(sizeof(struct ListaDupla));
3      if (nova_lista != NULL) {
4          nova_lista->inicio = NULL;
5          nova_lista->fim = NULL;
6          nova_lista->tamanho = 0;
7      }
8      return nova_lista;
9  }
```

Fonte: elaborado pelo autor.

A única diferença, para esta função, é que agora temos que inicializar também o ponteiro “fim” da *struct* “*ListaDupla*” com o valor NULL, pois a lista é criada vazia.

O código da função “*vazia*” se mantém praticamente o mesmo, a não ser pelo nome da *struct* que mudou para “*ListaDupla*” (Código 4.13). Isso porque a forma de verificar se uma lista está vazia é a mesma, tanto para a lista simplesmente ligada quanto para a duplamente ligada.

Código 4.13 | Função para verificar se uma lista duplamente encadeada está vazia

```
1  bool vazia(struct ListaDupla* li) {
2      assert(li != NULL);
3      if (li->inicio == NULL) {
4          return true;
5      } else {
6          return false;
7      }
8  }
```

Fonte: elaborado pelo autor.

O mesmo ocorre com as funções "*tamanho*" e "*liberar*". Assim, o código dessas funções não será apresentado aqui novamente.

Vamos, então, olhar para aquelas funções que sofreram maiores modificações, a saber, as funções "*inserir*" e "*remover*". Isso porque agora nós temos que atualizar também os ponteiros "anterior", da *struct* "*No*", e "fim", da *struct* "*ListaDupla*".

Vamos começar pela função "inserir", cujo código encontra-se no Código 4.14.

Código 4.14 | Função inserir um elemento em uma lista duplamente encadeada

```
1  void inserir (struct ListaDupla* li, int pos, int item) {
2      assert(li != NULL);
3      assert(pos >= 0 && pos <= li->tamanho);
4
5      struct No* novo_no = (struct No*) malloc(sizeof(struct
   No));
6      novo_no->info = item;
7      if (pos == 0) {
8          novo_no->anterior = NULL;
9          novo_no->proximo = li->inicio;
10         li->inicio = novo_no;
11         if (li->fim == NULL) {
12             li->fim = novo_no;
13         }
14     } else if (pos == li->tamanho) {
15         novo_no->anterior = li->fim;
16         novo_no->proximo = NULL;
17          li->fim->proximo = novo_no;
18         li->fim = novo_no;
19     } else {
20        struct No* aux = li->inicio;
21        for(int i = 0; i < pos - 1; i++) {
22            aux = aux->proximo;
23        }
24        novo_no->anterior = aux;
25        novo_no->proximo = aux->proximo;
26        aux->proximo = novo_no;
27     }
28     li->tamanho++;
29 }
```

Fonte: elaborado pelo autor.

No trecho de código das linhas 8 a 13, que trata da inserção do início da lista, foi preciso incluir instruções para atualizar o ponteiro “anterior” do novo nó para NULL, pois não há nós anteriores ao nó inicial. Também foi

preciso atualizar o ponteiro "*fim*" da lista, no caso de ser a inserção do primeiro elemento da lista, ou seja, quando a lista tem apenas um elemento, tanto o ponteiro "*inicio*" quanto o "*fim*" apontam para o mesmo elemento.

O bloco *else-if* das linhas 14 a 18 é inédito e trata da inserção no final da lista. Agora, como temos acesso direto ao último elemento da lista, não precisaremos mais percorrê-la até o fim. Basta atualizarmos o ponteiro "*fim*" da lista. Essa é a grande vantagem da lista duplamente ligada: a inserção de um elemento tanto no início da lista quanto no seu fim tem um desempenho muito bom.

Por fim, o trecho de código das linhas 19 a 27, que trata da inserção de um elemento em qualquer posição entre o primeiro e o último elemento, é bem similar ao da lista simplesmente ligada, com a exceção que, agora, é preciso atualizar o ponteiro "*anterior*" do nó que está sendo inserido.

Passando agora para a função "*remover*", o código para a lista duplamente ligada está no Código 4.15.

Código 4.15 | Função remover um elemento em uma lista duplamente encadeada

```c
int remover(struct ListaDupla* li, int pos) {
    assert(vazia(li) == false);
    assert(pos >= 0 && pos < li->tamanho);
    struct No* aux = NULL;

    if (pos == 0) {
        aux = li->inicio;
        li->inicio = aux->proximo;
        if (li->inicio == NULL) {
            li->fim = NULL;
        } else {
            li->inicio->anterior = NULL;
        }
    } else if (pos == li->tamanho - 1) {
        aux = li->fim;
        li->fim = aux->anterior;
        li->fim->proximo = NULL;
    } else {
        struct No* ant = NULL;
        aux = li->inicio;

        for(int i = 0; i < pos; i++) {
            ant = aux;
            aux = aux->proximo;
        }

        ant->proximo = aux->proximo;
        aux->proximo->anterior = ant;
     }

    int elemento = aux->info;
    li->tamanho--;
    free(aux);
    return elemento;
}
```

Fonte: elaborado pelo autor.

Ver anotações

De forma análoga ao que fizemos na função "*inserir*", também precisamos verificar se se trata de uma remoção no início da lista, no fim ou em qualquer outra posição entre o início e o fim. Isso é feito por meio do comando *if-else-if*, contido no código. Caso seja uma remoção no início da lista, além do que já fizemos para a lista simplesmente ligada, devemos atualizar o ponteiro "*anterior*" do nó que estava na segunda posição da lista, bem como do ponteiro "*fim*", da lista (caso a lista tenha ficado vazia) – linhas 8 a 13. Também criamos uma sequência de instruções para o caso de a remoção ocorrer no final da lista (linhas 15 a 17). Nesse caso, devemos atualizar o ponteiro "*fim*" da lista (linha 16), bem como o ponteiro "*próximo*" do penúltimo nó da lista, fazendo-o apontar para NULL, pois ele passará a ser o último nó da lista (linha 17). Por último, das linhas 19 a 28, temos o caso de remoção para qualquer elemento entre o primeiro e o último elemento. As diferenças para a função remover da lista simplesmente ligada são que, agora, precisamos atualizar o ponteiro "anterior" do elemento subsequente ao que será sendo removido (linha 28). A sequência de comandos responsável por liberar espaço da memória e retornar a informação contida no elemento removido (linhas 31 a 34) mantém-se a mesma da função da lista simplesmente ligada.

Para terminar, vejamos o código da função "*obter*" para a lista duplamente ligada (Código 4.16).

Código 4.16 | Função obter um elemento em uma lista duplamente encadeada

```
1  int obter(struct ListaDupla* li, int pos) {
2     assert(li != NULL);
3     assert(pos >= 0 && pos < li->tamanho);
4     struct No* aux;
5
6     if (pos == 0) {
7        aux = li->inicio;
8     } else if (pos == li->tamanho - 1) {
9        aux = li->fim;
10    } else {
11       aux = li->inicio;
12       for(int i = 0; i < pos; i++) {
13          aux = aux->proximo;
14       }
15    }
16    return aux->info;
17 }
```

Fonte: elaborado pelo autor.

A diferença dessa função é que, agora, temos condições de acessar a última posição da lista sem ter que percorrê-la, como era feito na lista simplesmente ligada. Isso é feito acessando diretamente o ponteiro "*fim*" na lista, como mostra a linha 9.

Chegamos ao final desta seção. Você pode encontrar o código completo da lista duplamente ligada a seguir, utilizando a ferramenta Paiza.io.

Para aprimorar seus conhecimentos, não deixe de resolver os exercícios do final da seção, assim como a situação-problema apresentada a seguir.

Bons estudos!

FAÇA VALER A PENA

Questão 1

Um mecanismo utilizado para organizar nossa informação e prover operações convenientes e eficientes para acessá-la e manipulá-la é conhecido como estrutura de dados. Diversos tipos de estruturas de dados têm sido propostos e o conhecimento das características dessas estruturas é um fator importante para a escolha da estrutura que melhor se encaixa na solução de um determinado problema.

A respeito das estruturas de dados, analise as afirmativas a seguir:

I. A estrutura de dados lista representa um conjunto dinâmico, cujos elementos podem ser inseridos e retirados de qualquer parte da estrutura.

II. A estrutura de dados lista é uma estrutura de dados pouco flexível, em termos de liberdade para a manipulação de elementos.

III. Estruturas de dados implementadas com vetores alocam espaços na memória do computador à medida que novos elementos precisam ser adicionados e os desalocam à medida que não precisamos mais deles.

É correto o que se afirma em:

a. I, apenas.

b. I e II, apenas.

c. I e III, apenas.

d. II e III, apenas.

e. I, II e III.

Questão 2

Entre as principais operações que fazem parte de uma ED do tipo lista, estão:

- inserir(li, pos, item): adiciona o elemento “item” na posição “pos” da lista “li”.

Ver anotações

- remover(li, pos): remove e retorna o elemento da posição “pos” da lista “li”.
- obter(li, pos): retorna (sem remover) o elemento da posição “pos” da lista “li”.

Uma sequência de operações possíveis em uma lista é mostrada a seguir:

- inserir(li, 0, 9)
- inserir(li, 0, 5)
- obter(li, 1)
- inserir (li, 2, 10)
- remover(li, 1)
- inserir(li, 1, 9)

Após a execução, em ordem, da sequência de instruções em uma lista inicialmente vazia, o estado atual da lista será

a. Um erro de execução.

b. [5].

c. [5, 9].

d. [5, 9, 10].

e. [5, 10, 9].

Questão 3

O código a seguir apresenta uma versão simplificada da função “inserir” de uma lista simplesmente ligada, a qual insere elementos no apenas início da lista.

```
1  void inserir_inicio(struct Lista* li, int item) {
2     assert(li != NULL);
3     struct No* novo_no = (struct No*) malloc(sizeof(struct
   No));
4     novo_no->info = item;
5     ____________________________;
6     ____________________________;
7     li->tamanho++;
8  }
```

Assinale a alternativa que completa o que está faltando no código mostrado.

a.
li->inicio = novo_no;
novo_no->proximo = li->inicio

b.
novo_no->proximo = li->inicio

c.
novo_no->proximo = li->inicio
li->inicio = novo_no;

d.
li->inicio = novo_no;

e.
novo_no = li->inicio
li->inicio = novo_no;

REFERÊNCIAS

CORMEN, T. H. *et al*. **Algoritmos**: teoria e prática. Elsevier, 2012.

LINKED List Animation. [S. l.], 14 ago. 2017. (3 min. 11 seg.). Publicado pelo canal Technology Premiere. Disponível em: https://bit.ly/36juGFo. Acesso em: 16 jan. 2021.

THE HUXLEY. Página inicial. The Huxley, [s. d.]. Disponível em: https://www.thehuxley.com. Acesso em: 12 jan. 2021.

THE HUXLEY. **Unindo listas**. The Huxley, 2018. Disponível em: https://bit.ly/3agGIAw. Acesso em: 16 jan. 2021.

FOCO NO MERCADO DE TRABALHO

LISTAS

Paulo Afonso Parreira Júnior

LISTA SIMPLESMENTE LIGADA

Desenvolvimento de funções para a estrutura de dados do tipo lista simplesmente ligada com utilização de estruturas de repetição.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Você foi incumbido de desenvolver mais duas funções para a estrutura de dados do tipo lista simplesmente ligada, a saber, "*posicao_menor*" e "*existe*". Em cada uma delas, você precisará percorrer a lista ligada. Para isso, nós vimos que podemos usar a estrutura de repetição *for*. Isso será útil para a função "*posicao_menor*", pois temos que percorrer todos os elementos da lista para descobrir qual é a posição do menor elemento, ou seja, nós sabemos a priori quantas iterações serão realizadas.

Já para a função "*existe*", nós podemos usar a estrutura de repetição *while*, uma vez que não sabemos quantas iterações teremos que fazer para encontrar o elemento desejado, se é que ele existe.

Uma possível solução para cada função requisitada é apresentada a seguir:

Código 4.17 | Funções "*posicao_menor*" e "*existe*"

```c
bool existe(struct Lista* li, int item) {
  assert(li != NULL);
  struct No* aux = li->inicio;
  while(aux != NULL) {
   if (aux->info == item) {
     return true;
   }
   aux = aux->proximo;
  }
  return false;
}

int posicao_menor(struct Lista* li) {
  assert(li != NULL);
  struct No* aux = li->inicio;
  int pos_menor = 0, menor = aux->info;
  for(int i = 0; i < li->tamanho; i++) {
   if (aux->info < menor) {
     pos_menor = i;
     menor = aux->info;
   }
   aux = aux->proximo;
  }
  return pos_menor;
}
```

Fonte: elaborado pelo autor.

Para demonstrar o funcionamento da função "*posicao_menor*", foi pedido a você que imprimisse o nome do inimigo que está mais próximo do jogador em determinado momento (cabe lembrar que você trabalha em uma empresa desenvolvedora de jogos digitais). Considerando que os inimigos, bem como sua distância do jogador, estão armazenados em um vetor, uma possível solução para esse problema é o seguinte:

Código 4.18 | Imprimir nome do inimigo

```c
int main() {
  struct Inimigo i1, i2, i3;
  i1.nome = "Vampiro";
  i1.distanciaDoJogador = 10;

  i2.nome = "Morcego Assassino";
  i2.distanciaDoJogador = 2;

  i3.nome = "Zoombie";
  i3.distanciaDoJogador = 3;

  struct Inimigo inimigos[3] = {i1, i2, i3};

  struct Lista* lista_distancias = criar();
  inserir(lista_distancias, 0, i1.distanciaDoJogador);
  inserir(lista_distancias, 0, i2.distanciaDoJogador);
  inserir(lista_distancias, 0, i3.distanciaDoJogador);

  int posicaoMenor = posicao_menor(lista_distancias);

  printf("Inimigo mais próximo: %s\n",
inimigos[posicaoMenor].nome);

  return 0;
}
```

Fonte: elaborado pelo autor.

AVANÇANDO NA PRÁTICA

ORDENAR LISTA

Na situação-problema anterior, nós implementamos a função "*posicao_menor*", que retorna a posição do menor elemento da lista. Utilize essa função, juntamente com as já existentes na lista simplesmente ligada, para implementar a função "*ordernar(li)*", que

ordena a lista “li” em ordem crescente. Por exemplo, dada a lista [3, 2, 1, 4, 5], após a execução da função “ordenar”, a lista ficará assim: [1, 2, 3, 4, 5].

RESOLUÇÃO

Uma possível solução para essa situação problema pode-se ser vista a seguir:

Código 4.19 | Função “*ordenar(li)*”

```
1  void ordenar(struct Lista* li) {
2    assert(li != NULL);
3    struct Lista* lista_aux = criar();
4    int i = 0, pos_menor, elemento;
5    while(vazia(li) == false) {
6     pos_menor = posicao_menor(li);
7     elemento = remover(li, pos_menor);
8     inserir(lista_aux, i, elemento);
9     i++;
10   }
11   i = 0;
12   while(vazia(lista_aux) == false) {
13    elemento = remover(lista_aux, 0);
14    inserir(li, i, elemento);
15    i++;
16   }
17   liberar(lista_aux);
18 }
```

Fonte: elaborado pelo autor.

Para resolver a questão, inicialmente foi criada uma lista auxiliar, denominada “*lista_aux*”. O primeiro laço *while* remove os elementos da lista “*li*”, do menor para o maior, adicionando-os na lista “*lista_aux*”. Isso fará com que a lista “li” fique vazia e que a lista

"*lista_aux*" fique com todos os elementos em ordem crescente. Logo após, os elementos da lista "*lista_aux*" são copiados novamente para a lista "*li*". Por fim, a lista "*lista_aux*" é liberada da memória.

NÃO PODE FALTAR

PILHAS

Paulo Afonso Parreira Júnior

O QUE SÃO AS ESTRUTURAS DE DADOS DO TIPO PILHA?

A pilha representa um conjunto dinâmico cujos elementos são inseridos e retirados de acordo com o seguinte protocolo: o último elemento que entra no conjunto é o primeiro que sai, por exemplo, a função desfazer do seu editor de textos, que desfaz o último comando de uma pilha de comandos realizados no editor.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Você teve sucesso em sua primeira empreitada como desenvolvedor de uma empresa de jogos digitais e tem atuado criando soluções que ajudam os desenvolvedores a ser mais produtivos em suas tarefas. Dessa vez seu chefe lhe apresentou o seguinte problema: os dados de um jogador (nome, idade, recursos etc.) de determinado jogo que está sendo produzido pela empresa são armazenados em um arquivo de texto denominado *"playerData.txt"*. Antes de iniciar, o jogo lê esse arquivo e faz as configurações necessárias no sistema. Contudo, *bugs* estão ocorrendo em virtude de erros existentes nesse arquivo de configuração.

Alguns símbolos de agrupamento, como chaves e colchetes, são usados nesse arquivo para indicar as configurações do jogo. No exemplo a seguir, os colchetes são usados para indicar um conjunto de recursos que o jogador possui e as chaves são usadas para descrever cada tipo de recurso.

```
Jogador: {
  Nome: "José"
  Idade: 20
  Recursos: [
   { Nome: "Arma com laser", HP: 20 }
   { Nome: "Poção mágica", HP: 55 }
  ]
}
```

Uma questão importante a respeito do uso dos símbolos de agrupamento é descobrir se eles foram utilizados de forma correta, isto é, para cada símbolo de abertura deve haver um símbolo corresponder de fechamento. Além disso, esse símbolo de fechamento deve aparecer na ordem correta, "combinando" com seu símbolo de abertura equivalente.

Os exemplos a seguir ilustram o uso (correto e incorreto) de símbolos de agrupamento. O problema é decidir se uma expressão está ou não correta, com relação ao uso dos símbolos de agrupamento:

(()) – Correto

([]) – Correto

(() – Incorreto, falta um parêntese de fechamento

()) – Incorreto, falta um parêntese de abertura

([)) – Incorreto, o primeiro parênteses de fechamento não combina com o colchete de abertura.

Seu novo desafio como estagiário é escrever um programa capaz de receber um conjunto de caracteres de símbolos de agrupamento e decidir se esse conjunto de símbolos corresponde a uma expressão correta ou não. Assim, os desenvolvedores poderão verificar a consistência do arquivo de configuração do jogo antes de enviar uma nova versão para os jogadores.

CONCEITO-CHAVE

PILHA

Caro aluno, nesta seção você vai aprender mais uma ED, conhecida como pilha. A pilha representa um conjunto dinâmico cujos elementos são inseridos e retirados de acordo com o seguinte protocolo: o último elemento que entra no conjunto é o primeiro que sai. Esse protocolo é amplamente conhecido como LIFO, do inglês *Last in*, *First out* (último a entrar, primeiro a sair).

É possível inserir um elemento na pilha a qualquer momento, mas somente o elemento inserido mais recentemente pode ser removido a qualquer momento. O nome pilha deriva-se da metáfora de uma pilha de pratos em uma cantina ou restaurante (GOODRICH; TAMASSIA, 2013; CORMEN *et al.*, 2012).

A pilha é uma das estruturas de dados mais simples, apesar de estar entre as mais importantes, uma vez que são utilizadas em diversos tipos de situações, desde aplicações de escritório (o comando “desfazer” do editor de texto utiliza o conceito de pilha para armazenar a última alteração feita pelo usuário sobre uma parte do texto) até sistemas operacionais e compiladores (as chamadas a funções de um programa são armazenadas em uma pilha de execução). Há inclusive, operações de pilha implementadas em hardware, com o intuito de melhorar o desempenho das máquinas modernas.

As principais operações que devem estar presentes em uma ED do tipo pilha são:

- ***criar()***: cria e retorna uma pilha vazia.
- ***empilhar(p, item)***: também conhecida como *push*, essa função é responsável por empilhar um “item” no topo da pilha “p”.
- ***desempilhar(p)***: também conhecida como *pop*, essa função é responsável por desempilhar o elemento do topo da pilha “p” e retorná-lo.
- ***topo(p)***: retorna o elemento do topo da pilha “p”, sem retirá-lo da mesma pilha.
- ***tamanho(p)***: retorna a quantidade de elementos existentes na pilha “p”.
- ***vazia(p)***: retorna se a pilha “p” está vazia ou não.
- ***liberar(p)***: desaloca a memória ocupada pela pilha “p”.

O Quadro 4.2 apresenta uma série de operações e seus efeitos sobre uma pilha inicialmente vazia. Considere que o elemento mais à direita do conjunto de dados da pilha representa o topo da pilha.

Quadro 4.2 | Sequência de operações em uma ED pilha

Operação	Saída	Conteúdo da Pilha
criar()	p	()
empilhar(p, 5)	-	(5)
empilhar(p, 3)	-	(5 3)
desempilhar(p)	3	(5)
empilhar(p, 7)	-	(5 7)
desempilhar(p)	7	(5)
topo(p)	5	(5)
desempilhar (p)	5	()
desempilhar (p)	Erro	()
vazia(p)	True	()
empilhar (p, 9)	-	(9)
empilhar (p, 7)	-	(9 7)
tamanho(p)	2	(9 7)

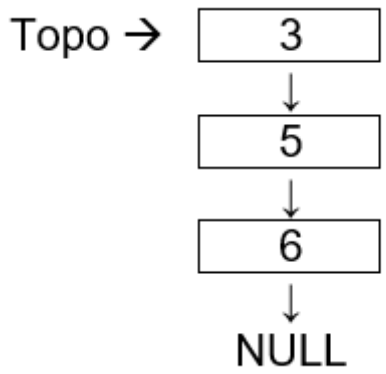
Fonte: elaborado pelo autor.

Assim como no caso da ED lista, podemos implementar uma pilha de pelo menos duas formas: **utilizando vetores** ou utilizando **estruturas ligadas (ou encadeadas)**. Nesta seção, será apresentada a implementação por meio de estruturas ligadas.

PILHA LIGADA

Observe a ilustração de uma pilha ligada, na Figura 4.7.

Figura 4.7 | Ilustração de uma pilha ligada



Fonte: elaborada pelo autor.

Uma pilha ligada consiste em uma sequência de elementos, denominados **nós** (representados por retângulos com uma seta em uma de suas extremidades na Figura 4.7). Cada nó tem uma informação (valor que aparece dentro do retângulo) e um ponteiro para o próximo nó da pilha (seta na extremidade direita do retângulo). O último nó da pilha (que está na sua base) aponta para **NULL**, representando que não existe um próximo nó na pilha. A pilha propriamente dita é representada apenas por um ponteiro para o nó do que está no topo da pilha, chamado de **Topo** na Figura 4.7.

> **REFLITA**
>
> Se em uma ED pilha nós podemos acessar apenas o nó que está em seu topo, como você faria para acessar os demais elementos da pilha sem perder os elementos?

Caro aluno, chegou a hora de você implementar a sua primeira **pilha ligada**. Você pode iniciar criando algumas *structs*. Vamos criar uma *struct* para representar cada nó da nossa pilha ligada e outra para representar a pilha propriamente dita. Mas, antes disso, vamos importar todas as bibliotecas que usaremos nos códigos desta seção.

```
#include <stdio.h>  // para operações de entrada e saída
#include <stdlib.h>  // para alocação dinâmica de memória
#include <stdbool.h> // para uso do tipo de dados “bool”
#include <assert.h>  // para uso da instrução “assert”
```

Feito isso, agora podemos observar o conteúdo das *structs* criadas para a pilha.

Ver anotações 0

```
struct No {
  int info;
  struct No* proximo;
};

struct Pilha {
  struct No* topo;
  int tamanho;
};
```

A *struct* "*No*" é idêntica à que foi criada para a lista simplesmente ligada.

A *struct* "*Pilha*" também é bastante similar, diferenciando apenas o nome da estrutura, bem como o nome do ponteiro para a *struct* "*No*", que aqui se chama "*topo*", indicando que ele aponta para o topo da pilha.

Assim como fizemos com a ED lista, partiremos para a implementação das funções da pilha, uma a uma, comentando uma delas ao longo do texto. No Código 4.20 encontra-se o código da função "criar".

Código 4.20 | Função para criar uma pilha vazia

```
1  struct Pilha* criar() {
2      struct Pilha* nova_pilha = (struct Pilha*)
   malloc(sizeof(struct Pilha));
3      if (nova_pilha != NULL) {
4          nova_pilha->topo = NULL;
5          nova_pilha->tamanho = 0;
6      }
7      return nova_pilha;
8  }
```

Fonte: elaborado pelo autor.

Essa função é responsável por instanciar dinamicamente uma variável do tipo *struct* "*Pilha*" na memória (linha 2) e configurar os valores de seus campos. Antes, porém, de configurar os valores dos campos da *struct* "*Pilha*", é preciso testar se a memória foi corretamente alocada para a pilha (linha 3). O campo "*topo*", que é um ponteiro, deve apontar para NULL, uma vez que estamos criando uma pilha vazia (linha 4). Analogamente, o campo "*tamanho*" deve ser inicializado com o valor igual a 0 (zero) – linha 5. Feito isso, deve-se retornar o endereço da memória alocado para a variável do tipo "*Pilha*" (linha 7). Com isso, nós já somos capazes de criar uma pilha vazia na memória do computador.

Passemos, então para a implementação das duas funções mais importantes da pilha, a saber "empilhar" e "desempilhar". O código dessas funções pode ser visualizado no Código 4.21.

Quadro 4.21 | Funções para empilhar e desempilhar elementos em/de uma pilha

```
1  void empilhar(struct Pilha* p, int item) {
2      assert(p != NULL);
3      struct No* novo_no = (struct No*) malloc(sizeof(struct
   No));
4      if (novo_no != NULL) {
5          novo_no->info = item;
6          novo_no->proximo = p->topo;
7          p->topo = novo_no;
8          p->tamanho++;
9      }
10 }
11 int desempilhar(struct Pilha* p) {
12     assert(p != NULL);
13     assert(p->topo != NULL);
14     struct No* aux = p->topo;
15     int elemento = aux->info;
16     p->topo = aux->proximo;
17     p->tamanho--;
18     free(aux);
19     return elemento;
20 }
```

Fonte: elaborado pelo autor.

Tanto a função "*empilhar*" quanto a "*desempilhar*" inicialmente verificam se o ponteiro para a pilha passada como parâmetro não é nulo (NULL) – linhas 2 e 12, respectivamente.

Feito isso, a função "*empilhar*" cria um novo nó dinamicamente, o que é feito na linha 3, por meio da função "*malloc*". Caso o novo nó tenha sido criado com sucesso, isto é, se o valor do ponteiro "*novo_no*" é diferente de NULL (linha 4), parte-se para o processo de empilhamento do nó. Como vimos, em uma pilha os nós são inseridos apenas em seu topo. Por isso, deve-se apontar o ponteiro "*proximo*" do novo nó para o topo da pilha (linha 6) e, depois, apontar o ponteiro "*topo*" da pilha para o

novo nó (linha 7). Esse procedimento fará com que o antigo topo da pilha passe a ser o segundo elemento dela e o novo nó passe a ocupar o topo da pilha. Por fim, incrementa-se o tamanho da pilha, na linha 8.

A função "*desempilhar*" faz o inverso da função "*empilhar*". Ou seja, ela remove o elemento que está no topo da pilha, fazendo com que o segundo elemento passe a ocupar o topo da mesma pilha. Para isso, inicialmente é verificado se a pilha não está vazia (linha 13).

Uma vez feito isso, cria-se um nó auxiliar que aponta para o topo de pilha (linha 14). Isso é necessário, pois como o ponteiro "topo" da pilha será atualizado, precisamos guardar a posição de memória do elemento que estava anteriormente no topo. Logo após, o valor do elemento que está no topo da pilha é armazenado na variável "elemento" (linha 15) e o ponteiro "topo" da lista é atualizado (linha 16). A partir de então, o topo da pilha passa a ser ocupado pelo segundo elemento, caso haja um. Caso contrário, o topo da pilha passa a ser igual a NULL, indicando que a pilha está vazia. Contudo, ainda restam algumas instruções a serem executadas, como decrementar o tamanho da pilha (linha 17), liberar a memória alocada para o nó que ocupava o topo da pilha anteriormente (linha 18) e retornar o valor armazenado na variável "*elemento*" (linha 19).

Com isso, já podemos começar a usar nossa pilha, cujo código completo se encontra a seguir, na ferramenta Paiza.io.

Para usar a ES pilha, crie uma função "*main*" em seu programa, com o código do Código 4.22.

Código 4.22 | Função que usa uma ED do tipo pilha

```c
int main() {
    struct Pilha* minha_pilha = criar();
    empilhar(minha_pilha, 1);
    empilhar(minha_pilha, 2);
    empilhar(minha_pilha, 3);

    printf("%d ", desempilhar(minha_pilha));
    printf("%d ", desempilhar(minha_pilha));
    printf("%d ", desempilhar(minha_pilha));

    return 0;
}
```

Fonte: elaborado pela autora.

Inicialmente, cria-se uma pilha vazia (linha 2). Logo após, os elementos 1, 2 e 3 são empilhados, nessa ordem, por meio da função “empilhar” (linhas 3 a 5). Por fim, todos os elementos da pilha são desempilhados e impressos na tela (linhas 7 a 9).

Teste o programa no Paiza.io, cuja saída esperada é a impressão dos números 3, 2 e 1, nessa ordem.

EXEMPLIFICANDO

A seguir temos um exemplo de algoritmos com ED pilha que seja capaz de imprimir uma sequência de 5 (cinco) números informados pelo usuário na ordem inversa da entrada, ou seja, de trás para frente. Por exemplo, dada a sequência [1, 2, 4, 0, 3], seu programa deve imprimir [3, 0, 4, 2, 1].

Uma possível solução para esse problema encontra-se a seguir. Inicialmente, criamos um laço *for* para ler os dados de entrada, adicionando-os em uma pilha. Depois, desempilhamos os elementos da pilha, imprimindo-os na tela. A própria característica da ED pilha fará com que os elementos sejam impressos na ordem inversa.

Código 4.24 | Pilha invertida

```c
int main() {
    struct Pilha* minha_pilha = criar();
    int num;

    for(int i = 0; i < 5; i++) {
        scanf("%d", &num);
        empilhar(minha_pilha, num);
    }

    for(int i = 0; i < 5; i++) {
        printf("%d ", desempilhar(minha_pilha));
    }
    return 0;
}
```

Fonte: elaborado pela autora.

O código completo pode ser visto a seguir, utilizando a ferramenta Paiza.io.

Agora podemos concluir a implementação da pilha com as funções que ainda restam, a saber "*tamanho*", "*topo*", "*vazia*" e "*liberar*". Vamos começar pelo código das funções "*tamanho*" e "*topo*" (Código 4.24).

Código 4.24 | Funções que retornam, respectivamente, o elemento do topo e a quantidade de elementos de uma pilha.

```
1  int topo(struct Pilha* p) {
2      assert(p != NULL);
3      assert(p->topo != NULL);
4      struct No* topo = p->topo;
5      return topo->info;
6  }
7  int tamanho(struct Pilha* p) {
8      assert(p != NULL);
9      return p->tamanho;
10 }
```

Ver anotações

Fonte: elaborado pelo autor.

Ambas as funções verificam se o ponteiro "*p*", passado como parâmetro, não é nulo (linhas 2 e 8). A função "*topo*", entretanto, também verifica se o ponteiro "*topo*" da pilha não é nulo (linha 3). Isso é importante, pois o valor do nó apontado por "*topo*" será lido e retornado pela função. Nas linhas 4 e 5 da função "*topo*", é criada uma variável auxiliar que aponta para o nó que está no topo da pilha (linha 4) e, depois, o valor armazenado nesse nó é retornado pela função (linha 5). Para a função "*tamanho*", após verificar se o ponteiro "*p*" não é nulo, o valor do campo "*tamanho*" da *struct* "*Pilha*" é retornado (linha 9).

ASSIMILE

A diferença entre as funções "topo" e "desempilha" é que a primeira retorna o elemento que está no topo da pilha, sem removê-lo, enquanto a segunda remove o elemento do topo e, posteriormente, retorna o seu valor.

Chegamos, então às últimas duas funções da ED pilha, "*vazia*" e "*liberar*", as quais são apresentadas no Código 4.25.

Código 4.25 | Funções que respectivamente verificam se uma pilha está vazia e libera a pilha da memória

```
1  bool vazia(struct Pilha* p) {
2      assert(p != NULL);
3      return (p->topo == NULL);
4  }
5
6  void liberar(struct Pilha* p) {
7      assert(p != NULL);
8      while(vazia(p) == false) {
9         desempilhar(p);
10     }
11     free(p);
12 }
```

Ver anotações 0

Fonte: elaborado pelo autor.

Novamente, ambas as funções verificam se o ponteiro "*p*", passado como parâmetro, não é nulo (linhas 2 e 7). Para checar se a pilha está vazia, a função "*vazia*" simplesmente compara o ponteiro "*topo*" da pilha a NULL (linha 3). O resultado dessa comparação lógica será retornado como resultado da linha, ou seja, se "*topo*" for igual a NULL, então a função "*vazia*" retornará "*true*", indicando que a pilha se encontra vazia. Caso contrário, a função retornará "*false*".

Quanto à função "*liberar*", ela faz uso da função "*vazia*", recém-apresentada. A lógica é simples: enquanto a pilha não estiver vazia (linha 8), isto é, enquanto o retorno da função "*vazia*" for igual a "*false*", então a função "*desempilhar*" deve ser invocada (linha 9). Isso é importante para que todos os elementos da ED sejam desalocados corretamente. Ao final, o espaço de memória reservado para a pilha também deve ser desalocado (linha 11).

Assim a nossa pilha está completa e funcional. Você pode acessar o código dela a seguir, utilizando a ferramenta Paiza.io.

EXEMPLIFICANDO

Analise o código a seguir, que utiliza a ED pilha que acabamos de implementar, e descreva qual será a saída impressa na tela.

Código 4.26 | Verifica pilha vazia

```
int main() {
    struct Pilha* minha_pilha = criar();

    printf("Está vazia (1 - SIM; 0 - NÃO)? %d\n", vazia(minha_pilha));

    printf("Empilhando 1... \n");
    empilhar(minha_pilha, 1);
    printf("Empilhando 2... \n");
    empilhar(minha_pilha, 2);
    printf("Empilhando 3... \n");
    empilhar(minha_pilha, 3);

    printf("Está vazia (1 - SIM; 0 - NÃO)? %d\n", vazia(minha_pilha));

    printf("Topo = %d\n", topo(minha_pilha));
    printf("Tamanho = %d\n", tamanho(minha_pilha));

    printf("Desempilhando elementos: ");
    printf("%d ", desempilhar(minha_pilha));
    printf("%d ", desempilhar(minha_pilha));
    printf("%d ", desempilhar(minha_pilha));

    liberar(minha_pilha);

    return 0;
}
```

Fonte: elaborado pelo autor.

Ver anotações 0

Inicialmente é criada uma pilha vazia, depois, verifica-se se a pilha está vazia. Como a pilha está vazia, a seguinte mensagem será impressa na tela:

Está vazia (1 - SIM; 0 - NÃO)? 1

Então, as seguintes mensagens são impressas, indicando que os elementos 1, 2 e 3 estão sendo empilhados:

Empilhando 1...

Empilhando 2...

Empilhando 3...

O status da pilha é novamente verificado e, como agora a pilha não está mais vazia, a mensagem será:

Está vazia (1 - SIM; 0 - NÃO)? 0

Por fim, a mensagem a seguir será exibida na tela, indicando que os elementos da pilha foram desempilhados:

Desempilhando elementos: 3 2 1

É possível observar que a ordem de impressão dos elementos é o inverso da ordem em que eles foram inseridos, comprovando que o funcionamento da pilha está correto.

FAÇA VALER A PENA

Questão 1

A pilha representa um conjunto dinâmico cujos elementos são inseridos e retirados de acordo com o seguinte protocolo: o último elemento que entra no conjunto é o primeiro que sai. Este protocolo é amplamente conhecido como LIFO, do inglês *Last in*, *first out* (último a entrar, primeiro a sair).

A respeito da pilha, analise cada uma das afirmativas a seguir:

I. É possível inserir um elemento na pilha a qualquer momento, mas somente o primeiro elemento inserido na pilha pode ser removido a qualquer momento.

II. As chamadas a funções de um programa são armazenadas em uma pilha de execução. Esse é um exemplo de uso de uma ED pilha.

III. Algumas das operações que devem estar presentes em uma ED do tipo pilha são: empilhar (insere um elemento no topo da pilha), desempilhar (remove o elemento do topo da pilha) e base (retorna, sem remover, o elemento localizado na base da pilha).

É correto o que se afirma em:

a. I e II, apenas.

b. I, apenas.

c. I e III, apenas.

d. II, apenas.

e. III, apenas.

Questão 2

Dentre as principais operações que fazem parte de uma ED do tipo pilha, estão:

- empilhar(p, item): insere o “item” no topo da pilha “p”.
- desempilhar(p): remove e retorna o elemento do topo da pilha “p”.
- topo(p): retorna (sem remover) o elemento do topo da pilha “p”.

A seguir há uma sequência de operações possíveis em uma lista.

➔ empilhar(p, 9)

➔ empilhar(p, 5)

➔ topo(p)

➔ empilhar(p, 10)

Ver anotações

➔ desempilhar(p)

➔ empilhar(p, 9)

Após a execução, em ordem, da sequência de instruções mostradas em uma pilha inicialmente vazia, o estado atual da pilha será:

a.
5 ← topo
9

b.
9 ← topo
5

c.
5 ← topo
9
10

d.
10 ← topo
9
5

e.
9 ← topo
5
9

Questão 3

O código a seguir apresenta a função “empilhar” de uma pilha ligada, a qual insere elementos no topo da pilha.

```
1  void empilhar(struct Pilha* p, int item) {
2     assert(p != NULL);
3     struct No* novo_no = (struct No*) malloc(sizeof(struct
   No));
4     if (novo_no != NULL) {
5         novo_no->info = item;
6         ____________________
7         ____________________
8         p->tamanho++;
9     }
10 }
```

Ver anotações

Assinale a alternativa que completa o as lacunas do código apresentado:

a.
p->topo = novo_no;
novo_no->proximo = p->topo;

b.
novo_no->proximo = p->topo;

c.
p->topo = novo_no;

d.
novo_no->proximo = p->topo;
p->topo = novo_no;

e.
O código já está completo e funcional

Ver anotações 0

REFERÊNCIAS

AULA 38 – Pilha: Definição. [*S. l.*], 10 dez. 2013. 1 vídeo (3 min. 43 s.). Publicado pelo canal Linguagem C Programação Descomplicada. Disponível em: https://bit.ly/2Yqd0Uq. Acesso em: 18 jan. 2021.

CORMEN, T. H. *et al*. **Algoritmos**: teoria e prática. São Paulo: Elsevier, 2012.

GOODRICH, M. T.; TAMASSIA, R. **Estruturas de dados & algoritmos em java**. 5. ed. Porto Alegre: Bookman Editora, 2013.

MILLER, B.; RANUM, D. Símbolos Balanceados. *In*: MILLER, B.; RANUM, D. **Resolução de problemas com algoritmos e estruturas de dados usando Python**. Símbolos Balanceados.Tradução de Andrew Toshiaki Nakayama Kurauchi, Carlos Eduardo Leão Elmadjian, Carlos Hitoshi Morimoto e José Coelho de Pina. IME-USP: [s. d.]. Disponível em: https://bit.ly/3t2oawy. Acesso em: 18 jan. 2021.

STACK using linked list animation. [*S. l.*], [s. d.]. 1 vídeo (1 min. 44 s.). Publicado pelo canal Acme Groups. Disponível em: https://bit.ly/3oyUe80. Acesso em: 18 jan. 2021.

THE HUXLEY. **Anagramas a partir de pilhas**. The Huxley, 18 ago. 2014. Disponível em: https://bit.ly/2M5a8tD. Acesso em: 18 jan. 2021.

THE HUXLEY. Página inicial. The Huxley, [s. d.]. Disponível em: https://www.thehuxley.com. Acesso em: 12 jan. 2021.

THE HUXLEY. **Pilhas com Listas**. The Huxley, 6 maio 2015. Disponível em: https://bit.ly/3tdzJBl. Acesso em: 18 jan. 2021.

FOCO NO MERCADO DE TRABALHO

PILHAS

Paulo Afonso Parreira Júnior

UTILIZAÇÃO DE PILHA PARA VERIFICAR A CONSISTÊNCIA DO ARQUIVO DE CONFIGURAÇÃO

Criação de um programa com a utilização de pilha, para verificar o arquivo com os dados de um jogador, quanto ao uso dos símbolos de agrupamentos, decidindo se eles estão corretos ou não.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Você foi contratado por ume empresa desenvolvedora de jogos digitais para escrever um programa que verifica se o arquivo com os dados de um jogador está correto ou não, quanto ao uso dos símbolos de

agrupamentos, tais como {, [,], }. Uma pilha pode ser convenientemente utilizada para resolver esse problema. Um exemplo de algoritmo capaz de avaliar uma expressão com os símbolos de agrupamento (), {} e [] e retornar "*true*" se ela é válida e "*false*" se é inválida, é apresentado no Código 4.27 a seguir.

O algoritmo funciona da seguinte forma: lendo a expressão da esquerda para a direita (isso é feito por meio do laço for da função "*valida*"), toda vez que se encontra um símbolo de abertura, insere-se esse símbolo na pilha (linhas 15, 16 e 17) e toda vez que se encontra um símbolo de fechamento, retira-se o símbolo do topo da pilha e verifica se os dois são do mesmo tipo (linhas 18 a 22). Caso sim, continue com o processo. Se a pilha estiver vazia após ter sido processada toda a sequência, então a expressão está correta.

As possíveis situações de erro, que indicam que a expressão é inválida, são:

- Quando tenta-se remover um símbolo da pilha e a pilha está vazia – linha 21 (ocorre quando há mais símbolos de fechamento do que de abertura).
- Quando um símbolo retirado da pilha não casa com o símbolo lido – linha 22 (ocorre quando tenta-se fechar um agrupamento com um símbolo diferentes daquele utilizado para abri-lo).
- Quando a pilha não está vazia após a leitura da expressão por completo – linha 26 (ocorre quando há mais símbolos de abertura do que de fechamento).

Código 4.27 | Algoritmo para verificação de dados de um jogador

```c
bool combina(char c1, char c2) {
    switch(c1) {
        case ')': return c2 == '(';
        case '}': return c2 == '{';
        case ']': return c2 == '[';
        default: return false;
    }
}

bool validar(char exp[], int tam) {
        struct Pilha* p = criar();
        for (int i = 0; i < tam; i++) {
            char c = exp[i];
            switch(c) {
                case '(':
                case '{':
                case '[': empilhar(p, c); break;
                case ')':
                case '}':
                case ']': {
                    if (vazia(p) == true) return false;
                    if (combina(c, desempilhar(p)) == false)
return false;
                }
            }
        }
        return (vazia(p));
}


int main() {
    char exp[] = "{([])}";
    printf("Resultado (1 = Correta; 0 = Incorreta): %d\n",
validar(exp, 6));

    return 0;
```

Ver anotações 0

```
35  }
```

Fonte: elaborado pelo autor.

AVANÇANDO NA PRÁTICA

CLONAR PILHA

Implemente uma nova função para a ED pilha, denominada "*clonar*". Essa função deve ser capaz de criar uma nova pilha, a partir de uma pilha já existente, clonando os elementos da pilha original para a nova pilha.

RESOLUÇÃO

Uma possível solução para o problema é apresentada a seguir. A ideia do algoritmo é a seguinte: inicialmente, cria-se duas pilhas, "*aux*" e "*clone*" (linhas 3 e 4). A pilha "*clone*" vai conter os elementos da pilha original e serão retornados no final da função (linha 17). A pilha "*aux*" serve como auxiliar para que os elementos da pilha original não sejam perdidos no processo de clonagem. Além disso, a pilha "*aux*" tem outra utilidade: garantir que os elementos da pilha original e da pilha clonada estejam na ordem correta. Isso acontece porque se simplesmente copiarmos os elementos de uma pilha para a outra, os elementos da pilha em que os elementos foram copiados estarão na ordem inversa. Após transferir todos os elementos da pilha passada por parâmetro para a pilha auxiliar (linhas 6 a 8), deve-se voltar os elementos para a pilha original "*p*" e para a pilha "*clone*", conforme pode ser visto nas linhas 10 a 14. Por fim, não se deve esquecer de liberar a pilha "*aux*", para que ela não fique ocupando espaço na memória do computador (lembre-se: toda memória alocada com a função "*malloc*" deve ser liberada pelo programador usando a função "*free*").

Código 4.28 | Pilha Clone

```c
struct Pilha* clonar(struct Pilha* p) {
    assert(p != NULL);
    struct Pilha* clone = criar();
    struct Pilha* aux = criar();

    while(vazia(p) == false) {
        empilhar(aux, desempilhar(p));
    }

    while(vazia(aux) == false) {
        int elemento = desempilhar(aux);
        empilhar(clone, elemento);
        empilhar(p, elemento);
    }

    liberar(aux);
    return clone;
}
```

Fonte: elaborado pelo autor.

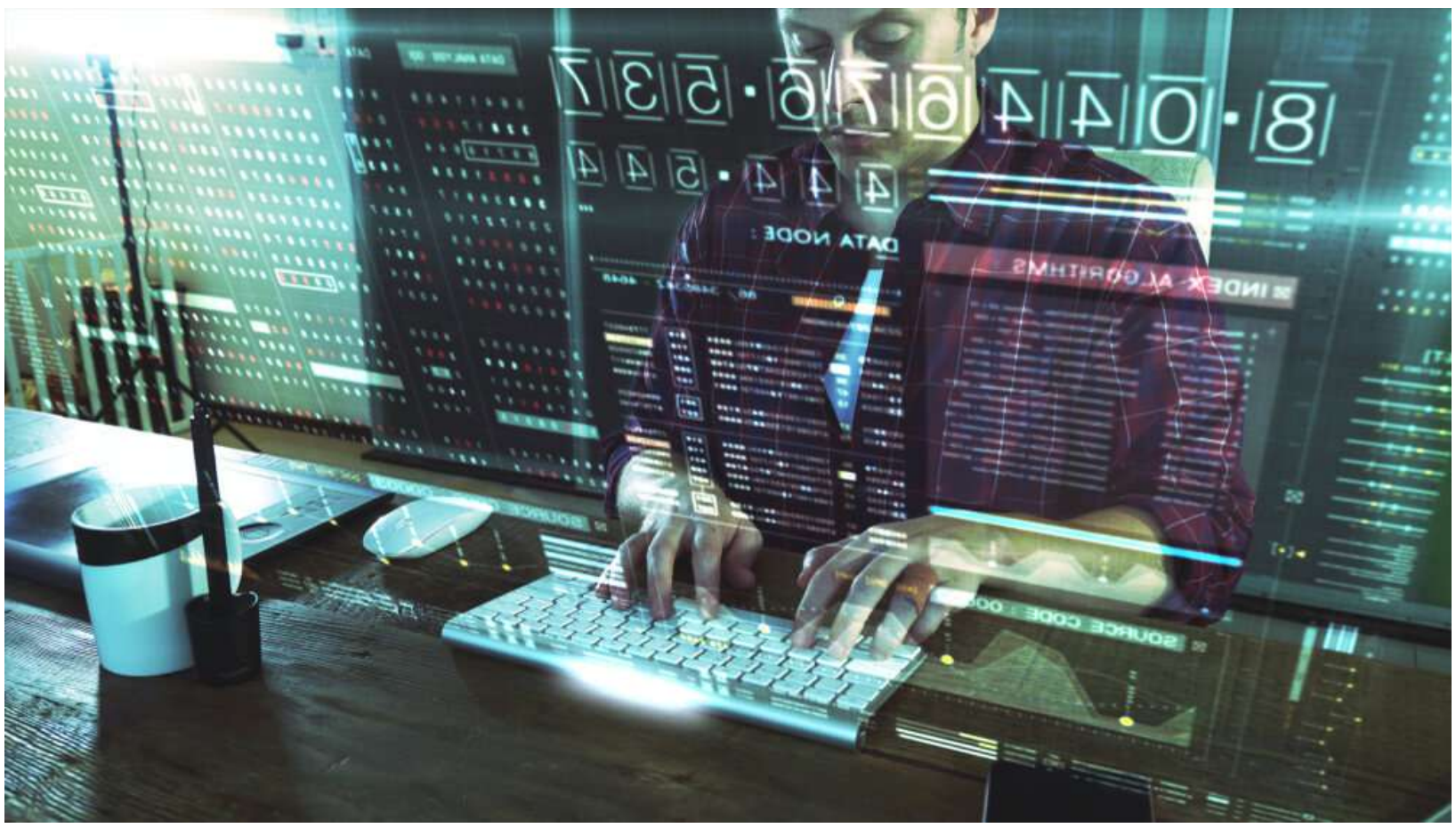
NÃO PODE FALTAR

FILAS

Paulo Afonso Parreira Júnior

O QUE SÃO AS ESTRUTURAS DE DADOS DO TIPO FILA?

A fila representa um conjunto dinâmico, cujos elementos são inseridos e retirados de acordo com o seguinte protocolo: o primeiro elemento que entra no conjunto é o primeiro que sai, assim como uma fila de pessoas em um banco.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Para colocarmos em prática os conhecimentos a serem aprendidos, vamos analisar a seguinte situação: você acaba de ser contratado pelo setor de TI (Tecnologia de Informação) de um grande banco. Ao chegar

no trabalho, seu supervisor lhe apresenta um problema pelo qual a agência está passando no momento.

A legislação em vigor requer que os bancos iniciem o atendimento a um cliente em no máximo 20 (vinte) minutos após a entrada do cliente na fila da agência. No banco em que você trabalha, a fila é única e há apenas um caixa ativo. Sabe-se ainda que um caixa não pode atender a mais de um cliente ao mesmo tempo. Para simplificar a solução do problema, seu chefe disse que, por enquanto, você pode ignorar o problema dos clientes prioritários, tais como idosos e gestantes.

Seu desafio será criar um programa que receberá o número de clientes e, para cada cliente, duas informações: o momento de entrada do cliente na fila e a duração do atendimento daquele cliente. De posse dessas informações, seu programa deve determinar o número de clientes que esperarão mais de 20 minutos para ter seu atendimento iniciado.

Suponha que 5 clientes procurarão a agência para atendimento. A seguir, é apresentado o momento em que cada cliente entrará na fila e o tempo necessário para atendê-lo, respectivamente.

Cliente 1: 0 e 10

Cliente 2: 0 e 10

Cliente 3: 1 e 10

Cliente 4: 2 e 10

Cliente 5: 30 e 10

Com base nesses dados, podemos ver, por exemplo, que os clientes 1 e 2 entrarão na fila da agência ao mesmo tempo e cada um tomará 10 minutos do caixa para ser atendido. A resposta esperada para o seu programa, neste caso, é 1. Ou seja, apenas um cliente terá que ficar esperando mais do que 20 minutos para ser atendido.

CONCEITO-CHAVE

Chegamos ao final de mais uma unidade. Até aqui, você aprendeu os recursos necessários para desenvolver soluções para problemas na linguagem de programação C. Lembre-se de que o mais importante não é se ater aos detalhes da linguagem de programação, mas aos conceitos fundamentais dos algoritmos, tais como atribuição, condicionais e laços, entre outros. É por meio deles que os problemas são efetivamente resolvidos. A linguagem nada mais é do que uma ferramenta para concretizar essa solução.

FILA

Nesta última seção, estudaremos a Estrutura de Dados conhecida como **fila**. A fila representa um conjunto dinâmico, cujos elementos são inseridos e retirados de acordo com o seguinte protocolo: o primeiro elemento que entra no conjunto é o primeiro que sai. Esse protocolo é amplamente conhecido como FIFO, do inglês *First in*, *First out* (primeiro a entrar, primeiro a sair). Ou seja, é possível inserir um elemento na fila a qualquer momento, mas somente o elemento que está na fila há mais tempo pode ser removido a qualquer momento. O nome fila deriva-se da metáfora de uma fila de pessoas em banco ou em um parque de diversões (GOODRICH; TAMASSIA, 2013).

A fila é outra estrutura de dados fundamental, que tem sido utilizada em diversos tipos de aplicações, desde comerciais (como em sistemas que controlam a chamada de pessoas para atendimento em um banco) até sistemas operacionais (uma das políticas mais simples de escalonamento de processos em um sistema operacional é baseada em uma fila simples).

As principais operações que devem estar presentes em uma ED do tipo fila são:

- ***criar()***: cria e retorna uma fila vazia.
- ***enfileirar(f, item)***: também conhecida como *enqueue*, enfileira o elemento "item" no final da fila "f".

- ***desenfileirar(f)***: também conhecida como *dequeue, desenfileira* o elemento do início da fila "f" e o retorna.
- ***inicio(f)***: retorna o elemento do início da fila "f", sem retirá-lo dela.
- ***tamanho(f)***: retorna a quantidade de elementos existentes na fila "f".
- ***vazia(f)***: retorna se a fila "f" está vazia ou não.
- ***liberar(f)***: desaloca a memória ocupada pela fila "f".

O Quadro 4.3 apresenta uma série de operações e seus efeitos sobre uma fila inicialmente vazia. Considere que o elemento mais à direita do conjunto de dados representa o final da fila e o elemento mais à esquerda, o início da fila (de onde os elementos são removidos).

Quadro 4.3 | Sequência de operações em uma ED fila

Operação	Saída	Conteúdo da Fila
criar()	f	()
enfileirar(f, 5)	-	(5)
enfileirar(f, 3)	-	(5 3)
desenfileirar(f)	5	(3)
enfileirar (f, 7)	-	(3 7)
desenfileirar(f)	3	(7)
inicio(f)	7	(7)
desenfileirar(f)	7	()
desenfileirar(f)	Erro	()
vazia(f)	True	()
enfileirar(f, 9)	-	(9)

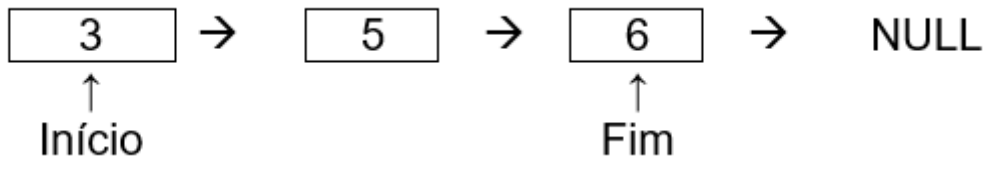
Operação	Saída	Conteúdo da Fila
enfileirar(f, 7)	-	(9 7)
tamanho(f)	2	(9 7)

Fonte: elaborado pelo autor.

Assim como no caso das EDs lista e pilha, podemos implementar uma fila de pelo menos duas formas: utilizando vetores ou utilizando estruturas ligadas (ou encadeadas). Nesta seção, será apresentada a implementação por meio de estruturas ligadas.

Observe a ilustração de uma fila ligada, na Figura 4.8.

Figura 4.8 | Ilustração de uma fila ligada



Fonte: elaborada pelo autor.

Uma fila ligada consiste em uma sequência de elementos denominados **nós** (representados por retângulos com uma seta em uma de suas extremidades na Figura 4.8). Cada nó contém uma informação (valor que aparece dentro do retângulo, na Figura 4.8) e um ponteiro para o próximo nó da fila (seta na extremidade direita do retângulo). O último nó da fila aponta para **NULL**, representando que não existe um próximo nó na fila. A fila, propriamente dita, é representada por dois ponteiros, um para o primeiro nó da fila, chamado de **Inicio** na Figura 4.8, e outro para o último nó da fila, chamado **Fim** da mesma figura.

E por que dois ponteiros? A resposta é que, diferentemente da lista simplesmente ligada e da pilha ligada, na fila os elementos são manipulados em ambas as extremidades. Por isso, é necessário um

ponteiro para o início e outro para o final da fila, permitindo, assim, a manipulação dos elementos sem a necessidade de percorrer toda a estrutura de dados.

> **ASSIMILE**
>
> Ao contrário da ED pilha, para a qual nós podemos ter apenas um ponteiro que aponta para o topo de pilha, no caso da fila nós precisamos de dois ponteiros, um para o início e outro para o final da fila. Isso faz sentido, pois o protocolo da fila é *First in*, *first out*, ou seja, nós inserimos elementos no final da fila e removemos os elementos do início dela.

FILA LIGADA

Chegou a hora de implementarmos nossa primeira **fila ligada**. Assim como fizemos nas seções anteriores, vamos começar criando algumas *structs*: uma para representar cada nó da nossa fila ligada e outra para representar a fila propriamente dita. Mas, antes disso, vamos importar todas as bibliotecas que usaremos nos códigos desta seção. Cada biblioteca dessas já foi usada e comentada ao longo do livro.

```
#include <stdio.h>  // para operações de entrada e saída
#include <stdlib.h>  // para alocação dinâmica de memória
#include <stdbool.h> // para uso do tipo de dados “bool”
#include <stdbool.h>  // para uso da instrução “assert”
```

Feito isso, agora podemos observar o conteúdo das *structs* criadas para a fila.

```
struct No {
  int info;
  struct No* proximo;
};


struct Fila {
```

Ver anotações 0

```
  struct No* inicio;
  struct No* fim;
  int tamanho;
};
```

A *struct* "No" é idêntica à que foi criada para a lista simplesmente ligada e para a pilha, vistas nas seções 1 e 2 desta unidade. A *struct* "*Fila*" também é bastante similar, diferenciando apenas o nome da estrutura e o nome dos ponteiros para a *struct* "*No*", que aqui se chamam "*inicio*" e "*fim*", indicando que eles apontam para o primeiro e para o último nós da fila.

Assim como fizemos com as EDs lista e pilha, partiremos agora para a implementação das funções da fila, uma a uma, comentando cada uma delas ao longo do texto. O Código 4.29 apresenta o código da função "*criar*".

Código 4.29 | Função que cria e retorna uma fila vazia

```
1  struct Fila* criar() {
2      struct Fila* nova_fila = (struct Fila*)
   malloc(sizeof(struct Fila));
3      if (nova_fila != NULL) {
4          nova_fila->inicio = NULL;
5          nova_fila->fim = NULL;
6          nova_fila->tamanho = 0;
7      }
8      return nova_fila;
9  }
```

Fonte: elaborado pelo autor.

O que essa função faz é instanciar dinamicamente uma variável do tipo *struct* "*Fila*" na memória (linha 2) e configurar os valores de seus campos. Antes, porém, de configurar os valores dos campos da *struct* "*Fila*", é preciso testar se a memória foi corretamente alocada

Ver anotações

para a fila (linha 3). Os campos "*inicio*" e "*fim*", que são ponteiros, devem apontar para NULL, uma vez que estamos criando uma fila vazia (linhas 4 e 5). Analogamente, o campo "*tamanho*" deve ser inicializado com o valor igual a 0 (zero) – linha 6. Feito isso, deve-se retornar o endereço da memória alocado para a variável do tipo "*Fila*" (linha 8). Com isso, nós já somos capazes de criar uma fila vazia na memória do computador.

Passemos, então para a implementação das duas funções mais importantes da fila, a saber "*enfileirar*" e "*desenfileirar*". O código dessas funções pode ser visualizado no Código 4.30 a seguir:

Código 4.30 | Funções que, respectivamente, inserem e removem um item em uma fila

```c
1  void enfileirar(struct Fila* f, int item) {
2      assert(f != NULL);
3      struct No* novo_no = (struct No*) malloc(sizeof(struct
   No));
4      if (novo_no != NULL) {
5         novo_no->info = item;
6         novo_no->proximo = NULL;
7
8          if (f->fim != NULL) {
9              f->fim->proximo = novo_no;
10       } else {
11            f->inicio = novo_no;
12       }
13       f->fim = novo_no;
14       f->tamanho++;
15     }
16 }
17
18 int desenfileirar(struct Fila* f) {
19     assert(f != NULL);
20     assert(f->inicio != NULL);
21     struct No* aux = f->inicio;
22     int elemento = aux->info;
23     f->inicio = aux->proximo;
24     if (f->inicio == NULL) {
25         f->fim = NULL;
26     }
27     f->tamanho--;
28     free(aux);
29     return elemento;
30 }
```

Fonte: elaborado pelo autor.

Para a função "*enfileirar*", inicialmente é feita uma verificação para garantir que o ponteiro passado por parâmetro não é nulo (linha 2). Logo após, é criado um novo nó dinamicamente e seu endereço de

memória é armazenado na variável “novo_no” (linha 3). Antes de continuar com a execução da função, verifica-se ainda se o novo nó foi alocado corretamente na memória do computador (linha 4). Caso sim, das linhas 5 a 15 realiza-se a inserção do novo nó no final da fila. Para isso, inicialmente, os campos “*info*” e “*proximo*” do novo nó são inicializados. Depois, verifica-se se o ponteiro “fim” da fila é igual a NULL. Isso é importante, pois caso o ponteiro “*fim*” seja diferente de nulo, isso significa que a fila não está vazia. Nesse caso, deve-se atualizar o ponteiro próximo do último nó para que ele aponte para o novo nó (linha 9). Caso contrário, a fila está vazia e trata-se da inserção do primeiro nó. Assim sendo, deve-se atualizar também o ponteiro início da fila para que ele aponte para o novo nó (linha 11). Logo após, o ponteiro “*fim* e o campo “*tamanho*” da fila são atualizados.

No caso da função “*desenfileirar*”, além de verificar se o ponteiro passado por parâmetro para a função não é nulo (linha 19), também verificamos se a fila não está vazia (linha 20), o que impediria a remoção de um nó dela. Logo após, cria-se um ponteiro auxiliar, denominado “*aux*”, que aponta para o início da fila (linha 21). Esse ponteiro será usado posteriormente para a correta desalocação da memória do nó removido, por meio da função “*free*” (linha 28). Com o ponteiro “*aux*” em mãos, fazemos com que o ponteiro “inicio” da fila aponte para o elemento da fila. Caso não haja outros elementos na fila, então o ponteiro “inicio” apontará para NULL. Assim, na linha 24 é verificado se isso acontece e, caso sim, o ponteiro “fim” também deve ser apontar para NULL, uma vez que a fila ficou vazia. Por fim, campo “*tamanho*” é decrementado (linha 27), a memória apontada pelo nó “*aux*” é desalocada (linha 28) e o valor do nó removido é retornado (linha 29).

> **REFLITA**
>
> Na vida real, existem aquele que “furam as filas”. E na programação, pode haver furo da fila? Por que ser tão rígido quanto aos protocolos?

No Código 4.31 serão apresentadas as demais funções da ED fila, a saber: "*vazia*", "*inicio*", "*tamanho*" e "*liberar*".

Código 4.31 | Funções utilizadas para verificar se uma fila está vazia, retornar o tamanho da fila, retornar o item do início da fila e liberar uma fila da memória, respectivamente

```
1  bool vazia(struct Fila* f) {
2      assert(f != NULL);
3      return (f->inicio == NULL);
4  }
5
6  int tamanho(struct Fila* f) {
7      assert(f != NULL);
8      return f->tamanho;
9  }
10
11 int inicio(struct Fila* f) {
12     assert(f != NULL);
13     assert(f->inicio != NULL);
14     return f->inicio->info;
15 }
16
17 void liberar(struct Fila* f) {
18     assert(f != NULL);
19     while(f->inicio != NULL) {
20         desenfileirar(f);
21     }
22     free(f);
23 }
```

Fonte: elaborado pelo autor.

As funções apresentadas no Código 4.31 verificam se o ponteiro para a fila, passado por parâmetro, não é igual a NULL (linhas 2, 7, 12 e 18). Além disso, a função "*inicio*" verifica também se a fila não está vazia, pois não é possível retornar a informação do primeiro nó de uma fila vazia (linha 13). O restante das funções são bastante parecidas com as

funções similares das EDs lista e pilha. Dois destaques a serem feitos aqui estão nas linhas 3 e 14. Na linha 3, a função "*vazia*" retorna o resultado da avaliação da expressão `f->inicio == NULL`. Essa é uma forma simplificada de escrever o código a seguir:

```
if (f->inicio == NULL) {
  return true;
else {
  return false;
}
```

Na linha 14, tem-se o acesso à informação de um nó da seguinte forma: `f->inicio->info`. Essa é uma forma simplificada (e mais eficiente, pois não se utiliza variáveis auxiliares) do código a seguir:

```
struct No* aux = f->inicio;
return aux->info;
```

O código completo da ED fila pode ser encontrado a seguir, na ferramenta Paiza.io.

> **EXEMPLIFICANDO**
>
> A seguir, no Código 4.32 é apresentado o código da função "*main*", que faz uso do código da ED fila, criado anteriormente. Analise esse código e pense em qual seria a sua saída.

Código 4.32 | Função "*main*" utilizando a estrutura de dados fila.

```c
int main() {
    struct Fila* minha_fila = criar();
    enfileirar(minha_fila, 1);
    enfileirar(minha_fila, 2);
    enfileirar(minha_fila, 3);
    printf("Tamanho: %d ", tamanho(minha_fila));

    printf("\nMinha fila: [%d ",
desenfileirar(minha_fila));
    printf("%d ", desenfileirar(minha_fila));
    printf("%d]", desenfileirar(minha_fila));

    printf("\nEstá vazia (1 - Sim, 0 - Não)? %d ",
vazia(minha_fila));

    liberar(minha_fila);

    return 0;
}
```

Fonte: elaborado pelo autor.

Nesse código é criada uma fila vazia (linha 2), na qual são enfileirados os elementos 1, 2 e 3 (linhas 3 a 5). Após, é impresso na tela o tamanho da fila (linha 6), usando a função “tamanho”. Nas linhas 8 a 10, os três elementos anteriormente inseridos na fila são desenfileirados e impressos na tela. Por fim, a situação da fila (se está vazia ou não) é impressa e a fila é liberada da memória do computador (linha 14). Assim, a saída do código anterior é:

```
Tamanho: 3
Minha fila: [1 2 3]
Está vazia (1 - Sim, 0 - Não)? 1
```

```
return aux->info;
```

Ver anotações 0

Você pode testar o código a seguir, utilizando a ferramenta Paiza.io.

FAÇA VALER A PENA

Questão 1

A fila representa um conjunto dinâmico, cujos elementos são inseridos e retirados de acordo com o seguinte protocolo: o primeiro elemento que entra no conjunto é o primeiro que sai. Esse protocolo é amplamente conhecido como FIFO, do inglês *First in*, *First out* (primeiro a entrar, primeiro a sair).

A respeito da fila, analise as afirmativas a seguir:

- É possível inserir um elemento na fila a qualquer momento, mas somente o elemento que está na fila há mais tempo pode ser removido a qualquer momento.
- O nome fila deriva-se da metáfora de uma lista de compras de supermercado.
- A fila tem sido utilizada em diversos tipos de aplicações, desde aplicações comerciais (como em sistemas que controlam a chamada de pessoas para atendimento em um banco) até sistemas operacionais (uma das políticas mais simples de escalonamento de processos em um sistema operacional é baseada em uma fila simples).

É correto o que se afirma em:

a. I, II e III.

b. I e II, apenas

c. I, apenas.

d. II, apenas.

e. I e III, apenas.

Questão 2

Ver anotações

Dentre as principais operações que fazem parte de uma ED do tipo fila, estão:

- enfileirar(f, item): insere o “item” no final da fila “f”.
- desenfileirar(f): remove e retorna o elemento do início da fila “f”.
- inicio(f): retorna (sem remover) o elemento do início da fila “f”.

A seguir, há uma sequência de operações possíveis em uma lista.

➔ enfileirar(f, 9)

➔ enfileirar(f, 5)

➔ inicio(f)

➔ enfileirar(f, 10)

➔ desenfileirar(f)

➔ enfileirar(f, 9)

Após a execução, em ordem, da sequência de instruções em uma fila inicialmente vazia, o estado atual da fila será:

a. início -> [9, 5, 10, 9] <- fim

b. início -> [10, 9] <- fim

c. início -> [5, 10, 9] <- fim

d. início -> [9, 10, 5] <- fim

e. início -> [9, 5, 9] <- fim

Questão 3

O código a seguir apresenta uma nova função da ED fila, denominada “desenfileirar_se_igual”, que remove um elementos do início da lista caso ele seja igual a um valor passado por parâmetro para a função.

Ver anotações 0

```
1  void desenfileirar_se_igual(struct Fila* f, int n) {
2      assert(f != NULL);
3      assert(f->inicio != NULL);
4      struct No* aux = f->inicio;
5      int elemento = aux->info;
6
7      if (_____________) {
8          f->inicio = aux->proximo;
9          if (f->inicio == NULL) {
10             _____________;
11         }
12         f->tamanho--;
13         free(aux);
14     }
15 }
```

Assinale a alternativa que completa o que está faltando no código apresentado.

a.
elemento == n
f->fim = NULL;

b.
elemento != n
f->fim = NULL;

c.
elemento == n
f->fim = aux->proximo;

d.
elemento != n
f->fim = aux->proximo;

e.
elemento == n
f->fim = aux;

REFERÊNCIAS

Ver anotações

AULA 31 – Fila: Definição. [S. l.], 3 nov. 2013. 1 vídeo (3 min. 43 s.). Publicado pelo canal Linguagem C Programação Descomplicada. Disponível em: https://bit.ly/3tariXs. Acesso em: 19 jan. 2021.

BEGIN, T.; BRANDWAJN, A. **Solutions to Queueing Systems**. [*S. l.*], jan. 2015. Disponível em: https://bit.ly/2YqB8q0. Acesso em: 19 jan. 2021.

BIANCHI, F. **Estrutura de Dados e Técnicas de Programação**. Grupo GEN, 2014. 9788595152588. Disponível em: https://bit.ly/39r5pLy. Acesso em: 18 jan. 2021.

GOODRICH, M. T.; TAMASSIA, R. **Estruturas de dados & algoritmos em java**. 5. ed. Porto Alegre: Bookman Editora, 2013.

UNIVERSITY OF SAN FRANCISCO. **Queue (Linked List Implementation)**. USF Computer Science, [s. d.]. Disponível em: https://bit.ly/3ovsGAj. Acesso em: 19 jan. 2021.

VISUALGO. **Linked list**. Visuago.net, [s. d.]. Disponível em: https://visualgo.net/en/list. Acesso em: 19 jan. 2021.

FOCO NO MERCADO DE TRABALHO

FILAS

Paulo Afonso Parreira Júnior

TEMPO DE ESPERA NA FILA DO BANCO

Criação de um programação com a utilização de fila, para descobrir quantos clientes de uma agência bancária teriam que esperar mais do que 20 minutos na fila do banco.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

Você foi encarregado de resolver um problema em seu novo emprego, que consiste em criar um programa para descobrir quantos clientes de uma agência bancária teriam que esperar mais do que 20 minutos na fila do banco, contrariando, assim, a nova legislação brasileira. Para isso, você tem como informações a quantidade de clientes e, para cada cliente, o momento que ele entrou na fila o tempo necessário para seu atendimento.

Para resolver esse problema, vamos iniciar lendo o número de clientes que entrarão na agência. Logo depois, você pode adicionar suas informações, uma a uma, em duas filas. Na primeira fila, armazenaremos o tempo em que o cliente entrou na fila. Na segunda, será armazenado o tempo necessário para seu atendimento. Observe parte da solução no Código 4.33 a seguir.

Código 4.33 | Solução para descobrir o número de clientes que precisam esperar mais que 20 minutos.

```c
int main() {
    int numClientes, tEntrada, tAtendimento;
    printf("Informe o número de clientes: ");
    scanf("%d", &numClientes);

    struct Fila* filaTEntrada = criar();
    struct Fila* filaTAtendimento = criar();

    for(int i = 0; i < numClientes; i++) {
        printf("\nInforme o tempo em que o cliente %d entrou na fila: ", i + 1);
        scanf("%d", &tEntrada);
        enfileirar(filaTEntrada, tEntrada);

        printf("\nInforme o tempo de atendimento do cliente %d: ", i + 1);
        scanf("%d", &tAtendimento);
        enfileirar(filaTEntrada, tAtendimento);

        printf("\n");
    }
    return 0;
}
```

Fonte: elaborado pelo autor.

Após isso, vamos criar uma função que recebe as duas filas como parâmetros e retorna um número inteiro, representando a quantidade de clientes que terão que esperar mais do que 20 minutos na fila.

Código 4.34 | Função que recebe duas filas.

```
1   int calcularClientesEmEsperaMaxima(struct Fila* filaTEntrada,
    struct Fila* filaTAtendimento) {
2
3       int esperaTotal = 0;
4       int clientesEmEsperaMaxima = 0;
5
6       while(!vazia(filaTEntrada)) {
7           int tEntrada = desenfileirar(filaTEntrada);
8           int tAtendimento = desenfileirar(filaTAtendimento);
9
10          if (esperaTotal - tEntrada > 20) {
11              clientesEmEsperaMaxima += 1;
12          }
13
14          esperaTotal += tAtendimento;
15      }
16
17      return clientesEmEsperaMaxima;
18  }
```

Fonte: elaborado pelo autor.

Entre as linhas 6 e 15, os elementos das filas são removidos e o seguinte cálculo é feito: para cada cliente, subtrai-se o tempo de espera total (que é acumulado na linha 14, somando-se o tempo de atendimento de cada cliente) do tempo de entrada do cliente na fila. Por exemplo: supondo que um cliente seja o quarto a ser atendido e que os três antes dele apresentam tempo de atendimento de 10 minutos, ele teria que ficar esperando por 30 minutos, caso tenha chegado no minuto 0. Supondo, entretanto, que ele tenha chegado no minuto 10, seu tempo de espera será de 20 minutos.

Então, nas linhas 10 a 12, verificamos se o tempo de espera do cliente será maior do que 20 minutos. Caso sim, o valor da variável "*cientesEmEsperaMaxima*" é incrementado de 1. Ao final, o valor dessa variável é retornado para quem chamou a função (linha 17).

O código completo, incluindo a chamada na função "*calcularClientesEmEsperaMaxima*" é apresentado a seguir:

Código 4.35 | Função calcularClientesEmEsperaMaxima

```c
int calcularClientesEmEsperaMaxima(struct Fila* filaTEntrada, struct Fila* filaTAtendimento) {

    int esperaTotal = 0;
    int clientesEmEsperaMaxima = 0;

    while(!vazia(filaTEntrada)) {
        int tEntrada = desenfileirar(filaTEntrada);
        int tAtendimento = desenfileirar(filaTAtendimento);

        if (esperaTotal - tEntrada > 20) {
            clientesEmEsperaMaxima += 1;
        }

        esperaTotal += tAtendimento;
    }

    return clientesEmEsperaMaxima;
}

int main() {
    int numClientes, tEntrada, tAtendimento, clientesEmEsperaMaxima;

    printf("Informe o número de clientes: ");
    scanf("%d", &numClientes);

    struct Fila* filaTEntrada = criar();
    struct Fila* filaTAtendimento = criar();

    for(int i = 0; i < numClientes; i++) {
        printf("\nInforme o tempo em que o cliente %d entrou na fila: ", i + 1);
        scanf("%d", &tEntrada);
        enfileirar(filaTEntrada, tEntrada);

```

Ver anotações

```
34          printf("\nInforme o tempo de atendimento do cliente
%d: ", i + 1);
35          scanf("%d", &tAtendimento);
36          enfileirar(filaTAtendimento, tAtendimento);
37
38          printf("\n");
39      }
40
41      clientesEmEsperaMaxima =
calculaClientesEmEsperaMaxima(filaTEntrada,
filaTAtendimento);
42      printf("\nNeste dia, %d cliente(s) esperarão mais do que
20 minutos na fila.", clientesEmEsperaMaxima);
43
44      return 0;
45  }
```

Fonte: elaborado pelo autor.

Você pode testar a solução completa utilizando a ferramenta Paiza.io.

AVANÇANDO NA PRÁTICA

O PROBLEMA DE JOSEPHUS

São inúmeras as aplicações da ED Fila. Nesta seção, você vai conhecer o problema clássico conhecido como o problema de Josephus. Esse problema postula um grupo de soldados circundados por uma força inimiga esmagadora. Não há esperanças de vitória sem a chegada de reforços e existe somente um cavalo disponível para escapar. Os soldados entram em um acordo para determinar qual deles deverá escapar e trazer ajuda. Eles formam um círculo e um número N é sorteado em um chapéu. Um de seus nomes também é sorteado. Iniciando pelo soldado cujo nome foi sorteado, eles começam a contar ao longo do círculo em sentido horário. Quando a contagem alcança N,

esse soldado é retirado do círculo e a contagem reinicia com o soldado seguinte. O processo continua de maneira que, toda vez que N é alcançado, outro soldado é retirado do círculo. Todo soldado retirado do círculo não entra mais na contagem. O último soldado que restar deverá montar no cavalo e escapar. Com base nesse contexto, crie uma função que, dada uma fila de soldados e um número N, o retorne o soldado que escapará.

RESOLUÇÃO

Você foi encarregado de resolver o tradicional problema de Josephus. Na prática, você deve criar uma função que, dada uma fila de soldados (representados por números inteiros) e um número inteiro N, sua função retorne o número do soldado que escapará para pedir ajuda contra o exército inimigo.

Utilizando a ED, a situação-problema pode ser resolvida da seguinte forma:

Código 4.36 | O problema de Josephus

```c
int josephus(struct Fila* f, int n) {
    while(tamanho(f) > 1) {
        for(int i = 0; i < n; i++) {
            enfileirar(f, desenfileirar(f));
        }
        desenfileirar(f);
    }
    return desenfileirar(f);
}

int main() {
    struct Fila* minha_fila = criar();
    enfileirar(minha_fila, 1);
    enfileirar(minha_fila, 2);
    enfileirar(minha_fila, 3);
    enfileirar(minha_fila, 3);
    printf("Soldado salvo: %d ", josephus(minha_fila, 3));
    liberar(minha_fila);

    return 0;
}
```

Fonte: elaborado pela autora.

Teste o código utilizando a ferramenta Paiza.io.

A função "*josephus*" recebe uma fila "*f*", contendo todos os soldados, e um número inteiro "*n*". A partir disso, cria-se um laço *while* que irá iterar enquanto o tamanho da fila for maior do que 1 (linha 2). Isso é feito porque a ideia do problema é ir removendo os soldados um a um até que sobre apenas 1. Uma vez terminada a execução do laço *while*, sabe-se que resta apenas um soldado na fila, a saber, aquele que será salvo. Portanto, na linha 8 remove e retorna-se o valor do único elemento da fila como resultado do problema de *Josephus*.

Dentro do laço *while*, há um laço *for* (linha 3), que é responsável por iterar na fila até encontrar o elemento que será removido. Esse laço fará sempre “n” iterações e, em cada uma delas, um elemento será removido da fila e reinserido nela. Como o protocolo da fila é FIFO, então cada elemento será retirado do início e inserido no final dela. Assim que o contato “i” do laço *for* atingir o valor igual a “n”, o laço é interrompido. Isso indica que o soldado a ser retirado encontra-se no início da fila. Então, basta removê-lo, conforme mostra a linha 6.

A função “”*main*” simplesmente cria uma lista vazia (12) e insere 4 elementos nela (linhas 13 a 16). Após isso, é invocada a função “*josephus*”, passando a fila criada como parâmetro e o valor de “n” igual a 3. A saída esperada desse código é “*Soldado salvo: 2*”.