

Algoritmos e Programação Estruturada

Algoritmos e Programação Estruturada

Kleber Ricardi Rovai
Marcio Aparecido Artero
Vanessa Cadan Scheffer

© 2020 por Editora e Distribuidora Educacional S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidência Rodrigo Galindo	Gerência Editorial Fernanda Miglioranza
Vice-Presidência de Produto, Gestão e Expansão Julia Gonçalves	Editoração Gráfica e Eletrônica Renata Galdino Luana Mercurio
Vice-Presidência Acadêmica Marcos Lemos	Supervisão da Disciplina Marcilyanne Moreira Gois
Diretoria de Produção e Responsabilidade Social Camilla Veiga	Revisão Técnica Marcilyanne Moreira Gois Marcio Aparecido Artero

Imagens

Adaptadas de Shutterstock.

Todos os esforços foram empregados para localizar os detentores dos direitos autorais das imagens reproduzidas neste livro; qualquer eventual omissão será corrigida em futuras edições.

Conteúdo em websites

Os endereços de websites listados neste livro podem ser alterados ou desativados a qualquer momento pelos seus mantenedores. Sendo assim, a Editora não se responsabiliza pelo conteúdo de terceiros.

Dados Internacionais de Catalogação na Publicação (CIP)

Rovai, Kleber Ricardi

R873a Algoritmos e programação estruturada / Kleber Ricardi Rovai,
Vanessa Cadan Scheffer, Marcio Aparecido Artero. - Londrina : Editora e
Distribuidora Educacional S.A., 2020.

248 p.

ISBN 978-85-522-1682-7

1. Programação Estruturada. I. Scheffer, Vanessa Cadan.
II. Artero, Marcio Aparecido. III. Título.

CDD 005.113

Jorge Eduardo de Almeida CRB-8/8753

2020

Editora e Distribuidora Educacional S.A.

Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>

Sumário

Unidade 1

Fundamentos de algoritmos e das linguagens de programação	7
---	---

Seção 1

Conceitos de algoritmos e programação	9
---	---

Seção 2

Componentes e elementos de linguagem de programação	39
---	----

Seção 3

Operações e expressões.....	61
-----------------------------	----

Unidade 2

Estruturas de decisão e repetição	77
---	----

Seção 1

Estruturas de decisão condicional.....	78
--	----

Seção 2

Estruturas de repetição condicional.....	96
--	----

Seção 3

Estruturas de repetição determinísticas	111
---	-----

Unidade 3

Funções e recursividade	129
-------------------------------	-----

Seção 1

Procedimentos e funções.....	130
------------------------------	-----

Seção 2

Escopo e passagem de parâmetros	144
---------------------------------------	-----

Seção 3

Recursividade.....	160
--------------------	-----

Unidade 4

Estrutura de dados	177
--------------------------	-----

Seção 1

Listas.....	178
-------------	-----

Seção 2

Pilhas	209
--------------	-----

Seção 3

Filas.....	227
------------	-----

Palavras do autor

Caro aluno, bem-vindo ao estudo dos algoritmos e da programação estruturada. Esse livro representa um marco na sua caminhada, pois ao apropriar-se dos conhecimentos que aqui 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*), desde sempre, 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 desse 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 permitirão a você organizar e otimizar seu programa, por meio das estruturas de dados com listas, filas e pilhas.

Nossa 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 complexos. Para que possamos começar nosso trabalho, convido você a se desafiar de forma que alcance excelentes resultados e seja um profissional diferenciado nessa área de grandes oportunidades.

Bons estudos!

Unidade 1

Marcio Aparecido Artero
Vanessa Cadan Scheffer

Fundamentos de algoritmos e das linguagens de programação

Convite ao estudo

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

Não tem como negar: 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 execução das mais diversas tarefas computacionais.

Nesta unidade, você terá o prazer em conhecer uma empresa de tecnologia de informação cujo foco principal é o desenvolvimento de *software* 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 então atender à demanda de mercado. Você, sendo um profissional exemplar na área de tecnologia da informação, foi incumbido de treinar os estagiários.

A empresa não exigiu nenhuma experiência para os candidatos e, por este motivo, você deverá apresentar as definições e aplicações dos algoritmos, as inovações e os diferentes paradigmas para 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 parte da 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 de linguagem de programação e você terá a oportunidade de aprender as variáveis e a introdução à prática de programação. Na terceira seção, serão apresentadas as operações e expressões para um programa de computador. Muito bem, essa é a proposta, você aceita o desafio?

Sigamos em frente.

Seção 1

Conceitos de algoritmos e programação

Diálogo aberto

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

Você inicia agora o seu trabalho em uma empresa responsável por criar softwares educacionais. Assim, você foi incumbido de treinar os estagiários contratados para trabalhar com desenvolvimento de sistemas em empresa, lembrando que eles não têm qualquer experiência com algoritmos ou linguagens de programação.

Para provocar a curiosidade nos estagiários, construa um algoritmo que realize o cadastro dos dados pessoais dos alunos que deverão ser cadastrados: nome, endereço, cidade e estado. Após o cadastro dos dados pessoais ser realizado, é necessário mostrar (print) os dados na tela do computador com o resultado do cadastro. O algoritmo deverá ser elaborado nas formas descritas a seguir:

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

De posse dos resultados, os algoritmos podem contribuir de forma significativa na elaboração de códigos de programação.

Agora chegou o momento de iniciar essa jornada, muita atenção às aulas e um ótimo estudo!

Não pode faltar

Olá! 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. 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, a partir da construção de algoritmos para a resolução de algum problema, é possível traduzir o algoritmo para alguma linguagem de programação.

Como mencionado, para qualquer tarefa a ser executada no dia a dia, podemos desenvolver um algoritmo. Por exemplo: a sequência de passos para o cozimento de um arroz, conforme a seguir:

1. Acender o fogo; 2. Refogar os temperos; 3. Colocar o arroz na panela;
4. Colocar a água; 5. Cozinhar o arroz; 6. Abaixar o fogo; 7. Esperar o ponto;
8. Desligar o fogo; 9. 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. Aguardar a água secar; 11. Baixar o fogo; 12. Fechar a panela com a tampa; 13. Aguardar o ponto; 14. Desligar o fogo; 15. Servir o arroz.

Observe que não existe somente uma forma de elaborar um algoritmo, porém, existe uma sequência lógica para a execução da tarefa, por exemplo, 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 obter o objetivo determinado, o qual será sempre o mesmo.

Para melhor entendimento dos algoritmos, é necessário dividi-los em três partes: entrada, processamento e saída.

- **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 que o algoritmo quer chegar após o processamento dos dados de entrada, finalização do arroz (momento que será 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.

Linguagem natural

Segundo Santos (2001), a linguagem natural é uma forma de comunicação entre as pessoas de diversas línguas, ela pode ser falada, escrita, gestualizada entre outras formas de comunicação. A linguagem natural tem uma grande contribuição quando vamos desenvolver uma aplicação computacional, pois pode direcionar de forma simples e eficiente as descrições dos problemas e suas soluções.

Para reforçar os conceitos de linguagem natural, podemos considerar o cadastro das notas de alguns alunos do seu curso. O problema é o seguinte: o usuário deverá entrar com dois valores que representam as notas de cada bimestre e o computador retornará o resultado da média destes valores (média das notas), se a média for maior ou igual a seis, o aluno será aprovado, senão estará reprovado.

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

1. Início.
2. Entrar com o primeiro valor (nota do primeiro bimestre).
3. Entrar com o segundo valor (nota do segundo bimestre).
4. Realizar a soma do primeiro valor com o segundo.
5. Realizar a divisão da soma dos valores por dois (média das notas dos bimestres).
6. Armazenar o valor encontrado.
7. Mostrar na tela o resultado da média.
8. Se a média do aluno for maior ou igual a seis.
9. O aluno será considerado aprovado.
10. Senão está reprovado.
11. Fim.

Um outro exemplo de algoritmo que vale a pena citar de acordo com Piva Junior (2012) é o algoritmo euclidiano, Euclides, usando de sua sabedoria, criou um algoritmo para calcular o máximo divisor comum, o famoso “MDC” no qual pode ser resumida da seguinte forma:

1. Dividir um número “a” por “b”, onde o resto é representado por “r”.
2. Substituir a por b.
3. Substituir b por r.
4. Continuar a divisão de a por b até que um não possa ser mais dividido, então “a” é considerado o mdc.

Veja na Tabela 1.1 a solução do “MDC” do algoritmo acima:

Tabela 1.1 | Cálculo do MDC (480, 130)

a	b	R
480	130	90
130	90	40
90	40	10
40	10	0
10	0	

Fonte: adaptada de Piva Junior (2012).

Nesse caso, o resultado fica: MDC (480,130) = 10.

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

Antes de iniciar a explicação sobre o 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. Vamos lá?

As variáveis, como o próprio nome sugere, consistem em algo que pode sofrer variações, ou seja, estão relacionadas à identificação de uma informação e atribuição (\leftarrow), que tem a função de indicar valores para as variáveis, ou seja, atribuir informação para variável. Por exemplo:

```
valor1 ← 8
```

```
nome ← marcio
```

Significa que a o número “8” está sendo atribuído para variável “valor1” e que o texto “marcio” está atribuído para variável “nome”.

Muito bem, dando sequência ao seu estudo de algoritmos, veja agora o funcionamento dos diagramas de blocos que também pode ser descrito como fluxograma.

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 que os símbolos utilizados devem estar em harmonia e serem de fácil entendimento. Para que os diagramas de blocos tenham certa coerência, os seus símbolos foram padronizados pela ANSI (Instituto Norte Americano de Padronização) conforme mostra o Quadro 1.1.

Veja agora a definição dos principais símbolos utilizados em um diagrama de blocos:

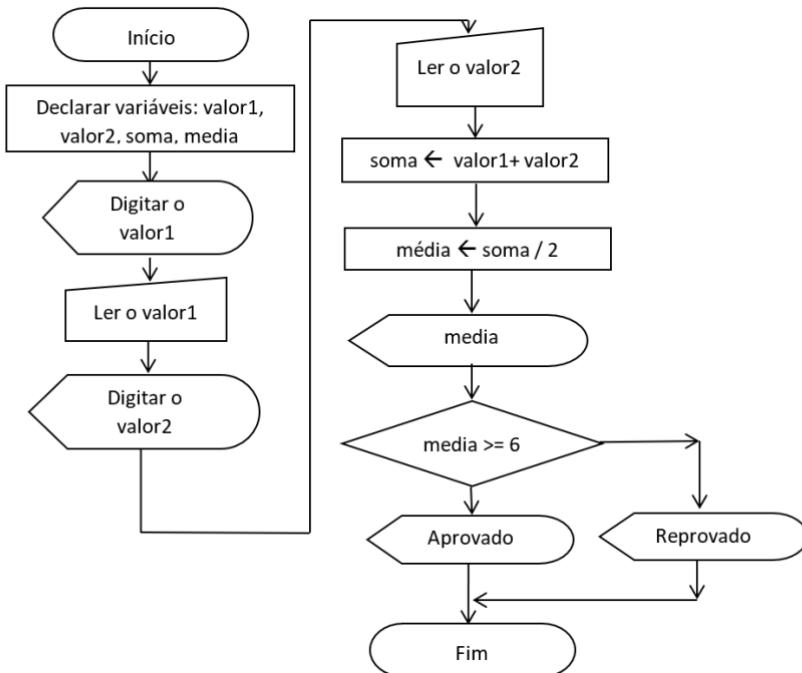
Quadro 1.1 | Descrição e significados de símbolos no diagrama de blocos

Símbolo	Significado	Descrição
	Terminal	Representa o início ou o fim de um fluxo lógico. Em alguns casos define as sub-rotinas.
	Entrada manual	Determina a entrada manual dos dados, geralmente através de um teclado.
	Processamento	Representa a execução de ações de processamento.
	Exibição	Mostra o resultado de uma ação, geralmente através da tela de um computador.
	Decisão	Representa os desvios condicionais nas operações de tomada de decisão e laços condicionais para repetição de alguns trechos do programa.
	Preparação	Representa a execução de um laço incondicional que permite a modificação de instruções do laço.
	Processo predefinido	Define um grupo de operações relacionadas a uma sub-rotina.
	Conector	Representa pontos de conexões entre trechos de programas, que podem ser apontados para outras partes do diagrama de bloco.
	Linha	Representa os vínculos existentes entre os símbolos de um diagrama de blocos.

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

A partir do momento que você utilizar os símbolos com as suas instruções, você vai aprendendo e desenvolvendo cada vez mais a sua lógica em relação aos problemas. O exemplo do Diagrama 1.1 traz a solução de um algoritmo utilizando diagrama de blocos:

Diagrama 1.1 | Diagrama de blocos (fluxograma)



Fonte: elaborado pelo autor.

Assimile

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

1. Estar atento aos níveis.
2. O diagrama de blocos (fluxograma) deve começar de cima para baixo e da esquerda para direita.
3. Ficar atento para não cruzar as linhas, principalmente as linhas de fluxos de dados.

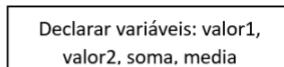
O diagrama de blocos apresentado mostra a execução da média de dois valores.

Vejamos então as representações de cada passo do diagrama:

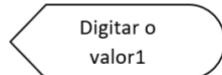
O símbolo terminal deu início ao diagrama de blocos.



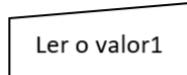
O símbolo de processamento definiu as variáveis.



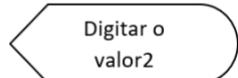
O símbolo exibição mostra na tela o que o usuário deve fazer.



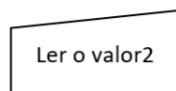
O símbolo de entrada manual libera para o usuário entrar com o primeiro valor.



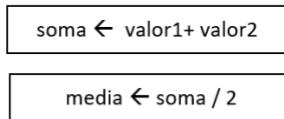
O símbolo exibição mostra na tela o que o usuário deve fazer.



O símbolo de entrada manual libera para o usuário entrar com o segundo valor.



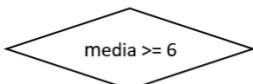
O símbolo de processamento realiza as atribuições dos valores calculados para suas respectivas variáveis. A soma dos valores 1 e 2 será atribuída à variável soma. E o resultado da soma será dividido por 2 e atribuído à variável media.



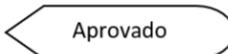
O símbolo de exibição mostra na tela o resultado de cada valor calculado.



O símbolo de decisão define a condicional (verdadeiro ou falso) para a variável.



Se a condição for verdadeira, a resposta será “Aprovado”.



Se a condição for falsa, a resposta será “Reprovado”.



Finaliza o programa.



Pseudocódigo

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

Na programação, os algoritmos também podem ser caracterizados pelos pseudocódigos, cuja intenção é chegar na solução de um problema.

Refletá

É importante estar atento para algumas regras básicas quando utilizar pseudocódigos:

- Escolher um nome.
- 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, mostra-lo na tela.
- Finalizar o algoritmo.

O uso de pseudocódigo pode ser aplicado para qualquer linguagem de programação?

Recordando o exemplo acima que calcula a média dos alunos da sua turma, veja como fica em pseudocódigo:

```
1. calculo_media;
2.         var
3.             real: valor1, valor2, soma, media;
4. Início
5.         escreva ("Digite o valor 1");
6.         leia valor1;
7.         escreva ("Digite valor 2");
8.         leia valor2;
9.         soma ← valor1 + valor2;
10.        media ← soma/2;
11.        escreva("A media do aluno e:"; media);
12.        se (media >=6)
13.            então escreva("Aluno Aprovado");
14.            senão escreva("Aluno Reprovado");
15.        Fim se;
16. Fim.
```

Veja agora os comentários deste algoritmo:

Linha 1: “calculo_media” 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 (valor1, valor2, soma, media), nesta linha também são definidos os tipos de variáveis (“real”, veremos com maior detalhe na próxima unidade do livro).

Linha 4: inicia os procedimentos dos algoritmos (início).

Linha 5: “escreva” é um comando de saída, este comando indica o que vai sair na tela do computador, geralmente o conteúdo do texto a ser mostrado fica entre aspas (“Digite valor 1”).

Linha 6: “leia” é comando de entrada, o valor digitado é armazenado na variável (valor1).

Linha 7: “escreva” é um comando de saída, este comando indica o que vai sair na tela do computador, geralmente o conteúdo do texto a ser mostrado fica entre aspas (“Digite valor 2”).

Linha 8: “leia” é comando de entrada, o valor digitado é armazenado na variável (valor2).

Linha 9: A adição das variáveis valor1 e valor2 é atribuído para variável soma ($soma \leftarrow valor1 + valor2$).

Linha 10: Realiza o cálculo da média e atribui o valor encontrado na variável média.

($media \leftarrow soma / 2$).

Linha 11: Escreve na tela o que está entre aspas. Escreva (“A media do aluno é:”; media). Perceba que a variável é colocada fora das aspas, para que seja representada pelo seu valor correspondente.

Linha 12: Utiliza o resultado da média para criar uma condição verdadeira ou falsa: se ($media \geq 6$).

Linha 13: se o resultado da média for maior ou igual a seis (condição verdadeira), o computador escreve na tela “Aluno Aprovado”.

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

Linha 15: Encerra a condição (fim se).

Linha 16: Encerra o algoritmo com a palavra “fim” e o ponto final.

Lembre-se, quando você escreve um algoritmo do tipo portugol, que é um pseudocódigo, é preciso escrever de forma clara para que as pessoas possam interpretar e, futuramente, possam passar para uma linguagem de programação.

Exemplificando

Veja abaixo um algoritmo escrito em pseudocódigo e executado em Visualg:

```
algoritmo "media"
var
    valor1, valor2, soma, media: real
inicio
```

```

Escreval("Digite o valor da nota 1: ")
Leia (valor1)
Escreval("Digite o valor da nota 2: ")
Leia (valor2)
soma <- (valor1 + valor2)
media <- (soma / 2 )
Escreval("A media do aluno e:" media)
se media >=6 entao
    escreval ("Aluno Aprovado média = ",
media)
senao
    escreval ("Aluno Reprovado média =
",media)
fimse
finalgoritmo

```

Perceba que os parâmetros utilizados também são considerados um algoritmo do tipo português estruturado, ou seja, de fácil entendimento e interpretação.

O *software* do Visualg é gratuito e o seu download está disponível na internet (VISUALG, 2020).

Após os estudos de algoritmos e as suas formas de construções, Manzano, Franco e Villar (2015) colocam em destaque os paradigmas de programação, que são caracterizados pelos paradigmas de programação estruturada, em que o algoritmo é construído como sequência linear de funções ou módulo. O outro paradigma é orientado a objetos ou simplesmente programação orientada a objetos, na qual o programador abstrai um programa como uma coleção de objetos que interagem entre si.

Conceitos de linguagem de programação

Após compreendermos os conceitos, aplicações e tipos de algoritmos, chegou o momento de 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, essas 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 computadores utilizadas foram as linguagens de máquina e a linguagem *assembly*, a partir da década de 1940. Desde então, muitas linguagens surgiram, bem como, criaram-se novos paradigmas de linguagem de programação.

Algumas definições para os paradigmas das linguagens de programação são apresentadas a seguir. 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 três décadas:

Programação imperativa: considerada 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 e a manipulação de exceções são seus componentes de programação.

Exemplo de programação imperativa: COBOL, Fortran, C, Ada e Perl.

Programação orientada a objeto: também conhecida na computação como (POO), como o próprio nome sugere, é considerado uma coleção de objetos que se inter-relacionam, facilitando assim a programação. São exemplos de POO: vSmalltalk, C++, Java e C#.

Programação funcional: caracterizada por possuir atuação matemática, cada uma com um espaço de entrada (domínio) e resultado (faixa). Exemplos de programação funcional: Lisp, Scheme, Haskell e ML.

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 programação lógica o Prolog.

Todas as linguagens de programação para criação de um programa possuem uma sintaxe, que nada mais é a forma de como o programa é

escrito, podemos ainda, definir sintaxe 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”.

Agora que você já viu que é importante conhecer as linguagens de programação e suas aplicações, vai conhecer os componentes de um programa de computador.

Componentes de um programa de computador

Segundo Damas (2016), um programa é uma sequência de instruções de tal forma que permita resolver um determinado problema. Um programa pode ser desenvolvido em módulos distintos e/ou em subprogramas.

Segundo Aguilar (2011), define-se um programa de computador como um conjunto de instruções. Essas instruções são ordenadas, em outras palavras, você direciona uma ordem para o computador e ele executa uma determinada tarefa de acordo com a ordem especificada.

Lembrando que para criar um programa de computador será necessário seguir as regras, segundo Aguilar (2011):

- Definir e analisar o problema a ser solucionado.
- Criar um algoritmo ou um diagrama de fluxo.
- Realizar o pseudocódigo.

Para executar um programa é importante estar atento à sequência que os dados percorrem:

- Entrada de dados: realiza as coletas de dados.
- Processamento: os dados são transformados em informação.
- Saída: onde todas as informações geradas pelo processamento de dados são apresentadas em um periférico.

Pois bem, para criação de um programa, você precisará definir as instruções a serem utilizadas na solução de um problema.

Segundo Aguilar (2011), as sintaxes (instrução) deverão ser escritas e armazenadas na memória do computador na mesma ordem que se espera serem executadas, ou seja, elas podem ser lineares (executadas sequencialmente) e não lineares (executadas de forma a serem redirecionadas, isto é, uma instrução de bifurcação).

Veja um exemplo de instrução linear:

Instrução 1

Instrução 2

.

.

Instrução n

No caso de um programa não linear, ele pode se comportar da seguinte maneira.

Instrução 1

Instrução x

Instrução y

Instrução 2

.

.

Instrução n

Muito bem, a partir de agora você irá conhecer as sequências de instruções para criação de um programa de computador. Vamos lá!

No contexto geral ficaria assim:

1. Início do programa.
2. Definição das variáveis e de possíveis atribuições.
3. Instrução de leitura dos dados.
4. Instrução do formato de escrita.
5. Demais instruções e funções.
6. Fim do programa.

Ou seja,

Início

variáveis;

comando1;

comando2;

comandoN;

Fim.

Em algoritmos, a forma de escrever (maiúsculas e minúsculas) não acarretará em erros, porém, em linguagem de programação C é preciso diferenciar as palavras em letras maiúsculas e minúsculas.

Veja no Quadro 1.2 a estrutura básica de um algoritmo e de um programa em linguagem C.

Quadro 1.2 | Estrutura de um algoritmo e da linguagem C

Algoritmo	Linguagem C
Nome_Programa Declaração das variáveis início Ações . . Fim.	Definição das bibliotecas Início do programa (main) Início das funções Declaração das variáveis Instrução . Termino das funções.

Fonte: elaborado pelo autor.

Quando iniciamos um programa na linguagem de programação C, segundo Manzano, Franco e Villar (2015), as primeiras linhas de programação são definidas pelas bibliotecas, também conhecidas como arquivos de cabeçalho.

Para inserir as bibliotecas no programa é necessário colocar #include (inclusão de um arquivo no programa fonte) e em seguida, entre os símbolos de menor “<” e maior “>” (quando se usa < e > o arquivo é procurado na pasta include) o nome da biblioteca.

Veja alguns exemplos de biblioteca em linguagem de programação C:

- **stdio** – essa biblioteca é responsável pelas funções de entradas e saídas, como é o caso da função printf e scanf que vamos aprender mais à frente.

Exemplo: #include <stdio.h>

- **stdlib** – essa biblioteca transforma as *strings* (vetores de caracteres) em números.

Exemplo: #include <stdlib.h>

- **string** – biblioteca responsável pela manipulação de *strings*.

Exemplo: #include<string.h>

- **time** – biblioteca utilizada para manipulação de horas e datas.

Exemplo: #include<time.h>

- **math** – biblioteca utilizada para operações matemáticas.

Exemplo: #include<math.h>

- **ctype** – biblioteca utilizada para classificação e transformação de caracteres.

Exemplo: #include <ctype.h>

Refletia

Para cada tipo de aplicação dentro do programa, será necessário inserir bibliotecas que interpretam as variáveis e os resultados que poderão surgir.

Diante desta situação é possível determinar a quantidade de bibliotecas e suas aplicações dentro da programação em linguagem C?

Após a definição das bibliotecas, o programa é inicializado pela função (main). Veja o seguinte exemplo:

```
main ()  
{  
}
```

Quando utilizamos a “{“ (chave aberta) indica o início de uma função em C e quando usamos a “}” (chave fechada) indica o término das funções e do programa, porém, elas podem indicar o início e o término de alguns contextos na programação.

Quando usamos a int antes de main () significa que retornará um número do tipo inteiro.

```
int main ()  
{  
}
```

Também pode ser utilizada a função void, esta é uma função sem retorno, ou seja, não recebe nenhum argumento.

```
void main ( )  
{  
}
```

Vamos iniciar pelos operadores aritméticos que são representados por binários e unários.

- Operadores binários

- + soma

- subtração

- * multiplicação

- / divisão

- % resto de divisão

- Operador unário

Segundo Mizrahi (2008), o operador unário também pode ser usado para representar a troca de sinais de uma determinada variável, por exemplo:

a= -5;

a= -a;

Após essa operação, o valor de “a” assume o valor de 5 positivo.

Os operadores relacionais são utilizados para realizar a comparação de valores. Eles podem ser classificados de acordo com o Quadro 1.3.

Quadro 1.3 | Operadores relacionais

Operador em linguagem C	Operador em algoritmos	Descrição
>	>	Maior
<	<	Menor
>=	>=	Maior ou igual
<=	<=	Menor ou igual
==	=	Igual
!=	<>	Diferente

Fonte: elaborado pelo autor.

Os operadores lógicos são utilizados quando nos deparamos com situações de conectividade lógica. Veja suas atribuições no Quadro 1.4:

Quadro 1.4 | Operadores lógicos

Operador em linguagem C	Operador em algoritmos	Descrição
&&	E	Lógico E – conjunção
	OU	Lógico OU – disjunção
!	NÃO	Lógico NÃO – negação

Fonte: elaborado pelo autor.

Segundo Mizrahi (2008), o operador lógico “NÃO” é considerado um operador unário e os operadores lógicos “E” e “OU” são binários.

Segundo Joyanes (2011), todas as linguagens de programação possuem elementos básicos que são utilizados como blocos construtivos, os quais formam regras para que esses elementos se combinem. Essas regras são chamadas de sintaxes de linguagem ou comandos para programação.

Em linguagem de programação C, segundo Soffner (2013), existem um total de 32 palavras reservadas para programação, conforme definido pelo padrão ANSI. Veja no Quadro 1.5 as palavras reservadas em linguagem de programação C.

Quadro 1.5 | Palavras reservadas em linguagem C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Fonte: Soffner (2013. p. 36).

Veja também, no Quadro 1.6, algumas palavras reservadas que são utilizadas em algoritmos.

Quadro 1.6 | Palavras reservadas para uso de algoritmos

Leia	para	faca	então	cos
escreva	de	enquanto	senão	maiusc
algoritmo	ate	fim_enquanto	fim_se	minusc
constantes	passo	repita, pare	sem	raiz
var	fim_para	se	compr	div
tan	trunc	abs	resto	e
ou	não	inteiro	real	logico
literal	vetor	matriz	verdadeiro	falso

Fonte: elaborado pelo autor.

Para atribuir um valor a uma variável em linguagem de programação C utilizamos o sinal de igual “=”.

Exemplos com linguagem C:

```
soma = valor1 + valor2;  
valor=5;
```

Para dar sequência aos nossos estudos, vamos entender de forma conceitual as variáveis, mas não se preocupe! Na próxima seção vamos nos aprofundar um pouco mais sobre o assunto. Pois bem, as variáveis são locais reservados na memória para armazenamento dos dados, cada uma possui um nome próprio para sua identificação.

Os tipos de variáveis mais usadas são:

- **inteiro**: armazena os números inteiros (negativos ou positivos). Em linguagem C é definida por “int”, em algoritmos é definida por “inteiro”. Veja as especificações no Quadro 1.7:

Quadro 1.7 | Exemplo de aplicação de variável

Algoritmos	Linguagem C
Nome_Programa; var inteiro: valor1, valor2,soma; inicio . . . Fim.	#include <stdlib.h> void main() { int valor1, valor2, soma; . . }

Fonte: elaborado pelo autor.

- **real**: permite armazenar valores de pontos flutuantes e com frações. Em linguagem C é definido por “float” e quando precisa do dobro de dados numéricos é utilizado o tipo “double” ou “long double”. Em algoritmo pode ser usado simplesmente a palavra “real”. Veja um exemplo no Quadro 1.8.

Quadro 1.8 | Exemplo de aplicação de variável

Algoritmo	Linguagem C
<pre>Nome_Programa; var valor1, valor2,soma: real; Inicio . . Fim.</pre>	<pre>#include <stdlib.h> void main() { float valor1, valor2, soma; . . }</pre>

Fonte: elaborado pelo autor.

- **caractere**: caracteriza os caracteres, números e símbolos especiais. São delimitadas por aspas simples (''). Em linguagem C é definida por “char”. Em algoritmo é utilizada a palavra “caractere”

Veja o exemplo no Quadro 1.9:

Quadro 1.9 | Exemplo de aplicação de variável

Algoritmo	Linguagem C
<pre>Nome_Programa; var nome1, nome2: caractere; inicio . . Fim.</pre>	<pre>#include <stdlib.h> void main() { char nome1, nome2; . . }</pre>

Fonte: elaborado pelo autor.

Agora você vai conhecer como funcionam as constantes. Vamos lá! O próprio nome já sugere, constante é algo que não se altera. Para Schildt (2005), as constantes são consideradas modificadores de tipo de acesso, ou seja, não podem ser alteradas, elas podem ser representadas pelo comando “const”.

Exemplo:

```
const int fixo=500;
```

As constantes também podem ser caracterizadas por qualquer tipo de dados básicos, por exemplo: as constantes do tipo texto são envolvidas por aspas simples (') ou aspas duplas, as aspas simples representam um único caractere, por exemplo

‘a’ ou ‘100’ e as aspas duplas caracterizam um conjunto de caracteres, por exemplo “A conversão da temperatura de graus centígrados para graus Fahrenheit”.

As constantes inteiras são representadas por números inteiros negativos ou positivos, exemplo: -30 e 30 são constantes inteiras (int).

Elas seguem as mesmas regras para as variáveis do tipo flutuante, você poderá utilizar os comandos float e double, por exemplo: 30.30 é um número em ponto flutuante.

Para dar sequência à seção, você conhecerá os principais operadores utilizados nos algoritmos e na linguagem de programação C.

Agora vamos estudar os comandos de entrada e saída de dados dentro da programação em linguagem C.

A função `printf()` é um comando de saída que possui um vínculo com a biblioteca `stdio.h`. É utilizada quando se pretende obter uma resposta na tela do computador.

A sua síntese é definida por:

`printf (“expressão de controle”, listas de argumentos);`

Existem algumas formatações na utilização da função `printf()`, conforme mostra o Quadro 1.10, estas formatações podem mudar de acordo com os tipos de variáveis declaradas.

Quadro 1.10 | Código de formatação para função `printf()`

Código	Função
%c	Permite a escrita de apenas um caractere.
%d	Permite a escrita de números inteiros decimais.
%e	Realiza-se a escrita de números em notação científica.
%f	É feita a escrita de números reais (ponto flutuante).
%g	Permite a escrita de %e ou %f no formato mais curto.
%o	Permite que números octais sejam escritos.
%s	Efetua-se a escrita de uma série de caracteres.
%u	Escreve-se um número decimal sem sinal.
%x	Permite a escrita de um número hexadecimal.
%n	Permite determinar entre colchetes quais caracteres podem ser ou não aceitos na entrada de uma sequência de caracteres, sendo “n” um valor opcional que determina o tamanho da sequência de caracteres.

Fonte: Manzano (2013. p. 38).

Veja no exemplo abaixo a aplicação de formatação da função printf():

```
printf ("O valor encontrado foi %d", valor1);
```

Perceba que o valor da variável “valor1” foi posicionado no local do “%d”, lembrando que “%d” é uma formatação para um dado do tipo inteiro.

Outras particularidades:

Na função printf você pode pular linhas com o comando “\n” e pode obter um resultado numérico determinando a quantidade de casas decimais. Veja os exemplos abaixo:

```
Printf (" \n Resposta: a = %.2f e b = %.2f \n",  
a,b);
```

Neste exemplo, antes de apresentar a frase o programa pulou uma linha “\n”, o “%f” é utilizado quando os dados numéricos são flutuantes, ou seja, valores fracionados e quando usamos %.2f significa que o valor será arredondado em duas casas decimais, ex.: 2,45.

Em algoritmos, a função utilizada para saída de dados é a palavra “escreva”, veja o exemplo a seguir:

```
Valor_program;
```

```
var
```

```
real: valor;
```

```
inicio
```

```
    escreva("Digite um valor"); // saída da  
informação na tela do computador
```

Fim.

A função scanf() é um comando de entrada, isto é, são informações que possibilitam a entrada de dados pelo teclado, assim, a informação será armazenada em um determinado espaço da memória, como o nome e tipo específico da variável. A sintaxe é definida por uma expressão de controle (sempre entre aspas duplas) e pela lista de argumentos.

A sintaxe da função scanf() é definida por:

```
scanf("expressão de controle", lista de argumentos);
```

A função `scanf()` faz uso de alguns códigos de formação, veja no Quadro 1.11:

Quadro 1.11 | Código de formatação para função `scanf()`

Código	Função
<code>%c</code>	Permite que seja efetuada a leitura de apenas um caractere.
<code>%d</code>	Permite fazer a leitura de números inteiros decimais.
<code>%e</code>	Permite a leitura de números em notação científica.
<code>%f</code>	É feita a leitura de números reais (ponto flutuante).
<code>%l</code>	Realiza-se a leitura de um número inteiro longo.
<code>%o</code>	Permite a leitura de números octais.
<code>%s</code>	Permite a leitura de uma série de caracteres.
<code>%u</code>	Efetua-se a leitura de um número decimal sem sinal.
<code>%x</code>	Permite que seja feita a leitura de um número hexadecimal.
<code>[%[código]]</code>	Permite que seja feita uma entrada formatada pelo código.

Fonte: Manzano (2013, p. 38).

Veja a sintaxe a seguir:

```
scanf ("%d", &valor);
```

Neste exemplo, o computador entrará com um valor decimal e retornará o valor da variável “`valor`”.

O “`&`” é utilizado na função `scanf()` na lista de argumentos, sua função é retornar o conteúdo da variável, ou seja, retorna o endereço do operando.

```
main()
{
    int valor;
    printf("Digite um número: ");
    scanf("%d",&valor);
    printf("\n o número é %d",valor);
    printf("\no endereço é %u",&valor);
}
```

Para realizar a entrada de dados em algoritmos você pode utilizar a palavra “leia”, veja o exemplo a seguir:

```
valor_program;
var
real: valor;
inicio
escreva("Digite um valor");
leia(valor); // valor de entrada, será armazenado
na memória do computador
escreva("O valor digitado foi:", valor);
fim.
```

Cabe ressaltar que você pode fazer comentários em qualquer lugar do seu programa, basta utilizar barras duplas “//”.

Exemplo:

```
#include <stdio.h> // biblioteca para entrada e saída de dados
int main() // comando de início e o mais importante do programa
{ // início de uma função
printf("Meu primeiro programa"); // comando para saída
de dados na tela
return 0; // indica que o processo está voltando para o Sistema
Operacional
} // fim de uma função ou de um programa
```

Exemplificando

Para um programa retornar ao sistema operacional, segundo Manzano, Franco e Villar (2015), será necessário utilizar a instrução retorna zero “return 0”, assim como, a instrução system (“pause”) tem a função de pausar a execução do programa, para que o resultado seja visualizado.

Veja o exemplo:

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

```
    printf ( "Digite a idade do candidato" ) ;
    scanf ( "%d" , &id ) ;
    printf ( "O candidato tem %d anos !\n" , id);
    system ("pause");
    return 0 ;
}
```

Não há como negar: todas as áreas estão voltadas para a tecnologia e são através 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 sempre buscar novos caminhos na sua área de atuação.

Você, com certeza, poderá ser um grande entusiasta em algoritmos e linguagens de programação.

Sem medo de errar

Caro aluno, chegou o momento de colocar em prática todo o conhecimento adquirido nesta seção. Lembrando que você foi incumbido de treinar os estagiários contratados pela empresa de desenvolvimento de softwares educacionais. Cabe ainda lembrar que os estagiários não possuem nenhuma experiência em algoritmos.

Para tal, como primeira proposta, você deverá apresentar aos estagiários um algoritmo utilizando a linguagem natural, ou diagramas de blocos (fluxogramas) e/ou pseudocódigos para cadastrar os dados pessoais do aluno (nome, endereço, cidade e estado) e mostrar na tela do computador o resultado do cadastro.

Seguem as dicas para realizar a solução da situação-problema.

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

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

Enfim, no pseudocódigo, faça uso das seguintes dicas:

- Escolher um nome.
- 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 mostrá-lo na tela.
- Finalizar o algoritmo.

Exemplo:

Inicio

```
nome, endereço, cidade, estado: caractere  
Escreva "Digite o nome do Aluno";  
Leia nome;  
Escreva "Digite o endereço do aluno";  
Leia endereço;  
Escreva "Digite a cidade do Aluno";  
Leia cidade;  
Escreva "Digite o estado do Aluno";  
Leia estado;  
Escreva "CADASTRO DE ALUNOS ";  
Escreva "Nome: ", nome;  
Escreva "Endereço: ", endereço;  
Escreva "Cidade: ", cidade;  
Escreva "Estado: ", estado;
```

Fim.

São várias as formas que você pode utilizar para a conclusão de um algoritmo. Seguindo este pensamento, resolva o seu algoritmo e se possível elabore outras formas de solução.

A troca

Um programador foi incumbido de realizar um algoritmo para coletar a quantidade de mulheres e de homens em um determinado evento, porém, algo deu errado. A variável que era para receber o número de mulheres acabou recebendo o número de homens e vice-versa. Agora, você precisa ajustar rapidamente esses valores. Qual a forma mais adequada para realizar essa troca utilizando um algoritmo em pseudocódigo?

Resolução da situação-problema

```
1.    inverter valores;  
2.        var  
3.            real: mulheres, homens, troca;  
4.    Início  
5.        escreva ("Acertar valores");  
6.        escreva("Digite a quantidade de  
mulheres");  
7.        leia homens; // Opa! Cometeu-se um erro:  
número de mulheres foi para homens.  
8.        escreva ("Digite a quantidade de  
homens");  
9.        leia mulheres; // Opa cometeu-se um  
erro: número de homens foi para mulheres.  
10.       troca ← mulheres;  
11.       mulheres ← homens;  
12.       homens ← troca;  
13.       Escreva("A quantidade correta de  
mulheres é:", mulheres);  
14.       Escreva("A quantidade correta de homens  
é:", homens);  
15. Fim.
```

Deixo como desafio realizar o diagrama de blocos e a linguagem natural do algoritmo acima.

Boa sorte!

Faça valer a pena

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 abaixo que calcula o bônus de 30% para os funcionários cadastrados, complete a linha de programação que está faltando:

```
1.    inicio
2.    caracter: nome;
3.    _____
4.    escreva ("Entre com o nome do funcionário");
5.    leia (nome);
6.    escreva ("Entre com o salario Bruto do Funcionario");
7.    leia (sal);
8.    _____
9.    totsal sal+bonus;
10.   escreva ("Seu salario com bonus será de:"; totsal);
11.   fim.
```

Assinale a alternativa correta.

- a. real: sal, bonus;
bônus ← sal*(30/100);
- b. real: sal, bonus, totsal;
sal ← bonus*(30/100);
- c. real: sal, bonus, totsal;
bonus ← sal*(30/100);

- d. real: sal, bonus, totsal;
- e. bonus ← sal*(30/100);

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 problemas. Para que os diagramas de blocos tenham certa coerência, os seus símbolos foram padronizados pela ANSI (Instituto Norte Americano de Padronização). Analise os símbolos a seguir:

I.



O símbolo significa um terminal, em que representa o início ou o fim de um fluxo lógico. Em alguns casos, definem as sub-rotinas.

II.



O símbolo significa exibição, em que mostra o resultado de uma ação, geralmente através da tela de um computador.

III.



O símbolo significa entrada manual, em que determina a entrada manual dos dados, geralmente através 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.

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 uma determinada linguagem de programação.

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

- I. **Programação imperativa:** considerada o paradigma mais antigo, a programação imperativa pode armazenar o programa e suas variáveis juntamente, assim como a abstração procedural, as atribuições, as sequências, os laços, os comandos condicionais e a manipulação de exceções são seus componentes de programação.
- 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:** caracterizada por possuir 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.
- b. II, III e IV.
- c. II e III.
- d. III.
- e. I, III e IV.

Seção 2

Componentes e elementos de linguagem de programação

Diálogo aberto

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 possui 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 empresa em que você atua contratou, recentemente, jovens programadores e ficou sob sua responsabilidade treiná-los. Para que a equipe se aproprie adequadamente do conhecimento, o primeiro passo é apresentar-lhes os conceitos de variáveis e constantes, bem como o que é um dado primitivo e quais tipos existem na linguagem C. Esses tipos são os mesmos de outra linguagem, por exemplo, Java? Como é possível saber quanto espaço uma variável ocupa na memória? Todos as variáveis de tipo primitivo ocupam o mesmo espaço na memória?

Você deverá criar um documento digital, em forma de slides, contendo esses conceitos, bem como uma lista com os tipos de dados primitivos que podem ser utilizados. Não se esqueça de apresentar exemplos de uso para cada tipo de dado.

Para completar essa etapa no trabalho, você aprenderá a diferença entre constantes e variáveis, bem como o que são dados primitivos e quais os tipos disponíveis na linguagem C.

Bons estudos!

Não pode faltar

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 1.2.

Tabela 1.2 | Alguns valores para área da circunferência

π (<i>pi</i>)	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.

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 de trabalho e, portanto, ocupa um espaço físico. O mesmo se aplica às constantes, porém, nesse caso, o valor armazenado nunca será alterado.

Na Figura 1.1 foi criada uma representação simbólica para a memória de trabalho e dentro dessa memória foram alocadas duas variáveis, A e B. Como você pode perceber, 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 1.1 | Alocação de variáveis na memória de trabalho



Fonte: elaborada pela autora.

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

- **Numérico inteiro:** são valores inteiros que podem ser positivos, negativos ou zero. Alguns exemplos são as variáveis que armazenam idade, quantidade de produtos, código de identificação, dentre outros.
- **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, dinheiro etc.
- **Caractere:** é o tipo usado para armazenar letras. 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 etc.

Uso de variáveis em linguagens de programação

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

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

Esse padrão é aceito por todas linguagens de programação, embora algumas permitam certas variações, tais como não definir o tipo explicitamente. Na

linguagem de programação C, esse padrão é obrigatório e podemos usar os seguintes tipos primitivos: ***int*** (inteiro), ***float*** ou ***double*** (ponto flutuante), ***char*** (caractere) e ***void*** (sem valor). O tipo booleano é representado pelo comando ***bool***, entretanto, para seu uso, é necessário incluir a biblioteca `<stdbool.h>`. Veja no Quadro 1.12 a criação de algumas variáveis na linguagem C.

Quadro 1.12 | Criação de variáveis na linguagem C

```
1. #include <stdbool.h>
2. void main(){
3.     int idade;
4.     float salario = 7500;
5.     double qtd_atomas;
6.     bool confirma = false;
7.     char genero = 'M';
8. }
```

Fonte: elaborado pela autora.

Destaca-se que ao criar uma variável, o programador pode optar por já atribuir ou não um valor. Mesmo existindo essa opção, é uma boa prática de programação sempre inicializar as variáveis para evitar que recebam dados que estejam na memória. Portanto, quando a variável for numérica, sempre iremos inicializar com zero, quando booleana, com falso, e quando do tipo caractere, iremos usar ‘ ‘ para atribuir vazio.

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 irá armazenar a idade, opte pelo termo completo. 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*”. No nome, não podem ser usados acentos nem caracteres especiais, como a interrogação.

Como já mencionado, a quantidade de espaço que será alocada 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 da 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 comandos usados na declaração da variável que modifica sua capacidade-padrão. Os três principais modificadores são: ***unsigned***, usado para especificar que a variável irá armazenar somente a parte positiva do número; ***short***, que reduz o espaço reservado pela memória; e ***long***, que aumenta a capacidade padrão. A Tabela 1.3 mostra o tamanho e os possíveis valores de alguns tipos, inclusive com os modificadores.

Tabela 1.3 | Tipos de variáveis e sua capacidade

Tipo	Tamanho (byte)	Valores
<i>int</i>	4	-2.147.423.648 até 2.147.423.648
<i>float</i>	4	-3,4 ³⁸ até 3,4 ³⁸
<i>double</i>	8	-1,7 ³⁰⁸ até 1,7 ³⁰⁸
<i>char</i>	1	-128 até 127
<i>unsigned int</i>	4	4.294.967.296
<i>short int</i>	2	-32.768 até 32.767
<i>long double</i>	16	-3,4 ⁴⁹³² até 1,1 ⁴⁹³²

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

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 o código no Quadro 1.13, em que foram criadas cinco variáveis (linhas 3 a 7) e, para cada tipo, usou-se um especificador na hora de fazer a impressão com a função `printf()`. Na linha 8, para o tipo inteiro, foi usado `%d`. Nas linhas 9 e 10, para os pontos flutuantes, foi usado `%f`. Na linha 11, para o caractere, foi usado o `%c`, e, na linha 12, o especificador de ponto flutuante ganhou o adicional `.3`, o que significa que o resultado será impresso com três casas decimais.

Quadro 1.13 | Impressão de variáveis

```
1. #include<stdio.h>
2.
3. void main(){
4.     short int idade = 18;
5.     float salario = 7500;
6.     double qtd_atomas = 123456789123;
7.     char genero = 'F';
8.     float altura = 1.63;
9.
10.    printf("\n idade: %d",idade);
11.    printf("\n salario: %f",salario);
12.    printf("\n qtd_atomas: %f",qtd_atomas);
13.    printf("\n genero: %c",genero);
14.    printf("\n altura: %.3f",altura);
}
```

Fonte: elaborado pela autora.

Constantes

Entende-se por constante um valor que nunca irá se alterar. 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> <valor>
```

Veja que não há ponto e vírgula no final. Outro ponto interessante é que a diretiva não utiliza espaço em memória, ela apenas cria um rótulo associado a um valor (MANZANO; MATOS; LOURENÇO, 2015). Se não é reservado um espaço na memória, logo o operador & não pode ser usado nesse caso.

A outra forma de se criar valores constantes é similar à criação de variáveis, porém, antes do tipo, usa-se o comando `const`, portanto a sintaxe ficará:

```
const <tipo> <nome>;
```

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 etc. 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 Quadro 1.14, há um exemplo das duas formas de sintaxes para constantes. Na linha 2 definimos uma constante (rótulo) chamada `pi` com valor 3.14. Na linha 4 criamos uma constante usando o comando `const`. Nas

linhas 5 e 6 imprimimos o valor de cada constante, veja que nada difere da impressão de variáveis.

Quadro 1.14 | Sintaxe para criação de constantes em C

```
1. #include<stdio.h>
2. #define pi 3.14
3. void main() {
4.     const float g = 9.80;
5.     printf("\n pi = %f",pi);
6.     printf("\n g = %f",g);
7. }
```

Fonte: elaborado pela autora.



Para consultar como incluir uma biblioteca matemática em linguagem C, acesse o QR Code ou o link:
http://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2020-1/algoritmos_programacao_estruturada/u1/s2/lkls201_u1s2_alg_pro_est_QRCode1.pdf

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 possui um endereço que o identifica. Podemos fazer uma analogia com os endereços das casas, já que cada casa possui 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 possui 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.

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 alocarmos o espaço de memória.



Para consultar como criar um programa que armazena as coordenadas x,y com os valores (5,10) e depois imprimir o endereço dessas variáveis, acesse o QR Code ou o link:
http://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2020-1/algoritmos_programacao_estruturada/u1/s2/lkls201_u1s2_alg_pro_est_QRCode3.pdf

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

```
scanf("especificador", &variavel);
```

A Figura 1.2 apresenta um exemplo no qual se utilizou o especificador “%d” para indicar ao compilador que o valor que será digitado deve ser um inteiro, e esse valor será guardado no endereço de memória da variável `x`.

Figura 1.2 | Armazenamento em variáveis



Fonte: elaborada pela autora.

Para ter acesso ao endereço de memória de uma variável e como armazenar valores digitados pelo usuário, 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 Quadro 1.15 apresenta a solução. Observe que na linha 3, como todas as variáveis eram do mesmo tipo, foram declaradas na mesma linha, separadas por vírgula, e todas foram inicializadas com zero. Nas linhas 5 e 7, os valores digitados pelo usuário serão armazenados nos endereços das variáveis `valor1` e `valor2`, respectivamente.

Quadro 1.15 | Impressão de valores armazenados

1.	#include<stdio.h>
2.	void main(){
3.	float valor1=0, valor2=0;
4.	printf("\n Digite o primeiro valor:");
5.	scanf("%f",&valor1);
6.	printf("\n Digite o segundo valor:");
7.	scanf("%f",&valor2);
8.	printf("Variavel 1 = %.2f",valor1);
9.	printf("Variavel 2 = %.2f",valor2);
10.	}

Fonte: elaborado pela autora.

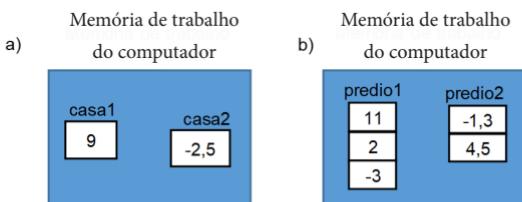
Variáveis compostas

Como já explicitado, as variáveis são usadas para armazenar dados na memória de trabalho e que 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*, a altura em um tipo *float* etc., mas e se fosse necessário armazenar quinze medidas da temperatura de um dispositivo, usariamos quinze variáveis? Com nosso conhecimento até o momento teríamos que criar as quinze, porém muitas variáveis podem deixar o código maior, além de não ser 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 em um endereço de memória (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 possui um endereço único que a identifica, 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 em seu endereço. Por outro lado, um prédio possui um endereço composto pelos mesmos parâmetros que a casa, porém, nesse mesmo endereço moram várias famílias. Assim são as variáveis compostas, em um mesmo endereço são armazenados diversos valores. Esses conceitos estão ilustrados na Figura 1.3: 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.

Figura 1.3 | a) Variáveis primitivas b) Variáveis compostas



Fonte: elaborada pela autora.

Reflita

Se, nas variáveis compostas, em um mesmo endereço são guardados muitos valores, como diferenciar um do outro? Assim como os apartamentos em um prédio possuem números para diferenciá-los, as variáveis compostas possuem **índices** que as diferenciam.

Nesse sentido, ao contrário de uma variável primitiva, uma variável composta possui 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, sendo a bidimensional 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 1.3, são homogêneas, pois a primeira armazena apenas valores do tipo inteiro e a segunda somente do tipo decimal. Além disso, elas também são unidimensionais, isso quer dizer que elas possuem 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 nome mais utilizado entre os profissionais, adotaremos essa forma no restante do livro.

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];
```

Exemplificando

Vamos criar dois vetores em C para armazenar a idade e a altura (em metros) de 3 pessoas. Veja no código do Quadro 1.16 que, na linha 3, foi criado o vetor *idade*, que é do tipo inteiro e com capacidade para

armazenar três valores. Já na linha 4, o vetor *altura* foi criado e foi inicializado com valores. Para armazenar valores no vetor no momento da criação, colocamos os elementos entre chaves separados por vírgula. Da linha 5 a 7 é feita a impressão dos valores guardados no vetor *altura*.

Quadro 1.16 | Vetor em C

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

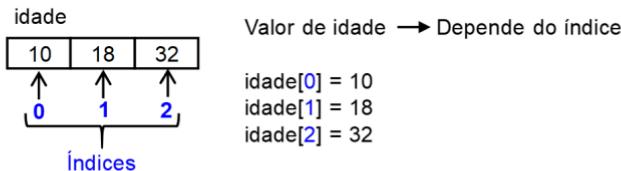
Fonte: elaborado pela autora.

Cada elemento no vetor é acessado por meio do seu índice, que sempre começará pelo valor zero, independentemente da linguagem de programação. Por isso, no código do Quadro 1.16, na linha 5, usou-se *altura[0]* para imprimir o primeiro valor, *altura[1]* para imprimir o segundo e *altura[2]* para imprimir o terceiro. O índice é usado tanto para leitura, como para escrita, por exemplo, podemos atribuir valores ao vetor *idade* da seguinte forma:

```
idade[0] = 10;
idade[1] = 18;
idade[2] = 32;
```

Para ajudar a compreensão, observe a Figura 1.4, que representa um esquema para o vetor *idade* na memória. 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 1.4 | Vetor *idade*



Fonte: elaborada pela autora.

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 no vetor, faríamos da seguinte forma:

```
printf("Digite uma idade: ");  
scanf("%d", &idade[1]);
```

Nesse caso, o valor seria armazenado no índice 1 do vetor, ou seja, no segundo bloco, pois o primeiro é o bloco zero.

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, mas lembre-se, quanto maior, mais espaço será reservado na memória de trabalho.

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 <variavel>[tamanho];
```

Exemplos:

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

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]* possui 15 “espaços” disponíveis para serem preenchidos.

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

Exemplos:

```
char nome[16]={ 'J','o','a','o' };  
char sobrenome[31] = "Alberto Gomes";  
char frase[101] = {"Disciplina de Algoritmos"};
```

Existem várias funções que fazem a leitura e impressão de *strings* em C. Iremos estudar 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*. Portanto, para armazenar o nome digitado por um usuário usamos:

```
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 obrigatório na função *scanf()*.

Essa forma de atribuição possui 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 possui a seguinte sintaxe:

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

O destino especifica o nome da *string* que será usada para armazenar. 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:

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

Veja que antes de usar o `fgets()`, usamos a função `fflush(stdin)`, não é obrigatório, mas garante que a entrada padrão (`stdin`) seja limpa antes de armazenar.

Variáveis compostas homogêneas bidimensionais (matrizes)

Observe a Tabela 1.4 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 implementarmos um programa que calcula a média dessas temperaturas, precisaríamos armazenar esses valores em uma variável composta bidimensional, ou, como é mais comumente conhecida, uma matriz bidimensional.

Tabela 1.4 | 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> <nome_da_matriz>[linhas][colunas];
```

Exemplos:

```
1. int coordenadas[3][2];
2. float temperaturas[7][2];
```

Na linha 1 é criada uma estrutura com 3 linhas e 2 colunas. Na linha 2 é criada a estrutura da Tabela 1.4 (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 Quadro 1.17 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 4 e 5 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 aluno, que são armazenados respectivamente na segunda e terceira linha.

Quadro 1.17 | Matriz em C

```
1. #include<stdio.h>
2. main(){
3.     float notas[3][2];
4. 
5.     //aluno 1
6.     notas[0][0] = 10;
7.     notas[0][1] = 8.5;
8. 
9.     //aluno 2
10.    notas[1][0] = 5.5;
11.    notas[1][1] = 2.7;
12. 
13.    //aluno 3
14.    notas[2][0] = 4;
15.    notas[2][1] = 10;
16. }
```

Fonte: elaborado pela autora.

A Figura 1.5 ilustra a estrutura de dados que é criada na memória para o código do Quadro 1.17. Veja que as linhas foram usadas para representar os alunos e as colunas para as notas.

Figura 1.5 | Esquema de atribuição de notas

O diagrama mostra uma matriz 3x2. As linhas são rotuladas como 'Aluno 1' (índice 0), 'Aluno 2' (índice 1) e 'Aluno 3' (índice 2). As colunas são rotuladas como 'Nota 1' (índice 0) e 'Nota 2' (índice 1). As células da matriz contêm os valores: (1,0) = 10,0, (1,1) = 8,5, (2,0) = 5,5, (2,1) = 2,7, (3,0) = 4,0 e (3,1) = 10,0. Linhas e colunas são rotuladas com índices em azul.

	Nota 1	Nota 2
	coluna 0	coluna 1
Aluno 1	linha 0 →	10,0
Aluno 2	linha 1 →	5,5
Aluno 3	linha 2 →	4,0
		8,5
		2,7
		10,0

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:

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

Nesse exemplo, a nota digitada será guardada na linha 1, coluna 0, e será exibida na tela usando duas casas decimais (comando linha 3).

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, as linguagens de programação oferecem 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 *struct*.

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

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

Nela, <nome> é o nome da estrutura, por exemplo, cliente, carro, fornecedor etc. e as variáveis internas são os campos que se deseja guardar dessa estrutura. Na prática, uma estrutura funciona como um “tipo de dado” e seu uso sempre será atribuído a uma ou mais variáveis.



A fim de consultar a implementação completa para guardar valores digitados pelo usuário na estrutura “automovel”, acesse o QR Code ou o link:
http://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2020-1/algoritmos_programacao_estruturada/u1/s2/lkls201_u1s2_alg_pro_est_QRCode2.pdf

Variável do tipo ponteiro

Além das variáveis primitivas e compostas, existe um tipo de variável muito importante na linguagem C, na qual, por meio dos ponteiros podem ser manipuladas as variáveis e outros recursos pelo endereço de memória (SOFFNER, 2013). Assim, 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 (*), usado para criação 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, 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 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* também estará ao ponteiro. A Figura 1.6 ilustra, de forma simplificada, o esquema de um ponteiro na memória. Veja que o conteúdo do ponteiro é o endereço da variável a que ele aponta e que ele também ocupa espaço na memória.

Figura 1.6 | 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("\nConteudo 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 (também poderia usar o `%x`).

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

```
printf("\n Conteudo da variavel 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("\nEnderecodoponteiro:%p", &ponteiro_para_ano);
```

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

Sem medo de errar

Após estudar as variáveis e constantes nesta seção, é hora de você estruturar a solução para o treinamento da sua equipe. Lembre-se, jovens programadores certamente já usaram variáveis em suas soluções, mas você deve garantir que eles compreendam todos os aspectos que envolvem esse importante elemento na programação. Seu treinamento deverá ser feito usando um documento digital em forma de *slides*, portanto a ordem lógica de apresentação do conteúdo é importante.

O primeiro item que você deve apresentar é a definição formal de variáveis e constantes. Essa informação ajuda a entender a relação que existe entre a criação de uma variável e a memória de trabalho de um computador. Para ajudar na solução, você pode usar imagens, como as das Figuras 1.1 e 1.2, assim ficará claro que os dados em um programa ocupam espaço na memória e que esse espaço irá variar conforme o tipo da variável.

O segundo passo é apresentar uma lista dos tipos de dados primitivos na linguagem C. Faça uma pesquisa e veja se os dados primitivos na linguagem C são os mesmos que a linguagem Java, já que essas duas linguagens pertencem a diferentes paradigmas de programação. Para essa etapa, é interessante construir um quadro com os tipos primitivos em ambas linguagens, por exemplo, conforme o Quadro 1.18.

Quadro 1.18 | Tipos primitivos em C e Java

Tipos primitivos em C	Tipos primitivos em Java	Exemplo
int	Int	int idade = 18;
float	Float	float salario = 2500;
...

Fonte: elaborado pela autora.

Outro ponto importante a ser abordado em seu treinamento é a questão do espaço que cada tipo de variável ocupa na memória. Faça um programa em C que utilize o comando `sizeof` para imprimir o tamanho de cada tipo primitivo, conforme exemplo no Quadro 1.19.

Inclua no código do programa em C exemplos de cada tipo de variável, bem como exemplos de como guardar dados inteiros e de ponto flutuante em variáveis para posterior impressão.

Quadro 1.19 | Espaço ocupado por variáveis

```
1. #include<stdio.h>
2.
3. void main(){
4.     int x = 10, y = 5;
5.     float peso = 75.5;
6.     char genero = 'M';
7.     //crie outras variaveis
8.
9.     printf("\n Espaco alocado para x = %d", sizeof(x));
10.    printf("\n Espaco alocado para y = %d", sizeof(y));
11.    printf("\n Espaco alocado para peso = %d", sizeof(peso));
12.    printf("\n Espaco alocado para genero = %d", sizeof(genero));
13. }
14.
```

Fonte: elaborado pela autora.

Todas essas informações alinharão a equipe para o uso consciente das variáveis para o armazenamento de dados.

Faça valer a pena

1. Variáveis são usadas para guardar valores temporariamente na memória de trabalho. 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.

- Todas as linguagens de programação possuem os mesmos tipos primitivos de dados.
- 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.

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. Na linguagem C, existem algumas constantes já definidas na biblioteca math.h.

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. A constante MPI é usada para referenciar o valor do pi.
- e. A constante definida pelo comando #define g 9.8 ocupará 4 bytes na memória.

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.

```
1. #include<stdio.h>
2. void main() {
3.     int idade = 0;
4.     float salario = 0;
5.     char a_letra = 'a';
6.     char A_letra = 'A';
7. }
```

- 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.

Seção 3

Operações e expressões

Diálogo aberto

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.

Parabéns pelo trabalho! Sua equipe já está dominando os tipos de dados que podem ser utilizados em um sistema computacional e já sabem como declará-los e guardá-los temporariamente em variáveis. Entretanto, a essência de um computador consiste em receber os dados, processá-los e exibir os resultados. Logo, chegou o momento de ensinar sua equipe a processar os dados por meio de operações e expressões matemáticas e a combiná-las usando operadores lógicos.

Para que o treinamento continue contribuindo com o aperfeiçoamento teórico e prático, crie um programa em C que calcule a quantidade média de veículos registrados no estado de São Paulo em cada ano, usando os dados da Tabela 1.5. Qual a média de veículos registrados em 2014? E em 2015? E 2016? Conseguimos saber qual ano obteve a maior média? Existe outra forma de comparar os dados? O programa criado deve ser mostrado a toda equipe.

Tabela 1.5 | Quantidade de veículos no estado de São Paulo

Tipo de veículo	Quantidade de veículos por ano		
	2014	2015	2016
Carro	16.319.979	16.834.629	17.247.123
Moto	4.133.366	4.268.872	4.378.772
Caminhão	658.713	664.617	669.056

Fonte: IBGE (2020).

Para cumprir com êxito a missão, você verá nesta seção como utilizar os operadores aritméticos, relacionais e lógicos, além de conhecer algumas funções predefinidas.

Bons estudos!

Não pode faltar

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 que serão 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 a partir de arquivos de texto, planilhas, arquivos pdf, entre outros.
- Acesso a banco de dados: caso em que são usados programas que fazem o gerenciamento da base de dados ao qual o sistema computacional possui acesso.

Nos três casos, a leitura dos dados é feita apenas para processar e gerar informações, e essa etapa é construída a partir da combinação de operações aritméticas, relacionais, lógicas e outras técnicas de programação que você conhecerá no decorrer do livro.

Operadores aritméticos em linguagens de programação

Vamos começar a aprimorar nossos algoritmos com as operações aritméticas. Veja no Quadro 1.20 algumas operações disponíveis nas linguagens de programação e seus respectivos exemplos.

Quadro 1.20 | Operações aritméticas básicas em linguagens de programação

Operador	Descrição	Exemplo	Resultado
+	Soma	$4 + 2$	6
-	Subtração	$4 - 2$	2
*	Multiplicação	$4 * 2$	8
/	Divisão	$4 / 2$	2
=	Atribuição	$x = 4$	$x = 4$
%	Módulo	$4 \% 2$	0

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

Exemplificando

Vamos criar um programa em C que soma a idade de duas pessoas e imprime na tela o resultado. No Quadro 1.21 está o código que realiza tal processo. Veja que, primeiramente, as idades foram solicitadas e armazenadas em duas variáveis (linhas de 6 a 9 – entrada pelo usuário), para depois ser feito o processamento (linha 10) e, por fim, a exibição do resultado na tela.

Quadro 1.21 | Soma da idade de duas pessoas

```
1. #include<stdio.h>
2. main(){
3.     int idade1=0;
4.     int idade2=0;
5.     int resultado=0;
6.     printf("Digite a primeira idade: ");
7.     scanf("%d",&idade1);
8.     printf("Digite a segunda idade: ");
9.     scanf("%d",&idade2);
10.    resultado = idade1 + idade2;
11.    printf("Resultado = %d",resultado);
12. }
```

Fonte: elaborado pela autora.

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

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

Refletia

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 se 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 um lucro ou prejuízo?

Das operações aritméticas apresentadas no Quadro 1.20, 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.

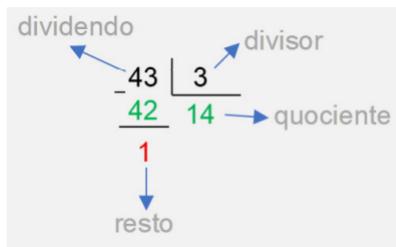
Exemplificando

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

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

Ao executar o código, obtém-se como resultado o resto da divisão, ou seja, nesse caso, o valor 1. Veja na Figura 1.7 o cálculo matemático que é efetuado e como o resultado é obtido.

Figura 1.7 | Operação aritmética módulo



Fonte: elaborada pela autora.

Os operadores aritméticos podem ser classificados em unário ou binário (MANZANO, 2015). Os binários, que nós já conhecemos no Quadro 1.20, são operadores que usam dois componentes; já os operadores unários usam apenas um componente. 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 um após a primeira execução.
 - Pós-decremento: `x --`; nesse caso, é decrementado um após a primeira execução.

- Antes da variável:
 - Pré-incremento `++ x`; nesse caso, é adicionado um antes da primeira execução.
 - Pré-decremento `-- x`; nesse caso, é decrementado um antes da primeira execução.

O Quadro 1.22 apresenta um resumo dos operadores unários.

Quadro 1.22 | Operadores aritméticos unários

Operador	Descrição	Exemplo	Resultado
<code>++</code>	Pós-incremento	$x++$	$x+1$
<code>++</code>	Pré-incremento	$+x$	$x+1$
<code>--</code>	Pós-decremento	$y--$	$y-1$
<code>--</code>	Pré-decremento	$--y$	$y-1$

Fonte: elaborado pela autora.

Operadores relacionais em linguagens de programação

Faz parte do processamento fazer comparações entre valores, para, a partir do resultado, realizar novas ações. Por exemplo, podemos criar um programa que soma a nota de dois bimestres de um aluno e efetua a média aritmética. A partir do resultado, se o aluno obteve média superior a seis, ele está aprovado; caso contrário, está 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 1.23 apresenta os operadores usados nas linguagens de programação (DEITEL; DEITEL, 2011).

Quadro 1.23 | Operadores relacionais em linguagens de programação

Operador	Descrição	Exemplo
<code>==</code>	igual a	$x == y$
<code>!=</code>	diferente de	$x != y$
<code>></code>	maior que	$x > y$
<code><</code>	menor que	$x < y$
<code>>=</code>	maior ou igual que	$x >= y$
<code><=</code>	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 ou falso. Quando fazemos uma comparação na linguagem C, o resultado será um ou zero, sendo que o primeiro representa um resultado **verdadeiro** e o segundo um **falso**.

Vamos criar um programa que solicita ao usuário dois números inteiros e faz algumas comparações com esses valores. Veja, no Quadro 1.24, que, na linha 9, comparamos se os números são iguais; na linha 10, se o primeiro é maior que o segundo e, na linha 11, se o primeiro é menor ou igual ao segundo.

Quadro 1.24 | Comparações entre dois números

```
1. #include<stdio.h>
2. main() {
3.     int n1=0;
4.     int n2=0;
5.     printf("Digite o primeiro numero: ");
6.     scanf("%d",&n1);
7.     printf("Digite o segundo numero: ");
8.     scanf("%d",&n2);
9.     printf("\n n1 e n2 sao 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.

Operadores lógicos em linguagens de programação

Além dos operadores relacionais, outro importante recurso para o processamento é a utilização de operadores lógicos, que possuem como fundamento a lógica matemática clássica e a lógica boolena (GERSTING, 2017). O Quadro 1.25 apresenta os operadores lógicos que podem ser usados nas linguagens de programação.

Quadro 1.25 | Operadores lógicos em linguagens de programação

Operador	Descrição	Exemplo
!	negação (<i>not</i>)	$!(x == y)$
&&	conjunção (<i>and</i>)	$(x > y) \&\& (a == b)$
	disjunção (<i>or</i>)	$(x > y) (a == b)$

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

Os operadores lógicos são utilizados juntamente com os relacionais, criando comparações 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 sejam verdadeiras. O operador de disjunção é usado para criar condições em que basta uma condição ser verdadeira para que o resultado também seja.

Veja, no Quadro 1.26, 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 a primeira condição não é verdadeira, logo o resultado da expressão será 0. Na linha 5, a condição criada será verdadeira caso uma das condições seja satisfeita, logo o resultado será 1. Por fim, na linha 6, invertemos esse resultado com o operador de negação.

Quadro 1.26 | Operadores relacionais e lógicos

```
1. #include<stdio.h>
2. 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 linguagem de programação

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.

C STANDARD Library Reference Tutorial. **Tutorials point:** simply easy learning. 2020.

Neste livro, 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 1.27).

Quadro 1.27 | Algumas bibliotecas e funções na linguagem C

Biblioteca	Função	Descrição
<code><stdio.h></code>	<code>printf()</code> <code>scanf()</code> <code>fgets(variavel, tamanho, fluxo)</code>	Imprime na tela. Faz leitura de um dado digitado. Faz a leitura de uma linha.
<code><math.h></code>	<code>pow(base,potencia)</code> <code>sqrtnumero)</code> <code>sin(angulo)</code> <code>cos(angulo)</code>	Operação de potenciação. Calcula a raiz quadrada. Calcula o seno de um ângulo. Calcula o cosseno de um ângulo.
<code><string.h></code>	<code>strcmp(string1, string2)</code> <code>strcpy(destino,origem)</code>	Verifica se duas <i>strings</i> são iguais. Copia uma <i>string</i> da origem para o destino.
<code><stdlib.h></code>	<code>malloc(tamanho)</code> <code>realloc(local,tamanho)</code> <code>free(local)</code>	Aloca dinamicamente espaço na memória. Modifica um espaço já alocado dinamicamente. Libera um espaço alocado dinamicamente.

Fonte: adaptado de Tutorials point ([s.d.]).

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 possuir o caractere, não coincidente e “menor” que a segunda, então o valor resultante é negativo. Por exemplo, o caractere “d” é menor que o “h”.

A função `malloc(tamanho)` é muito utilizada em estruturas de dados para alocar espaços dinâmicos na memória de trabalho. O parâmetro “tamanho” se refere à quantidade de bytes se deseja alocar. Algumas sintaxes possíveis para essa função são:

1. int x=0.
2. malloc(4).
3. malloc(sizeof(float)).
4. malloc(sizeof(x)).

Na linha 2, serão alocados 4 *bytes* na memória. Na linha 3, será alocado o tamanho de uma variável do tipo *float*. Na linha 4, será alocado o tamanho da variável x, que é do tipo *int*, ou seja, 4 bytes.

Assimile

Cada função, independentemente da linguagem de programação, precisa ter um tipo de retorno, por exemplo, retornar um inteiro, um real, um booleano, um void, que significa tipo genérico, dentre outros.

As funções de alocação dinâmica de memória, por exemplo, a malloc() possui como tipo de retorno um endereço de memória, por isso seu uso é comumente vinculado a ponteiros (lembando que são variáveis especiais que armazenam endereços).

Com esta seção, finalizamos a primeira unidade, na qual exploramos as formas de armazenamento temporário de dados em diversos tipos de variáveis e como podemos utilizar os operadores para realizar o processamento dos dados. Nas próximas seções, você aprenderá novas técnicas de programação que lhe permitirão criar estratégias de processamento ainda mais complexas.

Sem medo de errar

Chegou o momento de realizar uma etapa importante no treinamento dos jovens programadores, pois você deverá ensiná-los a implementar o processamento de alguns dados estatísticos. Como entrada, você tem acesso aos dados de veículos registrados no estado de São Paulo nos anos de 2014, 2015 e 2016.

Foi solicitado o cálculo da média de veículos cadastrados em cada ano e, para isso, o primeiro passo consiste em armazenar os dados em variáveis. Nesse ponto, vale a pena ressaltar que não existe uma única maneira de implementar uma solução, já que cada programador tem seu estilo e suas técnicas de implementação.

Vamos realizar o armazenamento dos dados usando vetores. Cada ano será um vetor com espaço para armazenar três valores: na posição zero, serão armazenados os carros; na posição um, as motos e, na posição dois, os caminhões. Portanto, crie três variáveis compostas, conforme exemplo:

```
int ano_2014[3]={0};
```

Vamos precisar de mais três variáveis para guardar a média de cada ano.

```
float media_2014 = 0;  
float media_2015 = 0;  
float media_2016 = 0;
```

Agora vamos inserir manualmente os valores, nos três vetores, conforme o comando:

```
ano_2014[0] = 16319979  
ano_2014[1] = 4133366  
ano_2014[2] = 658713
```

Fique à vontade para alterar o código para que o usuário digite os valores, em vez de fornecê-los manualmente. Caso opte por esse modo, use o comando:

```
scanf("%d", &ano_2014[0]);
```

Feita a inserção dos dados, agora podemos passar ao processamento das informações que, neste caso, consiste em calcular a média aritmética. Portanto, vamos somar os dados de cada ano, dividir pelo número de ocorrências (três) e guardar dentro da variável média do respectivo ano, conforme código abaixo:

```
media_2014 = (ano_2014[0] + ano_2014[1] +  
ano_2014[2])/3;
```

Veja que o somatório está entre parênteses, e isso é necessário porque a divisão tem precedência sobre a soma.

Após ter efetuado o cálculo das médias, você pode usar operadores relacionais e lógicos para descobrir qual ano obteve a maior média. Por exemplo, para verificar se 2014 foi o ano que obteve a maior média, você pode usar o comando:

```
printf("2014 obteve a maior media? %d", ((media_2014  
> media_2015) && (media_2014 > media_2016)));
```

Faça a impressão das médias e dos testes lógicos. Complete o código com os cálculos necessários e com as comparações necessárias, conforme os exemplos, execute e veja se todos os alunos estão com o programa funcionando.

Os operadores relacionais e lógicos podem ser trabalhados dentro de estruturas condicionais. Na próxima unidade, você aprenderá esse recurso e poderá aperfeiçoar suas implementações.

Sugestão de estrutura para esse tópico:

- Contextualize e relembrre a situação-problema da seção.
- Resolva-a ou, caso não seja possível uma única resposta, aponte caminhos para tal.
- Faça a ligação entre a resolução da situação-problema e o resultado de aprendizagem da unidade.

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 divide esse valor pela quantidade de integrantes da mesa e calcula o desconto concedido. O programa deve receber como dados 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 da situação-problema

O código, no Quadro 1.28, apresenta o resultado do problema. Um ponto importante é o cálculo do desconto feito na linha 13, para cuja montagem utilizamos uma regra de três simples. Outro ponto é o cálculo do valor por pessoa, feito na linha 15 diretamente dentro do comando de impressão. Esse recurso pode ser usado quando não é preciso armazenar o valor.

```
1. #include<stdio.h>
2. main(){
3.     float valor_bruto=0;
4.     float valor_liquido=0;
5.     float desconto=0;
6.     int qtd_pessoas=0;

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

13.    valor_liquido = valor_bruto - (valor_bruto * desconto/100);
14.    printf("\n Valor da conta com desconto = %f",valor_liquido);
15.    printf("\n Valor a ser pago por pessoa = %f",valor_li-
16.        quido/qtd_pessoas);
}
```

Fonte: elaborado pela autora.

Faça valer a pena

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 o comando `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.

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

- I. O resultado será um para $a = 30, b = 20, c = 10$

PORQUE

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

Assinale a alternativa correta

- a. As afirmações I e II são verdadeiras, e a segunda é uma justifica válida da primeira.
- b. As afirmações I e II são verdadeiras, mas a segunda não é uma justifica válida da primeira.
- c. Somente a afirmação I é verdadeira.
- d. Somente a afirmação II é verdadeira.
- e. As afirmações I e II não são verdadeiras.

3. Algumas soluções em software necessitam de alocação dinâmica de memória. Nesses casos, é comum utilizar estruturas de dados chamadas de listas, filas, pilhas, árvores e grafos (EDELWEISS; GALANTE, 2009). Para a alocação dinâmica, usam-se funções da biblioteca `<stdlib.h>`, tais como a função `malloc(tamanho)`.

```
1. #include<stdlib.h>
2. struct automovel{
3.     char modelo[20];
4.     int ano;
5. };
6. main(){
7.     struct automovel auto1;
8.     malloc(sizeof(auto1));
9. }
```

Considerando o código apresentado, escolha a alternativa correta:

- a. Esse código não pode ser executado.
- b. A função `malloc()` alocará 4 *bytes* na memória.
- c. A função `malloc()` alocará 20 *bytes* na memória.
- d. A função `malloc()` alocará 24 *bytes* na memória.
- e. A função `malloc()` alocará 0 *bytes* na memória.

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.
- C STANDARD Library Reference Tutorial. **Tutorials point:** simply easy learning. 2020. Disponível em: https://www.tutorialspoint.com/c_standard_library/index.htm. Acesso em: 23 jul. 2018.
- DAMAS, L. **Linguagem C.** Tradução: João Araújo Ribeiro, Orlando Bernardo Filho. 10. ed. Rio de Janeiro: LTC, 2016.
- DEITEL, P.; DEITEL, H. C. **Como programar.** 6. ed. São Paulo: Pearson, 2011.
- DIA DIAGRAM EDITOR. Dia installer. 2020. Disponível em: <http://dia-installer.de/>. Acesso em: 17 mar. 2018.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação:** a construção de algoritmos e estruturas de dados. São Paulo: Makron, 2000.
- GERSTING, J. L. **Fundamentos matemáticos para a ciência da computação:** matemática discreta e suas aplicações. Rio de Janeiro: LTC, 2017.
- HOUAIS, A.; FRANCO, F. M. M.; VILLAR, M. S. **Dicionário Houaiss da Língua Portuguesa.** São Paulo: Objetiva, 2001.
- IBGE. Instituto Brasileiro de Geografia e Estatística. **Frota de veículos.** Pesquisas. 2020. Disponível em: <https://cidades.ibge.gov.br/brasil/sp/pesquisa/22/28120?ano=2014>. Acesso em: 10 mar. 2020.
- LOPES, A.; GARCIA, G. **Introdução à programação.** 8. reimpr. Rio de Janeiro: Elsevier, 2002.
- LUCIDCHART. 2020. Página inicial. Disponível em: <https://www.lucidchart.com/pages/pt>. 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. **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.
- MARÇULA, M. **Informática:** conceitos e aplicações. 4. ed. rev. São Paulo: Érica, 2013.
- MICROSOFT. Math Constants. Syntax. **Visual Studio.** 2016. Disponível em: <https://msdn.microsoft.com/pt-br/library/4hwaceh6.aspx>. Acesso em: 11 mar. 2020.
- MIZRAHI, V. V. **Treinamento em linguagem C.** 2. ed. São Paulo: Pearson Prentice Hall, 2008.

PEREIRA, A. P. **O que é algoritmo.** 2009. Disponível em <https://www.tecmundo.com.br/programacao/2082-o-que-e-algoritmo-.htm>. Acesso em: 19 mar. 2018.

PEREIRA, S. do L. **Linguagem C.** Ime: USP, 2017. Disponível em: <https://www.ime.usp.br/~svlago/slago-C.pdf>. Acesso em: 11 mar. 2020.

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.

SCHILD'T, 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.

TARIFA, A. O que é Big Data? **Endeavor**, [s.d.]. Disponível em: <https://endeavor.org.br/big-data-descubra-o-que-e-e-como-usar-na-sua-empresa/>. Acesso em: 23 jul. 2018.

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

VISUALG. 2020. Página inicial. Disponível em: <http://visualg3.com.br/>. Acesso em: 10 mar. 2020.

Unidade 2

Marcio Aparecido Artero

Estruturas de decisão e repetição

Convite ao estudo

Podemos observar que tudo que realizamos em nossa vida envolve uma condição, certo? 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, 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, tendo como diferencial 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 condicional (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!

Seção 1

Estruturas de decisão condicional

Diálogo aberto

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 condicional e de seleção, 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 sistema de computador em um dos seus clientes. Para que o tempo seja suficiente e você possa usufruir do seu presente, foram oferecidas as seguintes condições: se você instalar o sistema 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: realizarem um programa em linguagem C que calculasse o valor do salário bruto, levando em consideração os descontos de INSS e Imposto de Renda (tabelas 2.1 e 2.2). Para efeito de documentação, ao final da compilação do programa vocês devem realizar um relatório com o código-fonte.

Tabela 2.1 | Descontos INSS

SALÁRIO DE CONTRIBUIÇÃO (R\$)	ALÍQUOTA / INSS
Até R\$ R\$ 1.045,00	7,5%
De R\$ 1.045,00 até 2.089,60	9%
De 2.089,61 até R\$ 3.134,40	12%
De R\$ 3.050,53 a R\$ 6.101,06	14%

Fonte: elaborada pelo autor.

Tabela 2.2 | Descontos IR

SALÁRIO (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.

Diante dessas informações, cabe uma pergunta: a estrutura de decisão nos garante a viabilidade nas mais diferentes condicionais de programação? Ou existem formas mais rápidas para programar condicionais?

Muito bem, aproveite ao máximo seus estudos e boa sorte!

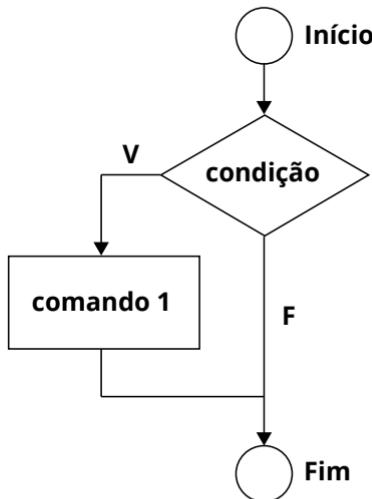
Não pode faltar

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 de decisão e seleção. Pois bem, vamos iniciar a seção com a estrutura de decisão condicional: if/else (se/então).

Segundo Manzano (2013), para a solução de um problema envolvendo situações, podemos utilizar a instrução “if” (em português “se”), cuja função é tomar uma decisão e criar 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 fim de uma instrução. Veja na Figura 2.1 como fica a estrutura condicional simples utilizando fluxograma:

Figura 2.1 | Fluxograma representando a função “if”



Fonte: elaborada pelo autor.

Na sequência, veja a sintaxe da instrução “if” (se) utilizada na linguagem C:

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

Muito bem, nada melhor do que visualizar uma aplicação condicional na prática. 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, não retornará nada. Veja no exemplo que segue a situação de um jovem que verifica se poderá ou não tirar a carteira de habilitação:

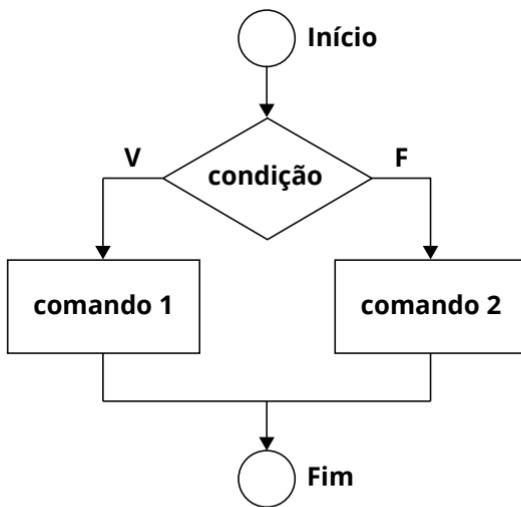
```
1. int main()
2. {
3.     float idade;
4.     printf("Digite sua idade: \n");
```

```

5.         scanf("%f", &idade);
6.     if (idade>=18)
7.     {
8.         printf("Voce ja pode tirar sua carteira de
9. Habilitacao, voce e maior de 18");
10.    }
11.    return 0;
12. }
```

Nesse exemplo, não é considerado o “se não” (*else*). Simplesmente, se a condição não for verdadeira, ela não exibirá nada como resposta. 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 2.2:

Figura 2.2 | Fluxograma representando as funções “if” e “else”



Fonte: elaborada pelo autor.

Agora, veja a representação da sintaxe:

```

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

```
else
{
<segundo conjunto de comandos>;
}
```

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 que R\$ 10.000,00 eles farão uma viagem internacional; caso contrário, deverão fazer uma viagem nacional.

```
1. #include <stdio.h>
2. int main() {
3.     float orcamento;
4.     printf("Digite o valor do orçamento para viagem
5. \n");
6.     scanf("%f", &orcamento);
7.     if (orcamento >=10000)
8.     {
9.         printf("\n Joao e maria possuem orçamento
10. para uma viagem internacional, pois seu orçamento
11. e de %f", orcamento);
12.     }
13.     else
14.     {
15.         printf("\n Joao e Maria irão optar por uma viagem
16. nacional, seu orçamento ficou em %f", orcamento);
17.     }
18.     return 0;
19. }
```

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 positivo ou negativo, representando uma estrutura condicional composta:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num>0)
    {
        printf ("\n\nO numero e positivo\n");
    }
    else
    {
        printf ("O numero e negativo");
    }
    return 0;
}
```

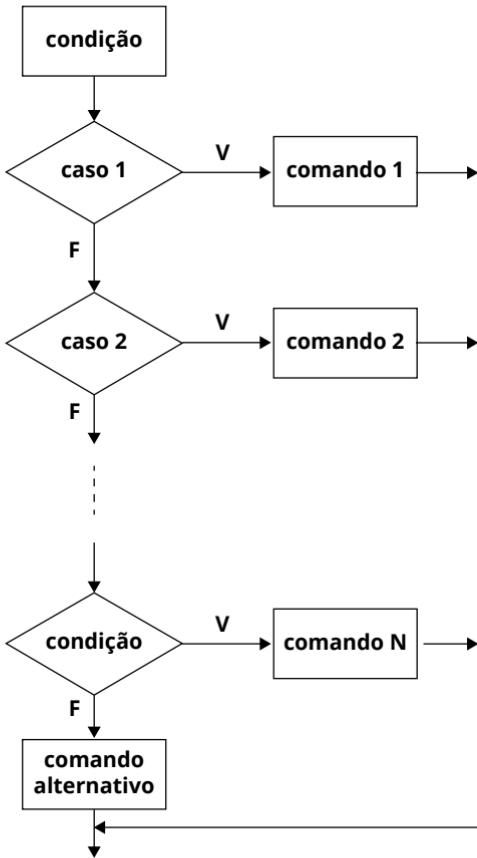
Dando sequência aos estudos, vamos conhecer a **estrutura condicional de seleção de casos**, *switch-case*, que, segundo Schildt (1997, p. 35), “testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere”. Quando os valores são avaliados o comando é executado.

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

- 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 2.3 o fluxograma representando a estrutura condicional de seleção de casos:

Figura 2.3 | 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;
    ...
}
```

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

Veja que o comando break é utilizado para forçar a saída do laço de repetição, 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.



Para consultar um exemplo de um código cuja finalidade é descobrir o desconto que um cliente terá, de acordo com a escolha de uma cor específica marcada no produto, acesse o QR Code ou o link:
<https://go.aws/2xO87u2>.

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 Inclusiva – OR

Fonte: elaborado pelo autor.

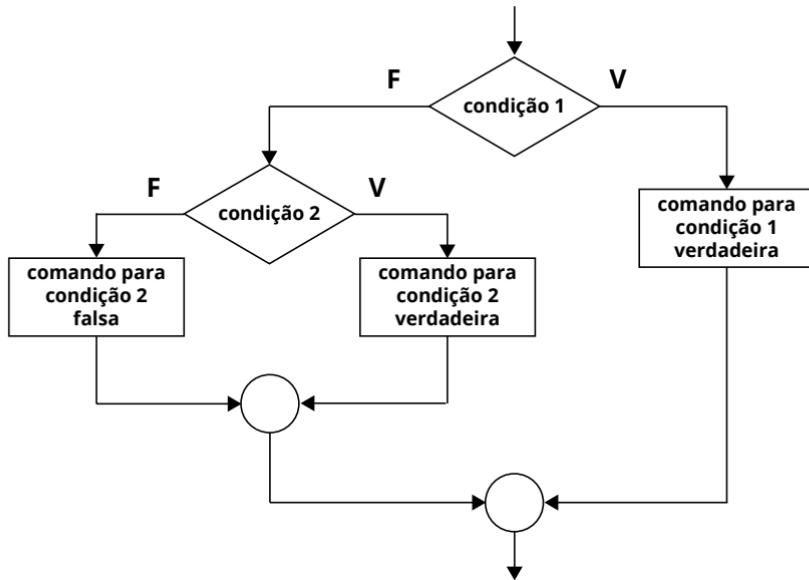
Quadro 2.2 | Tabela Verdade

A	B	A && B	A B	! A	! B
Verdade	Verdade	Verdade	Verdade	Falso	Falso
Verdade	Falso	Falso	Verdade	Falso	Verdade
Falso	Verdade	Falso	Verdade	Verdade	Falso
Falso	Falso	Falso	Falso	Verdade	Verdade

Fonte: elaborado pelo autor.

Para finalizar a seção que trata das estruturas de decisão e seleção, 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 2.4 um dos tipos de fluxogramas que representa uma estrutura condicional encadeada.

Figura 2.4 | Fluxograma estrutura condicional encadeada



Fonte: elaborada pelo autor.

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

```
if (condição) comando;  
else  
    if (condição) comando;  
    else (condição) comando;  
    .  
    .  
    .  
else comando;
```

Agora, veja no programa a seguir uma estrutura condicional encadeada, em que serão analisados os tipos de triângulo, partindo da premissa que ele deverá ser testado antes para verificar se forma ou não um triângulo:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main( ) {
4.     int a, b, c;
5.     printf("Classificacao do triangulo: informe a
6. medida dos lados
7. apertando a Tecla ENTER para cada medida:\n");
8.     scanf("%d %d %d", &a, &b, &c);
9.     if (a< b + c && b< a + c && c < a + b)
10.    {
11.            printf("\n\n Dadas as medidas: %d,
12. %d, %d, temos um triangulo", a, b, c);
13.
14.            if( a == b && a == c)
15.            {
16.                printf("Este     e      um      triangulo
17. EQUILATERO! \n");
18.            }
19.            else
20.                    if ( a==b || a == c || b
21. ==c)
22.                    {
23.                        printf("Este     e      um
24. triangulo ISOSCELES!\n");
25.                    }
26.                    else
27.                    printf("Este     e
28. um triangulo ESCALENO!
29. \n");
30.            }
31.            else
32.            printf("\n\n As medidas fornecidas,
33. %d,%d,%d nao formam
```

```
34. um triangulo", a, b, c);  
35. return 0;  
36. }
```

Refita

Pense nas possibilidades que você pode ter usando as estruturas de tomadas de decisão. “if-else”, if-else-if e “switch-case”. Lembre-se que para cada caso poderá haver uma particularidade diferente em desenvolver um programa. Imagine se você tivesse que criar um programa que calculasse o desempenho dos professores, de acordo com as suas produtividades: como você faria?

Quanta informação e quantas possibilidades foram criadas nesta seção! Porém, não pense que acabou. Estamos apenas começando. Bons estudos e até a próxima seção!

Sem medo de errar

Agora que você já conhece as estruturas de decisão condicional e de seleção, 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.

Veja as Tabelas 2.1 e 2.2 novamente e resolva o problema utilizando a estrutura de decisão condicional. Vamos lá!

Tabela 2.1 | Descontos INSS

SALÁRIO DE CONTRIBUIÇÃO (R\$)	ALÍQUOTA / INSS
Até R\$ R\$ 1.045,00	7,5%
De R\$ 1.045,00 até 2.089,60	9%
De 2.089,61 até R\$ 3.134,40	11%
De R\$ 3.050,53 a R\$ 6.101,06	14%

Fonte: elaborada pelo autor.

Tabela 2.2 | Descontos IR

SALÁRIO (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.

```

1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main()
4.  {
5.      float salario, inss, ir, sal_liquido;
6.
7.      printf("Calculo de Salario Liquido Com desconto do
8.          IR e INSS\n\n");
9.      printf("\nDigite seu salario Bruto\n");
10.     scanf_s("%f", &salario);
11.     //Calcular o INSS
12.     if ( salario <= 1693.72)
13.     {
14.         inss = salario * 0.08;
15.     }
16.     else
17.         if ( salario >= 1693.73 && salario <=
18.             2822.90)
19.         {
20.             inss = salario * 0.09;
21.         }
22.         else
23.             if ( salario >= 2822.91 && salario <=
24.                 5645.80)
25.             {
26.                 inss = salario * 0.11;
27.             }

```

```
28.         else
26.         {
27.             inss = 621.04;
28.         }
29.     //Calcular i IR
30.     if ( salario <= 1903.98 )
31.     {
32.         ir = salario*0;
33.     }
34.     else
35.         if ( salario >= 1903.99 && salario <=
36. 2826.65)
37.         {
38.             ir = salario * 0.075;
39.         }
40.     else
41.         if ( salario >= 2826.66 && salario <=
42. 3751.05 )
43.         {
44.             ir = salario * 0.15;
45.         }
46.     else
47.         if ( salario >= 3751.06 && salario <=
48. 4664.68 )
49.         {
50.             ir = salario * 0.225;
51.         }
52.     else
53.         if ( salario > 4664.69 )
54.         {
55.             ir = salario * 0.275;
56.         }
57.     //Calculo do Salario liquido
58.     sal_liquido = (salario - inss) - ir;
59.     //Resultados
60.     printf( "\nDesconto do INSS e: %.2f\n\n", inss);
```

```
61.     printf( "Desconto do imposto de renda e: %.2f\n\n",  
62.     ir);  
63.     printf( "Salario liquido: %.2f\n\n", sal_liquido);  
64.     return 0;  
65. }
```

Essa 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, e até mesmo os benefícios na sua folha de pagamento. Treine bastante, otimize ao máximo e ótimos estudo!

Avançando na prática

Semana do desconto

Na dinâmica do dia a dia de uma pizzaria, você resolveu realizar um programa em linguagem C para que em cada dia da semana fosse oferecido 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; e no domingo, ganha-se o refrigerante. Existem várias formas de criar esse programa em linguagem C, certo? Qual maneira você escolheria para criar esse programa?

Resolução da situação-problema

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

```
1. #include <stdio.h>  
2. int main ( )  
3. {  
4.     int compra;  
5.     printf ("Digite um numero que corresponde o dia  
6.     semana e seu respectivo desconto \n");  
7.     printf("Digite 1 para Domingo\n");  
8.     printf("Digite 2 para Segunda\n");  
9.     printf("Digite 3 para Terca\n");
```

```
10.     printf("Digite 4 para Quarta\n");
11.    printf("Digite 5 para Quinta\n");
12.    printf("Digite 6 para Sexta\n");
13.    printf("Digite 7 para Sabado\n");
14.    scanf("%d", &compra);
15.    switch ( compra )
16.    {
17.        case 1 :
18.            printf ("Domingo e dia de refri gratis\n");
19.            break;
20.        case 2 :
21.            printf ("Segunda o desconto sera de 40 por
22. cento no valor da pizza\n");
23.            break;
24.        case 3 :
25.            printf ("Terca o desconto sera de 30 por
26. cento no valor da pizza\n");
27.            break;
28.        case 4 :
29.            printf ("Quarta e dia de pizza em dobro
30. \n");
31.            break;
32.        case 5 :
33.            printf ("Quinta o desconto sera de 20 por
34. cento no valor da pizza \n");
35.            break;
36.        case 6 :
37.            printf ("Sexta o desconto sera de 10 por
38. cento no valor da pizza\n");
39.            break;
40.        case 7 :
41.            printf ("Sabado nao tem desconto\n");
42.            break;
43.        default :
44.            printf ("\n O valor digitado nao corresponde
45. a nenhum dia da semana\n");
```

```
46.     getch();
47.     return 0;
48. }
```

Pois bem, agora é sua vez de treinar um pouco. Boa sorte e ótimo estudo!

Faça valer a pena

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, no entanto, 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” é a afirmação de um comando “if”.

2. A estrutura condicional encadeada, também conhecida como ifs aninhados, segundo Schildt (1997), 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.

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

- a. if (condição) comando;
 if (condição) comando;
 else(condição) comando;
 else comando;
- b. if (condição) comando;
 else
 if (condição) comando;
 else(condição) comando;
 else comando;
- c. if (condição) comando;
 else
 if (condição) comando;
- d. else (condição) comando;
 if (condição) comando;
 else(condição) comando;
- e. if (condição) comando;
 else
 else (condição) comando;
 else(condição) comando;
 else comando;

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, porém, não é necessariamente obrigatório. Já o comando *break* determina o fim de uma das opções de comando.
- 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.

Seção 2

Estruturas de repetição condicional

Diálogo aberto

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

Assim como as estruturas decisão, as estruturas de repetição têm a função de otimizar as soluções de problemas. Imaginemos 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 nossa frase. Fácil! Vamos criar uma outra situação: no final do evento, você irá distribuir cinco livros de computação para os primeiros que chegarem com a solução de um problema proposto. Veja que várias pessoas terão a oportunidade de resgatar o livro, porém a quantidade de livros ficou condicionada ao final. Tanto a primeira situação como a segunda estão condicionadas à estrutura de repetição, mas uma realiza o teste no início e a outra no final.

Pois bem, agora você deverá auxiliar o seu atual funcionário (ex-estagiário) a criar um programa em linguagem C para ajudar a instituição de ensino para a qual vocês prestam serviço. Foi solicitada a elaboração de um programa que receberá as notas de um semestre de uma determinada disciplina. O professor poderá realizar quantas avaliações achar necessário para a composição da nota do aluno e, por fim, deverá apresentar a média final deste.

Pense nas soluções e execute o código em um compilador de linguagem C. Apresente o código livre de erros em um documento de texto. Nesse caso específico, qual a melhor solução: usar um teste de repetição no início ou no final? Faça o teste e otimize ao máximo as possíveis soluções para o problema.

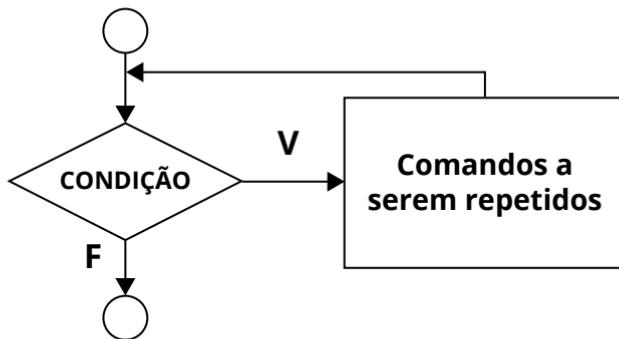
Boa sorte e uma ótima aula!

Não pode faltar

Após trabalhar as estruturas condicionais, 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 nenhuma 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 2.5 a forma simplificada do fluxograma do comando *while* direcionado para o teste no início.

Figura 2.5 | Fluxograma do comando *while*



Fonte: elaborada pelo autor.

Como o programa será elaborado em linguagem C, veja a seguir 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), irá somar 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.

Refita

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? Lembre-se, também, de que podem ocorrer os laços infinitos, nos quais as condições são sempre verdadeiras.

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

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int cont=0; // foi definido na declaração da variável
    um valor inicial de "0"
    while (cont < 10) // Será executado enquanto a cont
    for menor que 10
    {
        printf("PROGRAMA \n");
        cont++; // será necessário incrementar um valor,
        para dar sequência no programa
    }
}
```

```
    system("PAUSE");
    return 0;
}
```



O próximo exemplo é para checar se um número está entre um número escolhido e outro.

```
#include <stdio.h>
int main(void)
{
    char PARAR;
    int NUM;
    printf("\nDigite um Numero: ");
    scanf("%d", &NUM);
    while ((getchar() != '\n') && (!EOF)); // getchar()
!= '\n' está comparando esse caractere de leitura com o caractere de nova linha.
```



// EOF é um valor especial, para garantir que ele não continue tentando ler quando chegar ao final do arquivo.

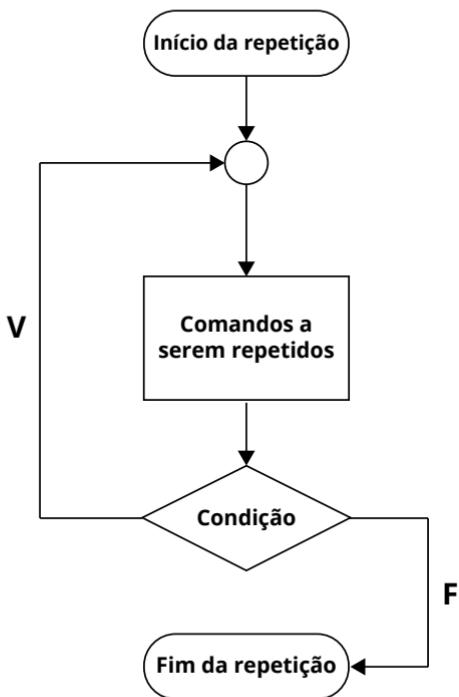
```
    if (NUM >= 10 && NUM <= 50)
        printf("O numero esta entre 10 e 50.\n");
    else
        printf("O numero nao esta entre 10 e 50.\n");
        printf("\n");
        printf("aperte <Enter> para parar... ");
    PARAR = getchar();
    return 0;
}
```

Vale ressaltar que a instrução `while ((getchar() != '\n') && (!EOF))` está sendo apresentada, sendo o operador relacional diferente de `(!=)`.

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, o usuário tem a possibilidade de digitar novamente uma nova informação.

Vamos ao fluxograma utilizando o teste de repetição no final:

Figura 2.6 | Fluxograma com teste de repetição no final



Fonte: elaborada pelo autor.

Veja como fica a sintaxe para realização da repetição com teste no final:

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

O exemplo abaixo realiza 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:

1. #include <stdio.h>
2. main() {
3. float metragem1,metragem2,resultado;

```

4.     int resp;
5.     metragem1 = 0;
6.     metragem2 = 0;
7.     resultado = 0;
8.     do
9.     {
10.         printf("C A L C U L O      D E      M E T R O S      Q
11. U A D R A D O S");
12.         printf("\n \n Digite a primeira metragem do
13. terreno: \n");
14.         scanf("%f", &metragem1);
15.         printf("\n Digite a segunda metragem do terreno:
16. \n");
17.         scanf("%f", &metragem2);
18.         resultado = (metragem1 * metragem2);
19.         printf("\n \n O Terreno tem = %.2f M2
20. \n", resultado);
21.         printf("Digite 1 para continuar ou 2 para
22. sair\n");
23.         scanf("%d", &resp);
24.     }while (resp==1);
25.     return 0;
26. }

```

Exemplificando

Segundo Soffner (2013), quando aplicado o comando *do-while*, as ações poderão ser executadas pelo menos uma vez antes do teste condicional. Nesse caso, é possível ter uma sequência de repetição de processos; sendo assim, é possível ter várias opções para a solução de um problema em um programa. No exemplo abaixo, você terá a opção de calcular a força digitando a massa e a aceleração e poderá, também, calcular a área e o perímetro de um círculo, digitando o raio.

```

1.     #include <stdio.h>
2.     #include <stdlib.h>
3.     main()
4.     {
5.         char material[60];

```

```

6.     float n, m, a;
7.     float raio, area, perimetro, pi;
8.     int opcao;
9.     do
10.    {
11.        printf("\t\t\n OPCOES DE CALCULOS \n");
12.        printf("\n 0. SAIR DO MENU \n");
13.        printf("\n 1. CALCULAR A FORCA \n");
14.        printf("\n 2. CALCULAR , AREA E
15. PERIMETRO DO CIRCULO \n");
16.        printf("\n 3. RETORNAR AO MENU \n");
17.        printf("\n Opcao: ");
18.        scanf("%d", &opcao);
19.        switch( opcao )
20.        {
21.            case 0:
22.                printf("SAIR...\n");
23.                break; 
24.            case 1:
25.                printf ("\n Digite a
26. massa do objeto: ");
27.                scanf("%f", &m);
28.                printf (" \n Digite a aceleracao: ");
29.                scanf("%f", &a);
30.                n=(m*a);
31.                printf("\n O calculo da forca e: %.2f \n",
32. n);
33.                break;
34.            case 2:
35.                printf("Digite o raio: ");
36.                scanf("%f", &raio);
37.                pi = 3.141592;
38.                area = pi*(raio * raio);
39.                perimetro = 2.0 * pi * raio;
40.                printf(" \n Raio: %.2f \n", raio);
41.                printf(" \n Area: %.2f \n", area);

```

```

42.         printf("\n Perimetro: %.2f \n",
43.     perimetro);
44.             break;
45.         case 3: system("cls"); 
46.             break;
47.         default:
48.             printf("OPÇÃO INVALIDA \n");
49.         }
50.     } while(opcao);
51. return 0;
52. }
```

Para dar continuidade aos nossos estudos, vamos considerar um problema que utilize a conjectura de Collatz: tomando um número natural “n”, se n for par, será dividido por 2; se n for ímpar, será multiplicado por 3 e ao resultado será somado 1. Repete-se o processo indefinidamente. A conjectura de Collatz estabelece que, com essas duas regras simples, todos os números naturais chegam até 1 (e a partir daí o laço 1,4,2,1,4,2,1,4,2,1... se repete indefinidamente). Matematicamente, as regras são:

$$(n) = n/2 \text{ se } n \text{ é par.}$$

$$(n) = 3n + 1 \text{ se } n \text{ é ímpar.}$$

Vamos, então, fazer um programa que calcula todos os números da sequência de Collatz para uma entrada qualquer. A sequência de comandos é:

- Entrar com um número inteiro positivo superior a 1.
- Se o número for par, dividir por 2.
- Se o número for ímpar, multiplicar por 3 e somar 1.
- Pela conjectura de Collatz, a sequência sempre termina em 1, como já foi explanado.
- Solicitar o número cuja sequência de Colatz será calculada.

Pois bem, agora veja a sua execução em linguagem C:

```

1. #include <stdio.h>
2. int main()
3. {
4.     int num,i;
```

```

5.     printf("\n\nDIGITE UM NUMERO PARA O PROBLEMA DE
6.     L. COLLATZ:\n");
7.     scanf("%d", &num);
8.     i=0;
9.     while(num>1)
10.    {   if(num%2==0)
11.        num=num/2;
12.      else
13.          num=3*num+1;
14.      printf("\n%d\n", num);
15.      i++;
16.    }
17.    return 0;
18. }
```

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. Ex: $y^* = x + 1$, que tem o mesmo efeito que $y = y^*(x + 1)$, neste caso, evitando colocar a variável à direita da atribuição.

Na sequência dos nossos estudos, vamos trabalhar algumas aplicações das estruturas de repetição condicional, realizando um programa que simula uma conta bancária (tela de opções das transações), adaptado do livro do Soffner (2013). Ele escreve um programa que repete uma entrada de dados até que determinada condição de saída seja atingida e, em seguida, acumule os valores digitados.

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. main()
4. {
5.     float soma=0;
6.     float valor;
7.     int opcao;
8.     do {
```

```

9.         printf("\n Digite uma Operacao");
10.        printf("\n 1. Deposito");
11.        printf("\n 2. Saque");
12.        printf("\n 3. Saldo");
13.        printf("\n 4. Sair");
14.        printf("\n Opcão? ");
15.        scanf("%d", &opcao);
16.        switch(opcao) {
17.            case 1: printf("\n Valor do deposito? ");
18.            scanf("%f", &valor);
19.            soma=soma+valor;
20.            break;
21.            case 2: printf("\n Valor do saque? ");
22.            scanf("%f", &valor);
23.            soma=soma-valor;
24.            break;
25.            case 3: printf("\n Saldo atual = R$ %.2f
26. \n", soma);
27.            break; 
28.        default: if(opcao!=4)
29.            printf("\n Opcão Invalida! \n");
30.        }
31.    }
32.    while (opcao!=4);
33.    printf("Fim das operações. \n\n");
34.    system("pause");
35.    return 0;
36. }

```

Observe que foi utilizado o laço *do-while* para implementar o menu do programa, em que sua função desejada é executada pelo menos uma vez dentro do laço, isto é, é aplicada uma estrutura de repetição usando comparativo.

Pois bem, chegamos ao final de mais uma seção do nosso livro. Agora é o momento de aplicar o conhecimento adquirido. Sucesso e ótimos estudos!

Sem medo de errar

Chegou o momento de resolver o problema solicitado pela instituição de ensino, para a qual você deverá criar um programa em linguagem C que calcule a média de um aluno de acordo com as avaliações realizadas.

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

- Criar uma variável para entrada das notas.
- Criar uma condição de entrada das notas até que um comando de saída seja executado.
- Após o lançamento das notas, digitar uma letra para sair e calcular a média do aluno.

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main()
4. {
5.     int avalia;
6.     char letra;
7.     float media;
8.     int cont=0;
9.     int soma=0;
10.    do
11.    {
12.        printf("Digite uma nota para avaliacao: \n");
13.        scanf("%d", &avalia);
14.        fflush(stdin); 
15.        cont++;
16.        soma = soma + avalia;
17.        printf("Digite qualquer letra pra continuar ou
18. '\s\' para encerrar: \n");
19.    }
20.    while (letra=getchar() != 's');
21.    printf("\n \nQuantidade de avaliacao = %d e
22. soma das notas = %d. \n", cont, soma);
23.    media = soma/cont;
24.    system("PAUSE");
25.    return 0;
```

26. }

parei aqui 09/09

Realize outros test

problema.

Avançando na prática

Valor nutricional do sorvete

Acreditamos que a maioria das pessoas aprecia um bom sorvete, porém 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 gostariam de consumir. Para solucionar esse problema, existe somente uma forma de programação ou há diversas formas permitidas?

Resolução da situação-problema

Com o intuito de resolver essa situação, vamos trabalhar a estrutura de repetição com teste no final, porém 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.

```
1. #include <stdio.h>
2. int main ()
3. {
4.     int i;
5.     do
6.     {
7.         printf ("\n\n INFORMACAO NUTRICIONAL
8. DO SORVETE\n\n");
9.         printf ("\n\n Digite um numero
10. que corresponde ao sabor desejado\n\n");
11.         printf ("\t(1)...flocos\n");
12.         printf ("\t(2)...morango\n");
13.         printf ("\t(3)...leite condensado\n\n");
14.         scanf("%d", &i);
15.     } while ((i<1) || (i>3));
16.     switch (i)
```

```
17.         {
18.             case 1:
19.                 printf ("\t\tVoce escolheu
20. flocos.\n");
21.                 break;
22.             case 2:
23.                 printf ("\t\tVoce escolheu
24. morango.\n");
25.                 break;
26.             case 3:
27.                 printf ("\t\tVoce escolheu
28. leite condensado.\n");
29.                 break;
30.             }
31.         return(0);
32.     }
```

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

Faça valer a pena

1. O comando *while* executa a rotina de um programa enquanto uma sintaxe dele for correta. Nesse caso, podemos afirmar que:

- I. O programa não executará nenhuma repetição (e as ações que ali dentro estiverem programadas) sem antes testar uma condição.
- II. Em alguns casos, quando utilizamos teste no início, pode ocorrer o famoso *loop* infinito.
- III. Geralmente usamos o comando while quando não sabemos quantas vezes o laço da condição deve ser repetido.

Com base no exposto, podemos afirmar que:

- a. Somente o item I está correto.
- b. Somente os itens I e II estão corretos.
- c. Somente o item II está correto.

- d. Os itens I, II e III estão corretos.
- e. Somente o item III está correto.

2. Levando em consideração que precisamos estar atentos para que não ocorra um loop infinito, analise as afirmações abaixo:

- 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), irá somar 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.

Com base no exposto, podemos afirmar que:

- a. Somente o item I está correto.
- b. Somente os itens I, III e IV estão corretos.
- c. Somente os itens II e III estão corretos.
- d. Somente o item IV está correto.
- e. Os itens I, II, III e IV estão corretos.

3. Segundo Soffner (2013), quando aplicado o comando *do/while*, as ações poderão ser executadas pelo menos uma vez antes do teste condicional. Nesse caso, é possível ter uma sequência de repetição de processos, de modo que é possível ter várias opções para a solução de um problema em um programa.

Analise o programa abaixo, que realiza a soma dos números positivos usando repetição com teste no final, e complete as partes que estão faltando.

```
1. #include<stdio.h>
2. int main()
3. {
4.     int n;
5.     int soma = 0;
6.     _____
7. }
```

```
8. printf("Digite um número positivo para ser somado  
9. ou negativo para sair: ");  
10. scanf("%d", &n);  
11. ___( n >= 0 )  
12. soma = soma + n;  
13. }while( _____ );  
14. printf("A soma eh %d\n", soma);  
15. return 0;  
16. }
```

Assinale a alternativa que completa o programa acima:

- 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".

Seção 3

Estruturas de repetição determinísticas

Diálogo aberto

Muito bem! Chegamos ao final de mais uma unidade. Nesta seção, você terá a oportunidade de estudar a estrutura de repetição usando “*for*” (para), o histórico e as aplicações da repetição determinística, assim como o comparativo com estruturas condicionais.

Pense na seguinte situação: uma grande empresa, no ramo de tecnologia, lança novo produto e, para popularizá-lo, resolve premiar os dez primeiros clientes que se interessarem por ele, ou seja, **para** os clientes de um **até** dez, essa ação será executada um a um até o último cliente. Bem tranquilo, certo?

Você e o seu funcionário (ex-estagiário) receberam uma nova missão da instituição de ensino à qual prestam serviços. Foi solicitado um programa em linguagem C para transformar o sobrenome digitado dos alunos em letras maiúsculas, assim, caso uma pessoa digite o sobrenome do aluno em letras minúsculas, o programa transformará automaticamente em maiúsculas. Após a criação do programa, entregue o código no formato TXT para documentação.

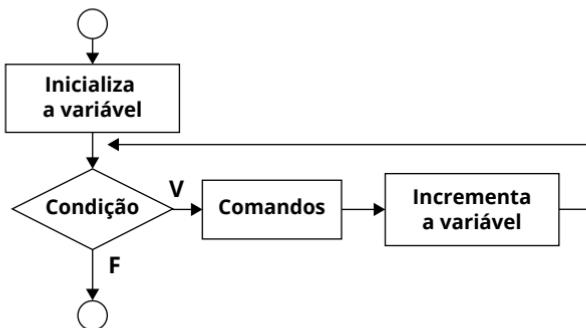
E agora? Qual função em linguagem C devemos usar para converter maiúsculas em minúsculas e vice-versa? Caro aluno, são tantas as possibilidades de aplicação de estrutura de repetição que não podemos deixar passar esse novo desafio. Então, vamos buscá-lo? Pratique bastante e boa sorte!

Não pode faltar

Chegou o momento de avançar em nossos estudos. Nesta seção, iremos aprender como a estrutura de repetição usando “*for*”, os históricos e aplicações de estruturas de repetição determinísticas e os comparativos com estruturas condicionais serão aplicados dentro da linguagem de programação.

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 uma informação por um número fixo de vezes, isto é, podemos determinar quantas vezes acontecerá a repetição. Veja na Figura 2.7 como é representada a estrutura de repetição usando o comando “*for*”.

Figura 2.7 | 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 final; incremento)
{
    comandos;
}
```

Na aplicação do comando “*for*”, você encontra três expressões separadas por ponto e vírgula. Veja abaixo o que significa cada uma delas:

- Inicialização: neste momento, coloca-se a instrução de atribuição. A inicialização é executada uma única vez antes de começar o laço.
- Condição final: 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.

Assimile

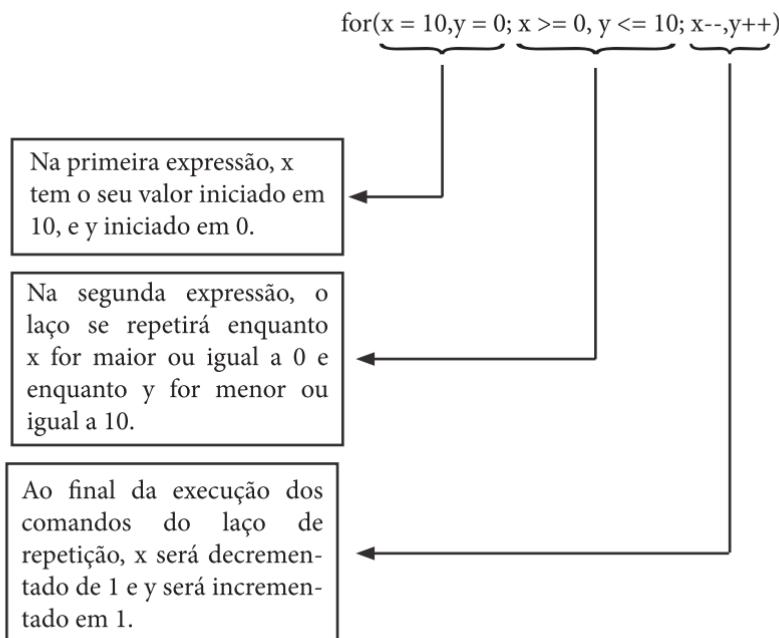
Para assimilar, vamos usar o comando iterativo *for* em várias situações, podendo-se citar:

- I. Realizar testes em mais de uma variável:

```
for (i=1, x=0; (i + x) < 10; i++, x++);
```

II. Utilizar o comando “for” para representações de caracteres, por exemplo:

`for(ch='d'; ch < 'm'; ch++)`.



Agora, vamos aplicar essa representação em um programa que mostra uma sequência de números, em que x vai de 10 a 0 e y vai de 0 a 10.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int x,y;
5.     for(x = 10, y = 0; x >= 0, y <= 10; x--, y++)
6.     {
7.         printf("x=%2d, y=%2d\n",x,y);
8.     }
9.     return 0;
10. }
```

Para o nosso próximo programa em linguagem C, criaremos uma contagem regressiva de um número qualquer, digitado pelo usuário:

Vejamos:

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int contador;
5.     printf("\nDigite um numero para contagem
6. regressiva\n\n");
7.     scanf("%d", &contador);
8.     for (contador; contador >= 1; contador--)
9.     {
10.         printf("%d ", contador);
11.     }
12.     getch();
13.     return(0);
14. }
```

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 abaixo:

```
#include <stdio.h>
main()
{
int w;
for ( w = 1; w <= 15; w++ )
{
if ( w == 8 )
{
break;
}
printf ( "%d ", w );
}
printf( "\n \n Parar a condicao de repeticao w =
%d \n", w );
return 0;
}
```

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 endereço na memória. Por armazenar diversos valores, também é chamado de variável composta ou, ainda, de estrutura matricial de dados. Veja a sintaxe abaixo para utilização de vetores homogêneos:

```
tipo variavel [n]
```

Na sintaxe acima, “[n]” representa a quantidade de colunas ou linhas.

Vamos ao nosso exemplo, que armazena valores em um vetor:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int num[5];
5.     printf("Entre com um numero\n");
6.     scanf("%d", &num[0]);
7.     printf("O valor digitado foi: %d", num [0]*2);
8.     getchar();
9. }
```

Que tal mesclar o comando *for* com *while*? O programa abaixo encontra a primeira posição para um determinado número inserido pelo usuário.

Vejamos, então, como fica a programação:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int numero;
5.     int i;
6.     int posicao=0;
7.     int vetor[10];
8.     printf("Entre com o numero de ate 3 casas
9. diferente de zero, a ser procurado em um vetor de
10. 10 posicoes: ");
11. a ser procurado em um vetor de 10 posicoes: ");
12.     scanf("%d", &numero);
13.     //Preenche o vetor com numeros
14.     for(i=0;i<10;i++)
```

```

15.      {
16.          printf("\nEntre com o numero para a posicao
17. %02d: ", i+1);
18.          scanf("%d", &vetor[i]);
19.      }
20.      //identifica a posicao do numero lido no vetor
21. de entrada
22.      while(vetor[posicao] != numero)
23.      {
24.          posicao++;
25.      }
26.      // Imprime vetor
27.      for(i=0;i<10;i++)
28.      {
29.          printf("%03d ", vetor[i]);
30.      }
31.      // Imprime espaços até) a posição do numero, e
32. em seguida um "*" sob o numero
33.
34.      printf("\n ");
35.      for(i=0;i<posicao;i++)
36.      {
37.          printf("    ");
38.      }
39.      printf("*");
40.      return 0;
41.  }

```

Segundo Damas (2016), uma instrução *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 e que, neste percurso, irá testar se foi digitado algum número ímpar, pois, caso seja ímpar, o programa continuará o teste até o fim do laço.

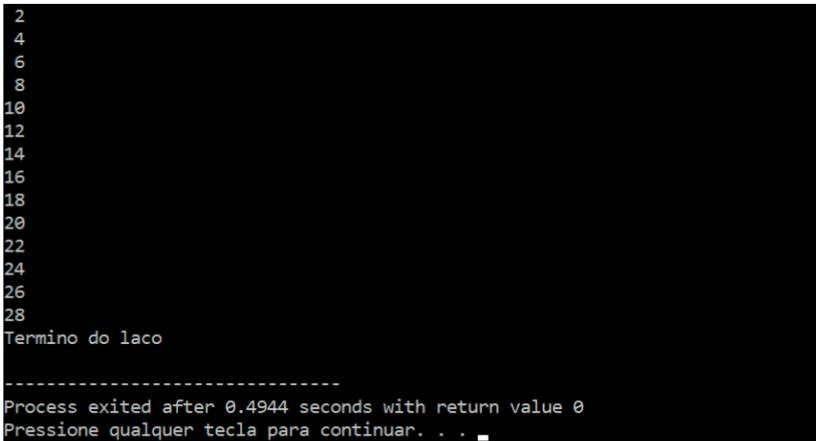
```

1. #include <stdio.h>
2. main()

```

```
3.  {
4.  int i;
5.  for (i=1; i<=100;i=i+1)
6.  if (i==30)
7.      break;
8.  else
9.      if (i%2==1)
10.         continue;
11.    else
12.        printf("%2d\n",i);
13.    printf("Termino do laco\n");
14. }
```

Figura 2.8 | Execução do laço com comando *continue*



```
2
4
6
8
10
12
14
16
18
20
22
24
26
28
Termino do laco
-----
Process exited after 0.4944 seconds with return value 0
Pressione qualquer tecla para continuar. . . -
```

Fonte: captura de tela do DEV C++.

Perceba que os valores apresentados foram os números de 2 a 28, ou seja, todos os números pares.

Reflita

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

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:

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.

Veja no programa abaixo uma matriz 3x3, em que os valores são lançados de acordo com a linha e coluna e que é montada no formato da matriz. Veja o resultado da programação na Figura 2.9.

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

```
18. {
19.     for (coluna=0;coluna<3;coluna++)
20.         printf("%d\t", matriz[linha][coluna]);
21.         printf("\n\n");
22. }
23. system("pause");
24. return 0;
25. }
```

Figura 2.9 | Resultado da programação da matriz 3x3

```
Digitar os valores da matriz para: linha 1, coluna 1: 1
Digitar os valores da matriz para: linha 1, coluna 2: 2
Digitar os valores da matriz para: linha 1, coluna 3: 3
Digitar os valores da matriz para: linha 2, coluna 1: 4
Digitar os valores da matriz para: linha 2, coluna 2: 5
Digitar os valores da matriz para: linha 2, coluna 3: 6
Digitar os valores da matriz para: linha 3, coluna 1: 7
Digitar os valores da matriz para: linha 3, coluna 2: 8
Digitar os valores da matriz para: linha 3, coluna 3: 9
Veja a sua Matriz
1      2      3
4      5      6
7      8      9

Pressione qualquer tecla para continuar. . .
```

Fonte: captura de tela do DEV C++.

Para encerrar a seção, vamos trabalhar um clássico da programação, o quadrado mágico. Ele tem vários elementos que utilizam as estruturas de decisão e repetição. O quadrado mágico é caracterizado por uma tabela quadrada em que a soma de cada coluna, de cada linha e das duas diagonais são iguais.

Uma das possíveis soluções para o problema apresentado, para quadrados de ordem $n \times n$, quando n é ímpar, é:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <ctype.h>
4. int main()
5. {
```

```
6. int Matriz[31][31], i, j, ordem=0, quadrado_da_
7. ordem=0, linha=0, coluna=0, soma=0;
8.         printf("Qual a ordem do quadrado magico?
9. (deve ser um numero impar positivo) \n");
10.        scanf("%d", &ordem);
11.        //encerra o programa se a ordem for par
12. (pois so funciona para numeros impares)
13.        if(ordem%2==0 || ordem <=0)
14.        {
15.            printf("Tem que ser numero impar positivo
16. \n");
17.            return(0);
18.        }
19.        quadrado_da_ordem = ordem * ordem;
20.        printf("Quantidade de numero no Quadrado Magico:
21. %d\n", quadrado_da_ordem);
22.        //calculo da posicao inicial (onde vai o numero
23. "1").
24.        // Quando a ordem e diferente de 1, sera sempre
25. na penultima coluna e na ultima linha.
26.        if (ordem == 1)
27.        {
28.            linha = 0;
29.            coluna = 0;
30.        }
31.        else
32.        {
33.            linha = ((ordem+1)/2) - 1;
34.            coluna = ordem - 1;
35.        }
36.        //Loop de loop para zerar a Matriz
37.        for (i=0; i<ordem; i++)
38.        {
39.            for (j=0; j<ordem; j++)
40.            {
41.                Matriz[i][j] = 0;
42.            }
```

```

43.      }
44.      //Preenchendo a Matriz com os valores do
45.      Quadrado Magico
46.      for(i=1; i<=quadrado_da_ordem; i++)
47.      {
48.          Matriz[linha][coluna] = i;
49.          //se estamos na ultima coluna, voltamos
50.          a coluna "0", do contrario, vamos uma coluna a
51.          frente.
52.          if (coluna == ordem - 1) { coluna = 0; }
53.          else { coluna++; }
54.      }
55.      // se estamos na primeira linha, vamos a
56.      ultima coluna (ordem-1), do contrario, vamos uma
57.      linha atras
58.      if (linha == 0) { linha = ordem - 1; } else
59.      { linha--; }
60.      //Se a proxima posicao ja esta; ocupada (e
61.      diferente de "0"), devemos ocupar a posicao
62.      //do lado esquerdo da ultima posicao
63.      preenchida.
64.      //Caso contrario, devemos preencher a
65.      posicao acima e a direita
66.      //da ultima posicao preenchida.
67.      if (Matriz[linha][coluna] != 0)
68.      {
69.          if (coluna == 0) { coluna = ordem - 2;
70.      } else { coluna = coluna - 2; }
71.
72.          if(linha == ordem - 1) { linha = 0; }
73.          else { linha++; }
74.      }
75.      }
76.  }
77.  printf("\nO quadrado magico de %dx%d e:\n",
78.  ordem, ordem);

```

```
79.         //Loop de loop para imprimir a Matriz
80.         for (i=0; i<ordem; i++)
81.         {
82.             for (j=0; j<ordem; j++)
83.             {
84.                 printf("%03d ", Matriz[i][j]);
85.             }
86.             printf("\n");
87.         }
88.         //Loop para calcular a soma de linhas, colunas
89.         e da diagonal principal do Quadrado magico.
90.         for(i=0; i<ordem; i++)
91.         {
92.             soma = soma + Matriz[i][i];
93.         }
94.         printf("A soma de cada linha, de cada coluna ou
95.         da diagonal principal e: %d", soma);
96.         return(0);
97.     }
```

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

Sem medo de errar

Acreditamos que você já esteja preparado para solucionar o desafio dado pela instituição de ensino em que você e seu funcionário prestam serviços. Lembrando que foi solicitado um programa em linguagem C para transformar o sobrenome digitado dos alunos em letras maiúsculas, ou seja, se o usuário digitar o sobrenome do aluno em minúsculas, o programa as transformará automaticamente em 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 irá proporcionar 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:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include < ctype.h>
4. int main( )
5. {
6.     char nome[30];
7.     int i;
8.     printf("Digite o sobrenome do aluno ou
9. aluna:\n");
10.    gets(nome);
11.    for(i=0; nome[i]!=' '; i++)
12.        nome[i] = toupper(nome[i]);
13.    printf("\n\nSobrenome convertido: %s\n\n", nome);
14.    getch();
15.    return 0;
16. }
```

Fácil! Agora tente fazer o contrário, utilizando a função “tolower”, que converte os caracteres maiúsculos em minúsculos.

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 CPFs 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. E agora, como resolver essa situação?

Resolução da situação-problema

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:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(int argc, char *argv[]) {
4.     char cpf1[14];
5.     char cpf2[11];
6.     int i=0,n=0;
7.     printf("Digite seu cpf na forma NNN.NNN.NNN-
8. NN: \n");
9.     scanf("%s",cpf1);
10.    for(i=0;i<14;i++) {
11.        if(cpf1[i] =='.' || cpf1[i]=='-') {
12.            continue;
13.        }
14.        else{
15.            cpf2[n] = cpf1[i];
16.            n++;
17.        }
18.    }
19.    printf("\n\n CPF formatado = %s",cpf2);
20.    return 0;
21. }
```

Muito bem, agora é com você! Modifique e tente otimizar ao máximo os seus programas.

Faça valer a pena

1. Segundo Manzano (2013), podemos dizer que o programa abaixo executa as seguintes instruções:

- I. Inicia o contador de índice, variável I, como 0 (zero), em um contador até 5.
- 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.

```
1. #include <stdio.h>
2. int main(void) {
3. }
4. int A[5];
5. int I, SOMA = 0;
6. printf("\nSomatorio de elementos\n\n");
7. for (I = 0; I <= 4; I++)
8. {
9. }
10. printf("Informe um valor para o elemento nr. %2d:
11. ", I);
12. scanf("%d", &A[I]);
13. for (I = 0; I <= 4; I++)
14. if (A[I] % 2 != 0)
15. SOMA += A[I];
16. printf("\nA soma dos elementos equivale a: %4d\n\n",
17. SOMA);
18. return 0;
19. }
```

Após a análise do código, é possível afirmar que:

- a. Somente o item I está correto.
- b. Somente os itens I e II estão corretos.
- c. Os itens I, II e III estão corretos.
- d. Somente o item II está correto.
- e. Somente o item III está correto.

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 final, 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.

Assinale a alternativa correta:

- a. Somatório.
- b. Finalização.
- c. Processamento.
- d. Incremento.
- e. Substituição.

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

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

Analisando o programa acima, qual é a leitura que podemos fazer da linha 5?

- 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.
- 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

BÓSON TREINAMENTOS. **13 - Programação em Linguagem C - Desvio Condisional Aninhado - if / else if.** 27 fev. 2015. Disponível em: <https://www.youtube.com/watch?v=7ZL-8tHlTTfs>. Acesso em: 26 jul. 2018.

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

DE ALUNO PARA ALUNO. **Programar em C - Como Utilizar “do while” - Aula 13.** 24 out. 2012. Disponível em: <https://www.youtube.com/watch?v=bjMD1QSVV-s>. Acesso em: 26 jul. 2018.

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

ESPERIDIÃO, H. **02 - Exercício - Estruturas de repetição em C.** 10 mar. 2017. Disponível em: <https://www.youtube.com/watch?v=PNeYeq4-Tt0>. Acesso em: 26 jul. 2018.

LOBATO, P. **Programação C - Aula 07 - while, do-while, for – Estruturas de repetição.** Disponível em: <https://www.youtube.com/watch?v=rCFh-tvoXIc>. Acesso em: 26 jul. 2018.

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.

PETERSON LOBATO. **Programação C - Aula 07 - while, do-while, for - Estruturas de repetição.** Disponível em: <https://www.youtube.com/watch?v=rCFh-tvoXIc>. Acesso em: 26 jul. 2018.

SCHILDIT, 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.

Unidade 3

Vanessa Candan Scheffer

Funções e recursividade

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 pensar sobre a 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 Android, 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 comandos. 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 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.

Prontos para mais um universo de conhecimento? Vamos lá!

Seção 1

Procedimentos e funções

Diálogo aberto

Olá!

Em uma empresa, quando você precisa resolver um problema na folha de pagamento você se dirige até o setor financeiro. Quando você vai sair de férias, você precisa ir até o RH para assinar o recibo de férias. Quando você 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, não acha? 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 possa encontrar. Uma das técnicas de programação utilizada para construir programas dessa forma é a construção de funções, que você aprenderá nesta seção.

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 engenheiros civis. Eles receberam uma demanda da construtora local para calcular e escolher qual guindaste deve ser usado em uma determinada construção para levantar as colunas de concreto armado. Eles possuem três tipos de guindastes: G1, G2 e G3. A escolha de qual utilizar depende do peso da peça que será alçado pelo equipamento, que é calculado pela fórmula $P = VR$, em que P é o peso da coluna, V é o volume e R é a constante utilizada que vale $25kN / m^3$. A regra para escolher o guindaste a ser usado está especificada na Quadro 3.1.

Quadro 3.1 | Regras para escolha do guindaste

Regra	Guindaste
Peso <= 500 kg	G1
500 < Peso <= 1500 Kg	G2
Peso > 1500 Kg	G3

Fonte: elaborado pela autora.

Como você poderá automatizar o processo, a partir da entrada das medidas da coluna (base, largura e altura)? Como o programa escolherá e informará o guindaste que deverá ser usado?

Para que você possa cumprir sua missão, você aprenderá, nesta seção, a diferença entre procedimento e função, bem como criá-los e utilizá-los.

Bons estudos!

Não pode faltar

Até esse momento você já implementou diversos algoritmos na linguagem C, não é mesmo? 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 Quadro 3.2 que imprime uma mensagem na tela. Veja o comando na linha 2, *int main()*. Esse comando especifica uma função que chama “*main*” e que vai devolver para quem a requisitou um valor inteiro, nesse caso, zero. Complicado? Calma! Vamos entender direito como construir e como funciona esse importante recurso.

Quadro 3.2 | Programa *Hello World*

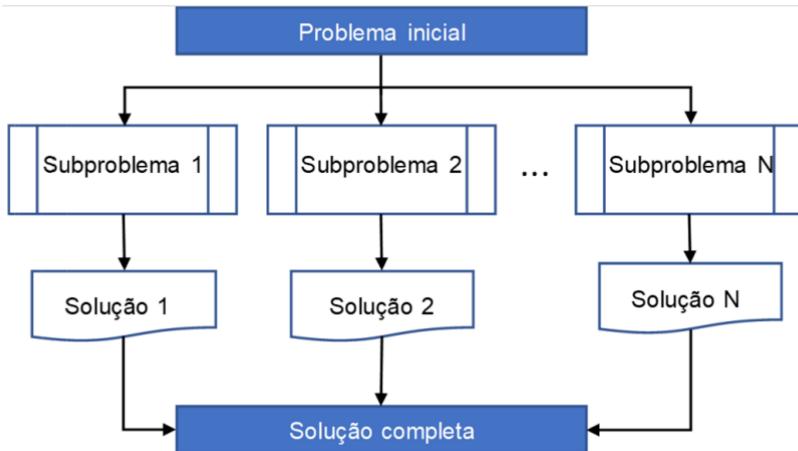
```
1. #include<stdio.h>
2. int main(){
3.     printf("Hello World!");
4.     return 0;
5. }
```

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 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 nenhum valor, usa-se o parâmetro *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 você conviá-la para sair você precisa “chamá-la pelo nome”. O nome não pode ter acento nem caractere especial, e não pode ser nome composto (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.
- <**parâmetros**> – Opcional. Veremos na próxima seção.
- <**comandos da função**> – Obrigatório. Só faz sentido criar uma função se ela tiver um conjunto de comandos para realizar.
- <**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.

Em qual parte do código a função deve ser programada?

A função *main()*, que traduzindo significa principal, é uma função de uso obrigatório em várias linguagens de programação, por exemplo, em C, em Java, em C# etc. 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 Quadro 3.3 um programa que utiliza uma função para calcular a soma entre dois números e, a seguir, sua explicação detalhada.

Quadro 3.3 | Programa com a função *somar()*.

```
1. #include<stdio.h>
2. int somar(){
3.     return 2 + 3;
4. }
5. int main(){
6.     int resultado = 0;
7.     resultado = somar();
8.     printf("O resultado da funcao e = %d",resultado);
9.     return 0;
10. }
```

Fonte: elaborado pela autora.

A função *somar()* no Quadro 3.3 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 adequada. Por exemplo: se os comandos tivessem sido escritos na forma:

```
int somar () {
    int x = 0;
    x = 2 + 3;
    return x;
}
```

a função teria o mesmo efeito. Mas veja: da forma que está no Quadro 3.3, escrevemos duas linhas a menos (imagine essa diferença sendo acumulada nas mais de 20 milhões linhas do Linux). Veja que no código do Quadro 3.3 foi usada uma variável a menos, o que significa economia de memória de trabalho e do processamento para a 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 Quadro 3.4 o código para essa solução. Todo programa sempre começa pela função *main()*, ou seja, esse programa começará a ser executado na linha 8. Na linha 10 a função *calcular()* é chamada, ou seja, a execução “pula” para a linha 2. Na função *calcular()* é solicitado um valor ao usuário (linha 4), armazenado em uma variável (linha 5) e retornado o valor multiplicado por ele mesmo (linha 6). Após retornar o valor, a execução do programa “volta” para a linha 10, pois foi nesse ponto que a função foi chamada. O resultado da função é armazenado na variável *resultado* e impresso na linha 11.

Quadro 3.4 | Função para calcular o quadro de um número

```
1. #include<stdio.h>
2. float calcular(){
3.     float num;
4.     printf("Digite um numero: ");
5.     scanf("%f", &num);
6.     return num*num;
7. }
8. int main(){
9.     float resultado = 0;
10.    resultado = calcular();
11.    printf("A potencia do numero digitado = %.2f ",resultado);
12.    return 0;
13. }
```

Fonte: elaborado pela autora.

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] 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). Lembrando que 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). Portanto, a sintaxe dessa função ficará da seguinte forma:

```
tipo* nome () {  
    tipo vetor[tamanho];  
    return vetor;  
}
```

Veja que ainda haverá um tipo primitivo no retorno da função, mas que estará acompanhado do asterisco, indicando o retorno de um ponteiro (endereço). Outra observação é que basta retornar o nome do vetor.

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 Quadro 3.5 a implementação dessa função. O programa começa sua execução pela linha 11, na função *main()*. Na linha 12 é criado um ponteiro do tipo inteiro, ou seja, este deverá apontar para um local que tenha número inteiro. Na linha 13 é criada uma variável para controle do laço de repetição. Na linha 14 a função *gerarRandomico()* é invocada; nesse momento a execução “pula” para a linha 2, a qual indica que a função irá retornar um endereço para valores inteiros (*int**). Na linha 3 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 5 é criada uma estrutura de repetição para percorrer as 10 posições do vetor. Na linha 6, para cada posição do vetor é gerado um valor aleatório por meio da função *rand()* e na linha 7 o valor gerado é impresso para que possamos comparar posteriormente. Na linha 9 a função retorna o vetor, agora preenchido, por meio do comando *return r;* com isso, a execução volta para a linha 14, armazenando o endereço obtido pela função no ponteiro *p*. Na linha 15 é criada uma estrutura de repetição para percorrer o vetor que será acessado pelo seu endereço. Veja que, na linha 16, para acessar o conteúdo do vetor foi usado o seguinte comando **(p + i)*. É importante ter em mente que o ponteiro retorna o endereço da primeira posição do vetor, por isso é usado *p + i*, para acessar o primeiro (*p*) mais os próximos, conforme *i* incrementa, imprimindo assim todos os valores do vetor.

Quadro 3.5 | Função que retorna vetor

```
1. #include<stdio.h>
2. int* gerarRandomico(){
3.     static int r[10];
4.     int a;
5.     for(a = 0; a < 10; ++a) {
6.         r[a] = rand();
7.         printf("r[%d] = %d\n", a, r[a]);
8.     }
9.     return r;
10. }
11. int main(){
12.     int *p;
13.     int i;
14.     p = gerarRandomico();
15.     for ( i = 0; i < 10; i++ ) {
16.         printf("\n p[%d] = %d", i, *(p + i));
17.     }
18.     return 0;
19. }
```

Fonte: elaborado pela autora.

O resultado do código do Quadro 3.5 pode ser conferido na Figura 3.2. Veja que a impressão de ambos será igual devido ao atributo *static*, usado na declaração do vetor.

Figura 3.2 | Resultado do código do Quadro 3.5

r[0] = 41	p[0] = 41
r[1] = 18467	p[1] = 18467
r[2] = 6334	p[2] = 6334
r[3] = 26500	p[3] = 26500
r[4] = 19169	p[4] = 19169
r[5] = 15724	p[5] = 15724
r[6] = 11478	p[6] = 11478
r[7] = 29358	p[7] = 29358
r[8] = 26962	p[8] = 26962
r[9] = 24464	p[9] = 24464

Fonte: elaborada pela autora.

Outro caso importante de ponteiros com funções é na alocação de memória dinâmica. A função *malloc()* pertencente à biblioteca *<stdlib.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=malloc(100);
```

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 função, o tipo de valor para o qual se deseja converter. Portanto, a forma totalmente correta de se utilizar a função é:

```
int*memoria=(int*)malloc(100);
```

Agora que vimos o tipo de retorno da função *malloc()*, vamos entender como usar essa função dentro de uma função criada por nós. Veja o código no Quadro 3.6: a função *alocar()* foi criada da linha 3 até 5, tendo seu tipo de retorno especificado como *int**; isso significa que o espaço será alocado para guardar valores inteiros. Veja que na linha 7 foi criado um ponteiro inteiro e na linha 8 a função *alocar()* foi chamada, tendo seu resultado armazenado no ponteiro chamado *memoria*. 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 9), então sabemos que a alocação foi feita com sucesso e imprimimos o endereço da primeira posição; caso contrário, o usuário é informado que a memória não foi alocada.

Quadro 3.6 | Função para alocar memória dinamicamente

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. int* alocar(){
4.     return malloc(200);
5. }
6. int main(){
7.     int *memoria;
8.     memoria = alocar();
9.     if(memoria != NULL){
10.         printf("Endereco de memoria alocada = %x",memoria);
11.     }
12.     else{
13.         printf("Memoria nao alocada");
14.     }
15.     Return 0;
16. }
```

Fonte: elaborado pela autora.

Dica

A utilização de funções para alocação dinâmica de memória é extremamente usada em estruturas de dados. Estude com atenção esse recurso, pois será muito útil a você nas próximas etapas do seu desenvolvimento profissional. Para se aprofundar, assista a uma série de seis vídeos sobre esse recurso, disponível no YouTube (LINGUAGEM C [...], 2012).

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.

Sem medo de errar

Recentemente 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 engenharia civil. Um cliente deseja saber qual guindaste usar para cada tipo de construção. Ele possui três modelos, que devem ser usados de acordo com o peso da coluna de concreto armado que levantarão. Os engenheiros lhe passaram a fórmula para calcular o peso da coluna, bem como uma tabela com as regras que devem ser seguidas, solicitando uma solução automatizada para o processo, baseada nos parâmetros de altura, largura e comprimento da coluna.

Para resolver esse problema você deve ter a fórmula para o cálculo do peso anotada: $P = VR$ e lembrar que o volume é calculado por meio da multiplicação entre os três parâmetros e que o parâmetro R possui valor 25. Também será preciso criar condicionais para verificar qual guindaste deve ser usado.

Primeiro construa a função principal do programa, seguindo as seguintes dicas:

1. Crie uma variável para armazenar um valor real, pois o peso certamente não será inteiro.
2. Guarde o resultado da função na variável criada.
3. Compare o valor armazenado com a primeira condição, para verificar se o guindaste a ser usado é o G1. Caso seja, imprima uma mensagem para o usuário.
4. Crie uma condicional encadeada com a primeira, comparando o valor armazenado com a terceira condição, para verificar se o guindaste a ser usado é o G3 e, caso seja, imprima uma mensagem para o usuário.

5. Crie o último “senão”, pois caso não seja nenhum dos anteriores o guindaste a ser utilizado será o G2, e imprima uma mensagem para o usuário.
6. Agora crie uma função para calcular o peso (antes da função *main()*).
7. Crie três variáveis para armazenar os valores da base, da altura e do comprimento.
8. Peça para o usuário digitar os valores e armazene nas variáveis criadas.
9. Calcule o preço da coluna usando a fórmula passada pelos engenheiros.



Para conferir uma possível solução para o problema, acesse o QR Code ou o link: https://cm-cls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/algoritmos-e-tecnicas-de-programacao/u4/s1/resolucaoSP_atp.pdf

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 da quantidade de letras “a” e “e” encontradas na frase, o algoritmo gerará uma senha aleatória.

Resolução da situação-problema

O programa para geração de senhas pode ser construído utilizando uma função; além disso, a função *rand()* também pode ser usada para gerar as senhas aleatórias. Veja no Quadro 3.7 uma possível solução para o problema.

Quadro 3.7 | Função para gerar senhas aleatoriamente

```
#include<stdio.h>

int gerarSenha() {
    char frase[40]="";
    int i, cont = 0;
    printf("\n Digite uma frase para gerar sua senha: ");
    fflush(stdin);
    fgets(frase,40,stdin);
    for(i=0; i < 40; i++) {
        if(frase[i]=='a' || frase[i]=='b') {
            cont++;
        }
    }
    srand(cont); //passando uma "semente" para a geração aleatória
    return rand() * rand();
}

void main(){
    int senha;
    senha = gerarSenha();
    printf("Sua senha é = %d",senha);
}
```

Fonte: elaborado pela autora.

Faça valer a pena

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 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.

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 funcionalidade.

Avalie o código a seguir.

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

```
int somar() {  
    return 2 + 3.23;  
}  
  
int main() {  
    int resultado = 0;  
    resultado = somar();  
    printf("O resultado da funcao e = %d", resultado);  
    return 0;  
}
```

Com base no contexto apresentado, é correto afirmar que:

- a. Será impresso na tela “O resultado da funcao e = 5.23”.
- b. Será impresso na tela “O resultado da funcao e = 2”.
- c. Será impresso na tela “O resultado da funcao e = 3”.
- d. Será impresso na tela “O resultado da funcao e = 5”.
- e. Será dado um erro de execução, pois a função espera retornar um *int*, e está sendo retornado um número real.

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.

Avalie o código a seguir.

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

```
int* retornarVetor() {  
    static int v[10];  
    int a;  
    for(a = 0; a < 10; ++a) {  
        v[a] = 2 * a;  
    }  
    return v;  
}
```

```
int main(){  
    int *p;  
    p = retornarVetor();  
    printf("Valor = %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”.

Seção 2

Escopo e passagem de parâmetros

Diálogo aberto

Caro estudante, 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 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.

Dando continuidade ao seu trabalho no laboratório de pesquisa multidisciplinar, após construir com êxito o programa para a equipe de engenharia civil, agora você trabalhará com o núcleo de química. Em uma reação química, a proporção entre os reagentes define o resultado. Nesse contexto, dados dois componentes químicos, A e B, a sua mistura resultará em um terceiro ($A + B \rightarrow C$). O núcleo foi procurado para otimizar as condições em uma reação chamada de proteção. Para obter a otimização, é preciso variar as condições do experimento, ou seja, usar diferentes massas dos compostos. Um *mol* do composto A possui massa 321,43 g e um *mol* de B possui massa 150,72 g. Você precisa criar uma função que facilite o cálculo dos químicos, ou seja, a partir de diferentes combinações de massa, o programa calculará a massa final do composto. Além do cálculo, seu programa deverá exibir os resultados das combinações do Quadro 3.8, pois estes valores são usados como referência pelos químicos.

Quadro 3.8 | Condições do experimento

Quantidade de mol de A	Quantidade de mol de B	Quantidade de mol de C
1,2	1,0	?
1,4	1,0	?
1,0	1,6	?

Fonte: elaborado pela autora.

Para cumprir essa missão, nesta seção você aprenderá sobre o escopo das variáveis, bem como a passagem de parâmetros em uma função. Com esse conhecimento seus programas ficarão mais organizados e otimizados. Bom estudo!

Não pode faltar

Escopo de variáveis

Já sabemos que 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 o código no Quadro 3.9.

Quadro 3.9 | Exemplo de variáveis em funções

```
1. #include<stdio.h>
2. int testar(){
3.     int x = 10;
4.     return x;
5. }
6. int main(){
7.     int x = 20;
8.     printf("\n Valor de x na função main() = %d",x);
9.     printf("\n Valor de x na função testar() = %d",tes
10. tar());
11.     return 0;
12. }
```

Fonte: elaborado pela autora.

Na implementação do Quadro 3.9 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.

Com esse exemplo em mente, podemos definir como sendo o escopo de uma variável “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). No exemplo do Quadro 3.9, ambas variáveis são locais, ou seja, elas existem e são “enxergadas” somente dentro do corpo da função onde foram

definidas. Para definir uma variável global, é preciso criá-la fora da função, assim ela será visível por todas as funções do programa. Nesse livro, vamos adotar criá-las logo após a inclusão das bibliotecas.

Veja no Quadro 3.10 um exemplo de declaração de uma variável global, na linha 2, logo após a inclusão da biblioteca de entrada e saída padrão. Veja que na função principal não foi definida nenhuma variável com o nome de “x” e mesmo assim pode ser impresso seu valor na linha 7, pois é acessado o valor da variável global. Já na linha 8 é impresso o valor da variável global modificado pela função *testar()*, que retorna o dobro do valor.

Quadro 3.10 | Exemplo de variável global

```
1. #include<stdio.h>
2. int x = 10;
3. int testar() {
4.     return 2*x;
5. }
6. int main() {
7.     printf("\n Valor de x global = %d",x);
8.     printf("\n Valor de x global alterado na funcao tes-
9. tar() = %d",testar());
10.    return 0;
11. }
```

Fonte: elaborado pela autora.

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.

Vejamos um exemplo prático da utilização do escopo global de uma variável. Vamos criar um programa que calcula a média entre duas temperaturas distintas. Veja o código no Quadro 3.11, na linha 2, em que foram declaradas duas variáveis. Lembrando que o programa sempre começa pela função principal e a execução inicia na linha 6. Na 7, é solicitado ao usuário digitar duas temperaturas, as quais são armazenadas dentro das variáveis globais criadas. Na linha 9, 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.

Quadro 3.11 | Calcular média de temperatura com variável global

```
1. #include<stdio.h>
2. float t1, t2;
3. float calcularMedia(){
4.     return (t1 + t2)/2;
5. }
6. int main(){
7.     printf("\n Digite as duas temperaturas: ");
8.     scanf("%f %f",&t1,&t2);
9.     printf("\n A temperatura media = %.2f",calcularMedia());
10.    return 0;
11. }
```

Fonte: elaborado pela autora.

Exemplificando

Vimos que 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:

```
int x = 10;
int main(){
    int x = -1;
    printf("\n Valor de x = %d",x);
}
```

Qual valor será impresso na variável x? A variável local sempre sobrecreverá o valor da global, portanto, nesse caso, será impresso o valor -1 na função principal.

Na linguagem C, para conseguirmos acessar o valor de uma variável global, dentro de uma função que possui uma variável local com mesmo nome, devemos usar a instrução **extern** (MANZANO, 2015). No Quadro 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 6 – 9), que atribui a nova variável o valor “externo” de x.

Quadro 3.12 | Variável global e local com mesmo nome

```
1. #include<stdio.h>
2. int x = 10;
3. int main(){
4.     int x = -1;
5.     int b;
6.     {
7.         extern int x;
8.         b = x;
9.     }
10.    printf("\n Valor de x = %d",x);
11.    printf("\n Valor de b (x global) = %d",b);
12.    return 0;
13. }
```

Fonte: elaborado pela autora.

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 Quadro 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 esses valores até o final da função. Na linha 7, 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”.

Quadro 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. int main(){
6.     int result;
7.     result = somar(10,15);
8.     printf("\n Resultado da soma = %d",result);
9.     return 0;
10. }
```

Fonte: elaborado pela autora.

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 todos 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 o código no Quadro 3.14. A execução do programa começa na linha 9, 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 14, 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 16, imprimimos novamente os valores após a chamada da função. A Figura 3.3 apresenta o resultado desse programa.

Quadro 3.14 | Variáveis em chamada de função com passagem de valores

```
1. #include<stdio.h>
2. int testar(int n1, int n2){
3.     n1 = -1;
4.     n2 = -2;
5.     printf("\n\n Valores dentro da funcao testar(): ");
6.     printf("\n n1 = %d e n2 = %d",n1,n2);
7.     return 0;
8. }
9. int main(){
10.     int n1 = 10;
11.     int n2 = 20;
12.     printf("\n\n Valores antes de chamar a funcao: ");
13.     printf("\n n1 = %d e n2 = %d",n1,n2);
14.     testar(n1,n2);
15.     printf("\n\n Valores depois de chamar a funcao: ");
16.     printf("\n n1 = %d e n2 = %d",n1,n2);
17.     return 0;
18. }
```

Fonte: elaborado pela autora.

Figura 3.3 | Resultado do código do Quadro 3.14

```
Valores antes de chamar a funcao:  
n1 = 10 e n2 = 20  
  
Valores dentro da funcao testar():  
n1 = -1 e n2 = -2  
  
Valores depois de chamar a funcao:  
n1 = 10 e n2 = 20
```

Fonte: elaborada pela autora.

Passagem por referência

A utilização de funções 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 o código no Quadro 3.15 um exemplo de uma função que passa variáveis como referência. Observe que a única diferença desse código para o do Quadro 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 3 e 4, usamos asterisco para acessar o conteúdo guardado dentro do endereço que ele aponta, pois se usássemos somente n1 e n2 iríamos acessar o endereço que eles apontam.

Quadro 3.15 | Variáveis em chamada de função com passagem de referência

```
1. #include<stdio.h>
2. int testar(int* n1, int* n2){
3.     *n1 = -1; //tem que usar o * para acessar o
4.     conteúdo
5.     *n2 = -2;
6.     printf("\n\n Valores dentro da funcao testar(): ");
7.     printf("\n n1 = %d e n2 = %d",*n1,*n2);
8.     return 0;
9. }
10. int main(){
11.     int n1 = 10;
12.     int n2 = 20;
13.     printf("\n\n Valores antes de chamar a funcao: ");
14.     printf("\n n1 = %d e n2 = %d",n1,n2);
15.     testar(&n1,&n2);
16.     printf("\n\n Valores depois de chamar a funcao: ");
17.     printf("\n n1 = %d e n2 = %d",n1,n2);
18.     return 0;
19. }
```

Fonte: elaborado pela autora.

Figura 3.4 | Resultado do código do Quadro 3.15

```
Valores antes de chamar a funcao:  
n1 = 10 e n2 = 20  
  
Valores dentro da funcao testar():  
n1 = -1 e n2 = -2  
  
Valores depois de chamar a funcao:  
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 percorre o vetor imprimindo o dobro de cada valor do vetor. Veja, no Quadro 3.16, o código que começa a ser executado pela linha 15. Na linha 18, 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 18 e depois para a 19, na qual é invocada a função `imprimir()`, novamente passando o vetor “numeros” como argumento. Mais uma vez, criamos a função na linha 9 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.

Quadro 3.16 | Função com passagem de vetor

```
1. #include<stdio.h>
2. void inserir(int a[]){
3.     int i=0;
4.     for(i=0;i<3;i++){
5.         printf("Digite o valor %d: ",i);
6.         scanf("%d",&a[i]);
7.     }
8. }
9. void imprimir(int b[]){
10.    int i=0;
11.    for(i=0;i<3;i++){
12.        printf("\n numeros[%d] = %d",i,2*b[i]);
13.    }
14. }
15. int main(){
16.     int numeros[3];
17.     printf("\n Preenchendo o vetor... \n ");
18.     inserir(numeros);
19.     printf("\n Dobro dos valores informados:");
20.     imprimir(numeros);
21.     return 0;
22. }
```

Fonte: elaborado pela autora.

Figura 3.5 | Resultado do código do Quadro 3.16

```
Preenchendo o vetor...
Digite o valor 0: 6
Digite o valor 1: 7
Digite o valor 2: 3

Dobro dos valores informados:
numeros[0] = 12
numeros[1] = 14
numeros[2] = 6
```

Fonte: elaborada pela autora.

Pesquise Mais

As técnicas apresentadas nesta seção fazem parte do cotidiano de um desenvolvedor em qualquer linguagem de programação. Leia o capítulo 10 do livro *Linguagem C: acompanhada de uma xícara de café* e aprofunde seu conhecimento (MANZANO, 2015).

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.

Sem medo de errar

Você foi contratado por um laboratório de pesquisa que presta serviço para diversas empresas, e agora precisa fazer um programa para a equipe de químicos. Para saber a massa exata de um determinado composto, eles precisam calcular a massa de cada elemento que será usado, bem como a proporção que será utilizada. Foi solicitado a você automatizar o cálculo de uma reação chamada de proteção. Nessa reação, um composto A, de massa 321,43 g/mol será somando a um composto B de massa 150,72 g/mol. Seu programa, além de calcular o composto com base nas informações do usuário, deverá também exibir os valores de referência das combinações: (1,2 : 1,0), (1,4 : 1,0) e (1,0 : 1,6) (valores da Quadro 3.8).

Para implementar essa solução você pode criar uma função que recebe como parâmetro a quantidade, em mol, dos componentes A e B, e retorna a massa da reação.

O primeiro passo é criar a função que fará o cálculo da função. Você pode criá-la da seguinte forma:

- `float calcularMassa(float a, float b)`

Nessa função, deverão ser impressos os valores de referência conforme Quadro 3.8, deverá ser feito o cálculo da massa e retornar o resultado.

Em seguida, é preciso criar a função principal, na qual será solicitado ao usuário que informe a quantidade dos elementos A e B. Em posse dos valores, a função `calcularMassa()` deverá ser invocada e os valores passados. O resultado dessa função deve ser guardado dentro de uma variável, da seguinte forma:

- `resultado = calcularMassa(a, b)`

Como último passo, deverá ser impressa a massa da reação.



Para conferir uma opção de implementação do problema, acesse o QR Code ou o link:

https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/algoritmos-e-tecnicas-de-programacao/u4/s2/resolucaoSP_atp.pdf

Avançando na prática

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 da situação-problema

Para implementar a solução, primeiro você deve saber a fórmula de juros simples $j = C \cdot i \cdot t$, na qual j é o juro, C é o capital inicial, i é a taxa e t é o tempo. Veja no Quadro 3.16 uma possível implementação para o problema.

Quadro 3.16 | Programa para calcular juros simples

```
#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("\n Dados invalidos!");
        return 0;
    }
    if((plano =='B' || plano =='b') && meses >=12){
        return valor * 0.008 * meses;
    }else{
        printf("\n Dados invalidos!");
        return 0;
    }
}
int main(){
    float valorInv = 0;
    float rendimento = 0;
    char plano;
    int tempo = 0;

    printf("\n Digite o plano: ");
    scanf("%c",&plano);

    printf("\n Digite o valor a ser investido: ");
    scanf("%f",&valorInv);

    printf("\n Digite o tempo para resgate: ");
    scanf("%d",&tempo);

    rendimento = calcularInvestimento(valorInv, plano, tempo);
    printf("\n Seu rendimento sera = %.2f",rendimento);

    return 0;
}
```

Fonte: elaborado pela autora.

Faça valer a pena

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 retornar 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. I – V; II – V; III – V.
- b. I – V; II – F; III – V.
- c. I – F; II – V; III – V.
- d. I – F; II – F; III – V.
- e. I – F; II – V; III – F.

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. int pensar(int a, int b) {
3.     a = 11;
4.     b = 12;
5.     return 0;
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.

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.  int pensar(int* a, int* b){
3.      a = 10;
4.      b = 20;
5.      return 0;
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.

Seção 3

Recursividade

Diálogo aberto

Caro estudante, 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 desse símbolo da matemática: \sum_a^b ? É o símbolo de somatória e representa que determinado termo será somado, uma certa quantidade de vezes, repetindo o processo no intervalo definido. Para você 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.

Após trabalhar com a equipe de engenheiros civis e químicos, 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 a raiz quadrada de um número usando o método de aproximações sucessivas de Newton. Dessa forma, você deverá entregar o arquivo compilado para que eles possam utilizar. Os matemáticos o ajudaram, fornecendo a fórmula

usada para tal tarefa: $x_n = \frac{x_{n-1}^2 + n}{2x_{n-1}}$, na qual x_n é o valor procurado, ou seja, a

raiz quadrada. O parâmetro x_{n-1} é a raiz anterior calculada pelo método e n é o número de que se deseja calcular a raiz. Após conhecer sua nova tarefa, seu gerente, que também é programador, pediu para que utilizasse a técnica de recursividade na implementação, portanto o usuário precisa informar o número de que deseja calcular a raiz quadrada. Quais valores precisam ser informados para que o programa seja executado de maneira correta? O que determinará o término do cálculo? Esse método exige algum tipo de valor específico?

Para que você cumpra essa missão, você aprenderá o que é uma função recursiva, como ela deve ser implementada e qual sua relação com as estruturas iterativas.

Aproveite mais essa oportunidade de aprimoramento e bons estudos!

Não pode faltar

Nesta unidade, apresentamos as funções a você; 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.

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, por exemplo, 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 se difere de outras funções simplesmente pelo fato dela possuir, em seu conjunto de comandos, uma chamada a si própria.

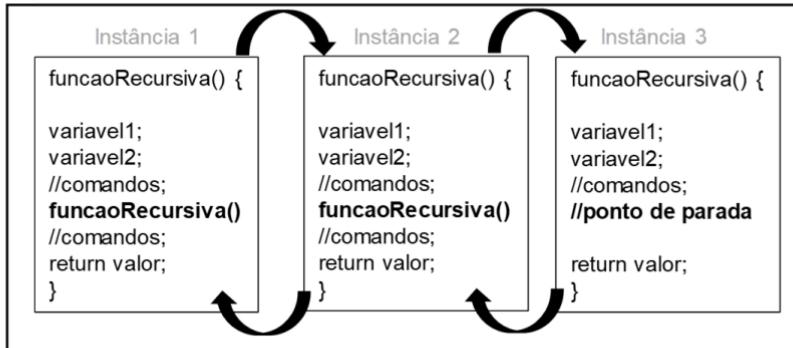
Embora a sintaxe da função recursiva seja similar às não recursivas, o funcionamento de ambas é bastante distinto e o mau uso dessa técnica pode acarretar em uso indevido de memória, muitas vezes chegando a travar a aplicação e o sistema (MANZANO, 2015). Para entender todo o processo, vamos estabelecer 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 possui 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 (instâncias) 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, possui 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

Memória de trabalho



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, obrigatoriamente, tem que 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.

Refletá

Toda função recursiva tem que possuir 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 em 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 $5 + 4 + 3 + 2 + 1 + 0$. A partir desse 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 Quadro 3.17 e a seguir sua explicação.

Quadro 3.17 | Função recursiva para soma

```
1. #include<stdio.h>
2. int somar(int valor){
3.     if(valor != 0){ //critério de parada
4.         return valor + somar(valor-1); //chamada recursiva
5.     }
6.     else{
7.         return valor;
8.     }
9. }
10. int main(){
11.     int n, resultado;
12.     printf("\n Digite um numero inteiro positivo : ");
13.     scanf("%d",&n);
14.     resultado = somar(n); //fazendo a primeira chamada da função
15.     printf("\n Resultado da soma = %d",resultado);
16.     return 0;
17. }
```

Fonte: elaborado pela autora.

A execução do programa no Quadro 3.17 começará na linha 11, pela função principal. 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 (\neq) de zero, a execução passa para a linha 4, na qual a função é invocada novamente, mas dessa vez passando como parâmetro o valor menos 1. Quando o valor for zero, as instâncias passam a retornar o valor para a que chamou.

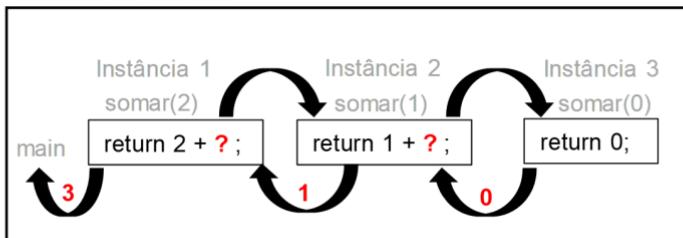
Exemplificando

A Figura 3.8 exemplifica o funcionamento na memória de trabalho 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 satisfeita, 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()*

Memória de trabalho



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 Quadro 3.17, poderia ser substituída por uma estrutura de repetição usando *for*? No exemplo dado, poderia ser escrito algo como:

```
for(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 consiste em indicar a possibilidade de um problema maior ser dividido em instâncias menores, porém considerando a utilização dos recursos computacionais que serão empregados por cada método.

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, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que resulta em 120. Observe o código, no Quadro 3.18, que implementa essa solução usando uma função recursiva. Em seguida a explicação.

Quadro 3.18 | Função recursiva para fatorial

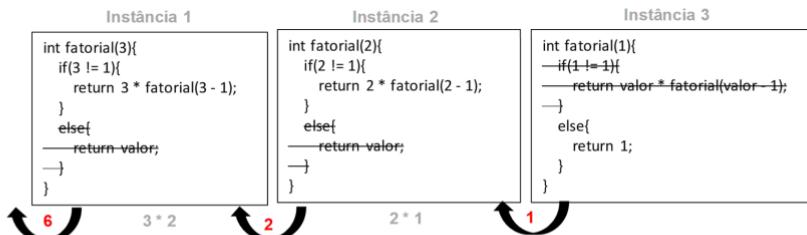
```
1. #include<stdio.h>
2. int fatorial(int valor){
3.     if(valor != 1){ //critério de parada
4.         return valor * fatorial(valor - 1); //chamada
5.     recursiva
6.     }
7.     else{
8.         return valor;
9.     }
10. }
11. int main(){
12.     int n, resultado;
13.     printf("\n Digite um numero inteiro positivo : ");
14.     scanf("%d",&n);
15.     resultado = fatorial(n);
16.     printf("\n Resultado do fatorial = %d",resultado);
17.     return 0;
18. }
```

Fonte: elaborado pela autora.

A execução do código, no Quadro 3.18, 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 novas instâncias na memória, passando a cada vez como parâmetro o valor decrementado de 1. Quando o valor chegar a 1, a função retorna à

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 função *fatorial()*



Fonte: elaborada pela autora.

Esse mecanismo é custoso para o computador, pois tem que alocar recursos para as variáveis e comandos da função, procedimento chamado de empilhamento, e tem também 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 no Quadro 3.19. Veja, na linha 3, 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. Também é possível perceber que foi preciso criar uma função auxiliar para efetuar o cálculo.

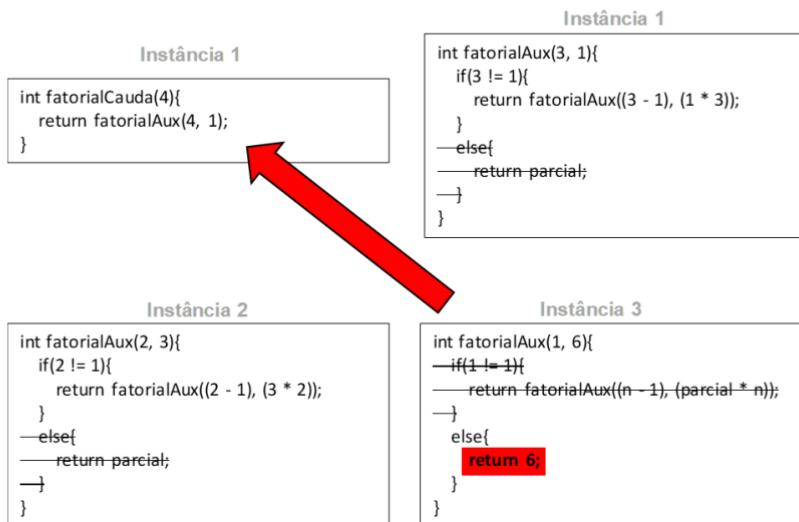
Quadro 3.19 | Recursividade em cauda

```
1. #include<stdio.h>
2. int fatorialCauda(int valor){
3.     return fatorialAux(valor, 1);
4. }
5. int fatorialAux(int n, int parcial){
6.     if(n != 1){
7.         return fatorialAux((n - 1), (parcial * n));
8.     }else{
9.         return parcial;
10.    }
11. }
12. int main(){
13.     int n, resultado;
14.     printf("\n Digite um numero inteiro positivo : ");
15.     scanf("%d",&n);
16.     resultado = fatorialCauda(n);//fazendo a primeira
17.     //chamada da função
18.     printf("\n Resultado do fatorial = %d",resultado);
19.     return 0;
20. }
```

Fonte: elaborado pelo autor.

Observe, na Figura 3.10, que o mecanismo de funcionamento da recursividade em cauda é diferente. A função *fatorialCauda()* possui apenas uma instância, a qual invoca a função *fatorialAux()*, passando como parâmetro 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, as funções não precisam retornar o valor para “quem” invocou, gerando otimização na memória, pois não precisa armazenar nenhum ponto para devolução de valores.

Figura 3.10 | Exemplo 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 irá se desenvolver e você terá maior facilidade para desenvolver novos programas. Agora você já tem conhecimento para solucionar seu próximo desafio! Boa sorte!

Sem medo de errar

Após conhecer a técnica de recursividade, chegou o momento de implementar o programa para os matemáticos do laboratório onde trabalha. Lembrando que foi solicitado a você implementar o método de Newton para o cálculo da raiz quadrada, porém usando funções recursivas. Os matemáticos lhe forneceram a

seguinte fórmula: $x_n = \frac{x_{n-1}^2 + n}{2x_{n-1}}$, na qual x_n é o valor procurado, ou seja, a raiz quadrada e n o valor informado para obter a raiz.

Para implementar essa solução, você precisa de um programa que solicite ao usuário um número. Você também deve especificar um valor inicial para a raiz e um critério de parada. Após pesquisar, você verificou que o método interativo de Newton, consiste em fazer comparações entre os resultados obtidos em cada instância. Será preciso comparar o valor calculado da raiz

com o obtido no passo anterior até que a diferença entre eles seja menor que o critério de parada. Por isso, também será necessário usar as funções *fabs()* e *pow()*, ambas pertencentes à biblioteca *math.h*. A primeira retorna o valor absoluto de um número e a segunda é usada para potenciação.

Como valor inicial, você pode adotar a metade do valor informado pelo usuário e como critério de parada, o valor 0.001. Quanto menor o valor, mais exato será o cálculo.

Você pode seguir os seguintes passos para a implementação:

1. Crie a função *calcularRaiz()*, de modo que ela receba dois parâmetros, o valor a ser calculado e a raiz anterior, que, na primeira chamada, será o chute inicial:

```
float calcularRaiz(float n, float raizAnt)
```

2. Implemente a fórmula passada pelos matemáticos usando a função *pow()*:

*raiz = (pow(raizAnt, 2) + n) / (2 * raizAnt);* Veja que a raiz anterior é elevada à segunda potência pela função.

3. Calcule a diferença entre o valor obtido no passo 2 e o valor da raiz anterior e veja se é menor que o critério de parada.

```
if (fabs(raiz - raizAnt) < 0.001)
```

4. Caso seja, o valor é satisfatório e pode ser retornado; caso não satisfaça, a função *calcularRaiz()* deve ser novamente invocada, passando como parâmetro o número e a raiz obtida no passo 2.

Veja, no Quadro 3.20, uma possível implementação para essa solução.

Quadro 3.20 | Cálculo da raiz quadrada por aproximações sucessivas

```
1. #include<stdio.h>
2. #include<math.h>
3. float calcularRaiz(float n, float raizAnt)
4. {
5.     float raiz = (pow(raizAnt, 2) + n)/(2 * raizAnt);
6.     if (fabs(raiz - raizAnt) < 0.001)
7.         return raiz;
8.     return CalcularRaiz(n, raiz);
9. }
10. void main(){
11.     float numero, raiz;
12.     printf("\n Digite um número para calcular a raiz: ");
13.     scanf("%f",&numero);
14.     raiz = calcularRaiz(numero,numero/2);
15.     printf("\n Raiz quadrada função = %f",raiz);
16. }
```

Fonte: elaborado pela autora.

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 da situação-problema

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:

$$\frac{16}{24} = 1, \text{ com resto } 16.$$

2. O divisor da operação anterior deve ser dividido pelo resto da divisão.

$$\frac{24}{16} = 1, \text{ com resto } 8.$$

3. O segundo passo deve ser repetido, até que o resto seja nulo e, então, o divisor será o MDC.

$$\frac{16}{8} = 2, \text{ com resto } 0.$$

Veja, no Quadro 3.21, uma possível implementação para o MDC.

Quadro 3.21 | MDC com recursividade

```
1. #include<stdio.h>
2. int calcularMDC(int a, int b) {
3.     int r = a % b;
4.     if(r == 0){
5.         return b;
6.     }else{
7.         return calcularMDC(b, r);
8.     }
9. }
10. void main(){
11.     int n1, n2, resultado;
12.     printf("\n Digite dois números inteiros positivos: ");
13.     scanf("%d %d",&n1,&n2);
14.     resultado = calcularMDC(n1,n2);
15.     printf("\n MDC = %d",resultado);
16. }
```

Fonte: elaborado pela autora.

Faça valer a pena

- I.** 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.

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 11.

```
1. #include<stdio.h>
2. int somar(int valor){
3.     if(valor != 0){
4.         return valor + somar(valor - 1);
5.     }
6.     else{
7.         return valor;
8.     }
9. }
10. int main(){
11.     printf("\n Resultado = %d", somar(6));
12.     return 0;
13. }
```

Com base no contexto apresentado, assinale a alternativa correta.

- a. Resultado = 21.
- b. Resultado = 0.
- c. Resultado = 12.
- d. Resultado = 6.
- e. Resultado = 5.

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 possua 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.

Referências

- ALMEIDA, R.; ZANLORENSSI, G. A evolução do número de linhas de telefone fixo e celular no mundo. **Nexo**, 2 maio 2018. Disponível em: <https://www.nexojornal.com.br/grafico/2018/05/02/A-evolu%C3%A7%C3%A3o-do-n%C3%BAmero-de-linhas-de-telefone-fixo-e-cellular-no-mundo>. Acesso em: 7 jul. 2018.
- COUNTER, L. **Lines of code of the Linux Kernel Versions**. Disponível em: <https://www.linuxcounter.net/statistics/kernel>. 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://www.ime.usp.br/~pf/algoritmos/aulas/recu.html>. Acesso em: 2 jan. 2018.
- LINGUAGEM C Programação Descomplicada. [C] **Aula 60 - Alocação Dinâmica - Parte 1 - Introdução**. 5 nov. 2012. Disponível em: <https://goo.gl/ps7VPa>. Acesso em: 30 jun. 2016.
- 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: http://www.ic.unicamp.br/~oliveira/doc/mc102_2s2004/Aula19.pdf. 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.

Unidade 4

Kleber Ricardi Rovai

Estrutura de dados

Convite ao estudo

Caro aluno, nesta unidade você vai aprender algumas estruturas de dados importantes para o desenvolvimento de software. Mas o que é uma estrutura de dados? A resposta para essa questão é: uma estrutura de dados é a maneira como o dado é organizado, os métodos de acesso aos dados para que eles sejam processados. Muitas vezes, sem saber, utilizamos estruturas de dados para o desenvolvimento de sistemas para várias atividades ou ramo de negócio. As listas, pilhas e filas são as estruturas de dados apresentadas nesta unidade, e elas possibilitam organizar os dados de diferentes formas. Quem nunca montou uma lista de compras, verificou uma lista telefônica ou montou uma lista de tarefas? Esses são exemplos uso de listas em nosso dia a dia. Logo, pode-se dizer que em uma lista os dados são organizados de forma sequencial. E a pilha? Em uma pilha os dados são organizados de tal forma que o último elemento a entrar é o primeiro a sair. Um exemplo prático seriam pratos empilhados: ao retirá-los para lavar, você deverá começar com o primeiro prato do topo e não com o prato mais abaixo. Por fim, uma fila é uma estrutura de dados em que o primeiro elemento a entrar é o primeiro a sair. Um exemplo para essa estrutura é uma fila de banco: os primeiros a chegar são atendidos, logo, são os primeiros a sair.

Essas são apenas algumas das aplicações dessas três importantes estruturas de dados. No decorrer desta unidade, essas estruturas serão apresentadas da seguinte maneira: na Seção 4.1, você vai conhecer o funcionamento das listas ligadas, bem como as operações que podem ser realizadas. Além disso, as listas duplamente ligadas também serão apresentadas, criando soluções por meio de aplicações. Na Seção 4.2 serão apresentados os conceitos, aplicações e operações com pilhas. Por fim, na Seção 4.3 serão apresentados os conceitos, aplicações e operações com filas.

Como desenvolvedor em uma empresa que projeta sistemas para diversos setores do comércio, como farmácias, lojas de roupas e padarias, entre outras, você precisará aplicar todo o seu conhecimento de estrutura de dados em listas, pilhas e filas para solucionar os desafios propostos. Certamente o estudo será fundamental para a compreensão e o aprendizado de Estrutura de Dados. Preparado para o desafio? Então vamos lá!

Seção 1

Listas

Diálogo aberto

Caro aluno, nesta seção você vai estudar alguns conceitos que são importantes para a compreensão e o aprendizado sobre as listas ligadas, assim como sua estrutura e suas aplicações.

Você vai conhecer e compreender as estruturas de dados dinâmicas essenciais e suas aplicações na solução de problemas. Poderá identificar como a estrutura de uma lista ligada funciona e, assim, desenvolver novas soluções para os mais variados sistemas que utilizam essa estrutura como base fundamental do seu funcionamento, ou, ainda, criar processos automatizados baseados em listas ligadas.

Poderá identificar as diferenças entre uma lista ligada simples e uma lista duplamente ligada, como funciona sua estrutura e quais as vantagens em utilizar essa estrutura dinâmica, permitindo que a utilização de listas seja mais simples de ser aplicada em sistemas mais complexos, como a criação de sistemas para controle de processos dentro do sistema operacional. Além disso, vai conhecer os conceitos e as aplicabilidades de operações com as listas ligadas, a fim de conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas.

Com as listas duplamente ligadas é possível identificar qual o processo anterior e qual o próximo processo a ser executado.

Como desenvolvedor, você precisa aplicar os conhecimentos que está adquirindo em estrutura de dados em listas ligadas, criando os relatórios necessários, para solucionar uma nova demanda de um cliente de autopeças.

Seu cliente solicitou que:

- O relatório seja gerado em tempo real.
- O relatório seja exibido todas as vezes que um produto entrar ou sair desta listagem.
- O relatório seja gerado por meio de venda ou compra de produtos.
- Seja possível pesquisar se um produto consta na listagem ou não.

Como desafio, agora você precisa implementar no relatório em que está trabalhando a adição ou remoção de produtos na listagem.

O principal desafio é utilizar a exibição de lista, inserir e remover, assim como utilizar seu conhecimento em pesquisar na lista para resolver a solicitação de seu cliente.

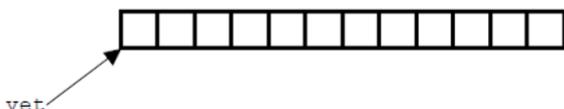
Vamos começar?

Bons estudos!

Não pode faltar

As estruturas de dados são formas de organização e distribuição de dados para tornar mais eficientes a busca e manipulação dos dados por algoritmos. As estruturas de dados são fundamentais para o desenvolvimento de programas e algoritmos. Existem diferentes tipos de estrutura de dados para diferentes aplicações específicas em sistemas. Uma das estruturas de dados muito utilizadas e importantes é a lista. Lista consiste em uma estrutura de dados dinâmica, uma vez que o número de elementos de uma lista é variável conforme eles são inseridos ou removidos. Para a implementação de listas, os vetores são muito utilizados para representar um conjunto de dados, permitindo definir um tamanho máximo de elementos a ser utilizados neste vetor. A Figura 4.1 ilustra a estrutura de um vetor com 12 posições.

Figura 4.1 | Exemplo de vetor



Fonte: Celes, Cerqueira e Rangel (2004, p. 134).

Ainda conforme Celes, Cerqueira e Rangel (2004), o uso do vetor, ao ser declarado, reserva um espaço contíguo na memória para armazenar seus elementos. Assim, é possível acessar qualquer um dos seus elementos a partir do primeiro elemento, por meio de um ponteiro.

Apesar de um vetor ser uma estrutura que permite o acesso aleatório aos elementos, ele não é uma estrutura flexível de dados, devido ao tamanho máximo que precisa ser definido. E quando temos um aumento de elementos acima do que foi determinado para o vetor, seria necessário o aumento da dimensão do vetor para alocá-los. Como quando em uma sala de aula com 25 carteiras disponíveis para alunos comparecem 30 alunos, sendo necessário alocar mais cadeiras de outras salas para acomodar os alunos excedentes.

Para solucionar este tipo de problema, é necessário utilizar uma estrutura de dados que permita o crescimento do vetor ou a sua redução de forma dinâmica.

Segundo Celes, Cerqueira e Rangel (2004), uma estrutura de dados que permite este tipo de armazenamento dinâmico de dados são as listas ligadas, as quais podemos implementar em diversas outras estruturas de dados.

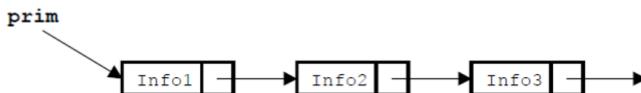
Definição de listas ligadas

Uma lista ligada é uma coleção $L: [a_1, a_2, \dots, a_n]$, em que $n > 0$. Sua propriedade estrutural baseia-se apenas na posição relativa dos elementos, que são dispostos linearmente.

De acordo com Silva (2007), é também conhecida como lista encadeada. É composta por um conjunto de dados dispostos por uma sequência de nós, em que a relação de sucessão desses elementos é determinada por um ponteiro que indica a posição do próximo elemento, podendo estar ordenado ou não.

Na Figura 4.2 podemos visualizar o modelo de uma lista ligada. Segundo Celes, Cerqueira e Rangel (2004), a estrutura da lista consiste em uma sequência de elementos encadeados, definida como nó de lista. O nó de lista é composto por duas informações: a informação armazenada e um ponteiro que indica o próximo elemento da lista.

Figura 4.2 | Modelo de lista ligada



Fonte: Celes, Cerqueira e Rangel (2004, p. 134).

Diferentemente dos vetores, em que o armazenamento é realizado de forma contígua, a lista ligada estabelece a sequência de forma lógica.

Conforme Silva (2007), para trabalharmos com a lista encadeada, definimos um ponto inicial ou um ponteiro para o começo da lista. A partir daí podemos inserir elementos, remover ou realizar buscas na lista.

As listas apresentam os seguintes procedimentos básicos de manipulação, segundo Silva (2007):

- Criação ou definição da estrutura de uma lista.
- Inicialização da lista.

- Inserção com base em um endereço como referência.
- Alocação de um endereço de nó para inserção na lista.
- Remoção do nó com base em um endereço como referência.
- Deslocamento do nó removido da lista.

Assimile

Uma das vantagens das listas ligadas é que os elementos alocados não precisam estar sequencialmente na memória, como é necessário ocorrer com os vetores. Cada elemento pode estar alocado em diferentes regiões da memória.

Quando uma lista está sem nós é definida como lista vazia ou lista nula, assim, o valor do ponteiro para o próximo nó da lista é considerado como ponteiro nulo (*null*).

Elementos de dados em listas ligadas

Os elementos de uma lista são armazenados em posições sequenciais de memória, sempre que possível e de forma dinâmica, e permitem que a lista seja percorrida em qualquer direção.

Toda lista precisa ter sua estrutura definida, sabendo que cada nó é composto de um conjunto de informações de tipos diferentes, por um campo de informação e outro de valor inteiro para o ponteiro.

Os elementos de informação de uma lista podem ser do tipo *int*, *char* e/ou *float*. Ao criar uma estrutura de uma lista, definimos também o tipo de dado que será utilizado em sua implementação. Vejamos, a seguir, um modelo de implementação de uma lista.

```
struct lista {  
    int info;  
    struct lista* prox;  
};  
typedef struct lista Lista;
```

Exemplificando

Exemplo de declaração para criar uma lista em C:

```
/*Cria a estrutura da lista*/
struct alunos {
    char nome[25];
    struct alunos* prox;
};

typedef struct alunos Classe;
```

Neste exemplo, podemos identificar que:

- Será criado uma *struct* (registro) *alunos*.
- Na *struct*, temos a variável *nome* do tipo *char*, que será nossa informação.
- Temos outra *struct* *prox* com ponteiro para a própria *struct* *alunos*, para receber o endereço de apontamento da próxima informação.

Precisamos inicializar a lista para ser utilizada após sua criação. Para isso, basta criarmos uma função em que inicializamos a lista como nula, pois esta é uma das possíveis formas de inicialização:

```
/* Função para inicialização: retorna uma lista vazia */

Lista* inicializa (void)
{
    return NULL;
}
```

Elementos de ligação (ponteiros)

Segundo Veloso (1996), os elementos de ligação em uma lista ligada são os ponteiros. Um ponteiro é um tipo de variável que armazena um endereço de memória e não o conteúdo da posição de memória.

A utilização dos ponteiros é indicada nos casos em que é preciso conhecer o endereço que está armazenando a variável. Podemos declarar um ponteiro utilizando a mesma palavra da variável, precedido do caractere * (asterisco). Vejamos o exemplo:

```
int *ptr; /* sendo um ponteiro do tipo inteiro*/
float *ptr; /* sendo um ponteiro do tipo ponto flutuante*/
```

```
char *ptr; /* sendo um ponteiro do tipo caracteres*/
```

Exemplificando

```
/* Exemplo de uma estrutura da lista  
declarada para armazenar dados de uma  
agenda. */  
typedef struct lista {  
    char *nome;          /*Declaração de um  
ponteiro do tipo char  
    int telefone;  
    struct lista *proxima;  
} Dados;
```

De acordo com Tenenbaum, Langsam e Augenstein (2007), um ponteiro é como qualquer outro tipo de variável. Pode ser utilizado de forma dinâmica, para armazenamento e manipulação de valores.

Para sabermos o endereço da memória reservada para a variável, utiliza-se o operador & juntamente ao nome de uma variável, enquanto o operador *(asterisco), utilizado juntamente com a variável do tipo ponteiro, acessa o conteúdo armazenado do endereço de memória, conforme Silva (2007). Temos:

```
int x = 10; /*variável  
int *p;      /*ponteiro  
p = &x;       /*ponteiro p aponta para o endereço da  
variável x
```

Em listas, além do uso de ponteiros, utilizamos também as alocações dinâmicas de memória, que são porções de memórias para utilização das listas.

Para entendermos como funciona um ponteiro em uma lista, precisamos entender a função *malloc()*, *Memory Allocation* ou Alocação de Memória. Ela é a responsável pela reserva de espaços na memória principal. Tem como finalidade alocar uma faixa de bytes consecutivos na memória do computador e retornar o endereço dessa faixa ao sistema.

O trecho do código a seguir apresenta um exemplo de utilização da função *malloc()* e do ponteiro:

```
char *pnt;  
pnt = malloc (2); /* Aloca 2 bytes na memória */  
scanf ("%c", pnt);
```

O endereço retornado pela função *malloc* é da posição inicial, onde se localizam os *bytes* alocados.

Em uma lista, precisamos alocar o tipo de dado no qual foram declarados seus elementos, e por este tipo de dados ocupar vários *bytes* na memória, precisaremos utilizar a função *sizeof*, que permite informar quantos *bytes* o tipo de elemento criado terá. Na Figura 4.3 temos um exemplo de código em C, com a utilização das funções *malloc()* e *sizeof*.

Figura 4.3 | Exemplo de programa em C com *malloc()* e *sizeof*

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
  
    int *p;  
    p=(int *) malloc(sizeof(int));  
  
    if (!p) {  
        printf("Erro de memoria insuficiente");  
    }else{  
        printf("Memoria alocada com sucesso");  
    }  
  
    return 0;  
}
```

Fonte: elaborada pelo autor.

Podemos classificar as listas de duas formas. Uma delas é a lista com cabeçalho, em que o conteúdo existente no primeiro elemento é irrelevante, pois serve apenas como marcador do seu início. Este primeiro elemento permanece sempre como ponto inicial na memória, independentemente se a lista está com valores ou não.

Assim, podemos considerar que *start* é o endereço inicial da nossa lista. Para determinar que a lista está vazia, consideramos: *start* → *prox* == *NULL*.

Como exemplo deste trecho de código, temos:

```
celula *start;
```

```
start = malloc (sizeof (celula));  
start → prox == NULL.
```

Exemplificando

Exemplo de um trecho de uma função do tipo inteiro, para inserir pacientes com prioridades na lista.

```
int insere _ com _ prioridade(Fila *F , Paciente *P)  
{  
    Lista *ptnodo, *aux , *ant;  
    ptnodo = (Lista*)malloc(sizeof(Lista));
```

Outra classificação de uma lista é chamada de lista sem cabeçalho, em que o conteúdo do primeiro elemento tem a mesma importância que os demais. Assim, a lista é considerada vazia se o primeiro elemento é NULL. A criação deste tipo de lista vazia pode ser definida por *start = NULL*.

Aplicações de listas ligadas

As aplicações de listas no nosso dia a dia são mais comuns do que parece. Podemos aplicar listas em muitas funções. Às vezes não percebemos, mas elas estão presentes.

A lista pode ser aplicada em nosso trabalho, quando precisamos listar todas as nossas tarefas do próximo dia de trabalho, definindo depois quais tarefas têm maior prioridade.

Outra possível aplicação de listas ligadas é a criação de uma lista de compras de supermercado. Ao anotar tudo o que você precisa comprar automaticamente está criando uma lista, na qual podem ser incluídos ou removidos produtos conforme a compra vai sendo realizada. Assim, é possível gerenciar uma lista de produtos de forma dinâmica, como na Figura 4.4, em que temos um trecho de código para iniciar uma lista de supermercado.

Figura 4.4 | Trecho de código para iniciar uma lista de supermercado

```
Dados *inicia_listaMerc(char *prod, int numpro) {  
    Dados *novo;  
  
    novo = (Dados *)malloc(sizeof(Dados));  
    novo -> prod = (char *)malloc(strlen(prod) + 1);  
    strncpy(novo -> prod, prod, strlen(prod) + 1);  
    novo -> numpro = numpro;  
    novo -> proximo = NULL;  
  
    return novo;  
}
```

Fonte: elaborada pelo autor.

Podemos também utilizar uma lista para checklist de itens para viagem. Você pode criar uma lista e elencar todos os itens de que precisa ou pode levar na viagem que vai realizar com seus amigos à praia. Após listar todos os produtos que você vai levar nessa viagem, ao separá-los, você pode checar ou remover os que podem parecer desnecessários e até adicionar novos produtos de que se lembrou posteriormente.

Vamos imaginar que você vai desenvolver um sistema para o site de uma empresa de casamentos e, por meio desse sistema, os usuários podem criar a lista de convidados para seu matrimônio, assim como a lista de presentes que os noivos selecionarem. Tanto a lista de convidados quanto a lista de presentes permitem aos noivos adicionarem convidados ou presentes e removê-los da lista, quando necessário. Na Figura 4.5 temos um trecho de um código para adicionar convidados em uma lista.

Figura 4.5 | Exemplo de trecho para inserir convidados em uma lista

```
void convidados inserirConvid(tipoitem elemento,int &cont)
{
    festa *novo,*aux,*aux1;
    aux = inicio;
    aux1 = inicio -> prox;

    while (aux1 != NULL) {
        if(strcmp(aux1 -> item.nome, elemento.nome) > 0)
            break;

        aux = aux -> prox;
        aux1 = aux1 -> prox;
    }

    if((novo = new(festa) == NULL)
       printf("\nMemoria insuficiente");
    else {
        novo -> prox = aux -> prox;
        aux -> prox = novo;
        novo -> item = elemento;
        cont++;
        printf("\nConvidado inserido com sucesso!");
    }

    if (aux1 == NULL)
        fim = novo;

    return;
}
```

Fonte: elaborada pelo autor.

É possível, também, desenvolver um sistema para uma instituição de ensino que permite gerar lista de alunos, adicionando ou removendo um aluno da classe, além de ordená-los por ordem alfabética, mesmo que os registros do aluno (RA) ou ordens de matrícula estejam em ordens distintas. Assim, a lista ligada permitiria trazer todos os alunos desta determinada classe com base no nome de cada aluno.

Podemos ver, com esses exemplos, que as aplicações das listas são diversas e estão presentes no nosso dia a dia.

Operações com listas ligadas

Podemos realizar várias operações para manipulação de listas ligadas, tais como adicionar, remover, buscar um elemento e percorrer toda a lista ligada.

Adicionar elementos à lista

Ao criarmos uma lista ligada, conforme a estrutura criada no box Exemplificando a seguir, nossa lista é criada sem valor algum, ou seja, ela é criada vazia.

Segundo Celes, Cerqueira e Rangel (2004), para inserirmos um elemento na lista ligada é necessário alocarmos o espaço na memória, de forma dinâmica, para armazenar o elemento e ligá-lo à lista existente.

Podemos inserir um elemento em uma lista em três situações diferentes. Ao inserirmos uma informação na lista ligada, é imprescindível que seja atualizado o valor do ponteiro desta lista, assim, a lista ligada deve apontar ao novo elemento da lista, segundo Celes, Cerqueira e Rangel (2004).

Inserir um novo elemento no início da lista é forma mais simples de inserção em uma lista ligada. Podemos implementar a função para adição de um novo elemento no início da lista, como a seguir:

Exemplificando

Veja o exemplo de um trecho de implementação do código para inserir um novo elemento no início da lista:

```
Lista* inserir (Lista* l, int i) {  
    Lista* novo = (Lista*) malloc(sizeof(Lista));  
    novo -> info = i;  
    novo -> prox = l;  
    return novo;  
}
```

Podemos visualizar que essa função aloca o espaço por meio da função *Malloc*, criando um ponteiro para armazenar o novo elemento na lista, armazenar a nova informação e apontar para o primeiro elemento da lista.

Na Figura 4.6 temos a lista vazia e depois a lista com um elemento inserido, o número 13.

Figura 4.6 | Inserindo um novo elemento no início da lista vazia



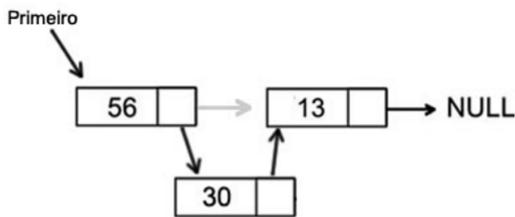
Fonte: elaborada pelo autor.

Assim, podemos implementar o trecho de código a seguir para criar a lista inicial com alguns elementos:

```
int main() {  
    Lista* listaFinal;  
    listaFinal = inicializar();  
    listaFinal = inserir(listaFinal, 13);  
    listaFinal = inserir(listaFinal, 56);  
}
```

Na Figura 4.7, podemos observar que também é possível inserirmos um valor no meio da lista.

Figura 4.7 | Inserindo um novo elemento no meio da lista



Fonte: elaborada pelo autor.

Para isso, podemos utilizar o seguinte código:

```
Lista* inserirPosicao(Lista* l, int pos, int v){  
    int cont = 1;  
    Lista *p = l;  
    Lista* novo = (Lista*)malloc(sizeof(Lista));  
    while (cont != pos){  
        p = p -> prox;  
        cont++;  
    }  
    novo -> info = v;  
    novo -> prox = p -> prox;
```

```

    p -> prox = novo;
    return l;
}

```

E sua implementação de chamada da função principal pode ser definida como:

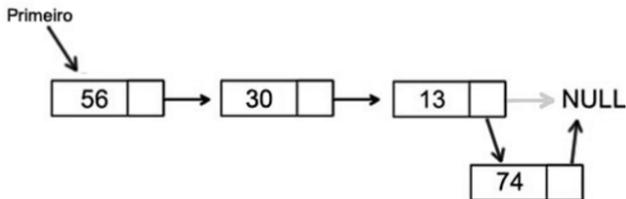
```
listaFinal = inserirPosicao(listaFinal, 1, 30);
```

Assim, no trecho implementado anteriormente, a função *inserirPosicao()* recebe como parâmetros:

- A lista ligada (*listaFinal*).
- A posição que desejamos inserir – neste caso, a primeira posição.
- O elemento – neste caso, o valor 30.

Em uma lista existe a possibilidade de inserção de um elemento no final, como na Figura 4.8, a seguir.

Figura 4.8 | Inserindo um novo elemento no fim da lista



Fonte: elaborada pelo autor.

Podemos visualizar a implementação da função a seguir:

```

Lista* inserirFim(Lista* l, int v) {
    Lista *p = l;
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    while (p -> prox != NULL) {
        p = p -> prox;
        cont++;
    }
    novo->valor = v;
    novo->prox = NULL;
    p->prox = novo;
}

```

```

    }

    novo -> info = v;

    novo -> prox = p -> prox;

    p -> prox = novo;

    return l;

}

```

E, assim, usarmos a implementação do trecho na função principal para chamar a função:

```
listaFinal = inserirFim(listaFinal, 74);
```

A chamada da função *inserirFim()* nos permite passar como parâmetro a lista ligada (listaFinal) e o valor do elemento para inserir, no final dessa lista, o valor 74 do exemplo.

Remover elementos da lista

Com uma lista ligada criada, podemos implementar o uso de funções para a remoção de elementos da lista. Segundo Celes, Cerqueira e Rangel (2004), a função para remover um elemento é mais trabalhosa e complexa e precisa de informações como parâmetros de remoção, o valor do elemento e a lista. Assim, a função atualiza o valor da lista sem o elemento removido.

Caso o primeiro elemento da lista seja o que deve ser retirado, atualizaremos o valor da lista com o ponteiro para o segundo elemento, liberando o espaço alocado do elemento retirado, como podemos observar na Figura 4.9.

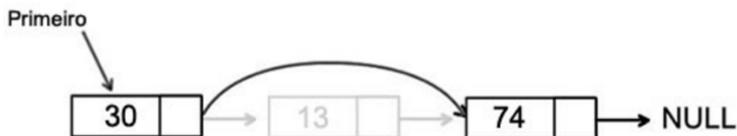
Figura 4.9 | Removendo o primeiro elemento da lista



Fonte: adaptada de Celes, Cerqueira e Rangel (2004, p. 140).

Se o elemento a ser retirado da lista pela função estiver no meio desta, o elemento anterior deve apontar para o elemento seguinte do qual será removido e, após isso, vamos liberar a alocação do elemento removido, como nos mostra a Figura 4.10.

Figura 4.10 | Removendo o elemento do meio da lista



Fonte: adaptada de Celes, Cerqueira e Rangel (2004, p. 140).

Podemos utilizar um único trecho de código em uma função para realizar as operações de remoção da lista, sendo do início ou de outra posição desta lista.

Criamos a função e já declaramos um ponteiro inicializado em *NULL* para ser o ponteiro que vai receber o elemento anterior ao que será excluído e, logo depois, criamos outro ponteiro para receber a lista passada por parâmetro:

```
List* remove (List* l, int v) {  
    List* anterior = NULL;  
    List* p = l;
```

Neste momento, implementamos o comando de repetição *WHILE* para procurar o elemento a ser excluído e guardar a posição do anterior no ponteiro criado anteriormente:

```
while (p != NULL && p -> info != v) {  
    anterior = p;  
    p = p -> prox;  
}
```

A função retorna à própria lista para a função principal, caso o elemento a ser excluído não seja encontrado na lista ligada:

```
if (p == NULL)  
    return l;
```

Sendo encontrado o elemento na lista, o comando de condição *IF* verifica se é o primeiro elemento ou não da lista. Se o valor armazenado no ponteiro for NULO, significa que é o primeiro elemento da lista; caso tenha outro valor, o elemento está em outra posição da lista. Essa função também remove o último valor da lista, já que ele armazena o ponteiro do elemento anterior:

```

if (anterior == NULL) {
    l = p -> prox;
} else {
    anterior -> prox = p -> prox;
}

```

E utilizamos o RETURN da lista para retornar à função principal:

```

return l;
}

```

Declaramos, na função principal, o seguinte trecho de código para chamar a função de retirada de um elemento da lista, onde passamos por parâmetro da lista o elemento que desejamos remover:

```

listaFinal = retira(listaFinal, 56);
listaFinal = retira(listaFinal, 13);

```

Percorrer a lista ligada

Talvez você precise saber quais os elementos que fazem parte da sua lista, em determinado momento do seu sistema. Para tal, é necessário percorrer toda a lista ligada para verificar esses elementos. A lista ligada pode ser impressa com todos os seus elementos, e podemos utilizar o trecho de código a seguir.

Por ser uma função que vai percorrer toda a lista e de impressão em tela, podemos declará-la como *VOID*, uma função que não vai retornar valor para a função principal:

```

void imprimir (Lista* l) {
    Lista* p;
    printf("Elementos:\n");
}

```

Neste trecho, uma condição de repetição *FOR* percorre a lista e imprime todos os elementos encontrados na lista:

```

for (p = l; p != NULL; p = p -> prox) {
    printf(" %d -> ", p -> info);
}

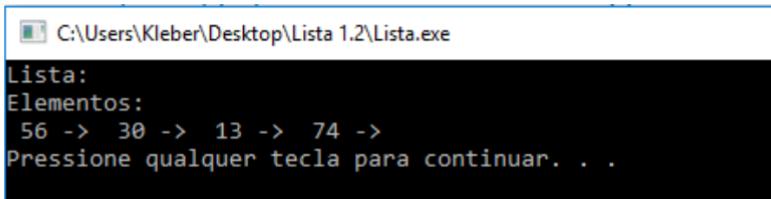
```

Na função principal, declaramos apenas a chamada da função *imprimir()*, passando como parâmetro a lista na qual desejamos imprimir:

```
imprimir(listaFinal);
```

Assim temos o resultado, que pode ser observado na Figura 4.11. Depois da lista percorrida, ela é impressa na tela.

Figura 4.11 | Lista impressa em tela



```
C:\Users\Kleber\Desktop\Lista 1.2\Lista.exe
Lista:
Elementos:
 56 -> 30 -> 13 -> 74 ->
Pressione qualquer tecla para continuar. . .
```

Fonte: elaborada pelo autor.

Verificar se um elemento consta na lista ligada

Outra função muito útil é verificar se determinado elemento consta na lista ou não, segundo Celes, Cerqueira e Rangel (2004). Essa função pode ser criada para receber a informação de qual elemento se deseja buscar e, caso encontre o valor, a função retorna o ponteiro do nó da lista em que representa o elemento ou sua posição na lista, ou, como no nosso exemplo, informa se o elemento foi encontrado ou não.

Podemos utilizar o trecho de código a seguir para implementar a função de busca:

```
Lista* buscar(Lista* l, int v) {
    Lista* p;
```

Criamos uma condição de repetição para procurar na lista o elemento solicitado, passado por parâmetro e até terminar a lista, comparando se o elemento é o que está consultado. Sendo verdadeiro, retorna o ponteiro; caso contrário, retorna *NULL*:

```
for (p = l; p != NULL; p = p -> prox) {
    if (p -> info == v)
        return p;
}
```

```
return NULL;
```

```
}
```

Já para a implementação na função principal, podemos utilizar o seguinte trecho de código para chamar a função e escrever se foi encontrado ou não o elemento. Se o retorno for *NULL*, será escrito *Elemento não encontrado*; caso encontre, será escrito *Elemento encontrado*, como é exibido na Figura 4.12.

```
if (busca(listaFinal, 74) == NULL) {  
    printf("\n Elemento não encontrado");  
}  
else {  
    printf("\n Elemento encontrado");  
}
```

Figura 4.12 | Lista impressa em tela

```
C:\Users\Kleber\Desktop\Lista 1.2\Lista.exe  
Lista:  
Elementos:  
56 -> 30 -> 13 -> 74 ->  
- Elemento 74 encontrado  
Pressione qualquer tecla para continuar. . .
```

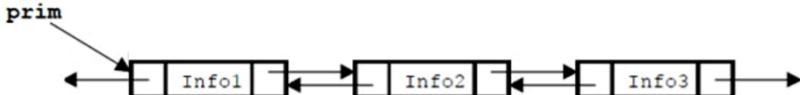
Fonte: elaborada pelo autor.

Definição de lista duplamente ligada

Segundo Celes, Cerqueira e Rangel (2004), em uma lista duplamente ligada, cada nó apresenta um elemento com informações, um ponteiro para seu próximo elemento e um ponteiro para seu elemento anterior.

Dessa forma, é possível acessar tanto o próximo elemento como o elemento anterior da lista, percorrendo-a também na ordem inversa, do final até o início. O primeiro elemento aponta para seu anterior *NULL* (valor nulo), assim como o ponteiro do último elemento da lista, como exibido na Figura 4.13.

Figura 4.13 | Lista duplamente ligada



Fonte: adaptada de Celes, Cerqueira e Rangel (2004, p. 149).

Segundo Tenenbaum, Langsam e Augenstein (2007), um nó em uma lista duplamente ligada consiste na criação de três campos:

- Um campo-elemento para a informação.
- Um ponteiro direcionando para o próximo elemento.
- Um ponteiro direcionando para o elemento anterior.

Na criação de uma lista duplamente ligada, é preciso criar – além do tipo de dado que será utilizado em sua implementação e o ponteiro que informa qual o próximo elemento – o ponteiro direcionando para o elemento anterior da lista, como o modelo de implementação a seguir:

```
struct Lista {  
    int info;  
    struct lista* ant;  
    struct lista* prox;  
};  
  
typedef struct lista Lista;
```

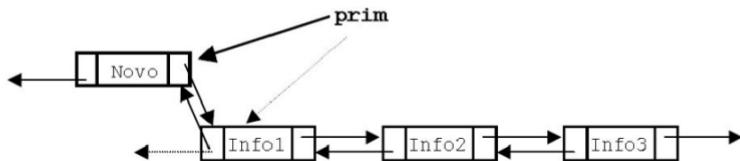
Como na lista ligada simples, também precisamos inicializar a lista duplamente ligada para a utilizarmos após a sua declaração. Uma das possíveis formas de inicialização é criar a função retornando a lista como nula:

```
/* Função para retornar uma lista vazia */  
Lista* inicializa (void)  
{  
    return NULL;  
}
```

Adicionar elementos à lista duplamente ligada

Podemos adicionar um novo elemento à lista duplamente ligada. Se a lista estiver vazia, este elemento terá como elementos anterior e próximo o valor *NULL*. No caso de a lista estar com elementos inseridos, ao adicionar um novo elemento, o elemento antigo passa a ser o próximo elemento da lista e o anterior passa a receber o valor *NULL*. A Figura 4.14 representa a adição de um novo elemento no início da lista.

Figura 4.14 | Adição de um novo elemento no início da lista



Fonte: adaptada de Celes, Cerqueira e Rangel (2004, p. 150).

O trecho de código a seguir representa a inserção de um novo elemento no início da lista:

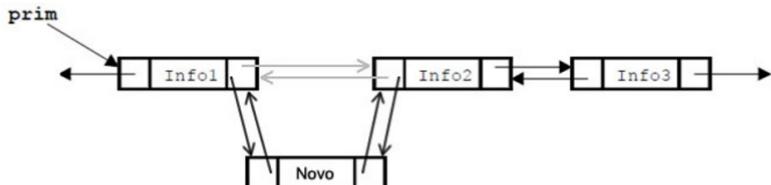
```
Lista* inserir (Lista* l, int i) {  
    Lista* novo = (Lista*) malloc(sizeof(Lista));  
    novo -> info = i;  
    novo -> prox = l;  
    novo -> ant = NULL;  
    //Verifica se lista não está vazia  
    if (l != NULL)  
        l -> ant = novo;  
    return novo;  
}
```

A chamada para a função dentro da função principal Main pode ser realizada pela linha:

```
int main() {  
    Lista* listaFinal;  
    listaFinal = inicializar();  
    listaFinal = inserir(listaFinal, 20);  
}
```

A adição de um novo elemento pode ser realizada tanto no início da lista como no seu final, ou em uma posição específica. Na Figura 4.15, podemos notar a representação da adição de um elemento no meio da lista.

Figura 4.15 | Inserindo no meio da lista



Fonte: elaborada pelo autor.



Para consultar um trecho do código referente à função de adição em uma posição da lista e um exemplo para adição no final da lista, acesse o QR Code ou o link: http://cm-cls-content.s3.amazonaws.com/ebook/embed/qr-code/2020-1/algoritmos_programacao_estruturada/u4/s1/l0kls201_u4s1_alg_pro_est_QRCODE1.pdf

Assimile

É bom salientar que em uma lista duplamente ligada é necessário sabermos o ponteiro para o elemento anterior e para o próximo elemento, quando for implementado um código de adição de elementos no meio da lista.

Remover elementos da lista duplamente ligada

Segundo Celes, Cerqueira e Rangel (2004), a função de remoção em uma lista permite remover um elemento da lista duplamente ligada apenas conhecendo o ponteiro para este elemento. Para facilitar a localização de um elemento na lista, podemos utilizar a função de busca e, em seguida, ajustar o encadeamento da lista, liberando, por fim, o elemento da alocação de memória.

Um trecho de uma função de busca pode ser implementado por:

```
Lista* busca(Lista* l, int v) {  
    Lista* p;  
  
    for (p = l; p != NULL; p = p -> prox) {
```

```

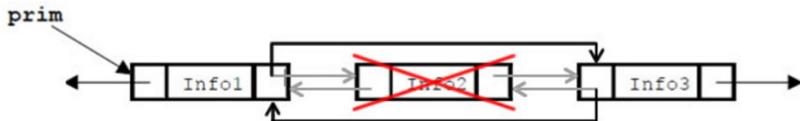
        if (p -> info == v)
            return p;
    }

    return NULL;
}

```

Assim, encontrado o elemento que se deseja remover, basta apontar o anterior para o próximo e o próximo para o anterior, permitindo que o elemento no meio da lista possa ser removido do encadeamento, como na Figura 4.16:

Figura 4.16 | Remoção de um elemento da lista



Fonte: elaborada pelo autor.

Conforme Celes, Cerqueira e Rangel (2004), se o elemento que desejamos remover estiver no início ou no final da lista, o apontamento para o anterior ao início será nulo, assim como se for o último, o apontamento para o próximo também será nulo.

Podemos implementar o trecho de código de remoção como segue:

```

Lista* retira (Lista* l, int v) {
    Lista* ant = NULL;
    Lista* p = l;
    while (p != NULL && p -> info != v) {
        ant = p;
        p = p -> prox;
    }
    if (p == NULL)
        return l;
    if (ant == NULL) {
        l = p -> prox;
    }
    else
        ant -> prox = p -> prox;
    delete p;
    return l;
}

```

```

    }

    else

    {

        p -> ant -> prox = p -> prox;

    }

    if (p -> prox != NULL)

        p -> prox -> ant = p -> ant;

    return l;

}

```

Ordenar a lista duplamente ligada

Em uma lista duplamente ligada, é possível realizar a ordenação de seus elementos de forma crescente ou decrescente, criando uma simples comparação entre os elementos e realizando a sua troca, em caso de aceitarem a condição de maior ou menor. Podemos utilizar o trecho de código a seguir para realizar a ordenação:

```

void Ordena(Lista* l) {

    Lista* p;

    Lista* aux;

    int temp;

    for (p = l; p != NULL; p = p -> prox) {

        for (aux = p -> prox; aux != NULL; aux = aux -> prox) {

            if ((p -> info) > (aux -> info)) {

                temp = p -> info;

                p -> info = aux -> info;

                aux -> info = temp;

            }

        }

    }

}

```

Para chamar a função de ordenação, passamos somente na função principal *Main* o parâmetro a seguir:

```
printf("\n Lista Ordenada");  
Ordena(listaFinal);
```

Para a utilização das funções apresentadas nesta seção, podemos criar uma função principal *Main* com a seguinte implementação (em QR Code):



Para melhor visualizar o conteúdo, acesse o QR Code ou o link: http://cm-cls-content.s3.amazonaws.com/ebook/embed/qr-code/2020-1/algoritmos_programacao_estruturada/u4/s1/lklis201_u4s1_alg_proest_QRCODE2.pdf

Neste exemplo, podemos observar que o sistema realiza a chamada da função e executa as rotinas, mas é possível, dentro da função principal *Main*, criar rotinas nas quais o usuário pode informar os elementos para serem inseridos, removidos ou para realizar a busca de um elemento.

Reflita

Sabemos que as listas duplamente ligadas permitem as mesmas funções que as listas ligadas simples, no entanto, podemos saber qual o elemento anterior e o próximo elemento devido aos ponteiros existentes. De que forma poderíamos imprimir a lista em ordem inversa em uma lista duplamente ligada?

Agora que você conheceu as operações sobre listas duplamente ligadas, você poderá desenvolver sistemas mais complexos. Vamos praticar?

Sem medo de errar

Você tem a função de atender à demanda de um cliente de um comércio de autopeças que possui duas unidades, uma matriz e uma filial, e deseja gerar um relatório para informação de estoque mínimo em seu sistema. O cliente solicitou que o relatório seja gerado em tempo real, apresentando um gráfico atualizado todas as vezes que um produto entrar ou sair desta listagem, por meio da venda ou compra de produtos. Este cliente também deseja pesquisar se um produto consta na listagem ou não.

Como desafio, você precisa implementar no relatório em que está trabalhando a adição ou remoção de produtos na listagem de produtos.

Como podemos adicionar ou remover produtos nesta lista? Pesquise sobre algoritmos de adição e remoção de dados em uma lista ligada.

Como é possível adicionar a função de busca por algum produto específico na listagem? Você vai precisar pesquisar e compreender o funcionamento do algoritmo para percorrer uma lista e verificar se determinado valor se encontra nela.

Para a resolução desta situação-problema, é necessário compreender os conceitos sobre as operações em listas ligadas e pesquisar outras técnicas para a criação das funções, para executar a adição e a remoção de elementos, assim como percorrer a lista e realizar uma contagem dos elementos e a criação da busca de elementos em uma lista ligada.

Para iniciar, implementamos a lista como:

```
#include <stdio.h>
#include <stdlib.h>

struct listaProd {
    int codigo;
    char produto[30];
    struct listaProd* prox;
};

typedef struct listaProd Produtos;
Produtos* inicializar (void) {
    return NULL;
}

Produtos* inserir (Produtos * l, int i, char* nprod) {
    Produtos* novo = (Produtos*) malloc(sizeof(Produtos));
    novo -> codigo = i;
    novo -> produto = nprod;
    novo -> prox = l;
    return novo;
}
```

```

Produtos* retira (Produtos* l, int v) {
    Produtos* ant = NULL;
    Produtos* p = l;
    while (p != NULL && p -> codigo != v) {
        ant = p;
        p = p -> prox;
    }
    if (p == NULL)
        return l;
    if (ant == NULL) {
        l = p -> prox;
    } else {
        ant -> prox = p -> prox;
    }
    return l;
}

Produtos* busca(Produtos* l, int v) {
    Produtos* p;
    for (p = l; p != NULL; p = p -> prox) {
        if (p -> codigo == v)
            return p;
    }
    return NULL;
}

```

Assim, podemos implementar a função principal *Main* com a seguinte solução:

```

int main() {
    int cont, codprod;
    char nprod[30];

```

```

Produtos* listaProdutos;

listaProdutos = inicializar(); /* inicializa
lista como vazia */

for (cont = 0; cont < 3; cont++) {
    printf("\nInforme o codigo do Produto: ");
    scanf("%d", &codprod);
    printf("\nInforme o nome do Produto: \n");
    scanf("%d", &nprod);

    listaProdutos = inserir(listaProdutos,
codprod, nprod);

}

printf("Lista Produtos:\n");
imprimir(listaProdutos);

printf("\nInforme o codigo do produto para
pesquisa: ");

scanf("%d", &codpro);

if (busca(listaProdutos, codprod) == NULL) {
    printf("\n\n- Produto não encontrado\n");
} else {
    printf("\n\n- Produto encontrado\n");
}

printf("\n"); system("PAUSE");
}

```

Após a criação desse algoritmo, e com o material desta seção, você poderá desenvolver seu algoritmo aprimorando as funções apresentadas em aula.

Pesquisa de mercado

Uma empresa que realiza pesquisa de mercado precisa solucionar um problema de demanda que surgiu. Com a alta nos valores dos produtos nos últimos meses, diversos supermercados trabalham com valores diferentes e, a fim de entender essa diferença de valores e a margem de um mercado para outro, esta empresa contratou você, um expert na área de programação, para desenvolver um sistema para cadastrar esses produtos de pesquisa, valores e em qual supermercado foi realizada a pesquisa. Esse sistema também deve permitir a geração de um relatório detalhado das pesquisas realizadas.

Sua função é criar esse sistema para cadastro desses produtos, para que possam ser inseridos produtos após o início da pesquisa e gerados relatórios com as informações detalhadas de cada produto.

Resolução da situação-problema

A criação desse sistema para a empresa de pesquisa pode nos dar um exemplo bem bacana de como criar um sistema utilizando lista duplamente ligada.

Podemos identificar que será necessário criar uma estrutura com um campo para descrição do produto, outro campo para descrição do supermercado no qual será realizada a pesquisa e um campo para informar o valor pesquisado, além dos ponteiros para o produto anterior e posterior da lista.

```
struct produto {  
    char[25] produto;  
    char[30] mercado;  
    float valor;  
    struct produto* ant;  
    struct produto* prox;  
};  
  
typedef struct produto Pesquisa;
```

Com base na estrutura da lista criada, criamos a implementação da função principal *Main*, em que o usuário informará o produto, o supermercado e o valor. Ao inserir esses dados na lista, um novo ponteiro para o

próximo elemento e para o anterior serão criados. Uma das funções a serem implementadas é a função para imprimir a lista duplamente ligada, de forma que o usuário possa escolher se ele deseja a apresentação em ordem crescente ou decrescente.

Faça valer a pena

1. Podemos inserir um elemento em uma lista em três situações diferentes. Ao inserirmos uma informação na lista ligada, é imprescindível que seja atualizado o valor do ponteiro dessa lista, assim a lista ligada deverá apontar ao novo elemento da lista, segundo Celes, Cerqueira e Rangel (2004).

Dado o trecho de código a seguir:

```
Lista* inserir (Lista* l, int i) {  
    Lista* novo = (Lista*) malloc(sizeof(Lista));  
    novo -> info = i;  
    novo -> prox = l;  
    return novo;  
}
```

Assinale a alternativa que define a funcionalidade do trecho de código informado:

- a. Inserir um elemento no meio da lista.
- b. Remover um elemento do início da lista.
- c. Apagar toda a lista para recomeçar.
- d. Inserir um elemento no final da lista.
- e. Inserir um elemento no início da lista.

2. Segundo Celes, Cerqueira e Rangel (2004), com a estrutura de dados das listas ligadas não há como percorrer os elementos de forma inversa na lista, iniciando pelo fim até o início. E, apesar de ser possível, a retirada de um elemento da lista com encadeamento simples é mais trabalhosa de ser realizada, pois é necessário percorrer toda a lista para encontrar o elemento anterior, pois dado o ponteiro para determinado elemento, temos acesso ao seu próximo elemento e não ao anterior.

Com base no texto, podemos afirmar que a alternativa correta quanto às listas duplamente ligadas é:

- a. Uma lista duplamente ligada requer obrigatoriamente a utilização de duas estruturas do tipo lista.
- b. Uma lista duplamente ligada possui em um nó somente um ponteiro para o próximo elemento e um ponteiro para o elemento anterior.
- c. Uma lista duplamente ligada não permite a impressão da lista de forma inversa, devido aos ponteiros que não mantêm a informação do elemento anterior.
- d. Uma lista duplamente ligada possui, em um nó, o campo para informação, um ponteiro para o próximo elemento e um ponteiro para o elemento anterior.
- e. Uma lista duplamente ligada, por manter informações do elemento anterior, não utiliza do recurso de alocação dinâmica de memória.

3. Em uma lista duplamente ligada é possível realizar a ordenação de seus elementos de forma crescente ou de forma decrescente, criando uma simples comparação entre os elementos e realizando sua troca em caso de aceitarem a condição de maior ou menor.

Considerando a estrutura de dados de ordenação em ordem crescente, o trecho de código que melhor define a ordenação na lista duplamente ligada é:

- a.

```
for (p = l; p != NULL; p = p -> prox) {
    for (aux = p -> prox; aux != NULL; aux = aux
-> prox) {
        if ((p -> info) < (aux -> info)) {
            temp = p -> info;
            p -> info = aux -> info;
            aux -> info = temp;
        }
    }
}
```
- b.

```
for (p = l; p != NULL; p = p -> prox) {
    for (aux = p -> prox; aux != NULL; aux = aux
-> prox) {
        if ((p -> info) > (aux -> info)) {
```

```
p -> info = aux -> info;
aux -> info = p -> info;
} } }

c. for (p = l; p != NULL; p = p -> prox) {
for (aux = p -> prox; aux != NULL; aux = aux
-> prox) {
if ((p -> info) > (aux -> info)) {
temp = p -> info;
p -> info = aux -> info;
aux -> info = temp;
} } }

d. for (p = l; p != NULL; p = p -> prox) {
for (aux = p -> prox; aux != NULL; aux = aux
-> prox) {
if ((p -> info) < (aux -> info)) {
p -> info = aux -> info;
aux -> info = p -> info;
} } }

e. for (p = l; p != NULL; p = p -> prox) {
for (aux = p -> prox; aux != NULL; p = p ->
prox) {
if ((p -> info) > (aux -> info)) {
temp = p -> info;
p -> info = aux -> info;
aux -> info = temp;
} } }
```

Seção 2

Pilhas

Diálogo aberto

As pilhas estão presentes no nosso dia a dia, mesmo que não percebamos. Você já deve ter visto os caminhões-cegonha pelas estradas, por foto ou até mesmo por televisão. Os carros que estão no caminhão são inseridos um por vez, e se for preciso remover o primeiro carro colocado no caminhão, será necessário remover o que está atrás dele.

Com o intuito de conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas, você vai conhecer a problematização sobre labirintos, desafios direcionados para soluções com pilhas.

Imagine poder utilizar a programação com a estrutura de pilhas para a solução de qualquer tipo de labirinto, com a utilização de funções de inserir e remover elementos de uma pilha.

Você vai poder implementar soluções para diversos tipos de problemas utilizando pilhas como Torre de Hanoi, empilhamento de caixas em um estoque ou até mesmo para a solução de montagem de produtos pelo método de empilhamento.

Como sabemos, você foi recomendado pela empresa de TI na qual trabalha para auxiliar um cliente no desenvolvimento de um sistema de prestação de serviços terceirizados em montagem de notebooks. Sua função será realizar um relatório com o levantamento para a linha de montagem e teste dos notebooks, com a criação de um módulo de testes nos notebooks, utilizando as operações de pilhas. Esse sistema vai precisar testar as peças montadas e realizar sua troca, caso haja algum problema.

Você tem o desafio de realizar o levantamento do sistema de testes dos notebooks, a fim de identificar as peças com defeitos e realizar sua troca.

Vamos começar?

Não pode faltar

Definição e elementos de pilhas

Caro aluno, em programação é fundamental o uso de estruturas de dados para a criação de sistemas, e a mais simples e utilizada é a estrutura de dados do tipo pilha.

Segundo Tenenbaum, Langsam e Augenstein (2007), uma pilha tem como definição básica ser um conjunto de elementos ordenados que permite a inserção e a remoção de mais elementos em apenas uma das extremidades da estrutura denominada topo da pilha, como exibido na Figura 4.17.

Figura 4.17 | Modelo de estrutura de dados pilha



Fonte: elaborada pelo autor.

Assim, um novo elemento que é inserido passa a ser o topo da pilha, e o único elemento que pode ser removido da pilha é o elemento que está no topo.

Conforme Tenenbaum, Langsam e Augenstein (2007), uma pilha é um objeto dinâmico em constante mutação. Ela é mutável porque novos itens podem ser inseridos em seu topo, assim como o elemento que estiver no topo pode ser removido.

Exemplificando

Para compreendermos melhor seu funcionamento, tome como exemplo os empilhamentos de produtos que um supermercado coloca no início das prateleiras ou no meio dos corredores. Quando vamos comprar um produto que está nesse empilhamento, conseguimos apenas pegar o produto que está sempre por cima. E quando o funcionário vai

abastecer, ele consegue somente inserir no topo da pilha, como pode ser observado na Figura 4.18.

Figura 4.18 | Pilha de caixas



Fonte: Shutterstock.

Para acessarmos algum outro elemento que esteja no meio da pilha ou no final, precisamos remover todos os elementos que estão acima deste elemento, até encontrá-lo.

Podemos fazer uma analogia muito comum de estrutura de dados do tipo pilha com uma pilha de pratos. Quando vamos a algum restaurante para comer e pegamos um prato, este prato está no topo da pilha. Se desejamos pegar o prato debaixo:

1. Removemos o prato do topo.
2. Retiramos o prato debaixo.
3. Retornamos o prato anterior para a pilha.

A criação de uma pilha simples em C++ pode ser declarada com:

- Uma estrutura de dados contendo dois elementos apenas.
- Um vetor para armazenar os elementos da pilha.
- Uma variável do tipo inteiro para armazenar a posição atual do topo da pilha.

A principal diferença entre uma pilha e um vetor está na forma de utilização dessas estruturas. Enquanto declaramos um vetor com um tamanho fixo, declaramos a pilha com um tamanho dinâmico que está sempre mudando conforme os elementos são inseridos ou removidos, por meio da alocação dinâmica de memória.

Regras para operação de pilhas

A Torre de Hanoi é um brinquedo pedagógico que representa uma pilha e é muito utilizado em estrutura de dados como exemplo de programação do tipo pilha, conforme exibido na Figura 4.19.

Figura 4.19 | Torre de Hanoi



Fonte: Shutterstock.

A Torre de Hanoi tem como objetivo transferir os elementos de uma pilha para outra, sempre mantendo o elemento de maior tamanho abaixo dos elementos de tamanho menor. Assim, todos os elementos devem ser movidos de uma pilha a outra, executando a transferência de uma elemento por vez, sempre mantendo esta regra.

Reflita

Como sabemos, a estrutura de dados do tipo pilha representa o empilhamento de elementos em uma estrutura de alocação dinâmica de memória, sendo possível identificarmos o elemento do topo. Como seria possível a contagem de elementos dentro desta pilha?

Os elementos inseridos em uma pilha apresenta, uma sequência de inserção. O primeiro elemento que entra na pilha só pode ser removido por último, após todos os outros elementos serem removidos.

Segundo Celes, Cerqueira e Rangel (2004), os elementos da pilha só podem ser retirados na ordem inversa da ordem em que nela foram inseridos. Isso é conhecido como LIFO (*last in, first out*, ou seja, o último que entra é o primeiro a sair) ou FILO (*first in, last out*, ou seja, primeiro que entra é o último a sair). Podemos observar a inserção na pilha do elemento sempre em seu topo na Figura 4.20:

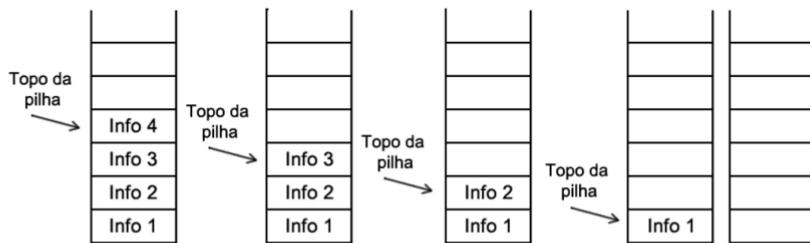
Figura 4.20 | Inserindo elemento na pilha



Fonte: elaborada pelo autor.

Podemos remover os elementos da pilha com identificação do elemento que está no seu topo, conforme a Figura 4.21.

Figura 4.21 | Removendo um elemento da pilha



Fonte: elaborada pelo autor.

Uma pilha também pode estar no estado de pilha vazia quando não houver elementos. Segundo Celes, Cerqueira e Rangel (2004), os passos para a criação de uma pilha são:

1. Criar uma pilha vazia.
2. Inserir um elemento no topo.
3. Remover o elemento do topo.
4. Verificar se a pilha está vazia.
5. Liberar a estrutura de pilha.

Conforme os mesmos autores, em uma estrutura de pilha devem ser implementadas duas operações básicas:

- Empilhar um novo elemento.
- Desempilhar um elemento.

Segundo Lorenzi, Mattos e Carvalho (2015), a operação de empilhar um novo elemento tem a função de inserir um novo elemento na pilha e é definida na programação em C++ como *push()*. Como colocar um livro em cima de uma pilha de livros: você colocará o livro no topo da pilha.

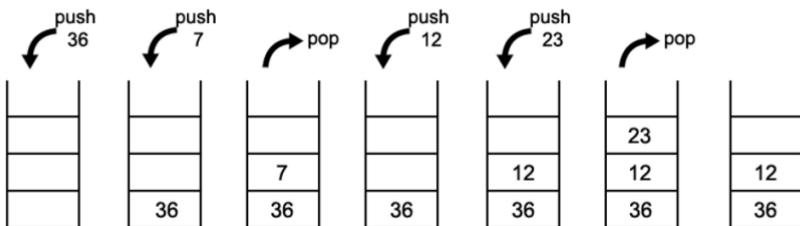
Já a operação de desempilhar tem a função de remover um elemento do topo da pilha e é utilizada na programação em C++ como *pop()*. Por exemplo, remover o livro que está no topo da pilha.

Conforme Drozdek (2016), outras operações que podem ser implementadas na pilha são:

- Limpar a pilha, utilizando a função *clear()*.
- Verificar se a pilha está vazia, com base na função *isEmpty()*.

Na Figura 4.22 a seguir, podemos observar uma sequência de operações para inserir um novo elemento na pilha com a função *push()* e remoção do elemento com a função *pop()*.

Figura 4.22 | Sequência de funções *push()* e *pop()* na pilha



Fonte: adaptada de Drozdek (2016, p. 116).

Você pode observar, na Figura 4.22, como funciona a função *push()*, que insere um novo elemento na pilha, e a função *pop()*, que remove o elemento do topo. No início, a pilha está vazia e, com a função *push*, é inserido o valor 36, depois, o valor 7. No próximo passo, com a função *pop*, é removido o valor 7. Novamente, com a função *push*, são inseridos os valores 12 e 23. Por fim, é removido o valor de 23 da pilha.

Exemplificando

Podemos dar como exemplo a utilização de cestas de compras em um supermercado. Elas ficam empilhadas uma dentro da outra, e para utilizar uma cesta o cliente retira a cesta que está no topo dessa pilha.

Após serem recolhidas pelo supermercado por um colaborador, as cestas são inseridas no topo da pilha, onde ficam alocadas.

Inserir elementos na pilha

Conforme Tenenbaum, Langsam e Augenstein (2007), uma pilha apresenta uma estrutura que pode ser declarada contendo dois objetos:

- Um ponteiro, que armazenará o endereçamento inicial da pilha.
- Um valor inteiro, que indicará a posição do topo da pilha.

Para utilização de uma pilha, primeiramente é necessário:

- Criar a declaração da estrutura da pilha.
- Criar a pilha com a alocação dinâmica.
- Criar as funções para inserir e remover da pilha.

A declaração da estrutura inicial para criação de uma pilha pode ser implementada por:

```
struct Pilha {  
    int topo;  
    int capacidade;  
    float * proxElem;  
};  
  
struct Pilha minhaPilha;
```

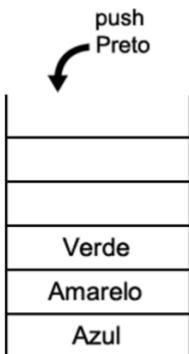
Segundo Celes, Cerqueira e Rangel (2004), com a estrutura declarada você pode criar a função para criar uma pilha. Essa função aloca dinamicamente na memória o espaço para utilização da pilha.

O trecho de código para criação da pilha pode ser implementado por:

```
void cria_pilha(struct Pilha *p, int c ){  
    p -> proxElem = (float*) malloc (c * sizeof(float));  
    p -> topo = -1;  
    p -> capacidade = c;  
}
```

Com a função para criar a pilha realizada, ela estará vazia, ou seja, não haverá elemento algum na pilha em sua criação. Assim, você pode criar a função que vai permitir ser inserido um novo elemento na pilha, como podemos observar na Figura 4.23:

Figura 4.23 | Inserindo um elemento na pilha



Fonte: elaborada pelo autor.

Observe, na Figura 4.23, que a função *push()* insere o elemento Preto no topo da pilha, passando esse elemento a ser o topo da pilha.

Em programação, para inserir um novo elemento na pilha você pode uma criar a função com o nome *push_pilha()*, como é demonstrado no trecho para implementação a seguir:

```
void push_pilha(struct Pilha *p, float v){  
    p -> topo++;  
    p -> proxElem [p -> topo] = v;  
}
```

No trecho apresentado, a função *push_pilha()* recebe a *struct* da pilha e o elemento a ser inserido por meio da passagem de parâmetros pela variável *v*.

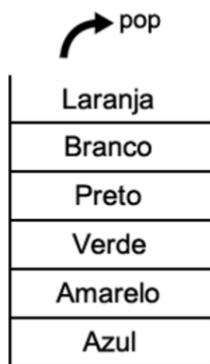
Assimile

Precisamos nos recordar de que a função para adicionar um novo elemento na pilha será realizada somente para inserir esse elemento no topo da pilha e não permite a inserção no seu meio.

Remover elementos da pilha

Agora que você já sabe como inserir um novo elemento na pilha, vai aprender a remover um elemento do seu topo. Como já estudamos, a remoção de um novo elemento é realizada somente pelo topo da pilha, como o exemplo na Figura 4.24, em que o elemento Laranja está no topo e será removido com a função *pop()*.

Figura 4.24 | Removendo elemento do topo da pilha



Fonte: elaborada pelo autor.

A implementação do trecho de código para a remoção de elementos do topo da pilha pode ser declarada como no modelo a seguir, utilizando a função *pop_pilha()*:

```
float pop_pilha (struct Pilha *p){  
    float aux = p -> proxElem [p -> topo];  
    p -> topo--;  
    return aux;  
}
```

No trecho apresentado, a função *pop_pilha()* recebe a *struct* da pilha, e a variável *aux* declarada recebe o elemento que está no topo. Na linha a seguir, o valor do topo é decrementado e é retornado o elemento removido da pilha.

Informar se a pilha está vazia

Uma pilha não possuirá nenhum elemento em sua inicialização, assim, não é possível remover elementos, apenas inseri-los. É interessante tratar este método para o sistema informar ao usuário que a remoção do elemento não é possível e que a pilha está vazia, como na Figura 4.25.

Figura 4.25 | Pilha vazia



Fonte: elaborada pelo autor.

O trecho para implementação do código para verificar se a pilha está vazia pode ser dado por:

```
/*Declaração da função pilha_vazia com passagem  
da pilha por parâmetro*/  
  
int pilha_vazia (struct Pilha *p ){  
    if( p -> topo == -1 )  
        return 1; /*Sendo o topo igual a -1, a função  
retorna verdadeiro*/  
  
    else  
        return 0; /*Caso contrário, a função retorna  
verdadeiro*/  
}
```

O código dado verifica se a posição do topo é -1 (menos um). Caso seja, a função retorna 1 (um) como sendo positivo para uma pilha vazia, ou 0 (zero), caso o valor do topo seja diferente, identificando, assim, que a pilha não está vazia.

Da mesma forma que podemos verificar se a pilha está vazia, é possível verificar se a está cheia, comparando se o valor do topo é igual ao valor total da sua capacidade. O trecho a seguir apresenta a implementação do código para verificar se a pilha está cheia:

```
int pilha_cheia ( struct Pilha *p ) {  
    if (p -> topo == p -> capacidade - 1)  
        return 1;  
    else  
        return 0;  
}
```

Refita

Conforme você pôde observar, em uma pilha é possível identificarmos e removermos o elemento do topo. Vamos supor que o usuário deseja saber quais os elementos da pilha e em quais posições eles estão. Como seria possível realizar essa operação?

Operações com pilhas: problema do labirinto

Um dos problemas mais comuns para solucionar com pilhas são os labirintos. Os labirintos são desafios criados como problematização de estrutura de dados. As pilhas podem ser aplicadas também no uso de algoritmos de *Backtracking*, que consiste em criar marcações para onde o algoritmo pode retornar.

Para, por exemplo, encontrar um caminho correto em um labirinto, podemos andar por ele até encontrarmos uma divisão nesse caminho. Assim, adicionamos a posição em que a divisão ocorre, junto com o caminho escolhido na pilha, e seguimos pelo caminho escolhido.

Caso o caminho escolhido não apresenta uma saída, é removido o ponto anterior da pilha, voltando ao último ponto em que o labirinto se dividiu, e recomeçamos por um outro caminho ainda não escolhido, adicionando na pilha o novo caminho.

O algoritmo de *Backtracking* pode ser aplicado também como operação de desfazer, existente em diversas aplicações de usuários, como a utilização deste algoritmo em sistema de GPS. Quando o motorista utiliza uma rota não indicada pelo programa, o algoritmo de *Backtracking* é aplicado para redefinir a nova rota.

Para implementar a operação de *Backtracking*, as ações são armazenadas em uma pilha e, caso a operação de desfazer seja realizada, o estado anterior do sistema pode ser restaurado, ou a ação contrária à realizada pode ser executada.

A seguir, um trecho de implementação de criação de uma solução para labirinto:

```
/*Chama a função inicLabirinto, passando o
labirinto, a pilha, o valor da linha e da coluna
como passagem de parâmetros*/

void inicLabirinto(Labirinto *l, Pilha *p_l, int
linha, int coluna) {

    int i, j, flag = 0;

    char aux;

    elem_t_pilha origem;

    /*Aplicamos uma rotina em matriz para verificar
    se a posição foi visitada (1) ou não (0)*/

    for(i = 0 ; i < linha ; i++){

        for(j = 0 ; j < coluna ; j++){

            if(l->p[i][j].tipo == '0'){

                l->p[i][j].visitado = 1; //Visitado

                origem.x = i;
                origem.y = j;

                /*Insere na pilha a posição de origem*/
                push(p_l, origem);

            }
        }
    }
}
```

Assim, o algoritmo de *Backtracking* tem como meta resolver o problema no menor intervalo de tempo possível, sem levar em consideração o esforço de operações para alcançar a solução do problema.

Agora é com você!

Sem medo de errar

Como sabemos, você foi recomendado pela empresa de TI na qual trabalha para auxiliar um cliente no desenvolvimento de um sistema de prestação de serviços terceirizados em montagem de notebooks. Sua função será realizar o levantamento para a linha de montagem e o teste dos notebooks, com a criação de um módulo de testes dos notebooks utilizando as operações de pilhas. Esse sistema vai precisar testar as peças montadas e realizar sua troca, caso haja algum problema.

Com isso, você tem o desafio de realizar o levantamento do sistema de testes dos notebooks, a fim de identificar as peças com defeitos e realizar sua troca.

O objetivo é entregar este desafio em forma de relatório com a implementação das operações de pilhas para verificar se o notebook está montando corretamente.

Para a resolução deste desafio, será necessário montar o esquema de montagem dos notebooks e a sequência de peças que serão inseridas.

A montagem tem como base criar um sistema em que as peças serão disponibilizadas na esteira, conforme a montagem do notebook, com a primeira peça a ser colocada entrando primeiro, assim, a última peça a ser inserida no notebook, será a peça do topo.

Caso algum notebook apresente alguma falha, o sistema vai identificar e desempilhar as peças colocadas até encontrar a peça com defeito para ser substituída. Ao trocar a peça defeituosa, o sistema remonta o notebook com base em sua pilha de peças, a partir da peça trocada.

```
struct peça {  
    int topo; /* posição elemento topo */  
    int qtd;  
    float *pPeca; /* aponta para a próxima peça */  
};  
  
void criarteste(struct Pecas *p, int c ){  
    p -> topo = -1;  
    p -> qtd = q;  
    p -> pPeca = (float*) malloc (c * sizeof(float));  
}
```

```

int note_vazio ( struct Pecas *p ){
    if( p -> topo == -1 )
        return 1; /* Caso não tenha pecas, o notebook
está sem pecas */
    else
        return 0; /* Caso contrário, o notebook está
com pecas */
}

int note_completo (struct Pecas *p){
    if (p -> topo == p -> capa - 1)
        return 1; /* Caso o notebook esteja
completo retorna 1 para verdadeiro */
    else
        return 0; /* Caso contrario retorna 0 para
falso */
}

void insere_peca (struct Pecas *p, float num){
    p -> topo++;
    p -> pPeca [p -> topo] = num;
}

float remove_peca (struct Pecas *p){
    float aux = p -> pPeca [p -> topo]; /* Aux recebe
a peca removida do topo */
    p -> topo--;
    return aux;
}

float retorna_peca (struct Pecas *p ){
    return p -> pPeca [p -> topo]; /* Retorna qual
peca está no topo */
}

int main(){

```

```

    struct Pecas TestePecas;
    int qtd, op;
    float num;
    printf( "\nInforme a quantidade de pecas do
notebook? " );
    scanf("%d", &qtd);
    criarteste (&TestePecas, qtd);
    while( 1 ){ /* Menu para teste */
        printf("1- Inserir Peca \n");
        printf("2- Remover Peca \n");
        printf("3- Mostrar ultima peca \n");
        printf("4- Sair\n");
        printf("\nopcao? ");
        scanf("%d", &op);
        switch (op) {
            case 1:
                if(note_completo( &TestePecas
) == 1)
                    printf("\nNotebook
Completo! \n");
                else {
                    printf("\nInformar o
numero da peca? \n");
                    scanf("%f", &num);
                    insere_peca
(&TestePecas, num);
                }
                break;
            case 2:
                if (note_vazio(&TestePecas)

```

```

== 1)
                                printf("\nNotebook
Vazio! \n");
else{
    valor = remove_peca
(&TestePecas);
    printf ("\\n%f Peca
removida!\n", num);
}
break;

case 3:
if (note_vazio(&TestePecas)
== 1)
printf("\nNotebook
Vazio! \n");
else {
    valor = retorna_peca
(&TestePecas);
    printf ("\nUltima peca
inserida: %f\n", num);
}
break;

case 4:
exit(0);

default: printf("\nOpcao Invalida!
\n");
}
}
}

```

Faça valer a pena

1. Os elementos inseridos em uma pilha apresentam uma sequência de inserção, sendo o primeiro elemento que entra na pilha só pode ser removido por último, após todos os outros elementos serem removidos. Assim, os elementos da pilha só podem ser retirados na ordem inversa da ordem em quem foram inseridos.

O método de funcionamento de uma pilha, em que só podem ser retirados os elementos na ordem inversa da ordem na qual foram inseridos, também é conhecido como:

- a. FIFO.
- b. FILO.
- c. FOLO.
- d. FILI.
- e. FOLI.

2. Segundo Tenenbaum, Langsam e Augenstein (2007), uma pilha tem como definição básica ser um conjunto de elementos ordenados que permite a inserção e a remoção de mais elementos em apenas uma das extremidades da estrutura denominada topo da pilha.

Com referência a pilhas, analise as sentenças a seguir:

- I. É a estrutura mais simples e utilizada dentro da estrutura de dados.
- II. O único elemento que pode ser removido é o elemento que está embaixo da pilha.
- III. Uma pilha é um objeto dinâmico e está em constantes mudanças.
- IV. Também conhecido como FIFO (*first in, first out*).
- V. A estrutura de dados pode conter dois elementos apenas, um vetor e uma variável do tipo inteiro.

Assinale a alternativa que contém as sentenças corretas:

- a. I, II e III apenas.
- b. II, IV e V apenas.
- c. I, III e V apenas.
- d. II, III e IV apenas.
- e. I, IV e V apenas.

3. Em uma estrutura de pilha devem ser implementadas duas operações básicas: empilhar um novo elemento e desempilhar um elemento. Conforme Tenenbaum, Langsam e Augenstein (2007), uma pilha apresenta uma estrutura que pode ser declarada contendo dois objetos: um ponteiro e um valor inteiro para indicar a posição do topo da pilha.

Com base nessa afirmativa, analise as sentenças a seguir:

- I. Criar o vetor para armazenamento dos elementos.
- II. Criar a declaração da estrutura da pilha.
- III. Criar a pilha com a alocação dinâmica.
- IV. Criar a função principal *Main*.
- V. Criar as funções para inserir e remover da pilha.

Assinale a alternativa que contém as sentenças utilizadas na declaração da estrutura inicial para criação de uma pilha:

- a. I, II e III apenas.
- b. I, III e IV apenas.
- c. II, IV e V apenas.
- d. II, III e V apenas.
- e. I, IV e V apenas.

Seção 3

Filas

Diálogo aberto

A estrutura de fila é amplamente usada no mundo real e é muito importante na organização dos elementos que a compõem. Empresas estão aderindo cada vez mais ao painel de senhas para a organização de sua fila de atendimento: você retira uma senha e é chamado pelo painel para atendimento, fazendo com que a fila seja organizada, mas sem a necessidade de as pessoas ficarem na mesma posição em que chegaram.

Para darmos prosseguimento em nossos estudos nesta seção, você vai conhecer e aprender sobre as operações de inserção, remoção e verificação se uma fila está vazia ou não.

Para a solução de desafios referentes a filas, você vai conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas.

Assim, você será capaz de solucionar problemas relacionados a filas, muito comuns em nosso dia a dia, como supermercados, bancos e em pedágios nas estradas, criando soluções que possam ser utilizadas para redução de filas ou até mesmo um melhor aproveitamento da utilização dessa estrutura.

Como é de seu conhecimento, você foi direcionado pela empresa de TI na qual trabalha para realizar uma consultoria, a fim de ajudá-lo na criação de um sistema para uma empresa de terceirização de serviços em montagem de notebooks, onde será feita a montagem e, após a conclusão, os testes nos equipamentos.

Como foi concluída a solução para o teste do notebook e a troca de peças com defeito, agora surgiu a demanda de essas peças de reposição serem novamente enviadas para a montagem.

Os produtos que apresentam defeitos seguem para uma nova linha de produção, entrando na fila para a reposição da peça com defeito e, após a peça ser repostada, este produto deve sair da fila. Utilizando as operações sobre filas, o sistema deve informar se existem novos produtos com defeitos ou se a fila está vazia e se esta fila trabalha de forma circular.

Como você é o responsável pela consultoria e dispõe de grande conhecimento neste tipo de sistema, seu desafio é solucionar o problema do seu cliente e entregar um relatório sobre a resolução do desafio para a empresa. Preparado?

Não pode faltar

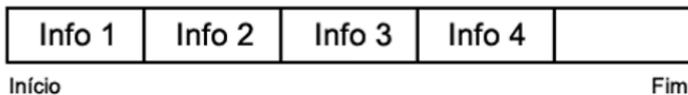
Definição e elementos de filas

Segundo Tenenbaum, Langsam e Augenstein (2007), uma fila é a representação de um conjunto de elementos. Podemos remover esses elementos deste conjunto por uma extremidade chamada de *início da fila*, e pela outra extremidade, na qual são inseridos os elementos, chamada de *final da fila*.

Assim como uma pilha, as filas também são estruturas dinâmicas com tamanho variável, podendo aumentar ou diminuir conforme são inseridos ou removidos elementos da fila.

Com certeza você já enfrentou uma fila no supermercado, em uma festa para comprar fichas de alimentação ou em bancos, para pagamento de uma fatura. As filas em programação são exatamente dessa forma: uma representação dessas filas do mundo real, criadas no mundo computacional, conforme podemos observar na Figura 4.26.

Figura 4.26 | Exemplo de uma estrutura de dados do tipo fila vazia



Fonte: elaborada pelo autor.

Exemplificando

Podemos utilizar como exemplo um aeroporto, onde passam diversas pessoas que viajam para todos os lugares do mundo. Mas antes de viajar, é necessário despachar as malas e realizar o check-in nos guichês. Para realizar tal atividade, você entra no final da fila e a primeira pessoa que entrou na fila está em outra ponta, na saída dela, para a realização do check-in. Esse processo é repetido a todo instante, e os elementos estão sempre sendo inseridos ou retirados dessa fila, como o exemplo da Figura 4.27.

Figura 4.27 | Fila em guichê de aeroporto



Fonte: Shutterstock.

Para conhecermos os elementos que estão na fila, precisamos percorrer a fila toda. Assim, para criarmos uma fila é necessário:

- Um vetor dinâmico.
- Uma variável do tipo inteiro para o início.
- Uma variável do tipo inteiro para o fim.

Regras para operação de filas

Podemos imaginar a fila em uma agência dos Correios. Diferentemente da fila de um aeroporto para despacho de malas, a fila dos Correios é realizada com base em senhas, em que um cliente adquire uma senha na entrada e aguarda ser chamado por ela no painel, dirigindo-se ao guichê.

Assimile

A utilização da estrutura de dados do tipo fila difere de uma estrutura de pilha na base de sua operação, como um FIFO (*first in, first out*, ou seja, o primeiro que entra é o primeiro que sai). Assim, adicionamos novos elementos ao fim da fila e removemos sempre os do princípio da fila. Poderíamos, então, inserir um novo elemento no meio da fila?

Segundo Silva (2007), em uma fila os elementos entram por uma extremidade e são removidos pela outra extremidade. Isso é conhecido como FIFO (*first in, first out*, ou seja, o primeiro que entra é o primeiro a sair).

No caso desta fila, sabemos quais os elementos com base em seu número de índice, que são as senhas sequenciais. Assim, a fila apresenta sua ordem de

entrada (fim da fila) e sua ordem de saída dos elementos (início da fila), como podemos observar na Figura 4.28.

Figura 4.28 | Entrada de elemento pelo final da fila

Info 1				
Início				Fim
Info 1	Info 2			
Início				Fim
Info 1	Info 2	Info 3		
Início				Fim
Info 1	Info 2	Info 3	Info 4	
Início				Fim

Fonte: elaborada pelo autor.

E a remoção dos elementos dá-se pela extremidade contrária, como Figura 4.29:

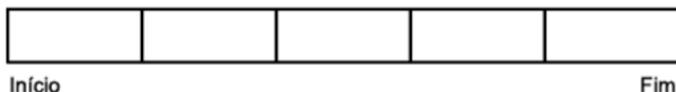
Figura 4.29 | Saída de elemento pelo início da fila

Info 1	Info 2	Info 3	Info 4	
Início				Fim
Info 2	Info 3	Info 4		
Início				Fim
Info 3	Info 4			
Início				Fim
Info 4				
Início				Fim

Fonte: elaborada pelo autor.

Uma fila pode estar no estado de vazia quando não contiver nenhum elemento, conforme na Figura 4.30.

Figura 4.30 | Fila vazia



Segundo Celes, Cerqueira e Rangel (2004), os passos para criação de uma fila são:

- Criar uma fila vazia.
- Inserir elemento no final.
- Retirar um elemento do início.
- Verificar se a fila está vazia.
- Liberar a fila.

Inserir elementos na fila

Segundo Drozdek (2016), a estrutura de dados de fila apresenta operações similares às da estrutura de pilha para gerenciamento de uma fila, como:

- Cria uma fila vazia.
- Insere um elemento no fim da fila.
- Remove o elemento do início da fila.
- Verifica se a fila está vazia.
- Libera a fila.

Conforme Celes, Cerqueira e Rangel (2004), podemos simplesmente utilizar um vetor para armazenar os elementos e implementarmos uma fila nessa estrutura de dados, ou podemos utilizar uma alocação dinâmica de memória para armazenar esses elementos.

Para iniciar uma fila é necessário definir o tipo da estrutura a ser utilizada. Essa definição da estrutura pode ser dada pela implementação do trecho a seguir:

```
#define N 100
struct fila {
    int n;
    int ini;
    char vet[N];
};
```

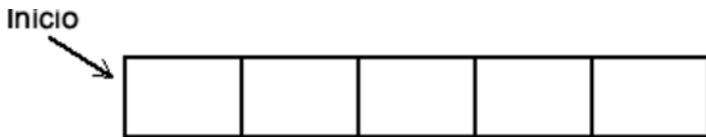
```
typedef struct fila Fila;
```

Após a definição da estrutura, é preciso inicializar a fila, para que possa receber os elementos e, utilizando a alocação dinâmica, ser implementado o trecho de código a seguir para a inicialização da fila como vazia:

```
Fila* inicia_fila (void){  
    Fila* f = (Fila*) malloc(sizeof(Fila));  
    f -> n = 0;  
    f -> ini = 0;  
    return f;  
}
```

Podemos observar na Figura 4.31 a estrutura da fila criada e vazia.

Figura 4.31 | Estrutura de fila inicializada e vazia



Fonte: elaborada pelo autor.

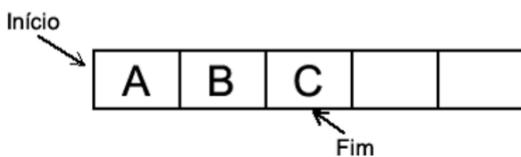
Já com a fila definida e inicializada, podemos implementar o trecho de código responsável pela inserção de elementos na estrutura criada:

```
void insere_fila (Fila* f, char elem) {  
    int fim;  
    if (f -> n == N) {  
        printf("A fila está cheia.\n");  
        exit(1);  
    }  
    fim = (f -> ini + f -> n) % N;  
    f -> vet[fim] = elem;  
    f -> n++;  
}
```

Na função apresentada, primeiramente passamos o ponteiro da fila e o elemento para ser inserido como parâmetros. Na estrutura de condição *IFI*, é preciso verificar se a posição do último elemento da fila é igual ao seu tamanho total. Em caso positivo, é apresentada a mensagem de que a fila está cheia e executar a saída da função.

Caso a fila não esteja completa, a função continua verificando a posição final da fila para receber o elemento a ser inserido, e incrementamos nela a quantidade de elementos. Na Figura 4.32, temos a representação da fila como elementos inseridos.

Figura 4.32 | Fila com elementos inseridos



Fonte: elaborada pelo autor.

Remover elementos da fila

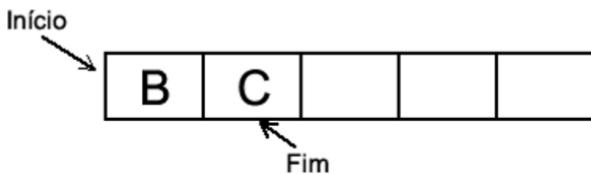
Como você já aprendeu, em uma fila só é possível remover um elemento pelo seu início. Podemos implementar o trecho de código a seguir para remoção do elemento e apresentar seu valor no retorno da função:

```
float remove_fila (Fila* f) {  
    char elem;  
    if (fila_vazia(f)) {  
        printf("A Fila esta vazia\n");  
        exit(1);  
    }  
    elem = f -> vet[f -> ini];  
    f -> ini = (f -> ini + 1) % N;  
    f -> n--;  
    return elem;  
}
```

Neste trecho, podemos observar que, antes de removermos o elemento da fila, precisamos verificar se ela possui elementos, chamando a função *fila_vazia*. Caso a fila esteja vazia, a função apresenta uma mensagem informando ao usuário de que não há elementos para serem removidos e finaliza a função.

Caso a fila possua elementos, a variável *elem* recebe o elemento da primeira posição da fila, e o início da fila passa para o próximo elemento, decrementando a quantidade de elementos e retornando o elemento removido. Na Figura 4.33 podemos visualizar que o elemento A foi removido e o elemento B passou a ser o primeiro elemento da fila:

Figura 4.33 | Fila com primeiro elemento removido



Fonte: elaborada pelo autor.

Informar se a fila está vazia

Outra função que podemos implementar na fila é verificar se ela está vazia ou não. No trecho de remoção de um elemento da fila já realizamos a chamada a esta função, e ela pode ser implementada com o trecho de código a seguir:

```
int fila_vazia (Fila* f) {  
    return (f -> n == 0);  
}
```

Nele, a função verifica se a quantidade de elementos da fila é igual a 0 (zero); caso seja positivo, a função retorna verdadeiro, caso contrário, retorna falso. Podemos chamar esta função sempre que precisarmos verificar se a fila está vazia ou em alguma funcionalidade em que ela não pode estar vazia para executar alguma rotina específica em um sistema.

Podemos utilizar uma função importante para podermos liberar a alocação de memória após o uso da fila e, assim, limparmos as variáveis utilizadas pelo sistema.

A implementação do trecho para liberação da alocação de memória pode ser dado por:

```
void libera_fila(Fila* f) {  
    free(f);  
}
```

No trecho dado, a chamada da função *free()* libera a memória.

Refletia

É muito importante realizarmos a liberação da alocação de memória do nosso sistema. Você saberia responder quais os problemas que podem vir a ocorrer em um sistema operacional se não for liberada a alocação dinâmica?

Filas circulares

Segundo Silva (2007), as filas não apresentam uma solução completa e, chegando ao final do vetor, poderemos ter a fila cheia mesmo que não esteja, uma vez que elementos podem ter sido removidos. Para isso, podemos utilizar as filas circulares como solução para esta situação.

Uma fila circular utiliza menos instruções a serem executadas, podendo ser mais adequada e eficiente na utilização de programação. Diferentemente da estrutura de fila, a fila circular apresenta a seguinte definição em sua implementação, quanto às variáveis:

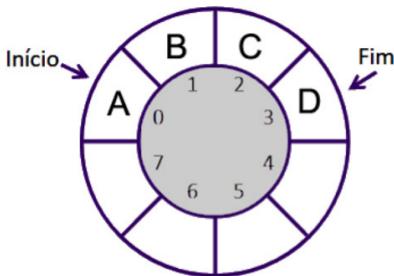
- Um vetor para os elementos.
- Um valor inteiro para o tamanho da fila.
- Um valor inteiro para o início da fila.
- Um valor inteiro para o fim da fila.

Exemplificando

Acredito que você já deve ter frequentando algum *drive-thru* (fast-food em que o cliente não sai do carro), correto? Ao entrar na fila com seu carro, a atendente anota seu pedido e a placa do seu carro, na próxima cabine, você realiza o pagamento e, por fim, você retira seus produtos na última cabine. Este processo todo se encaixa no modelo de fila circular, em que a numeração do pedido chega a um número limite, como 999, por exemplo, e o próximo volta a ser o 000.

Conforme Drozdek (2016), em uma fila circular o conceito de circularidade se baseia no fato de o último elemento da fila estar na última posição do vetor, adjacente à primeira. Assim, são os ponteiros, e não os elementos da fila, que se movem em direção ao início do vetor, como podemos observar na Figura 4.34:

Figura 4.34 | Estrutura da fila circular



Fonte: elaborada pelo autor.

Para implementar uma estrutura de fila circular, podemos utilizar como exemplo o código a seguir:

```
/* Vamos definir a constante N com valor de 10 */
#define N 10

struct filacirc { /* Criação da estrutura da Fila Circular */

    int tam, ini, fim;
    char vet[N];
};

typedef struct filacirc FilaCirc;

/* Função para inicializar a Fila */
void inicia_fila (FilaCirc *f){

    f -> tam = 0;
    f -> ini = 1;
    f -> fim = 0;
}
```

```

/* Função para inserir na Fila */
void insere_fila (FilaCirc* f, char elem) {
    if (f -> tam == N - 1){ /* Verifica se a Fila
está completa */
        printf("A fila esta cheia\n");
    } else { /* Caso a Fila não esteja completa,
inserimos o elemento */
        f -> fim = (f -> fim % (N - 1)) + 1;
        f -> vet[f -> fim] = elem;
        f -> tam++;
    }
}

int fila_vazia (FilaCirc* f) {
    return (f -> tam == 0); /* Retorna verdadeiro
se a Fila estiver vazia */
}

char remove_fila (FilaCirc* f) {
    if (fila_vazia(f)){ /* Verifica se a Fila es
tá vazia */
        printf("Fila vazia\n");
    } else { /* Caso a Fila contenha elemento, é
removido o primeiro */
        f -> ini = (f -> ini % (N-1)) + 1;
        f -> tam--;
    }
}

```

Considerando a utilização de uma fila, alguns problemas podem surgir com sua utilização. Se utilizarmos um vetor, teremos o problema de possuir um armazenamento de tamanho fixo e limitado, enquanto a fila pode crescer

com a necessidade de uso do sistema. Para resolver essa problemática, teríamos que limitar o tamanho máximo da fila ao tamanho do vetor.

Outra situação que pode ocorrer é adicionar um elemento em uma fila cheia ou remover um elemento de uma fila vazia. Em ambos os casos seria impossível realizar as operações. Como solução, é importante sempre implementar as funções para verificar se a fila está cheia (*fila_cheia(F)*) e para verificar se ela está vazia (*fila_vazia(F)*).

Podem surgir problemas relacionados aos controles de início e fim de fila, em que não é possível identificar as posições em que se encontram. Como solução, é preciso incluir duas variáveis (*inicio* e *fim*) para armazenar a posição do início e do fim da fila, e sempre atualizar esses valores conforme a fila aumenta ou diminui.

Com essas informações, será possível criar novos algoritmos muito bem estruturados e que serão utilizados para soluções de novos problemas.

Agora é com você!

Sem medo de errar

Retomando nosso desafio, você foi direcionado para realizar uma consultoria à empresa de terceirização de montagem de notebooks, a fim de ajudá-la na criação do sistema para a este serviço, em que será feita a montagem e, após a conclusão, os testes nos equipamentos.

Concluída a solução para o teste do notebook e troca de peças com defeito, você precisa solucionar agora a demanda de as peças de reposição serem novamente enviadas para a montagem.

Os produtos que apresentam defeitos seguem para uma nova linha de produção, entrando na fila para a reposição da peça com defeito. Após a peça ser repostada, este produto deve sair da fila. Utilizando as operações sobre filas, o sistema deve informar se existem novos produtos com defeitos ou se a fila está vazia e se esta fila trabalha de forma circular.

Assim, vamos pesquisar mais sobre como adicionar ou remover peças na linha de produção para atender à demanda das trocas realizadas por defeito.

Como grande conhecedor e responsável pela consultoria neste tipo de sistema, seu desafio é solucionar o problema do seu cliente.

A fim de que o problema seja resolvido, o sistema vai precisar verificar as peças com defeitos e descartá-las. Os notebooks que apresentam alguma peça com defeito seguirão um fluxo de filas circulares, pois com seu retorno às esteiras de produção precisaram ser remontados.

Com isso, os notebooks que estão na fila de produção serão montados normalmente e aqueles cujas peças precisaram ser trocadas, entraram no final desta fila de processo de montagem.

A remoção dos notebooks da fila de produção será realizada somente ao final de todo o processo de montagem.

É importante sempre verificar se a fila de produção não está vazia, a fim de que a esteira não fique ociosa e sem produção.

Avançando na prática

Fila de processos em um sistema operacional

Devido ao seu grande conhecimento em processos, você foi contratado por uma grande empresa de desenvolvimento de sistemas para realizar a implementação de uma fila de processos em um sistema operacional. Essa fila terá um tempo para utilizar cada um dos seus processos. No caso de um processo estar sendo executado e seu limite de tempo se encerrar, ele é removido e colocado na fila novamente; assim, o próximo processo passa a ser executado, e assim por diante, até que todos os processos tenham sido executados.

Sua função é analisar e implementar um algoritmo de uma fila circular de processos, a fim de solucionar o problema de requisições nos processos em execução, com base no tempo limite para cada processo ser executado, e garantir o funcionamento do sistema operacional da empresa.

Seu desafio será entregar um relatório com a análise realizada e o algoritmo de como implementar essa solução junto ao seu cliente.

Resolução da situação-problema

Para a resolução deste desafio, você vai precisar pesquisar como declarar e utilizar as funções de horas (*Time*) dentro da linguagem C++.

É importante realizar um fluxograma para entender como pode funcionar a fila circular para resolver o problema de processos e, a partir desse fluxograma, criar o algoritmo para a execução da solução.

A implementação do algoritmo a seguir é uma das formas possíveis de solucionar o desafio proposto, em que é verificado o tempo limite de cada

processo, e assim que o tempo se esgotar o sistema removerá o processo da fila e passar para o próximo processo. Com base neste algoritmo:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* Declaração das funções de
horas */
#define N 10
struct filacirc {
    int tam, ini, fim;
    int vet[N];
};
typedef struct filacirc FilaCirc;

void inicia_fila (FilaCirc *f) {
    f -> tam = 0;
    f -> ini = 1;
    f -> fim = 0;
}
void insere_fila (FilaCirc* f, char elem) {
    if (f -> tam == N - 1){
        printf("A fila está cheia\n");
    } else {
        f -> fim = (f -> fim % (N - 1)) + 1;
        f -> vet[f -> fim] = elem;
        f -> tam++;
    }
}
int fila_vazia (FilaCirc* f) {
    return (f -> tam == 0);
```

```

}

int remove_fila (FilaCirc* f){

    if (fila_vazia(f)){
        printf("Fila vazia\n");
    } else {

        f -> ini = (f -> ini % (N-1)) + 1;
        f -> tam--;
    }

}

int main (){

    FilaCirc* f;

    char processo[20];

    int tempo, tmpGasto;

        clock_t tInicio, tFim; /* Declaração
de variável do tipo hora */

        printf("\n Informe o tempo do processo
em execução: \n");

        scanf("%d", &tempo);

        tInicio = clock(); /* Inicia o relógio */

        while (f -> tam < N - 1){

            insere_fila(f, processo);

        }

        while (f -> tam <= N - 1){

            tFim = clock(); /* Finaliza o relógio */

            tmpGasto = ((int) (tFim -
tInicio)); /* Calcula o tempo gasto */

            if (tempo <= tmpGasto){ /* Se o tempo
for menor ou igual ao tempo gasto, remove da fila */
}
}

```

```

        remove_fila(f);

    } else {

        printf("Processando...");

    }

    system("Pause");
}

}

```

Faça valer a pena

1. Uma _____ é a representação de um conjunto de elementos no qual podemos remover elementos por _____ chamada de início da _____, e pela outra extremidade, chamada de _____, são inseridos os elementos.

Assinale a alternativa que contém as palavras que completam a sentença anterior:

- a. lista; um vetor; fila; topo da lista.
- b. pilha; uma extremidade; pilha; final da pilha.
- c. fila; um vetor; fila; final da lista.
- d. pilha; um vetor; fila; final da fila.
- e. fila; uma extremidade; fila; final da fila.

2. A estrutura de fila é do tipo FIFO (*first in, first out*), ou seja, o primeiro elemento que entra na fila é o primeiro elemento a ser removido; assim, todo primeiro elemento que entra na fila por uma extremidade sairá primeiro pela outra extremidade.

Considerando a estrutura de dados do tipo fila, assinale a alternativa que apresenta o exemplo de uso das filas.

- a. Diversos pratos um sobre o outro.
- b. Anotações de tarefas a realizar.
- c. Convidados de casamento.
- d. Carros parados em um pedágio.

e. Torre de Hanoi.

- 3.** As filas não apresentam uma solução completa. Ao final do vetor, poderemos ter a fila cheia, mesmo que ela não esteja, uma vez que elementos podem ter sido removidos. Podemos utilizar as filas circulares como solução para essa situação.

Com base na implementação das definições de fila e fila circular, assinale a alternativa que apresenta a principal diferente entre as estruturas:

- a. A declaração de uma variável para o fim da fila circular.
- b. A fila não permite o uso de alocação dinâmica.
- c. A fila circular utiliza mais instruções na programação do que a fila.
- d. A fila utiliza uma variável de controle boolean na estrutura.
- e. A fila circular não dispõe de uma programação adequada e eficiente.

Referências

AULA 32 – Criando e destruindo uma Fila Estática. [S.l.: s.n.], 7 nov. 2013. (6 min. 9 seg). Publicado pelo canal Linguagem C Programação Descomplicada. Disponível em: <https://www.youtube.com/watch?v=y93DzmBskGQ>. Acesso em: 18 mar. 2020.

CELES, W.; CERQUEIRA, R.; RANGEL, J. L. **Introdução à estrutura de dados:** com técnicas de programação em C. 6. reimpr. Rio de Janeiro: Elsevier, 2004.

DROZDEK, A. **Estrutura de dados e algoritmos em C++.** Trad. 4. ed. norte-americana. São Paulo: Cengage Learning, 2016.

ESTRUTURA de dados – Pilha em C#. [S.l.: s.n.], 28 mar. 2014. (9 min 18 seg). Publicado pelo canal Sthefane Soares – Vida Programação. Disponível em: https://www.youtube.com/watch?v=nHF_pgep9qE. Acesso em: 18 mar. 2020.

FORBELLONE, A. L. V. **Lógica de programação:** a construção de algoritmos e estrutura de dados. 3. ed. Prentice Hall: São Paulo, 2005.

LORENZI, F., MATTOS, P. N.; CARVALHO, T. P. **Estruturas de dados.** São Paulo: Cengage Learning, 2015.

SILVA, O. Q. **Estrutura de dados e algoritmos usando C:** fundamentos e aplicações. Rio de Janeiro: Ciência Moderna, 2007.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de dados usando C.** Pearson Prentice Hall: São Paulo, 2007.

VELOSO, P. A. S. **Estruturas de dados.** Rio de Janeiro: Campus-Elsevier, 1996.

ISBN 978-85-522-1682-7

A standard linear barcode representing the ISBN number 978-85-522-1682-7.

9 788552 216827 >