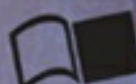


Algoritmos e programação em linguagem C

Renato Soffner



Editora
Saraiva

www.saraivauni.com.br



ALGORITMOS E PROGRAMAÇÃO EM LINGUAGEM C

www.saraivauni.com.br

ALGORITMOS E PROGRAMAÇÃO EM LINGUAGEM C

Renato Soffner



Rua Henrique Schaumann, 270

Pinheiros – São Paulo – SP – CEP: 05413-010

Fone PABX: (11) 3613-3000 • Fax: (11) 3611-3308

Televendas: (11) 3613-3344 • Fax vendas: (11) 3268-3268

Site: <http://www.saraivauni.com.br>

Filiais

AMAZONAS / RONDÔNIA / RORAIMA / ACRE

Rua Costa Azevedo, 56 – Centro

Fone/Fax: (92) 3633-4227 / 3633-4782 – Manaus

BAHIA / SERGIPE

Rua Agripino Dórea, 23 – Brotas

Fone: (71) 3381-5854 / 3381-5895 / 3381-0959 – Salvador

BAURU / SÃO PAULO (sala dos professores)

Rua Monsenhor Claro, 2-55/2-57 – Centro

Fone: (14) 3234-5643 – 3234-7401 – Bauru

CAMPINAS / SÃO PAULO (sala dos professores)

Rua Camargo Pimentel, 660 – Jd. Guanabara

Fone: (19) 3243-8004 / 3243-8259 – Campinas

CEARÁ / PIAUÍ / MARANHÃO

Av. Filomeno Gomes, 670 – Jacarecanga

Fone: (85) 3238-2323 / 3238-1331 – Fortaleza

DISTRITO FEDERAL

SIA/SUL Trecho 2, Lote 850 – Setor de Indústria e Abastecimento

Fone: (61) 3344-2920 / 3344-2951 / 3344-1709 – Brasília

GOIÁS / TOCANTINS

Av. Independência, 5330 – Setor Aeroporto

Fone: (62) 3225-2882 / 3212-2806 / 3224-3016 – Goiânia

MATO GROSSO DO SUL / MATO GROSSO

Rua 14 de Julho, 3148 – Centro

Fone: (67) 3382-3682 / 3382-0112 – Campo Grande

MINAS GERAIS

Rua Além Paraíba, 449 – Lagoinha

Fone: (31) 3429-8300 – Belo Horizonte

PARÁ / AMAPÁ

Travessa Apinagés, 186 – Batista Campos

Fone: (91) 3222-9034 / 3224-9038 / 3241-0499 – Belém

PARANÁ / SANTA CATARINA

Rua Conselheiro Laurindo, 2895 – Prado Velho

Fone: (41) 3332-4894 – Curitiba

PERNAMBUCO / ALAGOAS / PARAÍBA / R. G. DO NORTE

Rua Corredor do Bispo, 185 – Boa Vista

Fone: (81) 3421-4246 / 3421-4510 – Recife

RIBEIRÃO PRETO / SÃO PAULO
Av. Francisco Junqueira, 1255 – Centro
Fone: (16) 3610-5843 / 3610-8284 – Ribeirão Preto

RIO DE JANEIRO / ESPÍRITO SANTO
Rua Visconde de Santa Isabel, 113 a 119 – Vila Isabel
Fone: (21) 2577-9494 / 2577-8867 / 2577-9565 – Rio de Janeiro

RIO GRANDE DO SUL
Av. A. J. Renner, 231 – Farrapos
Fone: (51) 3371-4001 / 3371-1467 / 3371-1567 – Porto Alegre

SÃO JOSÉ DO RIO PRETO / SÃO PAULO (sala dos professores)
Av. Brig. Faria Lima, 6363 – Rio Preto Shopping Center – V. São José
Fone: (17) 3227-3819 / 3227-0982 / 3227-5249 – São José do Rio Preto

SÃO JOSÉ DOS CAMPOS / SÃO PAULO (sala dos professores)
Rua Santa Luzia, 106 – Jd. Santa Madalena
Fone: (12) 3921-0732 – São José dos Campos

SÃO PAULO
Av. Antártica, 92 – Barra Funda
Fone PABX: (11) 3613-3666 – São Paulo

304.790.001.001

ISBN 9788502207523

CIP-BRASIL. CATALOGAÇÃO NA FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ.

S664a
Soffner, Renato
Algoritmos e Programação em linguagem C / Renato Soffner. – 1. ed. – São Paulo: Saraiva, 2013.
200 p.; 27 cm

ISBN 9788502207523

1. Informática. 2. C (Linguagem de programação de computador). 3. Programação (Computadores). I. Título.
13-02309.

CDD: 005.1
CDU: 004.41

Copyright © Renato Soffner
2013 Editora Saraiva
Todos os direitos reservados.

| | |
|-----------------------|-------------------------|
| Direção editorial | Flávia Alves Bravin |
| Coordenação editorial | Rita de Cássia da Silva |
| Aquisições | Ana Paula Matos |

Editorial Universitário

Luciana Cruz
Patricia Quero

Editorial Técnico

Alessandra Borges

Editorial de Negócios

Gisele Folha Mós

Produção editorial

Daniela Nogueira Secondo
Rosana Peroni Fazolari

Produção digital

Nathalia Setrini Luiz

Suporte editorial

Najla Cruz Silva

Arte e produção

Negrito Produção Editorial

Capa

Weber Amendola

Produção gráfica

Liliane Cristina Gomes

Contato com o editorial

editorialuniversitario@editorasaraiva.com.br



Nenhuma parte desta publicação poderá ser reproduzida por qualquer meio ou forma sem a prévia autorização da Editora Saraiva. A violação dos direitos autorais é crime estabelecido na lei nº 9.610/98 e punido pelo artigo 184 do Código Penal.

Para Lourdes, Júlia e Ricardo
(e Elke, Dalilah, Dolly, Menina, Molly e Bella...)

Nihil sub sole novum. Si de quadam re dicitur: “ecce hoc novum est”, iam enim praecesit in saeculis, quae fuerent ante nos.

ECLESIASTES 1, 10.

The trouble with me mentally is that I do not know how to write a book. But this is better than thinking I do, when actually I do not.

VANNEVAR BUSH

Carta a James B. Conant

I heartly beg that what I have here done may be read with candour and that the defects in a subject so difficult be not so much reprehended as kindly supplied, and investigated by new endeavours of my readers.

SIR ISAAC NEWTON

Introdução aos *Philosophiae Naturalis Principia Mathematica*

SOBRE O AUTOR

O professor Renato Soffner é engenheiro agrônomo e mestre na aplicação de métodos quantitativos a sistemas mecanizados pela Universidade de São Paulo (USP); especializou-se em ciências da computação e doutorou-se na área de tecnologia e informática aplicadas à educação, na Universidade Estadual de Campinas (UNICAMP); foi pesquisador associado em um programa de pós-doutorado no Massachusetts Institute of Technology (MIT) – Media Laboratory.

Leciona as disciplinas de Lógica e Algoritmos, Programação, Estrutura de Dados, Sistemas Operacionais e Interação Humano-Computador na FATEC de Americana, do Centro Estadual de Educação Tecnológica Paula Souza – CEETEPS e atua como pesquisador no Mestrado em Educação do Centro Universitário Salesiano de São Paulo (UNISAL), na linha de pesquisa em tecnologias sociais educativas.

APRESENTAÇÃO

Tenho, nesta obra, o objetivo maior de trabalhar os conceitos, princípios, técnicas e ferramentas utilizados na programação de computadores dentro da abordagem estruturada e, em especial, o desenvolvimento de algoritmos e sua implementação utilizando uma linguagem procedural (a Linguagem C). Pretendo também capacitar interessados em desenvolver algoritmos e programas de manipulação de estruturas de dados básicas (vetores, matrizes, registros e listas), além dos algoritmos e técnicas de ordenação e busca.

Tentamos apresentar ao leitor uma linguagem acessível, que estabelecerá um vínculo de confiança na obra; como docente da área, tenho consciência do que um aluno de programação de computadores precisa durante o processo de desenvolvimento de autoconfiança suficiente para o domínio dos conteúdos, que são, em geral, complexos para a maioria das pessoas.

Nosso desafio é, portanto, mediar a aprendizagem de um assunto tido como *difícil* pela maioria das pessoas.

Que o leitor assuma a proposta de se empenhar nos exercícios propostos, em consonância com os objetivos instrucionais da obra e com a meta de contornar a dificuldade inicial de aprendizagem da programação de computadores, anteriormente citada.

Como na tarefa de se aprender um novo idioma – ou mesmo aprender a nadar, ou andar de bicicleta –, estudar programação demanda exercício, exercício, exercício...

Este livro é resultado de notas de aula acumuladas durante anos de exercício profissional e docente do autor, as quais sofreram ajustes contínuos dos pontos de vista didático e pedagógico. Esperamos que a obra possa suprir a demanda dos docentes de cursos introdutórios de programação do nível técnico, tecnológico e de bacharelado.

Sumário

CAPÍTULO 1 Lógica de programação e algoritmos ✓

- 1.1 Qual é o propósito da lógica? ✓
- 1.2 Conectivos e tabelas-verdade ✓
- 1.3 Lógica de programação ✓
- 1.4 Algoritmos ✓
- 1.5 Programação estruturada ✓
 - Exercícios ✓
 - Referências bibliográficas ✓
 - Anexo: IBM 1130 ✓

CAPÍTULO 2 Tipos primitivos, variáveis, constantes e operadores – estrutura geral de um programa

- 2.1 Tipos de dados primitivos ✓
- 2.2 Variáveis ✓
- 2.3 Constantes ✓
- 2.4 Operadores aritméticos ✓
- 2.5 Operadores relacionais ✓
- 2.6 Operadores lógicos ✓
- 2.7 Atribuição de valor ✓
- 2.8 Estrutura geral dos programas em Linguagem C ✓
- 2.9 Primeiro programa em Linguagem C ✓
 - Exercícios ✓
 - Referências bibliográficas ✓
 - Anexo: Paradigmas de linguagens de programação ✓

CAPÍTULO 3 Estruturas de controle de programação – decisão

- 3.1 Algoritmos de tomada de decisão
- 3.2 Algoritmos dos vários tipos de decisão
 - 3.2.1 *Decisão simples*
 - 3.2.2 *Decisão composta*
 - 3.2.3 *Decisão múltipla*
- 3.3 Estruturas de decisão encadeada: heterogênea e homogênea
 - 3.3.1 *Decisão encadeada heterogênea*
 - 3.3.2 *Decisão encadeada homogênea*
- Exercícios
- 3.4 Exemplos de programas completos

3.4.1 *Ilustração do conteúdo – exercício desafio*

Referências bibliográficas

Anexo: A Linguagem de Programação C

CAPÍTULO 4 Estruturas de controle de programação – repetição

4.1 Estruturas de repetição com teste no início

4.2 Estruturas de repetição com teste no final

4.3 Estruturas de repetição com variável de controle

Exercícios

Referências bibliográficas

Anexo: De Hackers e Nerds

CAPÍTULO 5 Variáveis indexadas: vetores e matrizes

5.1 Vetores e *strings*

5.1.1 *Passagem de vetores para funções*

5.1.2 *Strings*

5.2 Matrizes

Exercícios

Referências bibliográficas

Anexo: A Revolução da Internet (?)

CAPÍTULO 6 Funções (modularização)

6.1 Conceituando funções

6.2 Chamada por valor e chamada por referência

6.3 Argumentos da linha de comando

6.4 Recursividade

Exercícios

Referências bibliográficas

Anexo: Recursividade e humor

CAPÍTULO 7 Estruturas (registros)

7.1 Conceituando estruturas (ou registros)

7.2 Carga inicial automática da estrutura

7.3 Acessando os membros da estrutura

7.4 Vetores de estruturas

7.5 Definição de tipos (*typedef*)

7.6 Passagem de estruturas para funções

Exercícios

Referências bibliográficas

Anexo: Pensamento criativo

CAPÍTULO 8 Ponteiros e alocação dinâmica de memória

8.1 Ponteiros

- 8.1.1 *Ponteiros e parâmetros de funções*
- 8.1.2 *Aritmética de ponteiros*
- 8.1.3 *Ponteiros de ponteiros*
- 8.1.4 *Ponteiros e vetores*
- 8.1.5 *Ponteiros de strings*
- 8.1.6 *Ponteiros void*
- 8.1.7 *Manipulando estruturas com ponteiros*
- 8.1.8 *Eficiência dos ponteiros na Linguagem C*
- 8.1.9 *Vetores de ponteiros*
- 8.2 *Alocação dinâmica de memória*
- 8.3 *Alteração do tamanho da memória alocada*
- Exercícios
- Referências bibliográficas
- Anexo: O poder das redes

CAPÍTULO 9 Listas, pilhas e filas

- 9.1 *Listas ligadas (ou encadeadas)*
- 9.2 *Filas*
- 9.3 *Pilhas*
- Exercícios
- Referências bibliográficas
- Anexo: Gödel e a existência de Deus

CAPÍTULO 10 Arquivos

- 10.1 *Operações básicas sobre arquivos*
- 10.2 *Funções de manipulação de arquivos*
- 10.3 *Trabalhando com arquivos*
 - 10.3.1 *Abrindo arquivos*
 - 10.3.2 *Modos de abertura*
 - 10.3.3 *Fechando arquivos*
 - 10.3.4 *Escrevendo no arquivo e apagando o que existia*
 - 10.3.5 *Escrevendo no arquivo mas mantendo o que existia*
 - 10.3.6 *Escrevendo strings no arquivo*
 - 10.3.7 *Lendo o conteúdo do arquivo*
 - 10.3.8 *Solicitando o nome do arquivo*
 - 10.3.9 *Lendo e escrevendo no arquivo caractere a caractere*
 - 10.3.10 *Copiando um arquivo*
 - 10.3.11 *Entrada e saída formatadas*
 - 10.3.12 *Buscando determinada posição do arquivo e apontando para o seu início*
 - 10.3.13 *Manipulando erros*
 - 10.3.14 *Apagando um arquivo*
 - 10.3.15 *Escrevendo e lendo tipos de dados definidos pelo usuário (estruturas)*

Exercícios

Referências bibliográficas

Anexo: Modelagem e simulação – aprendendo sobre o mundo

CAPÍTULO 11 Busca e ordenação

11.1 Algoritmos de busca

11.1.1 Busca linear (sequencial)

11.1.2 Busca binária

11.2 Algoritmos de ordenação e classificação

11.2.1 Método de ordenação BubbleSort

11.2.2 Método de ordenação QuickSort

11.2.3 Outros tipos de algoritmos de ordenação

Exercícios

Referências bibliográficas

Considerações finais

Assuntos complementares

Enumerações

Unões

Aplicações

Um pequeno sistema estatístico

Jogo da Velha

Lógica de programação e algoritmos



VISÃO DO CAPÍTULO

Vamos iniciar nosso livro com a apresentação dos conceitos de lógica e de algoritmos. Ambos são a base das linguagens de programação dos computadores digitais. Qualquer problema a ser convertido em um programa precisa ser pensado, meditado, entendido e planejado antes de ser programado. É péssima prática iniciar a escrita de código de programação sem antes pensar o problema.

A lógica ordena nosso raciocínio e garante que as decisões que o programa tomará serão corretas. Já os algoritmos garantem que a sequência de passos seguida pelo programa seja apropriada e resolva o problema.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- « Entender os princípios de Lógica Matemática (valores lógicos, lógica proposicional e de predicados) que regem os programas de computador;
- « Usar a lógica na programação de computadores;
- « Representar problemas reais por meio de algoritmos, para depois programá-los.



Antes de mais nada, vamos indicar as ferramentas com as quais trabalharemos neste livro. São todas oriundas de *software livre* ou *open source*; portanto, podem ser baixadas e instaladas em sua máquina, sem que se desrespeite os devidos direitos autorais.

- Para o desenho de algoritmos, utilizaremos o SFC, que pode ser encontrado em <http://watts.cs.sonoma.edu/SFC/>;
- Como compilador, que é a ferramenta que gera o programa executável, podemos utilizar o Bloodshed DEV C++, que pode ser baixado em www.bloodshed.net;
- Também o compilador Code::Blocks pode ser utilizado; ele pode ser encontrado em www.codeblocks.org.

Antes de iniciar a leitura, instale os programas.



1.1 Qual é o propósito da Lógica?

A Lógica está relacionada ao pensamento racional e ordenado. Na Filosofia, ela se preocupa em estudar por que pensamos do jeito que pensamos. Como arte ou técnica, ela nos induz a usar corretamente as leis do pensamento. É a *arte do bem pensar*.

Quando utilizamos silogismos em Filosofia, estamos aplicando a lógica em um problema real:

Todo homem é mortal.

Sócrates é homem.

Portanto, Sócrates é mortal.

Da mesma forma que as outras ciências, a Computação utiliza a Matemática para determinar fatores precisos, além de uma notação poderosa que garanta abstrações corretas e raciocínios rigorosos. O objetivo da Lógica de programação é garantir nossa compreensão das linguagens e das ferramentas de programação.

A Lógica organiza de forma metódica o pensar de quem se preocupa com a atividade do raciocínio.

No linguajar comum, usamos afirmações e interrogações de acordo com o que desejamos expressar; o problema é que tais ações podem ser imprecisas ou duvidosas, mas um computador não pode agir com tal comportamento, dada a precisão que caracteriza a computação. Por isso precisamos de ferramentas e técnicas lógicas para programar a máquina.

Isso se faz por meio de sentenças (ou proposições), que só podem ser de dois tipos: *verdadeiras* ou *falsas*.

Considere os seguintes exemplos de expressões:

- a. *Dez é menor do que seis.*
- b. *Como vai?*
- c. *Fulano é muito competente.*
- d. *Existe vida fora da Terra.*

A frase (a) é uma sentença porque pode ter associado um *valor lógico*, que no caso é falso; já a do item (b) é uma *pergunta*, por isso não pode ter inicialmente um valor *nem verdadeiro nem falso*; assim, não é uma sentença dentro do campo da Lógica; a terceira expressão, (c), não é verdadeira nem falsa, pois não pode ser especificada, portanto, não é uma sentença; a frase (d) é uma sentença porque será *verdadeira* ou *falsa*.

1.2 Conectivos e tabelas-verdade

Nossas conversas não se limitam ao emprego simples de sentenças; adicionamos elementos chamados de *conectivos* para definirmos *sentenças compostas*.

O valor lógico de uma expressão composta dependerá dos valores de cada sentença individual pela qual ela é constituída, e também dos conectivos utilizados.

Exemplos de conectivos usuais são o “e”, o “ou” e o “não”.

Se combinarmos as sentenças “hipopótamos são grandes” e “a Terra é redonda”, teremos a sentença resultante “hipopótamos são grandes e a Terra é redonda”, que é verdadeira. Em Lógica, usaremos o símbolo “^” para representar o conectivo lógico “e”, e letras maiúsculas para representar as sentenças em estudo.

Se A e B são verdadeiras, A^B (lemos “A e B”) será verdadeira também.

A expressão A^B é chamada a conjunção de A e B, sendo A e B os fatores da expressão.

Podemos resumir os efeitos das conjunções (e de qualquer outro conectivo que utilizarmos) por meio da tabela-verdade correspondente, mostrada no exemplo a seguir. Ela nos ajuda a decidir o valor lógico resultante da combinação de diversas proposições.

TABELA 1.1 Tabela-verdade da conjunção (“e”).

| A | B | A^B |
|---|---|-----|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

Se tivermos a combinação de F e V, por exemplo, a resultante será F (dizemos: falsidade e verdade é falsidade)

Outro conectivo comum é o “ou”, representado pelo símbolo “v”. A expressão AvB (leia-se “A ou B”) é chamada a disjunção de A e B.

Vejam os a tabela-verdade da disjunção:

TABELA 1.2 Tabela-verdade da disjunção (“ou”).

| A | B | AvB |
|---|---|-----|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

Já o conectivo da negação é o “~”. A negação inverte o valor lógico da proposição (“não”).

TABELA 1.3 Tabela-verdade da negação (“não”).

| A | ~A |
|---|----|
| V | F |
| F | V |

As sentenças podem ainda ser combinadas na forma condicional do tipo “se sentença 1, então sentença 2”. A representação é a seguinte: $A \rightarrow B$ (leia-se “A implica B”).

O conectivo lógico aqui é a *implicação*, e indica que *a verdade de A implica a verdade de B*.

Por exemplo, se A representa a sentença “fogo é condição de fumaça”, poderia ser reformulada como “se há fumaça, então há fogo”. O antecedente é “há fumaça”, e o consequente é “há fogo”.

TABELA 1.4 Tabela-verdade da implicação.

| A | B | $A \rightarrow B$ |
|---|---|-------------------|
| V | V | V |
| V | F | F |
| F | V | V |
| F | F | V |

1.3 Lógica de programação

Os conectivos lógicos *e*, *ou* e *não* (ou, mais comumente, seus equivalentes em inglês *AND*, *OR* e *NOT*) são usados na programação de computadores e combinados entre si, sendo as tabelas-verdade que decidem o resultado de sua combinação.

Como veremos, vem daí o poder de decisão de um computador.

1.4 Algoritmos

Begin at the beginning ... and go till you come to the end: then stop.

Alice in Wonderland (LEWIS CARROLL)

Os algoritmos são o centro da computação, pois a principal tarefa, ao se escrever um programa para computador, é planejar um algoritmo que produza a solução e possa ser repetido indefinidamente.

Assim:

Algoritmo é um conjunto de passos, passível de repetição, que resolve um problema.

Os passos para a construção de um algoritmo qualquer são os seguintes:

- Analisar o problema;
- Identificar as entradas de dados;
- Determinar que transformações devem ser feitas pelo algoritmo (processamento);
- Identificar as saídas (solução);
- Construir o algoritmo com o diagrama de blocos (ou fluxograma).

Os algoritmos podem ser representados de forma gráfica, por meio de símbolos padronizados (*fluxogramas*, também chamados de *diagramas de blocos*).

Vejamos na figura a seguir como os algoritmos são representados:

FIGURA 1.1 Simbologia para diagramas de blocos.

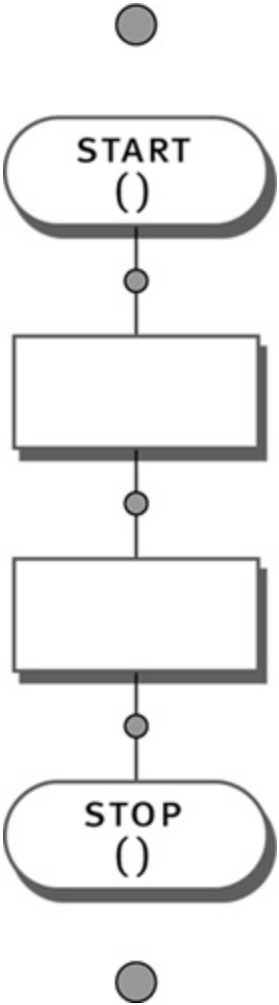


1.5 Programação estruturada

Qualquer programa de computador pode ser escrito usando-se apenas três tipos de *estruturas de controle do fluxo de programação*: *sequencial*, de *decisão* (ou *seleção*) e de *repetição*. A utilização dessas estruturas deve seguir algumas regras com o objetivo de evitar um código confuso e sem controle sistemático. A esse conjunto de regras chamamos *programação estruturada*.

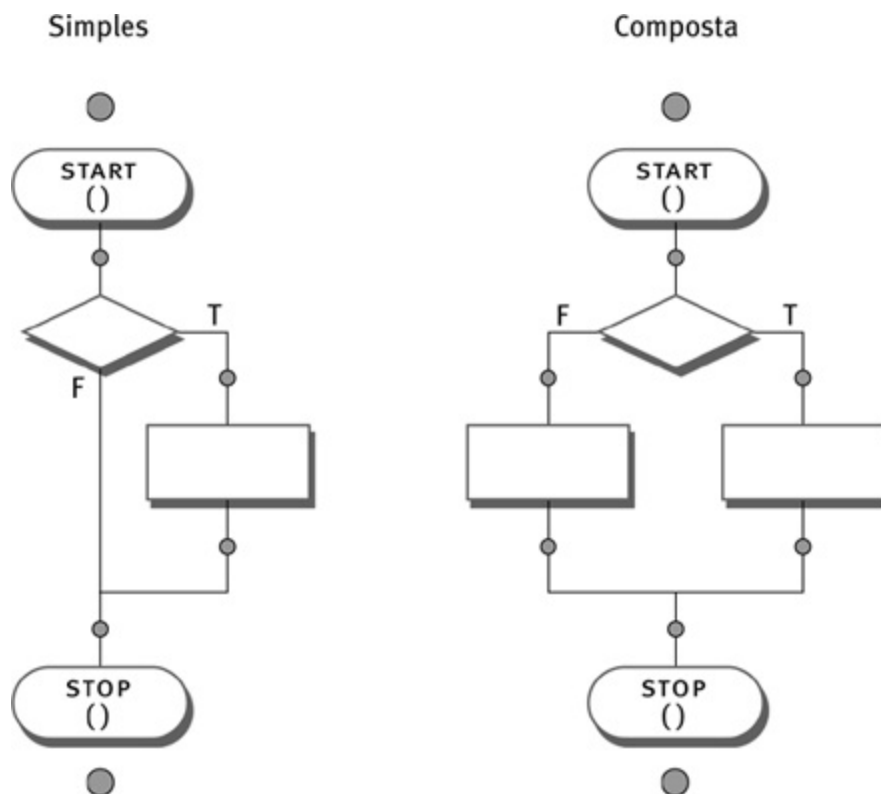
Eis as representações em diagramas de blocos dos componentes da programação estruturada, os quais garantem sua eficácia:

FIGURA 1.2 Sequência



Esse algoritmo garante que o programa seja executado do início ao fim, sem se perder por caminhos paralelos e desconexos. É a garantia da unicidade de ações e procedimentos, dentro da lógica pensada pelo programador.

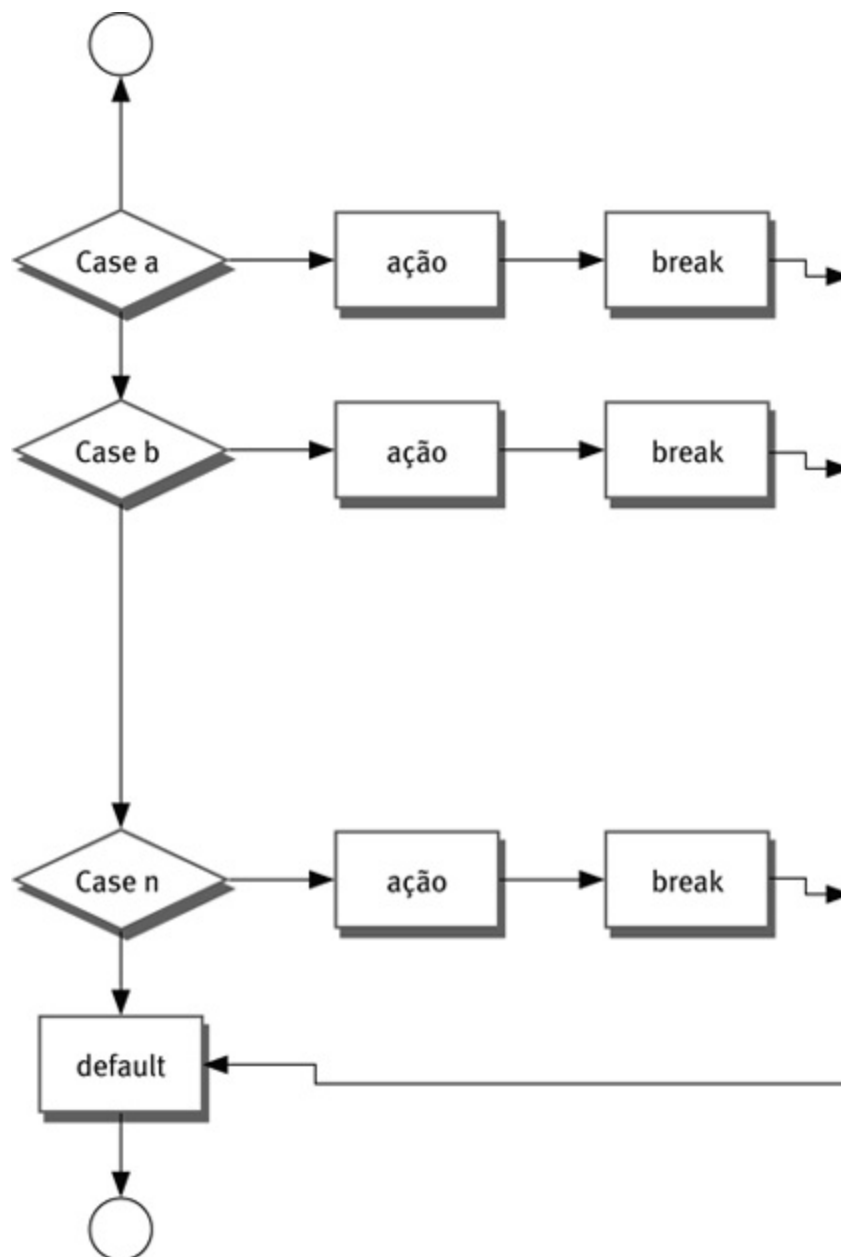
FIGURA 1.3 Decisão (ou seleção)



A decisão simples testa uma condição e realiza uma ação caso esta seja verdadeira, sem se preocupar em realizar uma ação no caso de verificação da condição oposta. Por exemplo, se um número digitado for menor que zero, solicito uma nova digitação; se não for, o programa simplesmente continua na próxima linha abaixo da decisão.

A decisão composta, ao contrário, tem uma ação prevista em caso de verificação da condição oposta. Por exemplo, se a média de um aluno for maior ou igual a seis, vou imprimir na tela “Aprovado”. Se não for (ou seja, se a média for menor que seis), imprimirei “Reprovado”.

FIGURA 1.4 Múltipla

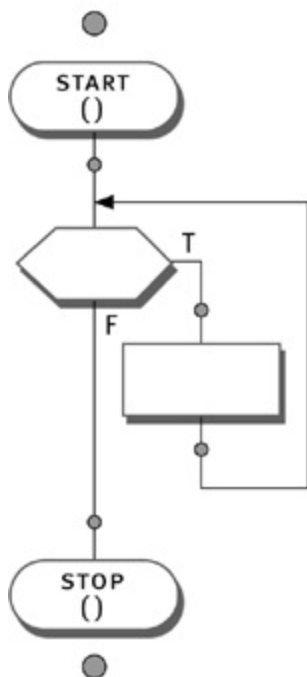


Na decisão múltipla, temos mais do que duas condições a serem testadas; precisamos, portanto, de uma estrutura maior do que a decisão composta – que é capaz de avaliar apenas duas condições.

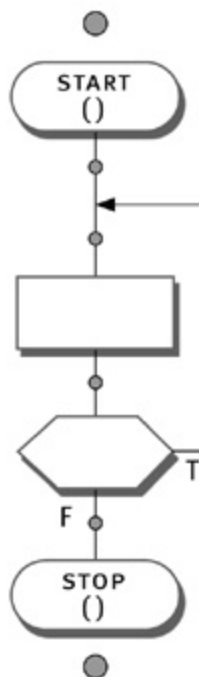
Utilizaremos, então, um aninhamento de decisões compostas, o que é uma interessante saída para o problema. Se a condição for *verdade*, realizam-se as ações previstas e abandona-se a estrutura, de forma a se evitar um novo teste fora de contexto. Se a condição for *falsidade*, testa-se a próxima possibilidade, e assim por diante. É possível, ainda, prever uma condição padrão (*default*), caso os testes anteriores não tenham verificado valores de verdade.

FIGURA 1.5 Repetição

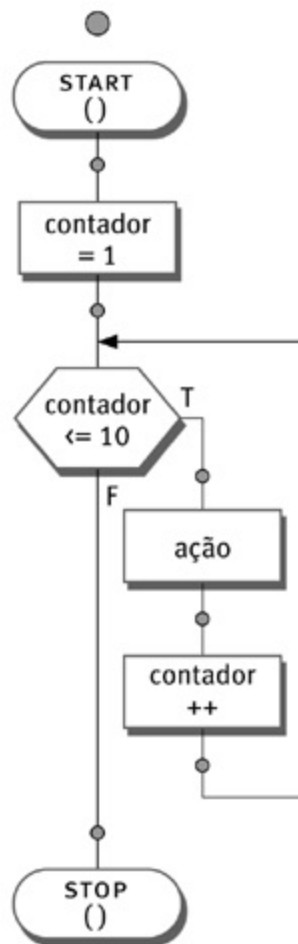
a) Com teste no início



b) Com teste no fim



c) Com variável de controle



A repetição com teste no início avalia uma condição antes de executar as ações previstas e repetitivas; se válida, o processamento entra em iteração (*loop*), até que tal condição não seja mais verdadeira, quando o programa seguirá normalmente para o restante das rotinas programadas. Essa repetição é perfeita para testes de senhas antes do acesso a funções repetitivas do programa.

Já a repetição com teste no fim executará uma ação pelo menos uma vez antes de decidir se ela continuará. É muito utilizada em validações de entradas de dados, antes que se dê a sequência ao programa. Todas serão trabalhadas no decorrer do livro.

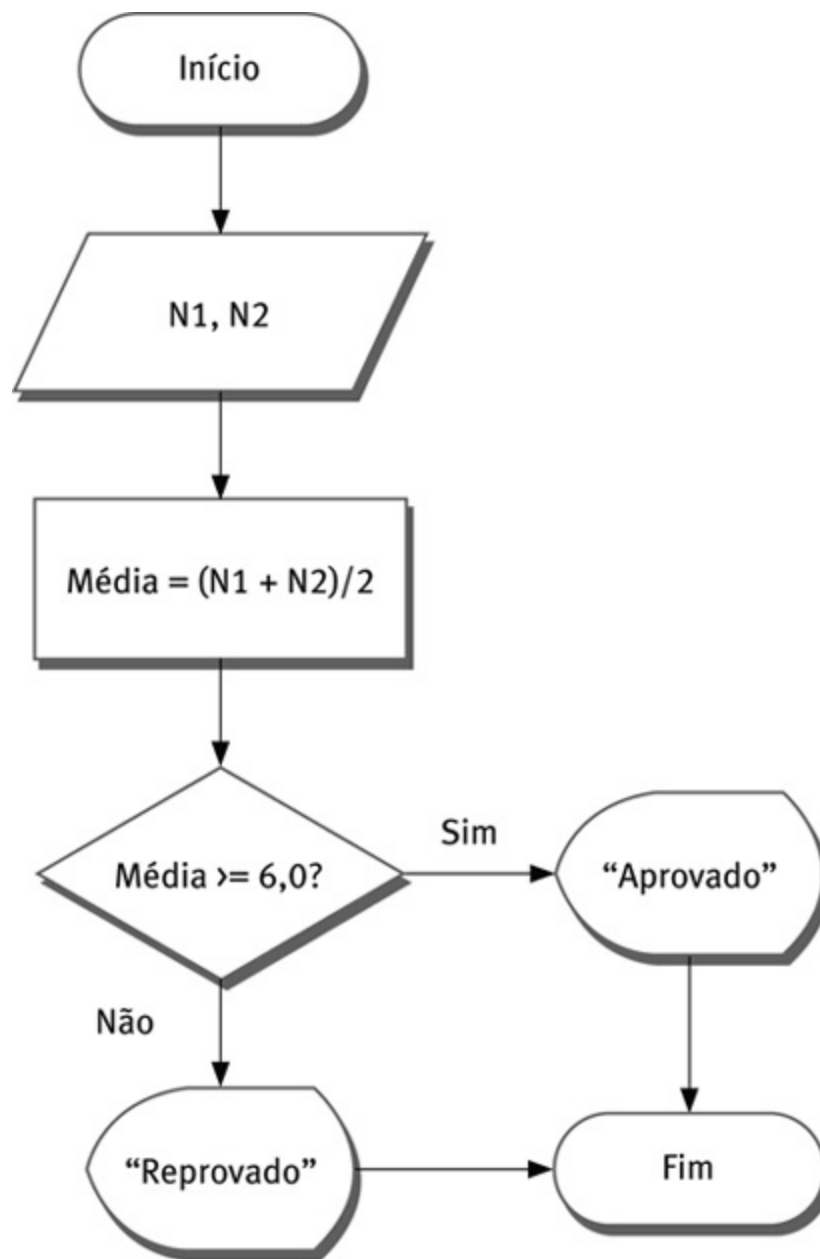
Apresentemos, agora, um exemplo real de algoritmo, a título de ilustração e modelo.

FIGURA 1.6 Cálculo da média das notas de um aluno.

Entradas: notas bimestrais

Processamento: soma das notas dividida pelo número de notas, e cálculo da média aritmética

Saída: situação do aluno, considerando sua aprovação ou reprovação



Se o programa de computador for preparado para seguir esse algoritmo, ele decidirá sempre, e sem dúvida, se o aluno foi aprovado ou não. O poder de um algoritmo vem daí: se for bem planejado, levará sempre a um resultado correto.

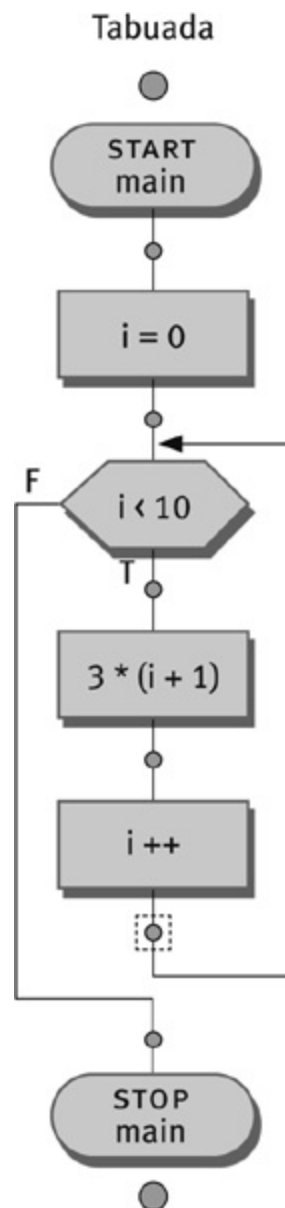
Vamos praticar, então. Em computação, é mandatório que se pratique tudo o que foi aprendido. Costumo dizer que aprender a programar um computador é como dirigir um automóvel: não basta ler o manual de instruções e sair dirigindo...

EXERCÍCIOS

1. Construa um algoritmo que calcule o valor do comprimento da circunferência, a partir do valor do raio.
2. Desenhe um algoritmo que receba dois números e exiba o resultado da sua soma.

3. Desenhe um algoritmo que receba dois números e mostre a soma, a subtração, a multiplicação e a divisão dos números.
4. Desenhe um algoritmo para determinar o consumo médio de um automóvel, considerando que a distância total percorrida e o total de combustível gasto são fornecidos.
5. Elabore um algoritmo que leia o nome de um vendedor, o seu salário fixo e o total de vendas efetuadas por ele no mês (em dinheiro). Sabendo que esse vendedor ganha 15% de comissão sobre suas vendas efetuadas, faça que o algoritmo informe o seu nome, o salário fixo e salário no final do mês.
6. Crie um algoritmo que leia o nome de um aluno e as notas das três provas que ele obteve no semestre, com pesos 2, 3 e 5. No final, deve-se informar o nome do aluno e a sua média ponderada.
7. Desenhe um algoritmo que leia uma temperatura em graus Celsius e a apresente convertida em graus Fahrenheit. A fórmula de conversão é: $F = (9 * C + 160) / 5$, sendo F a temperatura em graus Fahrenheit e C, a temperatura em graus Celsius.
8. Elabore um algoritmo que efetue a apresentação do valor da conversão em real (R\$) de um valor lido em dólar (US\$). O algoritmo deverá solicitar o valor da cotação do dólar e também a quantidade de dólares disponíveis com o usuário.
9. Faça um algoritmo que calcule o rendimento de um depósito após um mês de aplicação. Considere fixa a taxa de juros de remuneração, correspondente a 0,50% ao mês.
10. Faça um algoritmo que receba um número e mostre uma mensagem caso este seja maior que 10.
11. Crie um algoritmo que leia dois valores inteiros distintos e informe qual é o maior.
12. Faça um algoritmo que receba um número e diga se este está no intervalo entre 100 e 200.
13. [Resolvido] Desenhe um algoritmo que mostre na tela a tabuada do 3.
14. Desenhe um algoritmo que imprima dez vezes na tela do computador a frase “FATEC”.
15. Desenhe um algoritmo que, por dez vezes, receba um número digitado pelo usuário e o compare com o valor “5”; em caso de acerto do usuário, o algoritmo finaliza-se.

FIGURA 1.7 Tabuada do 3 (algoritmo).



REFERÊNCIAS BIBLIOGRÁFICAS

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

FORBELLONE, A; EBERSPACHER, H. *Lógica de programação – a construção de algoritmos e estrutura de dados*. 3.ed. São Paulo: Prentice Hall, 2005.

MANZANO, José Augusto Navarro Garcia; OLIVEIRA, Jayr Figueiredo de. *Algoritmos: lógica para desenvolvimento de programação de computadores*. 26. ed. São Paulo: Érica, 2012.

ANEXO: IBM 1130

Aprendi a programar em uma linguagem científica chamada FORTRAN (FORMula TRANslation), em meados da década de 1980, na USP. O computador era um IBM 1130, com entrada de dados em cartão perfurado (*punched card*), e saída impressa em relatório de 132 colunas.

Tínhamos algumas dezenas de *kilobytes* para escrever qualquer que fosse o programa, o que nos forçava a ser concisos em nosso código-fonte.

Revendo meu caderno dessa época, notei como tudo é direto e esclarecedor, sem muitos

rodeios, e como a evolução dos algoritmos é contínua e aplicada.

Comparo, então, a minha experiência do passado com os modernos recursos que um aluno tem hoje à sua disposição: máquinas potentes, excesso de memória, textos técnicos, mobilidade, Internet, fóruns de discussão... algo certamente mudou.

Para os interessados no assunto, existe um emulador do IBM 1130 em <http://ibm1130.org/sim/downloads>. Vale a pena saber como as coisas eram em um passado não tão distante!

Tipos primitivos, variáveis, constantes e operadores – estrutura geral de um programa

➔ VISÃO DO CAPÍTULO

Os dados trabalhados por um programa de computador podem estar em vários estados de representação (*tipos*); por isso, antes de utilizá-los, precisamos declará-los e inicializá-los.

O componente que guarda os dados trabalhados pelo programa é chamado de *variável*, e as possibilidades de transformação desses dados têm relação direta com os *operadores* aos quais recorreremos.

OBJETIVO INSTRUCIONAL

Após estudar e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- ◀◀ Conhecer e utilizar os diversos tipos de dados de um programa;
- ◀◀ Manipular variáveis e constantes;
- ◀◀ Utilizar os diversos operadores da Linguagem C;
- ◀◀ Editar e compilar seu primeiro programa, após conhecer a estrutura típica dos aplicativos em Linguagem C.



Observação: Qualquer dado trabalhado por um programa de computador deve ter seu tipo identificado e declarado, de forma a garantir que as operações sobre ele realizadas – bem como sua alocação em memória e precisão de cálculo – sejam respeitadas.

2.1 Tipos de dados primitivos

A maioria das linguagens declara os tipos inteiro, real, caractere, cadeia de caracteres e lógico. A Linguagem C não definiu como tipos originais o lógico e a cadeia de caracteres (que chamamos de *string* – exemplo: “FATEC”); assim, devemos utilizar funções predefinidas para manipular esse tipo de dado, em C.

Em Linguagem C, temos os seguintes tipos de dados primitivos:

- **Inteiro:** palavra reservada *int* (exemplos: 3, 6, 9, 27)

- **Real:** palavra reservada *float* (exemplos: 3.1416, 8.8)
- **Caractere:** palavra reservada *char* (exemplos: 'a', '8')
- **Real de precisão dupla:** palavra reservada *double*
- **Tipo sem um valor inicial:** palavra reservada *void*

Detalhando um pouco mais os tipos primitivos da Linguagem C, temos:

Inteiro (*int*): informação numérica do conjunto dos números inteiros. Podem ser *short* (2 bytes), *long* (4 bytes), *signed* (positivos e negativos – padrão da linguagem, não precisa declarar), *unsigned* (apenas positivos) e *register* (guarda valor em um registrador da CPU, caso seja muito utilizado pelo programa). Esses prefixos garantem o tamanho desejado independentemente da arquitetura do computador. O inteiro padrão varia de -32768 a 32767.

EXEMPLOS:

```
long int numero = 1234567L; // varia de -2147483648 a 2147483647
unsigned int maior; // usa de 0 a 65535
unsigned char var; // varia de 0 a 255
register int valor; // mantém o valor em um registrador da CPU, se utilizá-lo sempre
```

Real (*float*): conjunto dos números reais; variam de 3.4E-38 a 3.4E+38.

Caracter (*char*): caracteres alfanuméricos > (0...9) + (a...z) + (caracteres especiais, como #, \$, %, @, etc); armazenam valores inteiros (da tabela ASCII) de -128 a 127.

Real de precisão dupla (*double*): é um número real de grande precisão; variam de -1.7E-308 a 1.7E+308.

Podemos, também, definir nossos próprios tipos de dados pelo emprego de um recurso da Linguagem C chamado *typedef*:

```
typedef int PESSOAS;
```

O uso no programa seria feito da seguinte forma:

```
PESSOAS habitantes;
```

2.2 Variáveis

Variáveis são endereços de memória de trabalho que guardam, temporariamente, um valor utilizado pelo programa.

Toda variável deve ter atributos do tipo *nome*, *tipo*, *tamanho* (que vem do tipo escolhido) e *valor*.

EXEMPLO: *int numero = 8; // inicialização recomendável de uma variável*

As variáveis podem ter o chamado *escopo local*, quando são vistas apenas dentro da função em que foram chamadas e perdem seu valor quando o processamento sai de tal função, ou podem ser declaradas *static*, quando guardarão seu valor entre as diversas chamadas dessa função. Podem, ainda, ser de *escopo global*, quando têm validade em qualquer ponto do programa.

Quanto aos qualificadores, as variáveis podem ser *const* ou *volatile*. *Const* é usado em uma declaração de tipo de dado cujo valor não se altera.

EXEMPLO: `const double pi = 3.141593;`

Volatile especifica uma variável cujo valor pode ser alterado por processos externos ao programa atual – uma troca de dados com um dispositivo externo ao sistema, por exemplo.

Em relação à classe de armazenamento, podem ser *extern* (variável global entre os diversos arquivos que possam compor o programa), *static* (se globais, são reconhecidas apenas no arquivo em que estão escritas – o que permite, portanto, o emprego de nomes iguais para diversas variáveis), *register* e *auto*.

IMPORTANTE:

- É sempre recomendável que se inicialize uma variável com um valor adequado para evitarmos erros de tempo de execução, pois essa variável pode reter valores anteriores que gerarão surpresas desagradáveis durante a execução do programa;
- O nome da variável deve sempre lembrar o que vai guardar, e não pode ser igual às palavras reservadas da linguagem;
- A *vírgula* em Linguagem C é o *ponto*, e vice-versa (ao contrário do que fazemos no Brasil);
- Quando não se declara o tipo da variável, o compilador assume o tipo *int*.

Exemplos de declarações adequadas de variáveis:

```
int habitantes = 200000; // cidadãos de uma cidade
short int pessoas = 20; // grupo pequeno de pessoas
char letra = 'a'; // caractere 'a'
float salario = 1000.00; // valor monetário, com duas casas decimais
```

2.3 Constantes

Ao contrário das variáveis, *constantes* não podem ter seu valor modificado.

EXEMPLOS:

```
O valor de pi (3.1416);
A aceleração da gravidade na Terra (9.8 m/s2);
O número de Avogadro (6.02 x 1023).
```

2.4 Operadores aritméticos

São as já conhecidas operações que realizamos no dia a dia. Mostramos, a seguir, a notação sintática da Linguagem C para cada uma delas:

Adição (+)

Subtração (-)

Multiplicação (*)

Divisão (/)

Potenciação: `pow(x,y)` “Power of”

Radiciação: `sqrt(x)` “Square root”

Resto da divisão (%) ou módulo – exemplo: `9%4` resulta em 1

Quociente de divisão inteira (/) – exemplo: `9/4` resulta em 2

Incremento (++): incrementa o valor da variável em uma unidade

Decremento (--): decrementa o valor da variável em uma unidade

Os operadores, como na Aritmética, têm *prioridade de execução*:

PRECEDÊNCIA MAIOR PARA MENOR

- 1) Parênteses mais internos;
- 2) potenciação e radiciação;
- 3) multiplicação (*), divisão (/) e resto da divisão (%);
- 4) soma (+) e subtração (-).

2.5 Operadores relacionais

São utilizados para a comparação entre valores:

`=` igualdade

`<` menor

`>` maior

`>=` maior ou igual

`<=` menor ou igual

`!=` diferente

2.6 Operadores lógicos

Utilizados para a determinação de valores lógicos de expressões, como visto no capítulo que tratou de Lógica:

`not` (negação): `!`

and (conjunção): &&

or (disjunção): ||

2.7 Atribuição de valor

Atribui à variável o valor fornecido à direita da expressão, por meio de expressões aritméticas.

EXEMPLO: `x = 4;`

Podemos usar ainda: `+=`, `-=`, `*=`, `/=`

EXEMPLO: `x += y` é o mesmo que `x = x + y`

2.8 Estrutura geral dos programas em Linguagem C

Os programas são desenvolvidos em uma sequência de *edição*, *compilação*, *linkedição* e *execução*.

Depois que o código-fonte foi digitado em um editor de texto, o *compilador* verifica sua sintaxe e, se ela estiver correta, cria um arquivo intermediário chamado de *arquivo-objeto* (.obj). A partir desse, o *linkeditor* criará o *arquivo-executável* (.exe), a partir da ligação do arquivo-objeto com as funções providas pelas bibliotecas correspondentes.

Os erros que podem surgir em programação são do tipo *lógico* (*bug*) ou de *sintaxe*. No primeiro, o programa não gera o resultado esperado; portanto, é mais perigoso que o segundo, no qual as regras da linguagem são violadas mas o compilador barra a geração do arquivo executável até que o erro seja corrigido.

A Linguagem C foi desenvolvida por Dennis Ritchie e Brian Kernighan nos Laboratórios Bell, no início da década de 1960 (veja mais adiante informações sobre Ritchie).

Essa linguagem apresenta como vantagem o fato de ser simples: ela é dotada de bibliotecas-padrão que já vêm com o compilador C, trabalha dados de forma simples, permite acesso direto à memória e é portátil, considerando plataformas diferentes.

É representante da *programação estruturada ou modular*, baseada no conceito de blocos de código.

A Linguagem C possui um total de 32 *palavras reservadas*, conforme definido pelo padrão ANSI:

```
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

2.9 Primeiro programa em Linguagem C

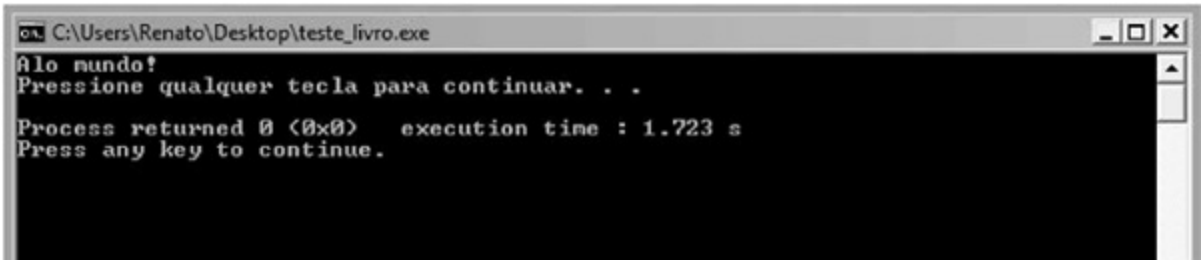
Um programa em Linguagem C é uma lista de instruções (conjunto de comandos) que o computador pode executar, agrupados em uma ou mais funções. A função *main()* é a função principal e obrigatória, e tem por tarefa ordenar a sequência de execução do programa como um todo. O programa tem um ou mais *header files* (arquivos com extensão *.h*, nos quais estão definidas as funções e respectivas listas de argumentos), blocos de instruções (*{...}*), instruções elementares terminadas com “;”, e comentários (*//* ou */* ... */*).

(**Atenção:** A Linguagem C é *case sensitive*, ou seja, diferencia os caracteres maiúsculos dos minúsculos.)

Vamos criar nosso primeiro programa em Linguagem C, que, por tradição, segundo o próprio criador da Linguagem C,¹ tem a missão de escrever na tela do computador a expressão *Alo Mundo!*.

```
/* Programa-exemplo para programação em C */  
#include <stdio.h>  
#include <stdlib.h>
```

```
main( )  
{  
    printf("Alo mundo! \n");  
  
    system("pause");  
}
```



A função *system()* executa um comando interno do sistema operacional, chamado a partir do programa em C.

O pré-processador do compilador e suas diretivas (*#include*) examinam o código e inserem o que for necessário no programa original. Os arquivos *.h*, como visto, são chamados de *arquivos de inclusão* ou de *cabeçalho* (*header files*).

Utilizamos a função *printf()* para a saída na tela, que está definida em *stdio.h* (entrada e saída-padrão – *standard input and output*); daí a necessidade de inclusão desse arquivo de cabeçalho antes da função *main()*.

Utilizamos também uma sequência (ou um caractere) de escape “*\n*”, que executa uma operação de formatação no texto impresso na tela. As principais sequências de escape são:

```
\n = newline (nova linha)  
\a = ASCII beep  
\b = backspace  
\f = form feed (avanço de página)  
\r = carriage return (enter)  
\t = tabulação horizontal  
\ = barra invertida  
\' = aspas simples  
\" = aspas duplas  
\? = interrogação  
\xnn = caractere ASCII em hexadecimal (nn); Exemplo: \xDC, \xC9, \xCD
```

Constantes simbólicas podem ser utilizadas para tornar valores fixos mais entendíveis e fáceis de se lembrar:

```
#define PI 3.14159  
#define EU "Renato"  
#define FALSE 0
```

Também faremos uso, mais adiante, dos especificadores de formato, que, como o nome diz, formatam a entrada e a saída-padrão. São eles:

%d = inteiro decimal
%f = real (ponto flutuante)
%o = inteiro octal
%x = hexadecimal
%c = caractere em formato ASCII
%s = *string* de caracteres
%p = valor ponteiro

Temos, ainda, a possibilidade de utilizar os especificadores de comprimento e precisão, de forma a dizer ao programa quantas casas decimais serão utilizadas na saída:

EXEMPLO: `printf ("%6.2f", 285978.0012)` mostrará *285978.00*.

Fizemos uso de uma função de biblioteca da Linguagem C, qual seja, `printf()`, que mostra uma expressão na saída-padrão (tela) do computador:

```
printf("Numero %d \n", var);
```

A função equivalente para receber dados da entrada-padrão (o teclado) é *scanf()*, a qual tem a seguinte sintaxe:

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

Note o emprego dos especificadores de formato e do `&`, os quais indicam que a variável será referenciada por seu endereço, e não pelo seu valor (também veremos isso mais adiante).

Podemos utilizar também a entrada e saída de caracteres simples, que recebem apenas o primeiro caractere digitado e o armazenam na variável de atribuição:

```
var = getchar();
```

Ou mostra na tela o conteúdo de tal variável:

```
putchar(var);
```

Para entrada e saída de cadeias de caracteres, temos:

```
char nome[30]; // cadeia ou vetor de caracteres  
gets(nome);  
puts(nome);
```

EXEMPLOS:

```
printf ("%s ... %d ...", "Venus", 67);  
printf ("%c", 'J');  
printf ("%s %c", "Jota", '.');
```

Vamos a mais alguns exemplos de programas:

```
/* Recebe numero do teclado e mostra na tela */
```

```

#include <stdio.h>
#include <stdlib.h>

main( )
{
    int num;

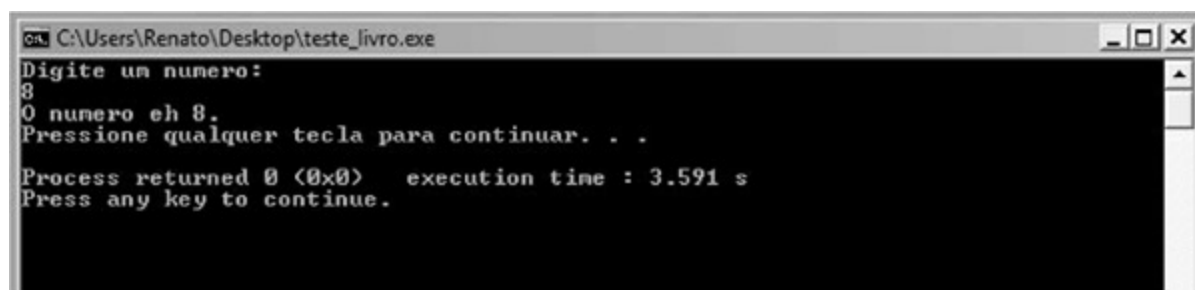
    printf("Digite um numero: \n");

    scanf("%d", &num);

    printf("O numero eh %d. \n", num);

    system("pause");
}

```



```

C:\Users\Renato\Desktop\teste_livro.exe
Digite um numero:
8
O numero eh 8.
Pressione qualquer tecla para continuar. . .

Process returned 0 (0x0)   execution time : 3.591 s
Press any key to continue.

```

```

/* Escrever um programa que receba o valor do peso e altura
de uma pessoa e calcule seu IMC pela fórmula
[ IMC = peso / (altura*altura) ]. Exibir o valor do IMC calculado. */

#include <stdio.h>

#include <stdlib.h>

main()

{

    float peso, altura, imc;

    scanf("%f %f", &peso, &altura);

    imc = peso/(altura*altura);

    printf("O IMC eh %f. \n", imc);

    system("pause");
}

```

```
C:\Users\Renato\Desktop\teste_livro.exe
00 1.00
0 IMC eh 24.691359.
Pressione qualquer tecla para continuar. . .

Process returned 0 (0x0)   execution time : 13.570 s
Press any key to continue.
-
```

Como atividade prática, rode o programa e calcule o seu próprio IMC, utilizando a tabela a seguir para analisar o resultado:

TABELA 2.1 Tabela IMC.

| RESULTADO | SITUAÇÃO |
|--------------------|--------------------------------|
| Abaixo de 17 | Muito abaixo do <i>peso</i> |
| Entre 17 e 18,49 | Abaixo do <i>peso</i> |
| Entre 18,5 e 24,99 | <i>Peso normal</i> |
| Entre 25 e 29,99 | Acima do <i>peso</i> |
| Entre 30 e 34,99 | <i>Obesidade I</i> |
| Entre 35 e 39,99 | <i>Obesidade II (severa)</i> |
| Acima de 40 | <i>Obseidade III (mórbida)</i> |

Fonte: <<http://www.calculoimc.com.br/>>.

EXERCÍCIOS

1. Escreva um programa em C que apresente um menu de opções para a escolha do time de futebol de alguém.
2. Escreva um programa que desenhe uma tela de abertura com a logomarca da empresa ACME (usar caracteres da Tabela ASCII) e, depois, as opções de operações administrativas que o usuário pode realizar.
3. Escreva um programa que receba dois números e mostre a soma, a subtração, a multiplicação e a divisão desses números.
4. Escreva um programa que receba um número qualquer e mostre o seu dobro.
5. Dadas as medidas de uma sala, informe sua área.
6. Dado um valor em reais e a cotação do dólar, converta esse valor para dólares.
7. Desenhe uma árvore de Natal usando “*”, em um programa em C.
8. Imprima na tela o seguinte quadro:

- 1) Clientes
- 2) Faturas
- 3) Sair

9. Escreva um programa que receba um valor em metros e o converta para milímetros.

10. Escreva um programa que imprima na tela a tabuada do 5.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. *Fundamentos da programação de computadores*. 2.ed. São Paulo: Prentice Hall, 2007.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

RITCHIE, D. M.; KERNIGHAN, B. W. *The C programming language*. 2.ed. Upper Saddle River: Prentice Hall, 1988.

ANEXO: PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

As diversas linguagens de programação disponíveis para quem aprende a arte de programar um computador ou qualquer dispositivo digital moderno podem ser classificadas dentro de seu desenvolvimento histórico. A isso chamamos de *paradigmas das linguagens*, dentro de um quadro que indica a justificativa para seu desenvolvimento em dado momento histórico. Assim, a *primeira geração* de linguagem de programação teve relação direta com o desenvolvimento do computador digital, e a ele se aplicava – foi a chamada Linguagem de Máquina, bastante trabalhosa e difícil de programar, pois utilizava o nível de *bits* e *bytes*.

A *segunda geração* foi marcada pelas Linguagens de Montagem (*Assemblers*), tendo a linguagem Assembly como representante. Aqui já pensamos em um nível maior de abstração, com a introdução de palavras reservadas que representam instruções no processador e nos demais componentes do sistema computacional.

A *terceira geração* se chamou Paradigma Estruturado, e teve como base o emprego de blocos bem delimitados e regras que organizam o código e as chamadas das funções – daí o termo estruturada. Como exemplos de linguagens estruturadas, temos Ada (DoD, EUA – sistemas de tempo real), o nosso C (início da década de 1970), FORTRAN (FORMula TRANslation, científica, 1957), Pascal (Wirth, 1971, recursos modernos, como ênfase em tipos e estruturas de dados), BASIC (Dartmouth College), COBOL (comercial). Em paralelo, e resultante da evolução dessas linguagens, surgiu o Paradigma Orientado a Objetos, que teve como representantes maiores o C++ (Bjarne Stroustrup, Bell Labs, década de 1980), Java (Sun Microsystems, *Java Platform*, início da década de 1990), Small-Talk (de Alan Kay) e C# (Microsoft, *.NET Framework*)

Finalmente, o Paradigma Declarativo compõe a *quarta geração* de linguagens de programação de computadores. São aquelas empregadas na Inteligência Artificial. Como exemplos, LISP, Logo, Scheme e PROLOG.

Estruturas de controle de programação – decisão



VISÃO DO CAPÍTULO

Aprendemos nos capítulos anteriores a representação de um problema qualquer no formato de um algoritmo e quais são os operadores da Linguagem C; aprenderemos agora o emprego das *estruturas de decisão* em um algoritmo computacional e, posteriormente, na linguagem de programação adotada neste livro.

OBJETIVO INSTRUCIONAL

Após estudar e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- « Entender o papel do processo de decisão dentro de um algoritmo ou programa de computador;
- « Aplicar o conceito de decisão em suas três variantes, apresentadas pelas linguagens de programação (*simples, composta e múltipla*);
- « Entender que as decisões podem ser encadeadas de forma homogênea e também heterogênea;
- « Resolver exercícios e praticar os comandos de decisão da Linguagem C (*if, else, switch*).



3.1 Algoritmos de tomada de decisão

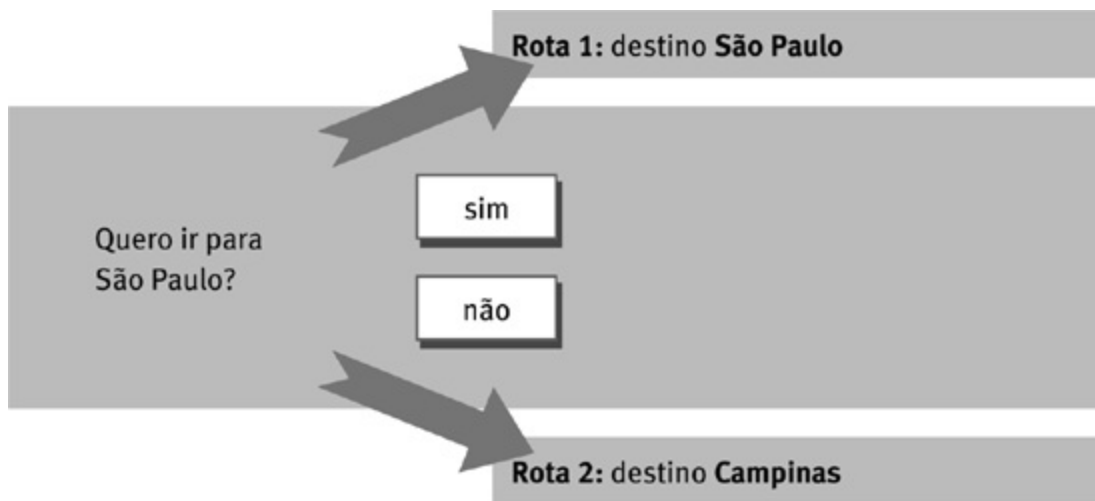
Você já reparou que tomamos *decisões* o tempo todo? Quando caminhamos dentro de casa, quando precisamos aplicar nosso dinheiro, quando aparece um problema em casa ou no trabalho, no trânsito – as decisões são parte da natureza de sobrevivência do animal neste planeta e, em especial, da nossa, dos seres humanos, em razão da racionalidade característica da espécie –; assim, decisões são tomadas a todo o momento, conscientemente ou não.

Existem até teorias e modelos que tentam representar o processo de decisão humano, considerando variáveis tangíveis e intangíveis, conflitos e interesses envolvidos.

Por exemplo, se nos depararmos com uma bifurcação em uma estrada, que nos oferece apenas dois caminhos possíveis, teremos que analisar as condições que cada opção nos oferece para que possamos decidir qual é a mais apropriada.

Vejamos:

FIGURA 3.1 Escolha de rotas.



Se viajo na direção de São Paulo, escolho a Rota 1; caso contrário, escolho a Rota 2.

Note que essa é uma decisão *binária*, pois só oferece duas possibilidades de escolha – e é exatamente assim que nossos computadores digitais tomam decisões, pois o sistema de lógica binária (como visto no Capítulo 1) só nos permite decidir entre *verdade* ou *falsidade*.

Vamos agora trabalhar esses conceitos dentro de nossa área de estudo: os algoritmos e a Linguagem C.

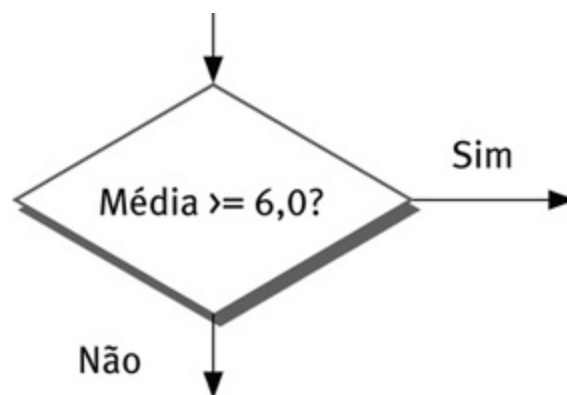
Um programa de computador faz, basicamente, três coisas: ou segue uma *sequência* de comandos, ou toma *decisões*, ou *repete* alguma ação.

Relembrando que:

Algoritmo é um conjunto de passos passíveis de repetição que resolve um problema.

A simbologia já apresentada para a representação de algoritmos pode agora ser utilizada, de forma prática e aplicada, no desenho dos algoritmos de *tomada de decisão*, assunto deste capítulo.

FIGURA 3.2 Algoritmo de tomada de decisão.



O *losango* representa a estrutura de decisão, e age como uma pergunta que só pode ter duas respostas: ou *verdadeiro* ou *falso*.

Vejamos agora um *exemplo de algoritmo* que decide se um aluno foi aprovado em dada disciplina escolar, em função das notas recebidas nas avaliações:

TESTE DA APROVAÇÃO DE UM ALUNO

- **Dados de entrada:** notas bimestrais
- **Processamento:** cálculo da média final e teste em relação à média de aprovação
- **Dados de saída:** “aprovado” ou “reprovado”

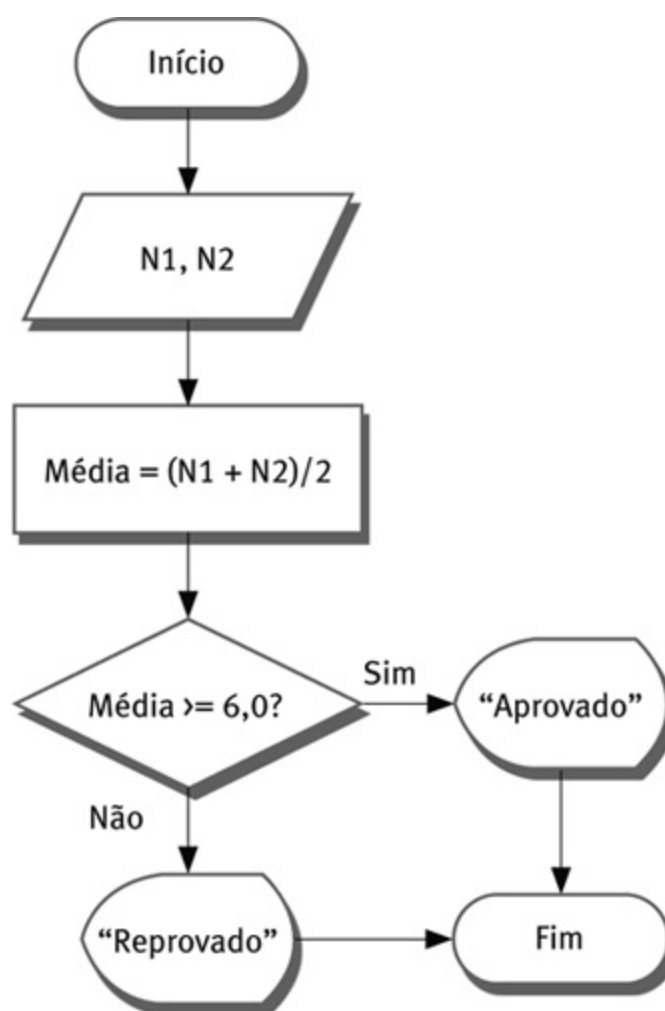
O algoritmo receberá duas notas de avaliações, calculará a média aritmética dessas notas e comparará a média com o valor 6,0 (que é a média de aprovação da escola).

Agora vem a *decisão*:

Se a média for maior ou igual a 6,0 (seis), o aluno está “aprovado”; **caso contrário**, ele está “reprovado”.

Vamos representar esses passos com o diagrama de blocos já conhecido:

FIGURA 3.3 Algoritmo de decisão (aprovação ou reprovação).



Criamos, dessa forma, o algoritmo de uma típica tomada de decisão, que poderá ser convertido agora em programa, por meio dos comandos e funções da sintaxe da Linguagem C:

Se a média for maior ou igual a 6,0 (seis), o aluno está “aprovado”; *caso contrário*, ele está “reprovado”.

```

if (media >= 6.0)
{printf("O aluno está aprovado! \n");}
else
{printf("O aluno está reprovado! \n");}

```

Um programa de computador toma muitas decisões em um algoritmo típico; isso é parte integrante da programação.

3.2 Algoritmos dos vários tipos de decisão

As decisões podem ser de três tipos: *simples*, *composta* ou *múltipla*.

Na *decisão simples*, existe ação a partir de apenas um dos valores lógicos testados (*verdade* ou *falsidade*); se a decisão for pelo outro valor lógico, o programa simplesmente continua no comando que sucede a estrutura de decisão, sem realizar ação alguma dentro desta.

Na *decisão composta*, os dois valores lógicos da decisão geram ações de algum tipo antes que o programa dê sequência.

Por fim, na *decisão múltipla*, temos mais do que duas possibilidades de escolha ou decisão; há, na verdade, um encadeamento de decisões binárias, como veremos.

Os algoritmos dos diversos tipos de decisão são mostrados a seguir:

FIGURA 3.4 Decisão simples

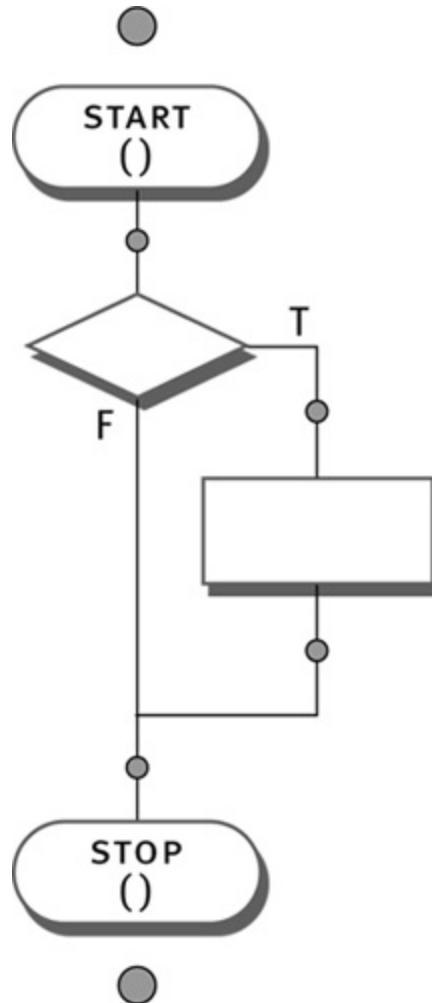


FIGURA 3.5 Decisão composta

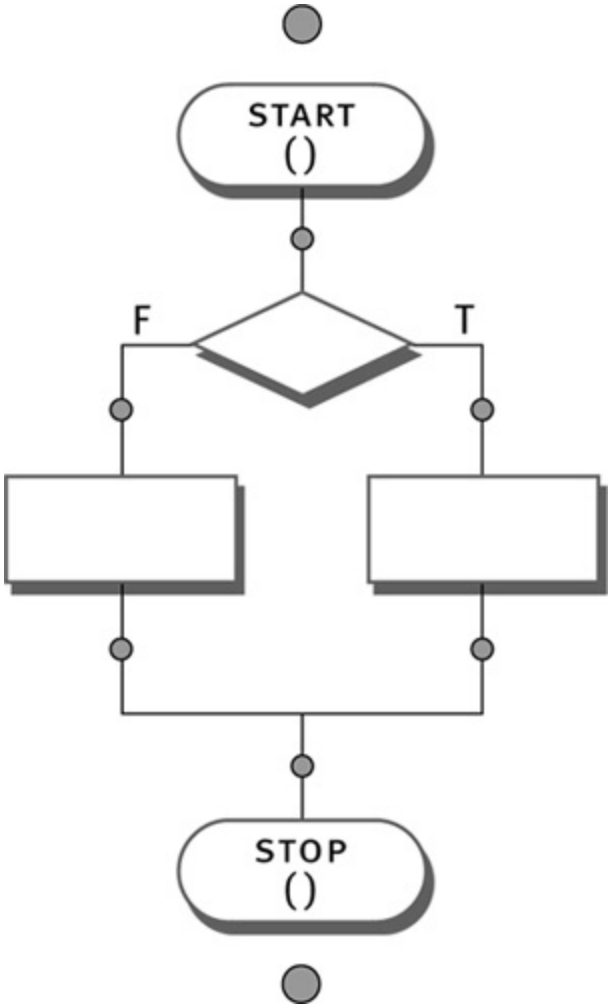
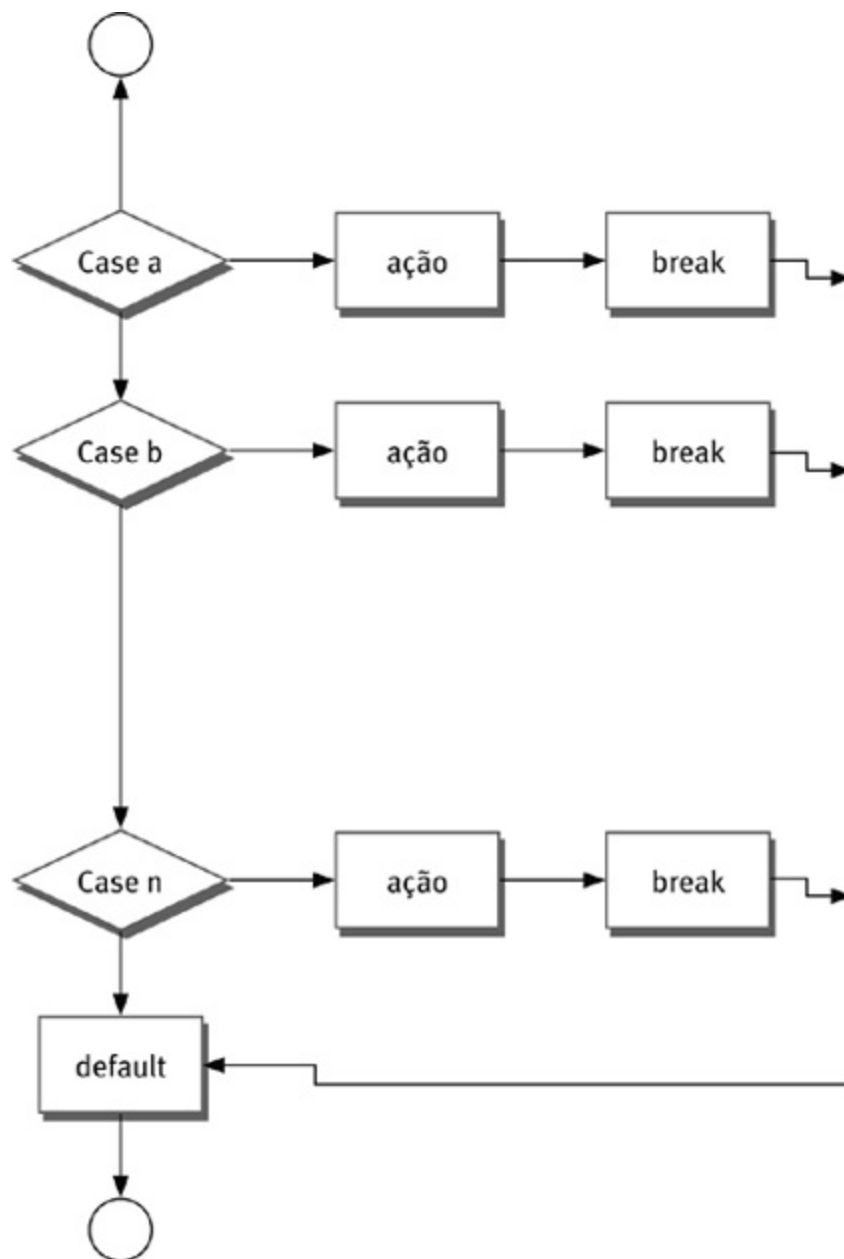


FIGURA 3.6 Decisão múltipla



Detalharemos agora esses tipos de decisão, explicando *quando* são aplicados e *como* são programados em Linguagem C.

3.2.1 *Decisão simples*

É usada para se testar uma condição, antes de se executar uma ação.

Formato padrão: *if (condição) {comandos};*

EXEMPLO:

```
if (média >= 6) {  
    printf("Um bom aluno! \n");  
}
```

```
}
```

3.2.2 *Decisão composta*

Ocorre quando duas alternativas dependem da mesma condição.

Formato padrão: *if (condição) {comandos} else {outros comandos};*

EXEMPLO:

```
if (média >= 6){
    printf("Aprovado \n");
}
else {
    printf("Reprovado \n");
}
```

3.2.3 *Decisão múltipla*

Ocorre quando um conjunto de valores discretos precisa ser testado e ações diferentes são associadas a esses valores.

Formato padrão: *switch (condição) {case 'x': comando(y); break;};*

EXEMPLO:

```
switch (var){
    case '1': printf("Opção 1 foi selecionada");
    break;
    case '2': printf("Opção 2 foi selecionada");
    break;
    default: printf("Nenhuma opção selecionada");
}
```

Vejamos um exemplo completo de decisão múltipla:

```
/* Exemplo de menu usando decisão múltipla */
#include <stdio.h>
#include <stdlib.h>
main ()
{
    char ch;

    printf("MENU de Escolhas\n\n");
```

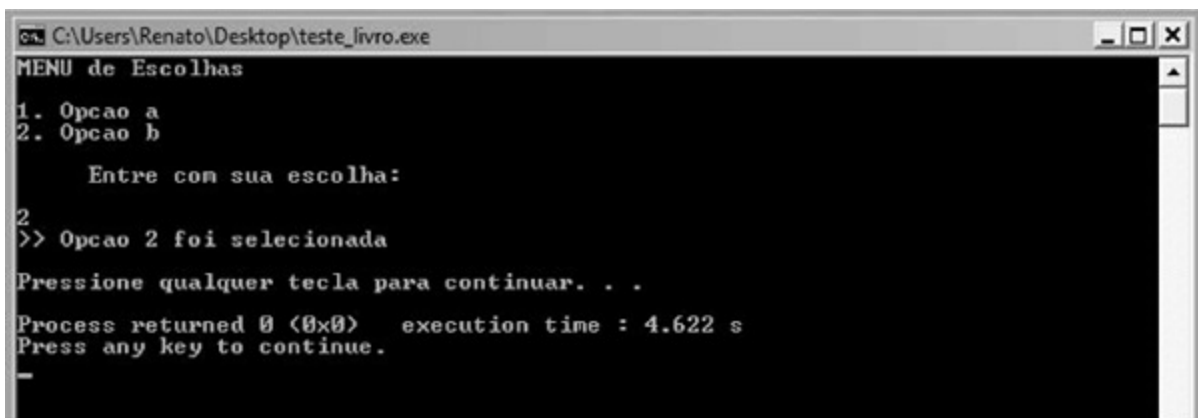
```

printf("1. Opcao a \n");
printf("2. Opcao b \n\n");
printf("Entre com sua escolha: \n\n");

ch = getchar();

switch(ch) {
    case '1':
        printf(">> Opcao 1 foi selecionada\n\n");
        break;
    case '2':
        printf(">> Opcao 2 foi selecionada\n\n");
        break;
    default:
        printf(">> Nenhuma opção selecionada\n\n");
}
system("pause");
}

```



3.3 Estruturas de decisão encadeada: heterogênea e homogênea

Temos que comentar ainda a possibilidade da ocorrência de encadeamentos dentro das estruturas de decisão, de forma a se obterem vários níveis de decisão.

Para isso utilizaremos as *estruturas de decisão encadeadas homogêneas e heterogêneas*.

Decisão encadeada, como citado, é o agrupamento de várias seleções, pois existe um número maior de possibilidades de ação.

3.3.1 Decisão encadeada heterogênea

São estruturas de decisão encadeadas que não seguem um padrão lógico.

EXEMPLO:

```

if (condição1)
{
    if (condição2)
        {comando1;}
}

```

```

else
{if (condição3)
    {comando2;}
    else
        {if (condição4)
            {comando3;}
            else
                {comando2;}}
}

```

3.3.2 *Decisão encadeada homogênea*

São estruturas de decisão que seguem um padrão lógico.

EXEMPLO 1: *se então se*

```

if (condição1) {
    if (condição2) {
        if (condição3) {
            comando;
        }
    }
}

```

EXEMPLO 2: *se senão se*

```

if (x=y) {comando1;}
else
if (x=z) {comando2;}
else
if (x=w) {comando3;}

```

OBSERVAÇÕES:

- As decisões múltiplas nada mais são do que encadeamentos homogêneos bem estruturados;
- Se o programa em execução saltar trechos de código, use logo depois das funções *scanf()* o comando *fflush(stdio)*, que limpa o *buffer* do teclado e resolve, assim, o problema.

Agora é hora de praticar tudo o que foi ensinado neste capítulo. Recomendamos que você tente programar todos os exercícios, usando um dos compiladores adotados.

EXERCÍCIOS

1. Escreva um programa que resolva o seguinte problema: uma cópia “xerox” custa R\$ 0,25 por folha, mas acima de 100 folhas esse valor cai para R\$ 0,20 por unidade. Dado o total de cópias, informe o valor a ser pago.

2. Escreva um programa que informe a categoria de um jogador de futebol, considerando sua idade: infantil (até 13 anos), juvenil (até 17 anos) ou sênior (acima de 17 anos).
3. Escreva um programa que diga qual é o maior de dois números distintos.
4. Escreva um programa que teste uma dada senha.
5. Escreva um programa que calcule a média aritmética de três números digitados pelo usuário.
6. Escreva um programa que leia três números do teclado e os imprima na tela na *ordem inversa* da entrada.
7. Escreva um programa que calcule a área de um círculo de raio r , testando se o valor do raio é menor que zero.
8. Escreva um programa que receba um número inteiro do teclado e diga se ele é par. **Dica:** Um número é par se o resto da divisão dele por 2 for zero.
9. Escreva um programa que calcule a velocidade de queda de um corpo em função do tempo, partindo da velocidade zero. **Dica:** Use equações de aceleração da Física.
10. Escreva um programa que calcule as raízes da equação do 2º grau ($ax^2 + bx + c$); os valores de a , b e c são fornecidos pelo usuário (veja a proposta de resolução mais adiante).
11. Escreva um programa que calcule o consumo médio de um automóvel; são fornecidos a distância total percorrida e o total de combustível gasto.
12. Escreva um programa que leia o nome de um aluno e as notas que ele obteve nas três provas do semestre. No final, o programa deve informar o nome do aluno e a sua média (aritmética).
13. Escreva um programa que receba um número e mostre uma mensagem caso este seja maior que 10.
14. Escreva um programa que receba um número e diga se ele está no intervalo entre 100 e 200.
15. Escreva um programa que conceda um aumento de 10% ao salário dos funcionários de uma empresa, os quais recebem até R\$ 1.000,00.
16. Desenhe um algoritmo que calcule a amplitude total (AT) de uma série de cinco números, considerando que $AT = \text{maior valor} - \text{menor valor}$.

3.4 Exemplos de programas completos

```
/* Programa de adivinhacao */

#include <stdio.h>
#include <stdlib.h>

main()
{
    int magic; /* número mágico */
    int guess; /* palpite do usuário */
    magic = rand(); /* gera número mágico */
    printf("Adivinhe o numero magico:");
```



```

scanf("%d", &guess);
if (guess == magic) {
    printf("*** Certo! ***");
    printf("%d eh o numero magico!\n", magic);
}
else {
    printf("Errado!");
    if (guess > magic) printf("Muito alto!\n");
    else printf("Muito baixo!\n");
}
system("pause");
}

```

/* Equacao do 2.o Grau */

```

#include<stdio.h>
#include<stdlib.h>
#include<math.h>

main ()
{
    float a, b, c, delta, x1, x2;

    printf ("\t\tResolucao da Equacao do Segundo Grau");
    printf ("\n\nInforme o valor de a: ");

    scanf ("%f", &a);
    printf ("Informe o valor de b: ");
    scanf ("%f", &b);
    printf ("Informe o valor de c: ");
    scanf ("%f", &c);

    delta=pow(b,2)-4*a*c;

    if(delta<0)
    {
        printf ("\nDelta e igual a %.2f", delta);
        printf ("\n\nDelta Negativo, impossivel calcular.\n\n");
    }
    else
    {
        if(delta == 0)
        {printf("Raizes iguais! \n");}
        else {printf("Raizes diferentes! \n");}
    }

    x1=(-b)+sqrt(delta)/(2*a);
    x2=(-b)-sqrt(delta)/(2*a);

    printf ("\nx1=%.2f e x2=%.2f\n\n", x1, x2);
}
system("pause");
}

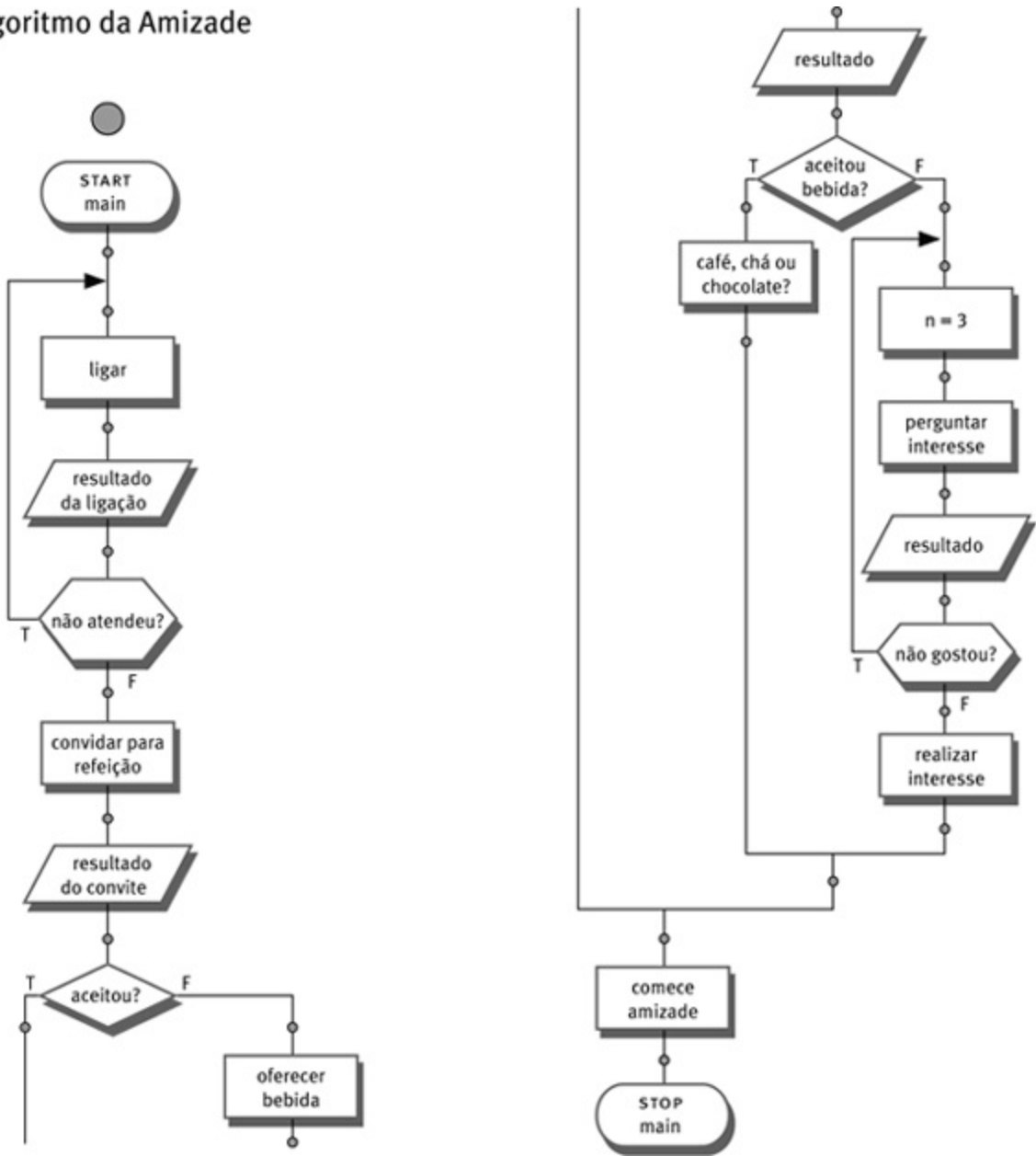
```

3.4.1 Ilustração do conteúdo – exercício desafio

O *Algoritmo da Amizade de Sheldon*, mostrado a seguir de forma adaptada, fez bastante sucesso na série *The Big Bang Theory*.¹ Escreva um programa em Linguagem C que implemente esse algoritmo.

FIGURA 3.7 Algoritmo da Amizade

Algoritmo da Amizade



Fonte: <http://www.cbs.com/shows/big_bang_theory/episodes/23381> (adaptado pelo autor).

PROPOSTA DE SOLUÇÃO

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

main()
{
    char casa, refeicao, bebida, compartilho;
    char interesse[30];
    int tipo;

    do {
```

```

printf("Alguem em casa? s/n\n");
scanf("%c", &casa);
fflush(stdin);
if (casa == 's'){ break;}
printf("Deixar mensagem ...\n");
printf("Esperar pelo retorno da pessoa ...\n");
system("pause");

}while (casa != 's');

printf("Gostaria de compartilhar a refeicao? s/n\n");
scanf("%c", &refeicao);
if (refeicao == 's'){
    printf("Vamos jantar...\n");
    printf("Inicio da amizade!!!\n");
    system("pause");
    exit(1);}
else {
    printf("Gostaria de uma bebida quente? s/n\n");
    fflush(stdin);
    scanf("%c", &bebida);

    if (bebida == 's'){
        printf("1. Cha \n");
        printf("2. Cafe \n");
        printf("3. Chocolate \n");

        fflush(stdin);
        tipo = getchar();

        switch(tipo){
            case '1':
                printf("cha...\n");
                break;
            case '2':
                printf("cafe...\n");
                break;
            case '3':
                printf("chocolate...\n");
                break;
            default:
                printf("opcao nao eh valida!\n");
                break;
        }
        printf("Inicio da amizade!!!\n");
        system("pause");
        exit(1);
    }

    else {
        printf("Conte-me um de seus interesses!\n");
        scanf("%s", interesse);
        printf("O interesse eh: %s\n", interesse);
        system("pause");}

    printf("Compartilho deste interesse?\n");
    fflush(stdin);

```

```

scanf("%c", &compartilho);
if (compartilho == 's'){
    printf("Por que nao fazemos aquilo juntos?\n");
    system("pause");}
else {
    printf("Nao compartilho este interesse...\n");
    system("pause");
    exit(1);
}

printf("Inicio da amizade!!!\n");
system("pause");
}

```

REFERÊNCIAS BIBLIOGRÁFICAS

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

FORBELLONE, A; EBERSPACHER, H. *Lógica de programação – a construção de algoritmos e estrutura de dados*. 3.ed. São Paulo: Prentice Hall, 2005.

MIZRAHI, Victorine Viviane. *Treinamento em Linguagem C*. 2.ed. São Paulo: Pearson/ Prentice Hall, 2008.

MANZANO, José Augusto Navarro Garcia; OLIVEIRA, Jayr Figueiredo de. *Algoritmos: lógica para desenvolvimento de programação de computadores*. 22.ed. São Paulo: Érica, 2009.

RITCHIE, Dennis; KERNIGHAN, Brian. *C: a linguagem de programação*. São Paulo: Campus, 1986.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

ANEXO: A LINGUAGEM DE PROGRAMAÇÃO C

The only way to learn a new programming language is by writing programs in it.

DENNIS RITCHIE

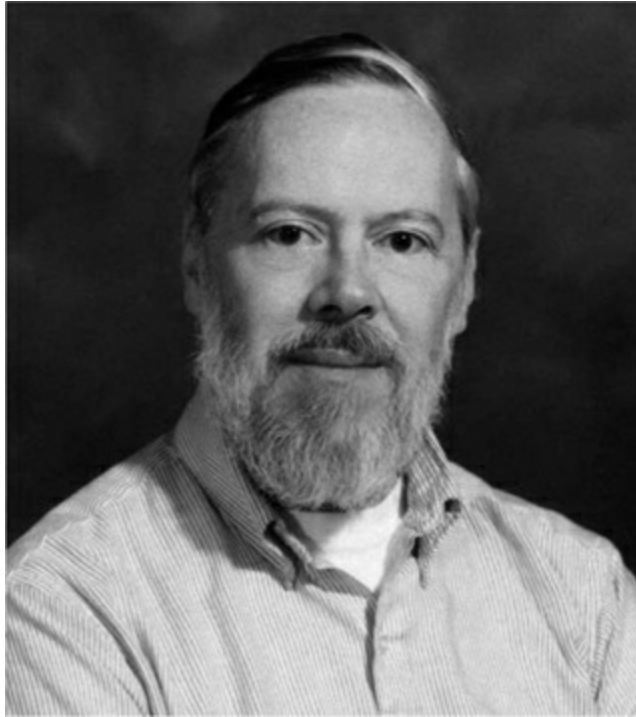
Criação de Dennis Ritchie, dos Laboratórios Bell, a Linguagem C é, de longe, a mais influente linguagem de programação moderna (mesmo tendo sido concebida e implementada na década de 1970). Ela é a base de criação de outras linguagens, aplicativos e até mesmo sistemas operacionais (daí a relação tão estreita entre a Linguagem C e o UNIX).

Dennis MacAlistair Ritchie (1941-2011), cientista da computação, participou de forma marcante no desenvolvimento das linguagens de programação B, BCPL e C, e em sistemas operacionais como o Multics e o UNIX. Físico e matemático de formação, fez carreira nos Laboratórios Bell. É ganhador, junto com Ken Thompson, do famoso Prêmio Turing.

Dennis Ritchie faleceu em 2011, na mesma época em que Steve Jobs. Pouco se falou dele naquele momento, o que consideramos uma injustiça. De minha parte, comentei com meus alunos sua vida, sua obra magna (a Linguagem C), seu papel no desenvolvimento do UNIX e sua influência na computação.

Fica, aqui registrada, nossa homenagem.

FIGURA 3A.1 Dennis Ritchie (1941-2011).



Fonte: <<http://www.computerhistory.org/fellowawards/hall/bios/Dennis,Ritchie/>>.

Estruturas de controle de programação – repetição



VISÃO DO CAPÍTULO

Aprenderemos neste capítulo o necessário sobre *estruturas de repetição*. Elas são de grande importância para os algoritmos computacionais, pois permitem que ações prévias possam ser repetidas toda vez que for necessário. Isso é o que confere grandes poderes a um computador, tornando-o capaz de repetir inúmeras operações e ações de forma rápida e eficiente.

OBJETIVO INSTRUCIONAL

Após estudar e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- « Entender a importância das repetições no processo de programação de computadores;
- « Discernir entre os diferentes tipos de estruturas de repetição (com teste no início, com teste no final e com variável de controle);
- « Resolver exercícios e praticar a repetição em Linguagem C (*while*, *do ... while*, *for*)



As repetições, assim como as decisões, são parte integrante do nosso cotidiano, pois estamos a todo momento repetindo ações, mesmo que de forma inconsciente.

O mesmo acontece em computação, quando um programa de computador ou algoritmo precisa repetir ações de acordo com as necessidades do problema trabalhado pelo programador.

Iniciemos nosso estudo sobre as repetições, já apresentando as diversas possibilidades de realizá-las.

4.1 Estruturas de repetição com teste no início

Pensemos em um *caixa eletrônico*: seria muito estranho se um deles realizasse operações bancárias sem testar nossa senha antes. Podemos dizer, portanto, que esse sistema implementa a repetição com *teste no início*, quando verifica nossa senha antes de realizar a operação, e depois nos pergunta se queremos continuar as operações. Ou seja, o sistema não executará nenhuma repetição (e as ações que ali dentro estiverem programadas) sem antes testar uma condição.

Ou seja, *enquanto* uma condição se verificar, repetiremos as ações desejadas:

```
while (condição)
```

Os conceitos adicionais importantes a serem considerados, quando trabalhamos repetições, são os de: a) *contador*; b) *incremento*; e c) *acumulador*. Podemos considerar importante, também, o chamado *critério de parada*.

O *contador* controla o número de repetições, quando sabemos quantas são.

O *incremento* (ou decremento, se for o caso) aumenta (ou reduz) o valor do contador, a fim de que o número de repetições desejado seja alcançado – caso contrário, entraríamos em uma repetição eterna (*loop infinito*), que geraria um resultado indesejável para o nosso programa.

O *acumulador*, como o nome diz, vai somando as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.

E, finalmente, o *critério de parada* é a condição de encerramento da repetição, quando *não* sabemos quantas são.

Os exemplos a seguir ilustram todos esses conceitos.

EXEMPLO 1:

Escreva cinco vezes a expressão “FATEC” na saída-padrão (tela).

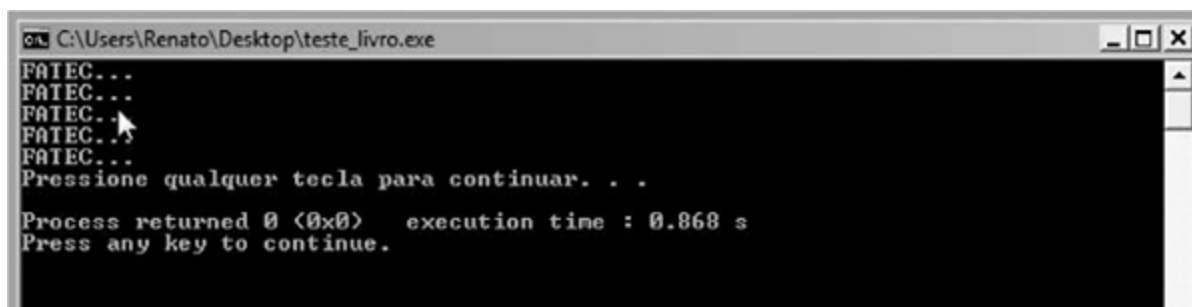
```
#include <stdio.h>

#include <stdlib.h>

main( )
{
    int contador=0; // aqui criamos o contador de repetições, com valor inicial de
    while (contador < 5) // enquanto a contagem for menor que 5 (ou seja, de 0 a 4
    {
        printf("FATEC... \n");

        contador++; // o contador incrementa de valor, a fim de passarmos adiante
    }

    system("PAUSE");
}
```



EXEMPLO 2:

Escreva um programa que apresente uma contagem regressiva de 10 a 0.

```
#include <stdio.h>

#include <stdlib.h>

main()
{
    int cont=10;

    while (cont >= 0)

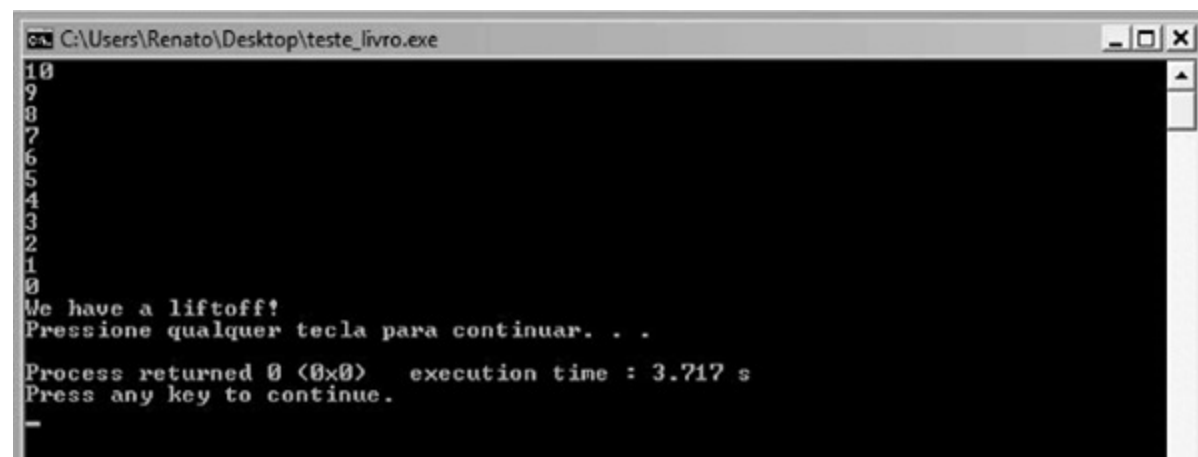
    {
        printf("%d \n", cont);

        cont--;

    }
    printf("We have a liftoff! \n");

    system("PAUSE");

}
```



```
C:\Users\Renato\Desktop\teste_livro.exe
10
9
8
7
6
5
4
3
2
1
0
We have a liftoff!
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0) execution time : 3.717 s
Press any key to continue.
```

EXEMPLO 3:

Escreva um programa que imprima “FATEC” na tela enquanto o caractere “q” não for digitado pelo usuário (portanto, condição de saída com caractere) e conte quantas repetições foram realizadas.

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



```

#include <stdlib.h>

main()
{
    int cont=0; // contador inicializado com valor zero - sempre recomendável

    char letra;

    while (letra=getchar() != 'q') // função getchar( ) captura o caractere digitado
    {

        printf("FATEC... \n");

        fflush(stdin);

        cont++; // incrementando o contador

    }

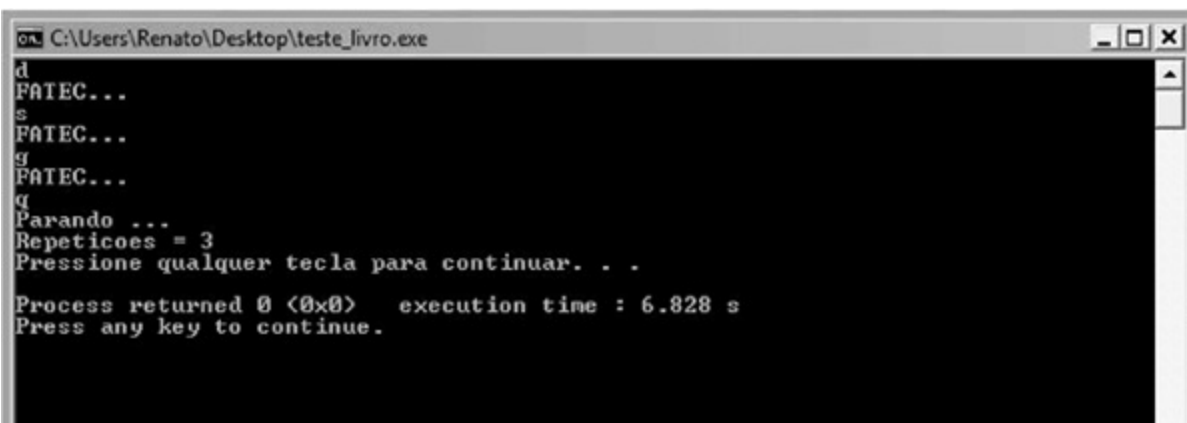
    printf("Parando ... \n");

    printf("Repeticoes = %d \n", cont); // mostrando o número de repetições efetuadas

    system("PAUSE");

}

```



```

C:\Users\Renato\Desktop\teste_livro.exe
d
FATEC...
s
FATEC...
g
FATEC...
q
Parando ...
Repeticoes = 3
Pressione qualquer tecla para continuar. . .

Process returned 0 (0x0)   execution time : 6.828 s
Press any key to continue.

```

EXEMPLO 4:

Escreva um programa de repetição que assuma uma condição de saída com caractere e acumule os valores inseridos em cada iteração da repetição.

```

#include <stdio.h>

```

```

#include <stdlib.h>

main()
{
    int x; int cont=0; int total=0;

    char letra;

    while (letra=getchar() != 'q')
    {
        printf("Digite um numero: \n");

        scanf("%d", &x);

        fflush(stdin);

        cont++;

        total = total + x;
    }

    printf("Parando ... \n");

    printf("Repeticoes = %d e total = %d. \n", cont, total);

    system("PAUSE");

}

```

```

C:\Users\Renato\Desktop\teste_livro.exe
e
Digite um numero:
4
f
Digite um numero:
4
q
Parando ...
Repeticoes = 2 e total = 8.
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0) execution time : 12.014 s
Press any key to continue.

```

4.2 Estruturas de repetição com teste no final

Neste caso, as ações ou comandos são executados pelo menos uma vez antes do teste de condição. É útil, por exemplo, para se garantir consistência da entrada de dados ou repetir um processo com

confirmação do usuário.

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

Como exemplo, podemos dar uma nova oportunidade para o usuário de um programa que exige uma entrada de dados diferente de zero, supondo que ele tenha digitado o valor proibido:

```
do  
  
{  
  
    printf("Digite o valor: \n");  
  
    scanf("%f", &a);  
  
}while (a == 0);
```

EXEMPLO 1:

Escreva um programa que repita uma entrada de dados até que determinada condição de saída seja atingida e, em seguida, acumule os valores digitados.

```
#include <stdio.h>  
  
#include <stdlib.h>  
  
#include <string.h>  
  
main()  
{  
    int x; int cont=0; int total=0;  
  
    char letra;  
  
    do  
  
    {  
        printf("Digite um numero: \n");  
  
        scanf("%d", &x);  
  
        fflush(stdin);
```

```

    cont++;

    total = total + x;

    printf("Digite uma letra: \n");

} while (letra=getchar() != 'q');

printf("Parando ... \n");

printf("Repeticoes = %d e total = %d \n", cont, total);

system("PAUSE");

}

```

EXEMPLO 2:

Escreva um programa que receba as notas de um aluno até que a condição de saída se confirme e que, em seguida, calcule a média do aluno.

```

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

main()
{
    int x; int cont=0; int total=0;

    char letra;

    float media;

    do

```

```

{

    printf("Digite uma nota: \n");

    scanf("%d", &x);

    fflush(stdin);

    cont++;

    total = total + x;

    printf("Digite uma letra qualquer ou \'q\' para encerrar: \n");

}while (letra=getchar() != 'q');

printf("Parando ... \n");

printf("Repeticoes = %d e total das notas = %d. \n", cont, total);

media = total/cont;

printf("Media = %f. \n", media);

system("PAUSE");

}

```

```

C:\Users\Renato\Desktop\teste_livro.exe
Digite uma nota:
6
Digite uma letra qualquer ou 'q' para encerrar:
e
Digite uma nota:
8
Digite uma letra qualquer ou 'q' para encerrar:
q
Parando ...
Repeticoes = 2 e total das notas = 14.
Media = 7.000000.
Pressione qualquer tecla para continuar. . .

Process returned 0 (0x0)   execution time : 17.372 s
Press any key to continue.

```

EXEMPLO 3:

Escreva um programa que teste a senha de um cliente por três vezes no máximo; se ele errar esse número de chances, o programa deve avisar da necessidade de procurar a agência do banco.

```

#include <stdio.h>

#include <stdlib.h>

main()

```

```
{

    int cont=1; char senha;

    do

    {

        printf("Digite a senha: \n");

        scanf("%c", &senha);

        fflush(stdin);

        cont++;

        if(senha==' f ')

        {

            printf("Certa! \n");

            break;

        }

        if(cont > 3)

        {

            printf("Senha incorreta - procure a agencia mais proxima. \n");

            system("PAUSE");

            exit(1);

        }

    } while (1); // enquanto for verdade

    system("PAUSE");

}
```

```
C:\Users\Renato\Desktop\teste_livro.exe
Digite a senha:
e
Digite a senha:
r
Digite a senha:
f
Certa!
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 8.031 s
Press any key to continue.
-
```

```
C:\Users\Renato\Desktop\teste_livro.exe
Digite a senha:
g
Digite a senha:
h
Digite a senha:
j
Senha incorreta - procure a agencia mais proxima.
Pressione qualquer tecla para continuar. . .
Process returned 1 (0x1)   execution time : 4.874 s
Press any key to continue.
-
```

EXEMPLO 4:

Escreva um programa que simule uma conta bancária.

```
/* Caixa eletronico */

#include <stdio.h>

#include <stdlib.h>

main()
{
    float s=1000.00, v;

    int op;

    do {

        printf("\n 1. Deposito");

        printf("\n 2. Saque");

        printf("\n 3. Saldo");

        printf("\n 4. Sair");

        printf("\n Opcao? ");

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

```
Switch(op)
```

```
{
```

```
    case 1: printf("\n Valor do deposito? ");
```

```
    scanf("%f", &v);
```

```
    s=s+v;
```

```
    break;
```

```
    case 2: printf("\n Valor do saque? ");
```

```
    scanf("%f", &v);
```

```
    s=s-v;
```

```
    break;
```

```
    case 3: printf("\n Saldo atual = R$ %.2f \n", s);
```

```
    break;
```

```
    default: if(op!=4) printf("\n Opcao Invalida! \n");
```

```
    }
```

```
    } while (op!=4);
```

```
printf("Fim das transacoes. \n\n");
```

```
system("pause");
```

```
}
```



```
C:\Users\Renato\Desktop\teste_livro.exe

1. Deposito
2. Saque
3. Saldo
4. Sair
Opcao? 2

Valor do saque? 400

1. Deposito
2. Saque
3. Saldo
4. Sair
Opcao? 1

Valor do deposito? 500

1. Deposito
2. Saque
3. Saldo
4. Sair
Opcao? 4
Fin das transacoes.

Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 18.706 s
Press any key to continue.
-
```

EXEMPLO 5:

Escreva um programa que gere na tela a Tabela ASCII.

```
/* Gerando a Tabela ASCII */

# include <stdio.h>

#include <stdlib.h>

main()

{

    int a=0;

    printf("Gerando Tabela ASCII ... \n\n");

    do{

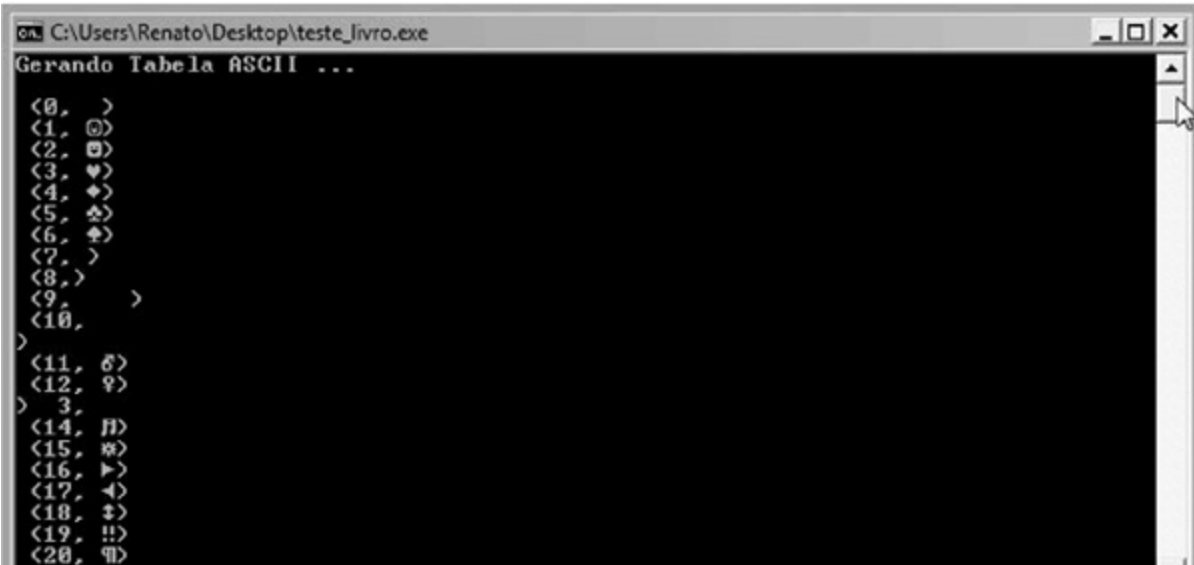
        printf(" (%d, %c) \n", a,a); // aqui imprimimos o valor inteiro do caracter

        a++; // próximo caractere

    } while(a<=255); // 255 é o último caractere da Tabela ASCII

    system("pause");

}
```



```
C:\Users\Renato\Desktop\teste_livro.exe
Gerando Tabela ASCII ...

<0, >
<1, @>
<2, @>
<3, ♥>
<4, ♦>
<5, ♠>
<6, ♠>
<7, >
<8, >
<9, >
<10, >
>
<11, ♂>
<12, ♀>
> 3,
<14, ♀>
<15, ♂>
<16, ♀>
<17, ♀>
<18, ♀>
<19, !!>
<20, ♀>
```



```
C:\Users\Renato\Desktop\teste_livro.exe

<47, \>
<48, @>
<49, 1>
<50, 2>
<51, 3>
<52, 4>
<53, 5>
<54, 6>
<55, ?>
<56, 8>
<57, 9>
<58, :>
<59, ;>
<60, <>
<61, =>
<62, >>
<63, ?>
<64, @>
<65, @>
<66, B>
<67, C>
<68, D>
<69, E>
<70, F>
<71, G>
```

4.3 Estruturas de repetição com variável de controle

Aqui utilizaremos uma *variável* que controlará as repetições; a estrutura de repetição terá uma condição inicial, uma final e o incremento dessa variável.

```
for(inicialização; condição final; incremento)
{
    ações;
}
```

INFORMAÇÕES ADICIONAIS:

O *for* pode conter vírgulas e testar mais de uma variável:

```
for(i=0, j=1; (i + j) < 100; i++, j++); printf ("%d", i + j);
```

a) Pode ser usado com caracteres:

```
for(ch='a'; ch <= 'z'; ch++); printf ("\n O valor ASCII de %c eh %d", ch, ch);
```

b) Pode também usar funções em sua estrutura:

```
for(ch=getch(); ch!= 'X'; ch=getch()); printf ("%c", ch + 1);
```

c) Pode ser utilizado para gerar *loops infinitos*:

```
for( ; ; ); printf ("FATEC");
```

EXEMPLO 1:

Escreva um programa que teste três vezes uma senha.

```
/* Testando uma senha */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

main()
{

    char senha[20]; // vetor de caracteres para receber a senha

    int x;

    for(x=1; x<=3; ++x)

    {

        printf("Digite a senha: ");

        gets(senha); // função gets( ) recebe cadeias de caracteres - strings

        if (strcmp(senha, "passwd") == 0) // comparando se a senha é correta

        {

            printf("OK! \n");

            system("pause");

            break;

        }

    }
```

```

else

{

    printf("Tente novamente!\n");

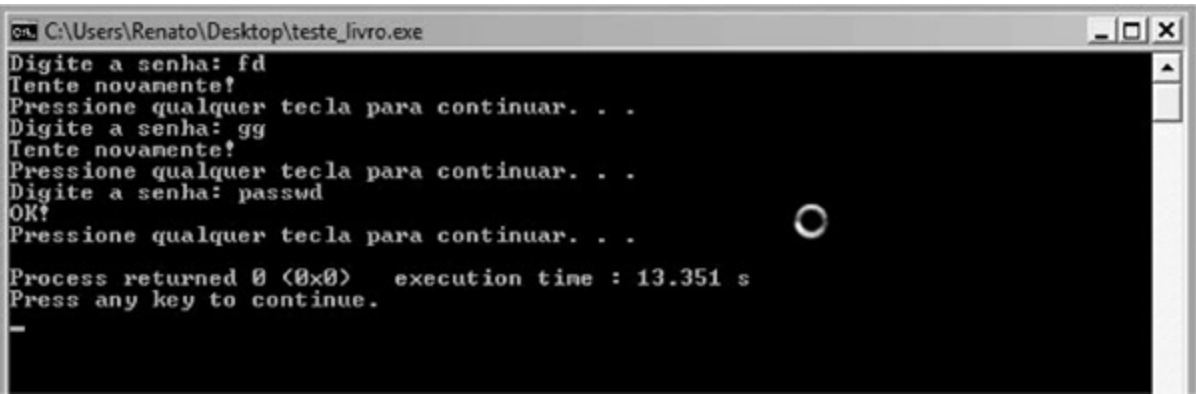
}

system("pause");

}

}

```



EXEMPLO 2:

Escreva um programa que converta caracteres minúsculos para maiúsculos.

```

/* Convertendo caracteres minúsculos para maiúsculos */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

main()

{

    char min, mai;

    int cont = 0;

    for(cont=1; cont<=10; cont++ )

    {

        min = getchar();

```

```

    mai = toupper(min);

    putchar(mai);

}

system("pause");

}

```

EXEMPLOS ADICIONAIS DE REPETIÇÃO:

```

/* Exemplo de rand(): adivinhacao com repeticao */

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

main ()
{
    int iSecret, iGuess;

    /* inicializando semente randomica: */

    srand ( time(NULL) );

    /* gerando numero secreto: */

    iSecret = rand() % 10 + 1; // rand( ) gera numero aleatorio entre 0 e RAND_MAX

    do {

        printf ("Adivinhe o numero (1 a 10): ");

        scanf ("%d",&iGuess);
    }

```

```

    if (iSecret<iGuess)

        puts ("O numero secreto eh menor!");

    else

        if (iSecret>iGuess) puts ("O numero secreto eh maior!");

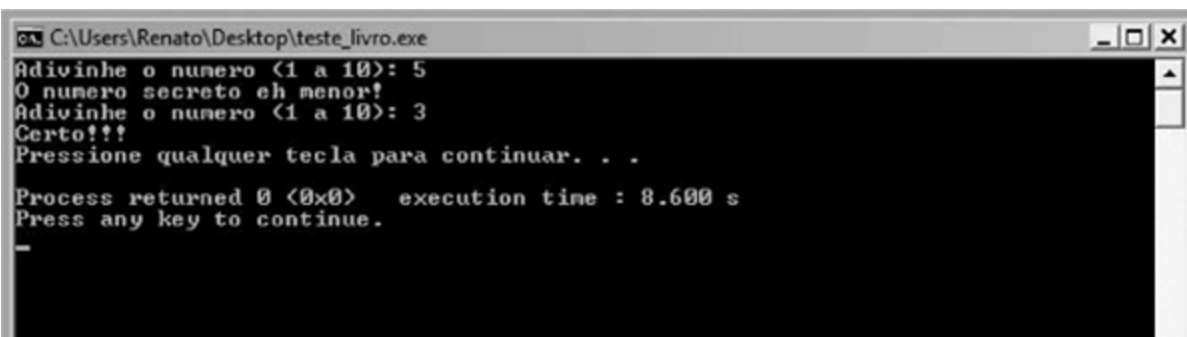
    } while (iSecret!=iGuess);

puts ("Certo!!!");

system("pause");

}

```



```

C:\Users\Renato\Desktop\teste_livro.exe
Adivinhe o numero <1 a 10>: 5
O numero secreto eh menor!
Adivinhe o numero <1 a 10>: 3
Certo!!!
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 8.600 s
Press any key to continue.
_

```

```

/* Geracao de numeros aleatorios */

#include <stdio.h>

#include <stdlib.h>

main()

{

    int i;

    printf("Valores de rand \n");

    for(i=0; i<100; i++)

    {

        printf("%d", rand());

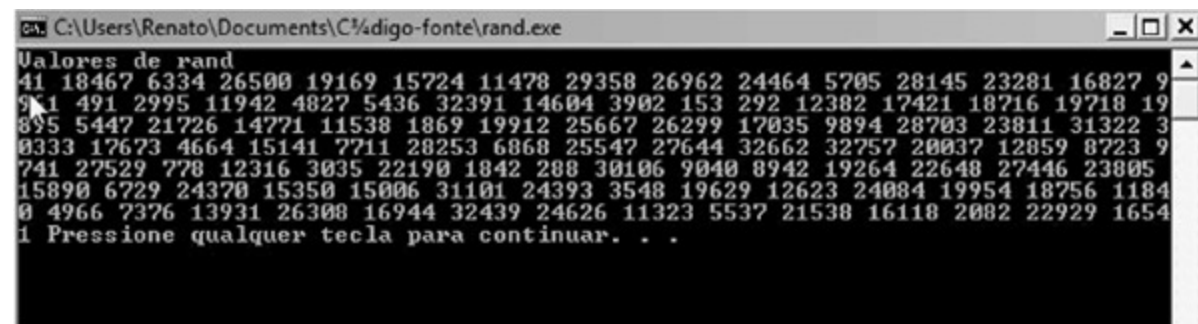
    }

    system("pause");

}

```

Tela de saída do exercício anterior:



REPETIÇÕES ENCAIXADAS

Estruturas de repetição podem estar subordinadas umas às outras (encaixadas) para formar estruturas mais complexas. Esse assunto será tratado em detalhes nas seções de estruturas de dados, mais adiante.

Vejamos o exemplo a seguir:

```
for(i=0; i<=20; i++)
{
    for(j=0; j<=30; j++)
    {
        printf("%c \n", caractere);
    }
}
```

Esse programa vai formar uma matriz de caracteres.

COMANDOS DE DESVIO

São utilizados para a saída ou encerramento de determinado processamento, antes que ele seja encerrado.

São eles:

- **return**: retorna a uma função (será visto mais adiante).

- **goto**: desvia para outro trecho do programa.

```
goto rótulo;
...
rótulo:
```

- **break**: termina um *case* em um *switch*, ou força o término imediato de um laço.

```
/* Exemplo de "break" com "for" */

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

```

main()
{
    int x;
    for ( x = 1; x <= 10; x++ ) {
        if ( x == 5 ) {
            break;
        }
        printf( "%d ", x );
    }
    printf( "\nSaiu do loop quando x == %d\n", x );
    system("pause");
}

```

■ **exit (código de retorno)**: serve para sair de um programa em execução.

■ **continue**: força a próxima iteração do laço, pulando o código intermediário.

```

/* Exemplo de "continue" com "for" */

#include <stdio.h>
#include <stdlib.h>
main()
{
    int x;
    for ( x = 1; x <= 10; x++ ) {
        if ( x == 5 ) {
            continue;
        }
        printf( "%d ", x );
        system("pause");
    }
    printf( "\nUsando o continue para evitar o valor 5\n" );
    system("pause");
}

```

EXERCÍCIOS

1. Escreva um programa de controle de senha que dê três chances de acerto ao usuário; se ele não conseguir, o programa deve avisar que o cartão foi bloqueado.
2. Use o comando *continue* em um programa de divisão que respeite a restrição de divisão por zero.
3. Escreva um programa que leia cinco valores e conte quantos desses valores são negativos, mostrando essa informação no final.
4. A prefeitura de uma cidade fez uma pesquisa entre seus habitantes, coletando dados sobre o salário e número de filhos. A prefeitura deseja saber:
 - a) média do salário da população;
 - b) média do número de filhos;
 - c) maior salário;

d) percentual de pessoas com salário até R\$ 100,00.

O final da leitura de dados se dará com a entrada de um salário negativo.

5. Escreva um programa que calcule a média dos números digitados pelo usuário, se eles forem pares. O programa deve terminar a leitura se o usuário digitar zero.
6. Escreva um programa que solicite ao usuário três números inteiros a , b e c , em que a seja maior que 1. O programa deve somar todos os inteiros entre b e c que sejam divisíveis por a .
7. Escreva um programa que leia três números do teclado e os imprima na tela na *ordem inversa* da entrada.
8. Escreva um programa que calcule a área de vários círculos de raio r , até que o usuário digite “n”.
9. Escreva um programa que imprima na tela o caractere (ASCII) correspondente a determinado número decimal entre 50 e 75.
10. Escreva um programa que receba dez números inteiros do teclado e diga quantos são pares e quantos são ímpares.
11. Escreva um programa que sorteie os números necessários para a Mega Sena. Dica: usar a geração de números aleatórios com *rand()*.

REFERÊNCIAS BIBLIOGRÁFICAS

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

FORBELLONE, A; EBERSPACHER, H. *Lógica de programação – a construção de algoritmos e estrutura de dados*. 3.ed. São Paulo: Prentice Hall, 2005.

MANZANO, José Augusto Navarro Garcia; OLIVEIRA, Jayr Figueiredo de. *Algoritmos: lógica para desenvolvimento de programação de computadores*. 26. ed. São Paulo: Érica, 2012.

MIZRAHI, Victorine Viviane. *Treinamento em Linguagem C*. 2.ed. São Paulo: Pearson/ Prentice Hall, 2008.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

ANEXO: DE HACKERS E NERDS

Existe um vínculo forte entre a imagem de um programador de computadores e os termos *hacker* e *nerd*.

O MIT (Massachusetts Institute of Technology) é considerado o lar dessa tradição. Um *hacker*, ao contrário do que se propagou pela mídia – e de forma incorreta – não é necessariamente um fora da lei, mas alguém que deseja conhecer profundamente determinado objeto de estudo, como um dispositivo eletrônico ou um computador. Já um *nerd* é aquele estereótipo do sujeito estudioso, antissocial e inteligente.

O primeiro *hacker* certamente foi Marvin Minsky, criador do campo da Inteligência Artificial,

junto com John McCarthy. Foi sucedido por Richard Stallman, seu discípulo, no famoso Laboratory for Computer Science do MIT. Stallman foi o criador do movimento do *software livre*.

A cultura *hacker* permanece no MIT, e também na forma de brincadeiras anônimas pelo *campus*, mas que seguem regras estabelecidas de respeito às pessoas e ao ambiente: são os *hacks*.

O *hack* é, portanto, uma brincadeira de alunos que provoca a admiração da comunidade acadêmica, pelo planejamento e execução únicos nas dependências do Instituto. Os principais *hacks* praticados pelos alunos do MIT estão catalogados, dada a importância que esse assunto tem para a própria cultura da instituição.

Para informações detalhadas sobre o assunto, e para conhecer os hacks mais famosos, visite <http://hacks.mit.edu/>.

FIGURA 4A.1 Hacks.



Fonte: <http://hacks.mit.edu/>.

FIGURA 4A.2 O *campus* do MIT, em Cambridge (Massachusetts).



Variáveis indexadas: vetores e matrizes



VISÃO DO CAPÍTULO

A partir deste capítulo adentramos o tema das *estruturas de dados*. São importantes quando se considera a necessidade de preservação de dados em dispositivos de armazenamento. Variáveis indexadas, ao contrário das variáveis tradicionais, armazenam mais de um valor de mesmo tipo, e são úteis para a manipulação de séries de valores semelhantes, sejam elas uni ou multidimensionais. Os vetores e as matrizes são seus representantes clássicos, e a eles daremos atenção nas próximas linhas.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- ◀◀ Conhecer as estruturas de dados baseadas em variáveis indexadas;
- ◀◀ Aprender a manipular vetores e matrizes;
- ◀◀ Trabalhar o tipo de dados não primitivo chamado de *string* (cadeia de caracteres);
- ◀◀ Exercitar o assunto com base nas tarefas propostas.



5.1 Vetores e *strings*

Um vetor é uma estrutura que armazena vários dados de mesmo tipo, ao contrário das variáveis comuns, que só podem armazenar um valor de cada vez. Em programação, é das estruturas mais simples.

Os elementos individuais são acessados por sua posição dentro do vetor. A posição é dada pelo chamado *índice*, que, em geral, utiliza uma sequência de números inteiros, que são acessados de forma rápida e eficiente.

O vetor é declarado da seguinte forma:

```
tipo_dos_elementos nome_do_vetor [número de elementos];
```

Por exemplo:

```
int numero[10]; // declara um vetor de números inteiros de 10 elementos
```

O vetor é, assim, uma sequência de memória, que poderia ter a seguinte analogia representativa:

FIGURA 5.1 Um vetor como trecho de memória.

| | |
|--|-----------|
| | |
| | 15 A[3,3] |
| | 14 A[2,3] |
| | 13 A[1,3] |
| | 12 A[0,3] |
| | 11 A[3,2] |
| | 10 A[2,2] |
| | 9 A[1,2] |
| | 8 A[0,2] |
| | 7 A[3,1] |
| | 6 A[2,1] |
| | 5 A[1,1] |
| | 4 A[0,1] |
| | 3 A[3,0] |
| | 2 A[2,0] |
| | 1 A[1,0] |
| | 0 A[0,0] |
| | |

Outros exemplos:

- `int vetor[5] = {1,2,3,4,5};` // declaração de vetor com carga inicial automática
- `char vogal[5] = {'a', 'e', 'i', 'o', 'u'};` // idem
- `char titulo[] = "The Raven and Other Poems by Edgar Allan Poe";` /* embora não tenha declarado o tamanho inicial do vetor, vai alocar tais caracteres */
- `char nome[100] = "FATEC";` /* vai alocar 100 caracteres, mas inicializa com 5 elementos */

Os elementos do vetor, como visto, são referenciados pelo *índice*, que começa em *zero*: assim, `vetor[0]` mostrará o valor do primeiro elemento da estrutura.

Tais elementos costumam ser acessados percorrendo-se a estrutura, item a item, por meio de uma rotina semelhante a essa:

```
void percorrer_vetor( )
{
    int i;
    for(i=0; i<10; i++)
    {
        s[i]=8; // vamos aqui preencher as posições do vetor com o número 8
    }
}
```

Exemplo de uso de vetor:

```
// Usando vetores na conversao de maiúsculas para minúsculas

#include <stdio.h>

#include <conio.h>

#include <stdio.h>

main()
{

char st[40] = "HELLO";

int i;

for(i=0; st[i]; i++)

{

st[i] = tolower(st[i]);

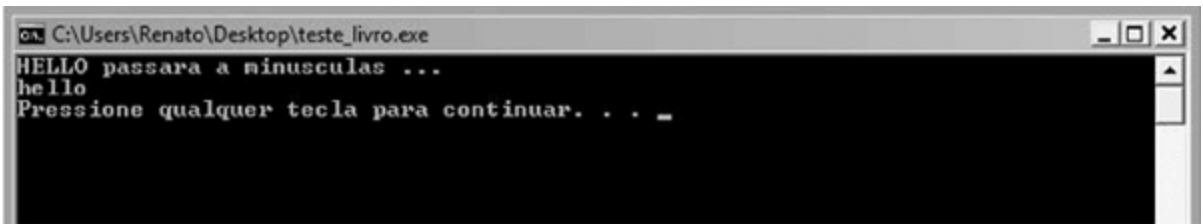
}

printf("HELLO passará a minúsculas ... \n", st); // mostra a string em minúscula

printf("%s \n", st); // mostra a string em minúsculas

system("pause");

}
```



5.1.1 *Passagem de vetores para funções*

Sempre que declaramos um vetor, seu espaço é alocado continuamente em memória. A posição que ele ocupa em memória pode ser obtida pelo nome do vetor, que contém o endereço do primeiro elemento.

Quando invocamos uma função (assunto que, mais adiante, será visto em detalhes) e lhe passamos o vetor como parâmetro, na realidade a função não recebe o vetor na sua totalidade, mas apenas o endereço inicial do vetor, pois estamos passando, por exemplo, `s = &s[0]` (endereço de memória do primeiro elemento do vetor `s`).

Se passamos um endereço, então a variável que o recebe terá que ser um ponteiro (variável que armazena endereços de memória, assunto que também veremos adiante) para o tipo dos elementos do vetor. É por essa razão que no cabeçalho de uma função, que recebe um vetor como argumento, aparece normalmente um ponteiro recebendo o respectivo parâmetro.

Como exemplo, a própria função *main()*:

```
int main(int argc, char * argv[])
```

Se tivermos o seguinte vetor:

```
int v[10];
```

Se quisermos inicializar esse vetor com um determinado valor em todos os elementos, poderíamos passá-lo como argumento de uma função que realizaria a seguinte tarefa:

```
#include <stdio.h>

#include <stdlib.h>

void inicializa(int s[10])
{
    int i;

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

        {
            s[i]=0; // vamos preencher as posições do vetor com '0'

        }
}

void mostra(int s[10])
{
    int i;

    printf("O vetor ficou assim: \n"); // vamos mostrar as posições do vetor

    for(i=0; i<10; i++)
    {
        printf("| %d ", s[i]);

    }

    printf("|");

    printf("\n \n");
}

main( )
{

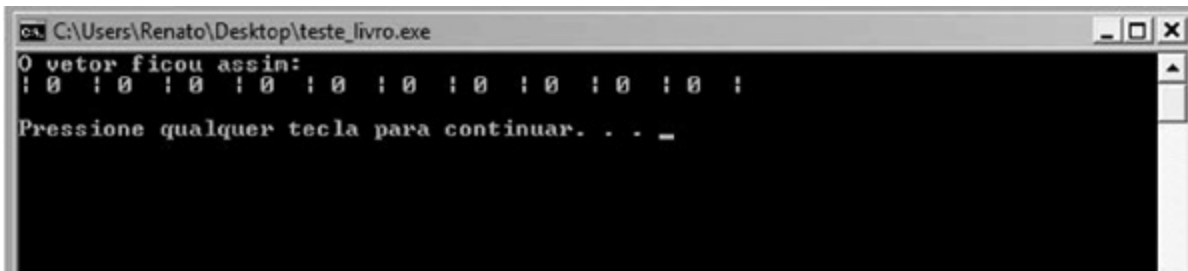
    int v[10];

    inicializa(v);
```

```
mostra(v);
```

```
system("pause");
```

```
}
```



5.1.2 Strings

A linguagem C, como visto anteriormente, não determina um tipo específico para a manipulação de *strings* (que são vetores ou cadeias de caracteres, terminados pelo caractere NULL). Para isso, fornece uma completa biblioteca de funções específicas (*string.h*).

Funções que manipulam *strings* a percorrem até encontrar o caractere NULL, quando saberão que ela terminou. Utilizamos, portanto, o caractere *zero* da tabela ASCII (‘\0’) para encerrar a *string*, e este ocupa um espaço que deve ser previsto pelo programador, como visto anteriormente.

A relação entre *strings* e vetores é, dessa forma, direta. Uma *string* é um vetor de caracteres, mas nem todo vetor de caracteres é uma *string*.

Declarando e inicializando strings

Vamos declarar uma *string* que poderá armazenar até 20 caracteres (incluindo o NULL). Usaremos inicialmente apenas cinco posições:

```
char frase[20] = "Teste"; // usando 5+1 posições
```

Eis um programa que pode ilustrar essa questão:

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

```
#include <stdlib.h>
```

```
main()
```

```
{
```

```
    char frase[50];
```

```
    int i;
```

```
    for(i=0; i < 30; i++)
```

```
{
```

```

        frase[i] = 'A' + i; /* a variável 'i' incrementa a posição do caractere n
    }

    frase[i] = NULL;

    printf("A string contem %s \n", frase);

    system("pause");
}

```

A saída do programa seria,



```

C:\Users\Renato\Documents\C3\4digo-fonte\string1.exe
A string contem ABCDEFGHIJKLMNOPQRSTUVWXYZ
Pressione qualquer tecla para continuar. . .

```

Se o mesmo programa forçasse o elemento frase[10] a ser NULL, a *string* pararia na letra J.



```

C:\Users\Renato\Documents\C3\4digo-fonte\string1.exe
A string contem ABCDEFGHIJ
Pressione qualquer tecla para continuar. . .

```

Podemos, agora, diferenciar *caracteres* de *strings*, ou seja, ‘A’ é diferente de “A”. ‘A’ é o caractere simples, enquanto “A” significa o caractere simples mais \0 (NULL). Assim, *aspas simples* indicam um caractere, mas *aspas duplas* indicam uma cadeia de caracteres (*string*).

A seguir temos um vetor de caracteres que não é uma *string*:

```

char nome[ ] = {'A', '\n', 'a'}; /* não possui o marcador de final de string; estes
caracteres serão utilizados de forma separada */

```

Neste próximo exemplo, determinaremos o comprimento da *string*:

```

#include <stdio.h>

#include <stdlib.h>

main()
{
    char string[50];

    int i;

    printf("Digite um conjunto de caracteres: \n");

    gets(string);
}

```



```

for(i=0; string[i] != NULL; i++)

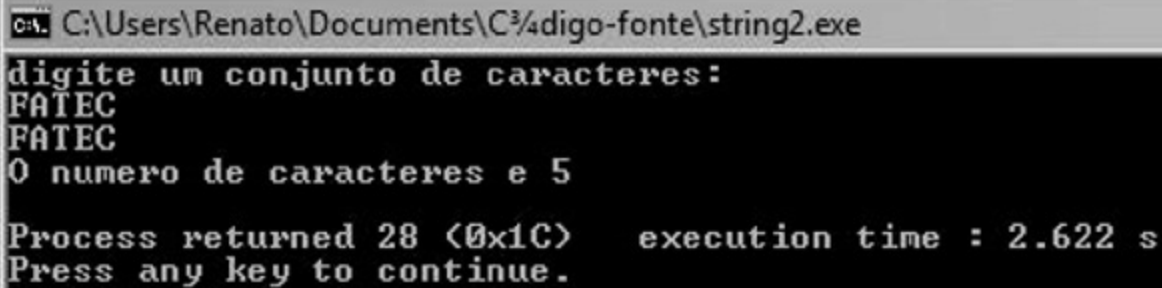
    putchar(string[i]);

printf("\n");

printf("O numero de caracteres e %d \n", i);

system("pause");
}

```



```

C:\Users\Renato\Documents\C³4digo-fonte\string2.exe
digite um conjunto de caracteres:
FATEC
FATEC
0 numero de caracteres e 5
Process returned 28 (0x1C)    execution time : 2.622 s
Press any key to continue.

```

A função de biblioteca `strlen()`, da mesma forma, conta o número de caracteres da *string*. Estas funções estão definidas em *string.h*.

Vejamos:

```

#include <stdio.h>

#include <stdlib.h>

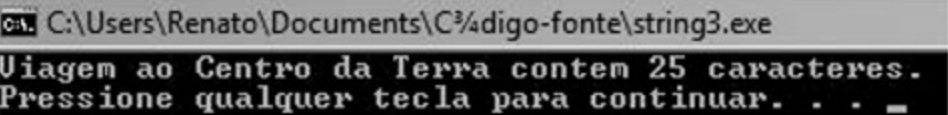
#include <string.h>

main()
{
    char livro[ ] = "Viagem ao Centro da Terra";

    printf("%s contem %d caracteres. \n", livro, strlen(livro));

    system("pause");
}

```



```

C:\Users\Renato\Documents\C³4digo-fonte\string3.exe
Viagem ao Centro da Terra contem 25 caracteres.
Pressione qualquer tecla para continuar. . . _

```

Repare que o caractere `NULL`, de final de *string*, não foi contado.

A função *strlen()* poderia ser implementada assim:

```
tamanho_str strlen(const char string)
{
    int i = 0;
    while(string[i])
        i++;
    return(i);
}
```

Outras importantes funções de manipulação de *strings* são:

```
strcpy( ): copia uma string em outra;
strcat( ): adiciona o conteúdo de uma string em outra;
strlwr( ): converte conteúdo para minúsculas;
strupr( ): converte conteúdo para maiúsculas;
strcmp( ): compara duas strings.
```

Exemplo de comparação de *strings*:

```
// Comparando duas cadeias de caracteres
#include <stdio.h>
#include <string.h>

int main()
{
    char x[10], y[10];
    printf("Entre a primeira string: \n");
    gets(x);
    printf("Entre a segunda string: \n");
    gets(y);
    if(strcmp(x, y) == 0)
        printf("Iguais! \n");
    else
        printf("Diferentes! \n");
    system("pause");
}
```

É possível converter caracteres de uma *string* para o formato de número:

```
atof( ): converte cadeia de caracteres para um valor real;
atoi( ): converte cadeia de caracteres para um valor inteiro;
```

As funções de entrada e saída padrão para *strings* (algumas já vistas) são:

```
printf() mais o emprego de %s
puts (string)
scanf("%s", string) // não usa o '&', e lê os caracteres até encontrar espaço
gets(string) // lê a string de forma completa
```

5.2 Matrices

Uma matriz é um vetor multidimensional. Alguns autores consideram ambas as expressões a mesma coisa, e o que muda é apenas o número de dimensões (como no termo em inglês, *array*, que vale para as duas coisas). Uma matriz bidimensional, por exemplo, pode ser declarada no formato de uma tabela de linhas e colunas, como a matriz seguinte:

```
int aluno[3][5]; // 3 linhas e 5 colunas
```

A posição de cada elemento será determinada pelo par de índices que representa sua localização dentro da matriz.

Por exemplo, o primeiro elemento será `aluno[0][0]`.

Podemos preencher essa matriz com valores diretos:

```
int aluno[3][5] = {{10,30,45,70,36}, {86,44,63,82,80},{70,61,52,63,74}};
```

Ou podemos manipular os elementos da matriz usando duas variáveis e uma repetição, como a seguir:

```
// Manipulando uma matriz bidimensional, supondo que os valores já existem
#include <stdio.h>
#include <stdlib.h>
main()
{
    int aluno[3][5];
    int i, j;
    for(i=0; i< 3; i++)
    {
        for(j=0; j<5; j++)
        {
            printf("Aluno[%d][%d]: aluno[i][j] \n", i, j, aluno[i][j]);
        }
        aystem("pause");
    }
}
```

No final deste livro apresentaremos o famoso jogo da velha, implementado pela ideia do tabuleiro representado por uma matriz.

EXERCÍCIOS

1. Escreva um programa que receba do usuário (interativamente) três notas para dado aluno, armazenando-as em um vetor; em seguida, o programa deve imprimir na tela tais notas.
2. Refaça o exercício anterior, agora calculando a média aritmética das notas recebidas pelo aluno.
3. Como seria uma matriz que representa os assentos de uma aeronave? Quais seriam as manipulações dessa matriz, em um sistema de reserva e compra de passagens aéreas?
4. Escreva um programa que receba uma *string* de caracteres minúsculos a partir do usuário e conte o número de elementos dessa *string*; na sequência, o programa deve converter os

caracteres para maiúsculos.

5. Escreva um programa que cadastre um aluno a partir de seu RA (Registro Acadêmico), nome e de três notas. Em seguida, gere um relatório na tela com essas informações na ordem em que foram digitadas.
6. Dada uma sequência de cinco números inteiros, mostre-os na tela na ordem inversa à da leitura.
7. Dada uma *string* digitada pelo usuário, determine o número de vezes que determinada letra de sua escolha ocorre na mesma.
8. Em dada turma há dez alunos, e cada um realizou três provas. Calcule a média de cada aluno e a média da turma.
9. Determine a soma de uma sequência de dez números inteiros.
10. Desenhe o tabuleiro do jogo da velha usando uma matriz.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7.ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

FEOFILOFF, Paulo. *Algoritmos em Linguagem C*. São Paulo: Campus, 2008.

MIZRAHI, Victorine Viviane. *Treinamento em Linguagem C*. 2.ed. São Paulo: Pearson/ Prentice Hall, 2008.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

ANEXO: A REVOLUÇÃO DA INTERNET (?)

Trabalho com a Internet e a acompanhei desde seus primórdios no país, ainda no âmbito das universidades públicas, quando a utilizávamos para o envio de correio eletrônico ou buscas muito simples, se comparadas aos *search engines* atuais. Isso foi no início da década de 1990.

A rede mudou em muito nossas vidas, mas se continuarmos a chamá-la de “revolução”, deveremos tratar o assunto de, pelo menos, dois pontos de vista: a) o acesso (ou *digital divide*, como o chamam os americanos); ou b) a real utilização dos recursos da rede para transformar a vida das pessoas (o que chamamos de *práxis*, em Educação). Quando penso em *práxis*, penso na real mudança que o Facebook e afins podem trazer para alguém. E me preocupo...

É esse, na verdade, o mesmo ponto de discussão que ocorre na área de tecnologia aplicada à educação; as políticas públicas insistem em garantir o acesso por meio da simples entrega de *hardware*; mas onde estaria a inclusão *social* advinda dessa inclusão *digital*? O que muda, de fato, na vida das pessoas, quando oferecemos o simples acesso à tecnologia? O discurso de um *digital*

divide parece ultrapassado. Os preços dos computadores caíram tanto, assim como as possibilidades de acesso, que tal problema não se nos apresenta mais.

Douglas Engelbart, um dos pais da Internet, sonhou com uma rede global de dispositivos interativos e colaborativos, que seriam utilizados para resolver os grandes problemas da humanidade e da ciência. Acabou sendo lembrado como o criador do *mouse* – para sua decepção, como declarou recentemente. As tecnologias digitais geraram produtividade operacional, mas não necessariamente cognitiva e estratégica.

Funções (modularização)



VISÃO DO CAPÍTULO

Funções são importantes em programação estruturada, pois permitem que um código já estabelecido seja reaproveitado quando da nova chamada à função, e que ações específicas possam ser realizadas de forma eficiente. Depois de escrita, uma função pode ser executada quantas vezes se fizerem necessárias, o que aumenta drasticamente a produtividade de um programador e de seu programa. Funções podem, ou não, retornar valores ao ponto do programa que as chamou.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- ◀◀ Entender o conceito e as propriedades de uma função;
- ◀◀ Criar suas próprias funções;
- ◀◀ Resolver os exercícios propostos.



6.1 Conceituando funções

Funções são trechos (ou blocos) de código que diminuem a complexidade de um programa, ou evitam a repetição excessiva de determinada parte do aplicativo. A isso chamamos de *modularização*. O conceito de modularização se aplica às linguagens estruturadas, em que as funções são elementos fundamentais para a construção de blocos ordenados e organizados. Em Linguagem C, *main()*, *printf()*, *scanf()* são funções que apresentam, como forma geral:

```
<tipo do retorno> nome (parâmetros)
{
    corpo da função
    retorno (não obrigatório)
}
```

EXEMPLO:

A função a seguir retorna a multiplicação de dois valores inteiros recebidos como argumentos:

```
int multiplica(int a, int b) // função de fato
{
    int resultado;
    resultado = a*b;
    return resultado;
}
```

ALGUMAS DEFINIÇÕES:

- *Parâmetros* são mostrados quando se cria a função (na declaração das variáveis que fazem parte dessa função); já *argumentos* tratam do uso efetivo dos elementos passados pela função, quando o programa é executado;
- O *retorno* da função é o valor que ela devolve ao ponto do programa que a chamou; se não for declarado, é do tipo *inteiro*; *void* não retorna qualquer valor, ou então não passa parâmetros;
- *Funções* retornam valores, *procedimentos*, não;
- Se a função for definida depois de *main()*, é preciso introduzir um *protótipo* dela antes da função principal.

Quanto a esse último conceito, vejamos:

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

int soma(int a, int b); // prototipo da funcao

int main()
{
    int x, y;

    printf("Digite o primeiro numero: \n");

    scanf("%d", &x);

    printf("Digite o segundo numero: \n");

    scanf("%d", &y);

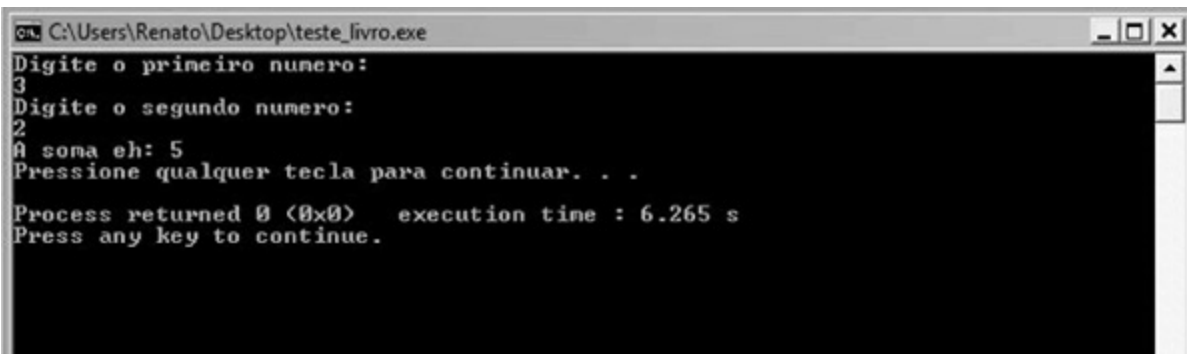
    printf("A soma eh: %d \n", soma(x, y));

    system("pause");
}

int soma(int a, int b) // função de fato
{
    int resultado;

    resultado = a + b;
```

```
    return resultado;
}
```



EXEMPLO 1:

Função da diferença entre dois números inteiros.

```
#include <stdio.h>

#include <stdlib.h>

int dif(int a, int b)
{
    int resultado;

    resultado = a - b;

    return(resultado);
}

main()
{
    int x, y, total;

    printf("Quais sao os numeros? \n");

    scanf("%d %d", &x, &y);

    total = dif(x,y);

    printf("Resultado = %d \n", total);

    system("pause");
}
```



```
C:\Users\Renato\Desktop\teste_livro.exe
Quais sao os numeros?
10 5
Resultado = 5
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 4.856 s
Press any key to continue.
-
```

Repare que o valor retornado pela função deve ser compatível com o tipo declarado da função. No exemplo, a função retorna um número inteiro. Isso é garantido quando se declara o tipo *int* antes do nome da função: `int soma(int a, int b)`.

EXEMPLO 2:

Função que retorna o maior entre dois números inteiros.

```
#include <stdio.h>

#include <stdlib.h>

int max(int n1, int n2)
{
    if(n1>n2)
        return n1;

    else
        return n2;

}

main()
{
    int x, y;

    printf("Digite o primeiro numero: \n");

    scanf("%d",, &x);

    printf("Digite o segundo numero: \n");

    scanf(";%d",, &y);

    printf("O maior eh: %d \n",, max(x, y));

    system("pause");
}
```

6.2 Chamada por valor e chamada por referência

A *chamada por valor* é a passagem normal do valor dos argumentos para a função. Os valores dos argumentos passados não são modificados, pois é fornecida uma cópia dos valores para a função.

Na *chamada por referência* são passados os endereços de memória dos argumentos. Portanto, os valores podem ser modificados.

EXEMPLOS:

```
/* Função com chamada por valor */

#include <stdio.h>

#include <stdlib.h>

int valor(int a, int b)
{
    a = a + 1; /* primeiro argumento foi modificado */
    b = b + 2; /* segundo argumento foi modificado */
    printf("Valores das variaveis dentro da funcao: \n");
    printf("Valor 1 = %d \n", a);
    printf("Valor 2 = %d \n", b);
}

main()
{
    int n1 = 1, n2 = 1, total;

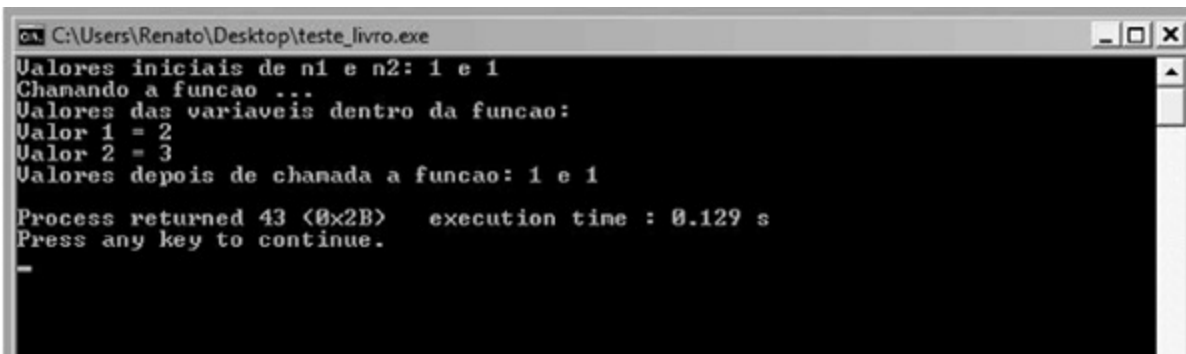
    printf("Valores iniciais de n1 e n2: %d e %d \n", n1, n2);

    printf("Chamando a funcao ... \n");

    valor(n1, n2);
}
```

```
printf("Valores depois de chamada a funcao: %d e %d \n", n1, n2);
```

```
}
```

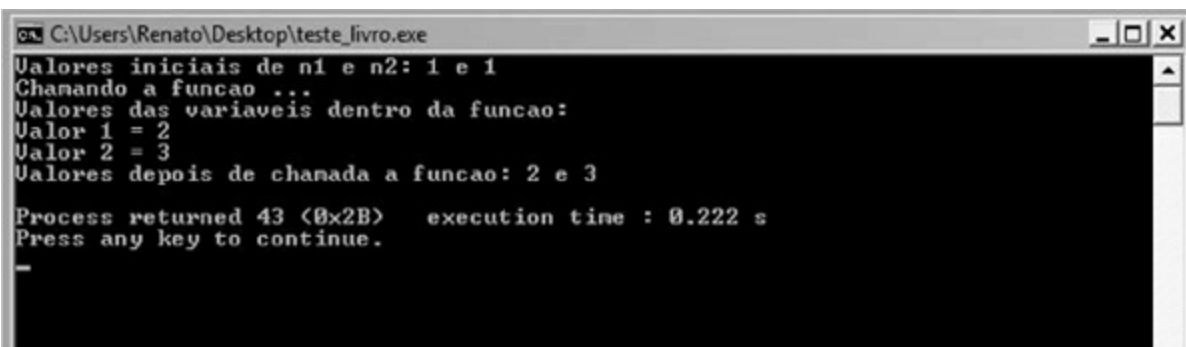


O que temos aqui é um paradoxo aparente, pois a chamada à função não alterou os valores iniciais das variáveis. O fato pode ser explicado pela chamada *por valor*.

A seguir mostraremos o mesmo problema, mas resolvido por passagem de parâmetros *por referência*, o que deve sanar a questão. Esse exercício utiliza os conceitos de *ponteiros e endereços de memória*, que serão vistos detalhadamente mais adiante.

```
#include <stdio.h>
#include <stdlib.h>
int valor(int * a, int * b)
{
    *a = *a + 1; /* primeiro argumento foi modificado */
    *b = *b + 2; /* segundo argumento foi modificado */
    printf("Valores das variaveis dentro da funcao: \n");
    printf("Valor 1 = %d \n", *a);
    printf("Valor 2 = %d \n", *b);
}

main()
{
    int n1 = 1, n2 = 1, total;
    printf("Valores iniciais de n1 e n2: %d e %d \n", n1, n2);
    printf("Chamando a funcao ... \n");
    valor(&n1, &n2); // passado o endereco da variavel
    printf("Valores depois de chamada a funcao: %d e %d \n", n1, n2);
}
```



No próximo exemplo, apresentamos o clássico algoritmo de troca de valores entre variáveis (*swap*); para que valores enviados para tal função possam retornar ao ponto de chamada – já trocados – é preciso passá-los por referência, e não por valor. Vejamos:

```
// Função de troca de valores
void swap (int * x, int * y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Na função *main()* utilizamos a chamada *swap (&i, &j)*, em que “&” representa o endereço da variável, e não seu valor.

IMPORTANTE:

- As *strings* e vetores são sempre *chamadas por referência*. Quando a Linguagem C passa um vetor ou uma *string* para uma função, o que se passa, na verdade, é o endereço de memória inicial do mesmo;
- As variáveis declaradas dentro de uma função são *locais*, visíveis apenas ali e destruídas após o término da função. Caso se deseje manter o valor da variável entre as chamadas a uma função, ela deve ser declarada *static*.

6.3 Argumentos da linha de comando

A função *main()* tem, em sua definição completa, a possibilidade de passar argumentos para a linha de comando do sistema operacional. Por ser a primeira função a ser chamada pelo programa, seus argumentos são passados junto com o comando que executa o programa (em geral, seu próprio nome). Vejamos um exemplo:

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

int main(int argc, char *argv[]) // aqui a definição completa de main( )
{
    printf("Numero de argumentos de main( ): %d \n", argc);
    printf("Valores dos argumentos de main( ): %s e %s \n", argv[0], argv[1]);
    return 0;
}
```

```
C:\Windows\system32\cmd.exe

C:\Users\Renato\Desktop>teste_livro argumento1
Numero de argumentos de main< >: 2
Valores dos argumentos de main< >: teste_livro e argumento1

C:\Users\Renato\Desktop>
```

Os argumentos *argc* e *argv* mostram, respectivamente, o número de argumentos passados para o programa (inclusive o nome do aplicativo) e os valores de tais argumentos. O parâmetro *argv* é, na verdade, um vetor de *strings* que armazena o nome do programa e os argumentos em forma de cadeia de caracteres; *argv[0]* é o nome do programa, *argv[1]*, o primeiro argumento passado ao programa, e assim por diante.

A função *main()* retorna um número inteiro para o processo que a chamou – geralmente, o próprio sistema operacional. Podemos declarar *main()* como void, se não retornar nenhum valor.

6.4 Recursividade

Recursividade é a possibilidade de uma função chamar a si mesma. Knuth,¹ por exemplo, apresenta o algoritmo (de Euclides) de determinação do MDC (Máximo Divisor Comum – o maior inteiro positivo que divide dois números inteiros positivos, *m* e *n*) como recursivo:

```
[Achar o resto da divisão] Dividir m por n e armazenar o resto da divisão em r (r = m % n)
[É zero?] Se r = 0, o algoritmo termina; n é a resposta.
[Reduzir] Faça m = n, n = r, e volte à etapa 1.
```

A implementação em Linguagem C poderia ser assim:

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

int mdc(int m, int n)
{
    int r = m%n;
    if(r == 0)
        return n;
    return mdc(n, (m%n));
}

main()
{
    int a=544; int b=119;
    printf("%d \n", mdc(a, b));
    system("pause");
}
```

Outro exemplo clássico de função recursiva é o cálculo do *fatorial de um número inteiro*:

```

#include <stdio.h>
#include <stdlib.h>

int fatorial(int valor)
{
    if(valor == 1);
        return(1);
    else
        return(valor * fatorial(valor -1));
}

main()
{

    int i;
    for(i=1; i <= 5; i++)
    {
        printf("O fatorial de %d e %d \n", i, fatorial(i));
    }
    system("pause");
}

```

```

C:\Users\Renato\Documents\C3\4digo-fonte\fatorial.exe
O fatorial de 1 e 1
O fatorial de 2 e 2
O fatorial de 3 e 6
O fatorial de 4 e 24
O fatorial de 5 e 120
Pressione qualquer tecla para continuar. . .

```

O que explica a recursividade dessa função é o fato de que o fatorial de 5 é igual a $5 \times 4 \times 3 \times 2 \times 1$. Ou seja:

- fatorial de 5 = $5 * \text{fatorial de } 4$
- fatorial de 4 = $4 * \text{fatorial de } 3$
- fatorial de 3 = $3 * \text{fatorial de } 2$
- fatorial de 2 = $2 * \text{fatorial de } 1$

Ou seja, o fatorial de um número = número * (fatorial do número – 1).

EXERCÍCIOS

1. Escreva um programa que implemente uma função que retorne a diferença entre dois números inteiros digitados pelo usuário.
2. Escreva uma função que retorne a divisão entre dois números. Atenção para a questão da *divisão por zero*!

3. Desenhe o algoritmo do cálculo recursivo do fatorial de determinado número inteiro (veja modelo anterior).
4. Escreva um programa que calcule a área de um círculo a partir de uma função especialmente desenhada para isso; essa função recebe o valor do raio e retorna a área do círculo.
5. Crie um sistema de caixa eletrônico, utilizando menus (*switch*) e outros recursos, que realizem operações financeiras a partir de funções especificamente projetadas para tal. Lembre-se de que o caixa eletrônico é um programa que roda como repetição contínua, até que o usuário decida encerrar as operações.
6. Crie uma função que determine se dado caractere está entre ‘a’ e ‘z’. **Dica:** Use a tabela ASCII.
7. Escreva um programa que implemente uma função que passe dado número inteiro como parâmetro, e este desenhe um número equivalente a “*” na tela.
8. Escreva uma função que retorne o cubo do valor passado como argumento.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7. ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

MIZRAHI, Victorine Viviane. *Treinamento em Linguagem C*. 2.ed. São Paulo: Pearson/ Prentice Hall, 2008.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidiah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

ANEXO: RECURSIVIDADE E HUMOR

Um sujeito faz dois pedidos a um gênio:

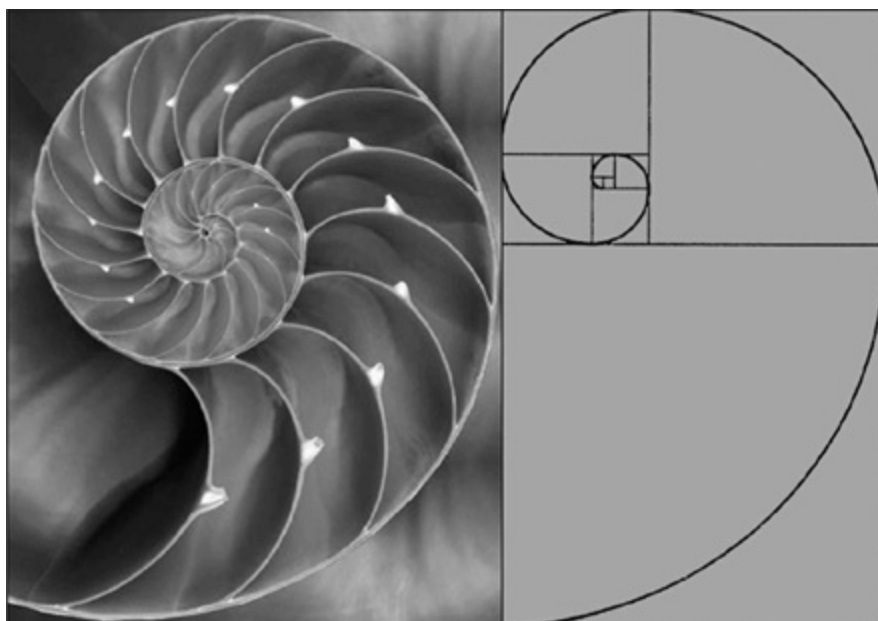
- 1) *Um supercomputador;*
- 2) *Fazer os dois pedidos de novo.*

Pode não parecer, mas a resposta do sujeito lembra, de certa forma, um processo recursivo!

Douglas Hofstadter trabalhou o conceito de recursividade em seu livro clássico *Gödel, Escher, Bach*. Neste ele mostra a recursividade na Matemática (lógica), na Arte e na Música.

Também a natureza usa a recursividade em suas construções, como as espirais que caracterizam as conchas. A concha *Nautilus*, mostrada a seguir, segue a série de Fibonacci na sua constituição.

FIGURA 6A.1 *Nautilus* e a série de Fibonacci.



Fonte: <blog.educastur.es>.

Estruturas (registros)



VISÃO DO CAPÍTULO

As estruturas permitem que armazenemos diversos tipos de dados *diferentes* na mesma estrutura, ao contrário dos vetores e matrizes que, como visto, trabalham apenas elementos do mesmo tipo. Isso aumenta em muito a capacidade de manipulação de dados do programa. As estruturas são chamadas por alguns autores de *registros*. Daremos preferência, neste livro, para o termo *estruturas*.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- «« Entender as propriedades das estruturas;
- «« Criar e manipular estruturas;
- «« Resolver os exercícios propostos.



7.1 Conceituando estruturas (ou registros)

Como já dito, as estruturas (*structs*) podem armazenar diferentes tipos de dados em uma mesma constituição, ao contrário dos vetores e matrizes, que trabalham o mesmo tipo.

Consideremos a seguinte estrutura:

```
struct Nome
{
    tipo a variável1;
    tipo b variável2;
    .....
    .....
};
```

Aqui estamos definindo uma estrutura chamada *Nome*, que contém diversos tipos de dados (*a*, *b* etc.) associados a variáveis (1, 2 etc.).

A implementação de uma estrutura desse tipo seria algo assim:

```
struct Pessoa
{
```

```
int rg;
char nome[30];
float salario;
};
```

Ou seja, uma *pessoa* tem variáveis características dos tipos int (RG), char (nome) e float (salário).

7.2 Carga inicial automática da estrutura

Vamos mostrar a inicialização de valores em uma estrutura, definida por meio desse exemplo:

```
struct Pessoa Pedro; (ou seja, Pedro é uma Pessoa)
```

Podemos então atribuir valores, como esses:

```
Pedro.rg = 2324;
Pedro.salario = 600.00;
strcpy(Pedro, "Pedro da Silva"); // usando a função strcpy( ) por ser uma string
```

A declaração da estrutura cria um *novo tipo de dado*. Poderemos, então, criar uma variável do tipo da estrutura criada:

```
struct Pessoa Pedro;
```

Repare que o tipo da variável é a nova estrutura criada. Pode-se também declarar a variável de outra forma – após a estrutura:

```
struct Aluno
{
    int RA;
    char nome[50];
} Paulo;
```

7.3 Acessando os membros da estrutura

Agora acessaremos os valores armazenados na estrutura:

```
printf("Salario = %f \n", Pessoa.salario);
printf("Nome = %s \n", Pessoa.nome);
```

Vamos agora a um exemplo completo:

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

```
struct Aluno
```

```

{
int RA;
char nome[50];
float nota;
};

int main()
{
    struct Aluno Paulo;
    printf("RA: \n");
    scanf("%d", &Paulo.RA);
    printf("Nota: \n");
    scanf("%f", &Paulo.nota);
    printf("Nome: \n");
    scanf("%s", Paulo.nome);
    // ou gets(Paulo.nome);

    printf("Exibicao dos dados do aluno: \n");
    printf("RA: %d \n", Paulo.RA);
    printf("Nota: %.2f \n", Paulo.nota);
    printf("Nome: %s \n", Paulo.nome);
    // ou puts(Paulo.nome);

system("pause");
}

```

```

C:\Users\Renato\Desktop\teste_livro.exe
RA:
123
Nota:
8
Nome:
Paulo
Exibicao dos dados do aluno:
RA: 123
Nota: 8.00
Nome: Paulo
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 13.855 s
Press any key to continue.

```

7.4 Vetores de estruturas

Uma possibilidade interessante é expressar as estruturas por meio de seus *vetores*, tornando possível um encadeamento de dados. Vejamos:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

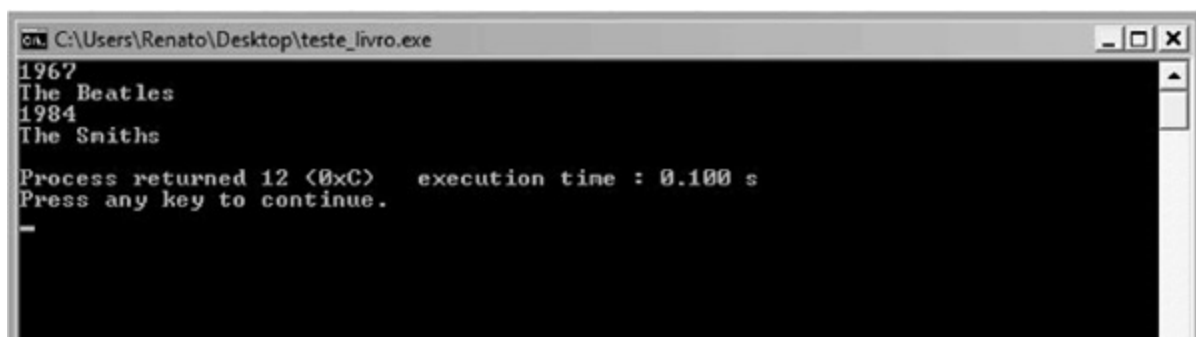
struct CD {
    char titulo[30];
    int ano;
}; // características de cada CD

struct CD colecao[5]; // cria vetor de 5 CD's

main()
{
    colecao[0].ano = 1967;
    printf("%d \n", colecao[0].ano);
    strcpy(colecao[0].titulo, "The Beatles");
    printf("%s \n", colecao[0].titulo);

    colecao[1].ano = 1984;
    printf("%d \n", colecao[1].ano);
    strcpy(colecao[1].titulo, "The Smiths");
    printf("%s \n", colecao[1].titulo);
}

```



7.5 Definição de tipos (*typedef*)

A palavra reservada *typedef* tem o propósito de associar nomes alternativos a tipos de dados já existentes. Um *typedef* pode ser usado da seguinte forma:

```

typedef int km_por_hora; // definindo novo tipo int
km_por_hora velocidade_atual; // usando o novo tipo

```

Podemos utilizar *typedef* para simplificar a declaração de estruturas já criadas, como esta:

```

struct MinhaStruct{
    int data1;
    char data2;
};

```

Uma nova declaração simplificadora poderia ser:

```

typedef struct MinhaStruct novo_tipo;

```

ou

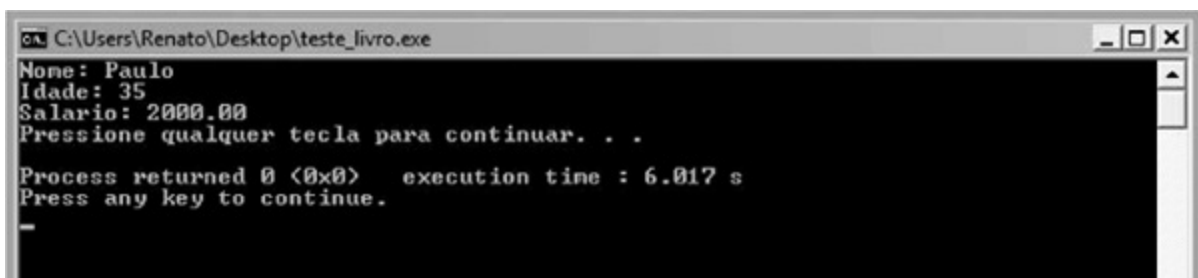
```
typedef struct MinhaStruct {  
    int dado1;  
    char dado2;  
} novo_tipo;
```

7.6 Passagem de estruturas para funções

É sempre interessante poder passar uma estrutura de dados como argumento de uma função, dadas as já comentadas vantagens que uma função apresenta em programação.

Vamos a um exemplo:

```
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct Pessoa  
{  
    char nome[30];  
    int idade;  
    float salario;  
} PESSOA;  
  
void mostrar(struct Pessoa x)  
{  
    printf("Nome: %s \n", x.nome);  
    printf("Idade: %d \n", x.idade);  
    printf("Salario: %.2f \n", x.salario);  
}  
  
main()  
{  
    struct Pessoa p = {"Paulo", 35, 2000};  
    mostrar(p);  
    system("pause");  
}
```



Poderíamos ter entrado com os dados de forma interativa, da seguinte forma:

```
void ler(PESSOA *ptr)
{
    printf("Nome: \n");
    gets(ptr->nome);
    printf("Idade: \n");
    scanf("%d", &ptr->idade);
    printf("Salario: \n");
    scanf("%f", &ptr->salario);
}
```

E na função *main()*:

```
main( )
{
    struct Pessoa p = {"Paulo", 35, 2000};
    mostrar(p);
    ler(&p);
    mostrar(p);
}
```

EXERCÍCIOS

1. Considere a seguinte estrutura:

```
struct Funcionario {
    int registro;
    float salario;
    char nome[30];
}
```

Escreva um programa que implemente essa estrutura.

2. Uma estrutura descreve os veículos de uma determinada montadora, utilizando os campos a seguir:

- marca
- ano de fabricação
- cor
- preço

- a) Escreva um programa com a estrutura *Carro*;
- b) Atribua valores aos campos;
- c) Imprima na tela um relatório com os dados entrados.

3. Escreva um programa de agenda de telefones. Para cada pessoa, devem ser declarados os seguintes dados:

- nome
- e-mail
- telefone

O programa deve apresentar um menu de opções que indique o campo a ser mostrado na tela (saída-padrão).

4. Implemente as funções `ler_pessoa()` e `mostrar_pessoa()`, as quais devem receber e mostrar os dados de uma determinada pessoa.
5. [Para pesquisar] É possível criar uma estrutura dentro de uma estrutura já existente? Se isso for possível, escreva um programa que mostre essa possibilidade.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7.ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

ANEXO: PENSAMENTO CRIATIVO

Programação tem a ver com o pensar de forma criativa, que é um processo de extremo valor em um momento em que tanto se fala de inovação e criatividade. Existe relação direta entre essas características e a geração de conhecimento, que acontece via criação, invenção, melhoria, inovação, *insight* e criatividade.

Não é sistemática nossa atuação nos processos de criatividade. Existe uma *charge* famosa que mostra duas pessoas bastante surpresas olhando pela fresta de uma porta de escritório semiaberta, onde um dos funcionários da empresa encontra-se com as pernas cruzadas sobre a mesa em posição de dormência, com as mãos atrás da cabeça. Um diz ao outro: “ – Dizem que ele ganha para pensar...!” O paradoxo da piada é que, na verdade, todos nós ganhamos para pensar em nossas atividades, mas essa ação fica tão rotineira e passiva que admiramos o fato de encontrar alguém que pensa de forma sistemática e organizada.

Um dos exemplos históricos que podem ser citados, considerando essa questão, é o de Newton, que costumava entrar em um estado de semidormência, no qual imaginava suas futuras descobertas. Outros pensadores passeavam, observavam paisagens, meditavam no calor de uma lareira.

É preciso que cada pessoa desenvolva seu próprio método de criação e invenção, em que o padrão de pensamento seja alterado de forma a liberar o potencial criativo. Ambientes de trabalho, e mesmo alguns ambientes domésticos, não são ideais para a invenção e a descoberta – precisam ser mudados.

Aqui estão algumas características básicas da criação e da descoberta:

- Mapeamento inicial do problema – essa etapa pode ter a participação de especialistas *ad hoc*

contratados para dar pareceres específicos. Nessa fase são acumulados dados e informação quantitativa e qualitativa. Um mapa conceitual é criado;

- Fase de incubação e geração de novas ideias a partir do problema – aqui, toda desconexão possível deve ser permitida, de forma a liberar a mente para o momento de *insight*. Quadros, murais, *post-it's*, conversas informais, *brainstormings* e outras técnicas pouco estruturadas são úteis nessa fase. Muitas ideias são geradas para trabalho posterior;
- Ahá! Eureka!

Ponteiros e alocação dinâmica de memória



VISÃO DO CAPÍTULO

Ponteiros são fundamentais para a Linguagem C; são parte do poder que essa linguagem tem em relação às demais. É por meio dos ponteiros que podemos manipular variáveis e outros recursos pelo endereço de memória, o que gera amplas capacidades e possibilidades. Em total relação com os ponteiros, a alocação dinâmica de memória permite a declaração de estruturas de dados dinâmicas, que poderão ter sua dimensão alterada durante a execução do programa.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- « Entender os conceitos relacionados aos ponteiros e trabalhar com eles de forma eficiente;
- « Aprender a alocar memória de forma dinâmica;
- « Resolver problemas reais de ponteiros e de gerenciamento de memória.



8.1 Ponteiros

Um ponteiro, como diz o nome, é uma variável que aponta para o endereço de outra variável de certo tipo (ou seja, armazena o endereço de memória da outra variável).

O compilador aloca um endereço automaticamente a toda variável declarada. Em geral, o programador não precisa se preocupar com a manipulação direta de endereços. Os ponteiros são a principal ferramenta de manipulação desses endereços, durante a execução do programa.

São declarados pelo uso do “*”:

```
int * pont; // declaramos aqui um ponteiro para um número inteiro
char * pont; // declaramos aqui um ponteiro para um caractere
int * float; // declaramos aqui um ponteiro para um número real
```

Casas e seus respectivos endereços são uma analogia interessante para a definição de ponteiros. Uma casa de determinado tipo pode ser encontrada pelo seu endereço postal; assim, “Rua da Alegria, 88”, por exemplo, identifica uma casa única, e esse endereço serve de referência para ela.

Os ponteiros manipulam diretamente os dados contidos em endereços específicos de memória, o

que lhes dá grande poder dentro dos programas escritos em Linguagem C.

FIGURA 8.1 Exemplo de ponteiro.



Neste exemplo, o ponteiro *ptr* aponta para a variável *x*, tendo como valor o endereço de memória de *x* (1024). Já o valor da variável à qual o ponteiro faz referência é 8.

Ponteiros utilizam um especificador próprio dentro da função *printf()*, que é o *%p*. Esse especificador mostra o endereço de memória do ponteiro, em formato hexadecimal.

Vejamos um exemplo:

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

main ()
{
    int x=8;
    int *p = &x;
    printf("Valor de x = %d \n", x);
    printf("Valor de x acessado pelo ponteiro = %d \n", *p);
    printf("Valor de p = %p \n", p);

    system("pause");
}
```

```
C:\Users\Renato\Desktop\teste_livro.exe
Valor de x = 8
Valor de x acessado pelo ponteiro = 8
Valor de p = 0022FF44
Pressione qualquer tecla para continuar. . .

Process returned 0 (0x0)   execution time : 14.862 s
Press any key to continue.
=
```

Se a variável *x* contém o endereço da variável *y*, é possível acessar o valor de *y* a partir de *x* pelo acréscimo de um asterisco antes de *x* (**x*). Pode-se, também, alterar diretamente o valor da variável apontada, como ilustra o exemplo a seguir:

```
*ptr = 20; //alteração do valor, supondo que ptr é o endereço de dada variável
```

Para se exibir o endereço de memória de uma variável, usamos o operador `&` (lembre-se que na função `scanf()` já utilizamos esse recurso). Assim, se considerarmos:

```
x = valor ou conteúdo de x
&x = endereço de memória de x
```

Poderíamos operar da seguinte forma:

```
int * ptr; // declaramos um ponteiro para o tipo int
ptr = &x; // ptr aponta para o endereço de x
int x = 8; // atribuímos um valor a x
int * ptr = &x; // outra forma de se apontar ptr diretamente para x
int * ptr = NULL; // ptr não apontará inicialmente para nenhuma variável
*ptr // forma de se trabalhar com o valor de x diretamente
```

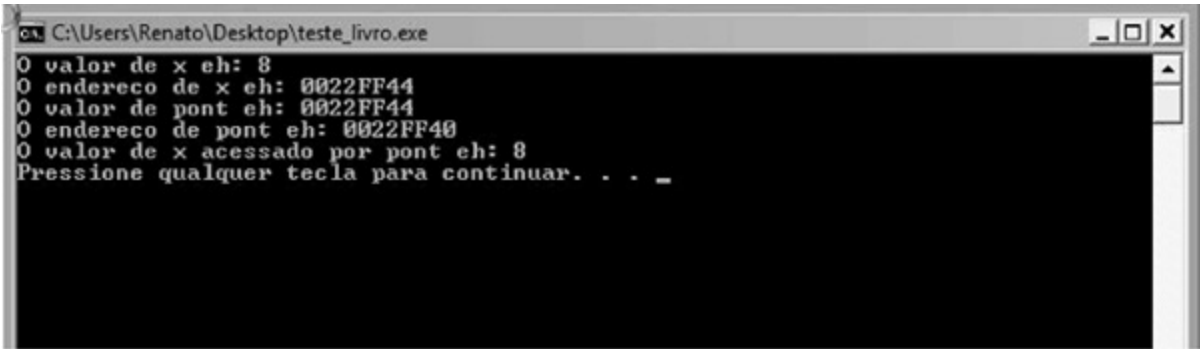
EXEMPLO:

```
/* Uso de ponteiros */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int valor = 8;
    int * pont = &valor;

    printf("O valor de x eh: %d \n", valor);
    printf("O endereco de x eh: %p \n", &valor); /* pode usar %x para formato hexadecimal */
    printf("O valor de pont eh: %p \n", pont);
    printf("O endereco de pont eh: %p \n", &pont);
    printf("O valor de x acessado por pont eh: %d \n", *pont);

    system("pause");
}
```



8.1.1 Ponteiros e parâmetros de funções

Para alterar o valor dos argumentos passados a uma função, é preciso definir os parâmetros da função como ponteiros. Na verdade, isso já foi visto no Capítulo 6, na forma de passagem de argumentos *por referência*. Ela só é possível, e agora entendemos bem o porquê, devido à manipulação direta de conteúdos de variáveis, pois estamos utilizando endereços de memória, e não valores absolutos que não são visíveis fora das funções com as quais trabalhamos.

8.1.2 Aritmética de ponteiros

Podemos manipular os ponteiros pelo emprego de aritmética tradicional nos seus valores e recursos; por exemplo, somar um número a um ponteiro é o mesmo que avançar dentro do trecho de memória alocado. Vejamos:

```
ptr++; // próximo endereço de memória
ptr = ptr + 2;
ptr += 4;
```

Mas atenção: o avanço real é sempre relativo ao tamanho da variável, que pode ser determinado por `sizeof(tipo de dado)`.

Vejamos alguns exemplos de aritmética de ponteiros:

| Operação | Exemplo | Observação |
|--------------|-------------|--|
| Atribuição | ptr=&x | recebe endereço |
| Incremento | ptr=ptr+2 | + 2*sizeof(tipo) de ptr |
| Decremento | ptr=ptr-10 | - 10*sizeof(tipo) de ptr |
| Apontado por | *ptr | acessa valor apontado |
| Endereço de | &ptr | manipula endereço |
| Diferença | ptr1 - ptr2 | número de elementos entre os dois |
| Comparação | ptr1 > ptr2 | compara dois elementos em uma estrutura pelo valor de seus endereços |

8.1.3 Ponteiros de ponteiros

Pode ser de interesse que um ponteiro armazene, como valor, o endereço de um outro ponteiro; isso seria indicado da seguinte forma:

```
int ** ptr_ptr; // um ponteiro para outro ponteiro do tipo int
```

A atribuição do ponteiro duplo se faria da seguinte maneira:

```
ptr_ptr = &ptr;
```

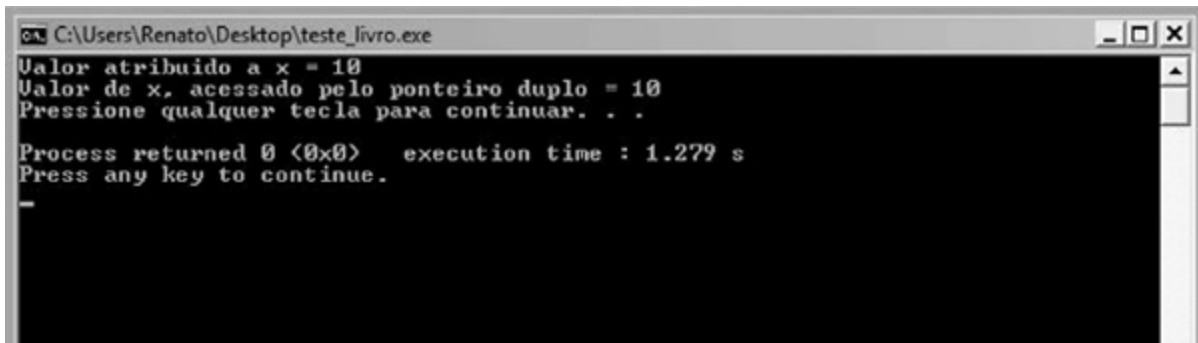
Já o acesso ao valor da variável ao qual o ponteiro original faz referência, poderia ser feito assim:

```
printf("%d \n", **ptr_ptr);
```

EXEMPLO:

```
/* Aqui vamos fazer o ponteiro duplo mostrar o valor da variável apontada pelo p
original */
#include <stdio.h>
#include <stdlib.h>

main()
{
    int x, *p, **q;
    x=10;
    p=&x;
    q=&p;
    printf("Valor atribuído a x = %d \n", x); //valor de x!
    printf("Valor de x, acessado pelo ponteiro duplo = %d \n", **q); //valor de x
    system("pause");
}
```



OUTRO EXEMPLO:

```
/* Aqui vamos percorrer um vetor de strings por meio de um ponteiro */
#include <stdio.h>
#include <stdlib.h>

main()
{
    char *banda[] = {"Beatles", "Smiths", "U2", "Led Zeppelin"};
    char **aponta_banda;
```

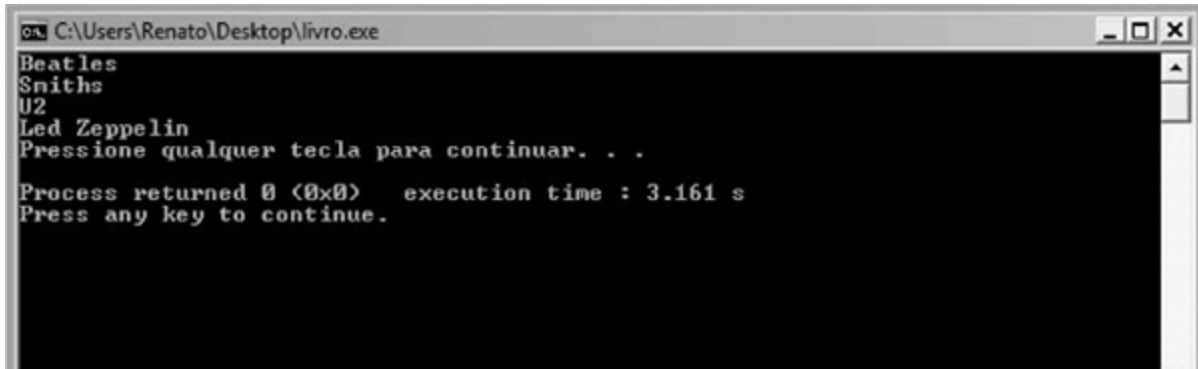
```

aponta_banda = banda;

while(*aponta_banda)
{
    printf("%s \n", *aponta_banda);
    aponta_banda++;
}

system("pause");
}

```



8.1.4 *Ponteiros e vetores*

O nome de um vetor corresponde ao endereço de seu primeiro elemento, ou seja, se V for um vetor, $V == \&V[0]$. Podemos, portanto, usar vetores como se fossem ponteiros.

Vejamos:

```

double * pdata;
double data[10];
pdata = data; // pois data = &data[0]
pdata = &data[0];

```

Um outro exemplo:

```

char str[10], *p;
p = str; // o ponteiro recebe o vetor, ou o endereço de seu primeiro elemento

```

A Linguagem C permite dois métodos de acesso a elementos de um vetor, quando do emprego de ponteiros:

a) Aritmética de ponteiros (mais rápida),

EXEMPLO:

```

void putstr(char * s)
{
    while(*s)

```

```

{
    putchar(*s++);
}
}

```

b) Indexação do vetor (mais clássica),

EXEMPLO:

```

char s[] = "FATEC";
char * ptr = s; // ptr aponta para &s[0] - endereço do primeiro elemento de s

```

Nesse caso, como acessaríamos o caractere ‘A’ da *string*?

- Uma possibilidade seria simplesmente *s[1]*;

```

vetor[0] == *(vetor)
vetor[1] == *(vetor + 1)
vetor[2] == *(vetor + 2)
vetor[n] == *(vetor + n)

```

- Outra forma seria **(ptr + 1)*;
- Ou **(s + 1)* pois *s = &s[0]*;

Na verdade, o endereçamento de elementos com a utilização de colchetes pode ser realizado também por ponteiros, como se considerássemos um vetor, ou seja, *ptr[2]*.

8.1.5 Ponteiros de strings

Tomemos o seguinte exemplo:

```

char * frase = "Fatec Americana";

```

Ao contrário do vetor típico constituído por caracteres (como *char frase[] = "Fatec Americana"*), o ponteiro para caracteres pode ser alterado, já que é referenciado pelo endereço de memória do primeiro elemento.

Vejamos um exemplo:

```

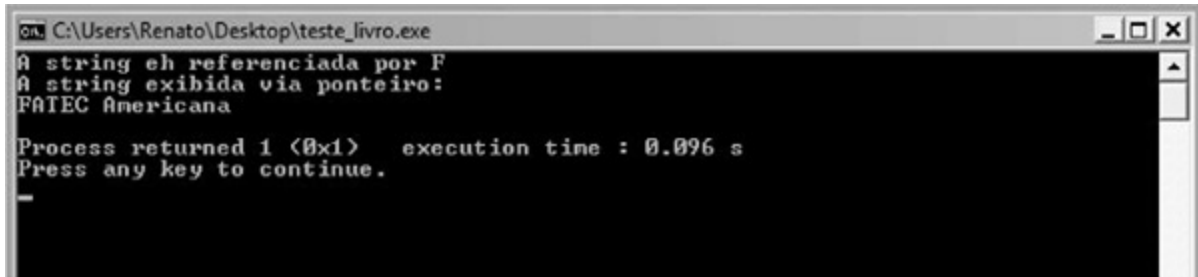
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char nome[30] = "FATEC Americana";
    char *ptr_str;

```

```
ptr_str = nome;
printf("A string eh referenciada por %c \n", *ptr_str);
printf("A string exibida via ponteiro: \n");
```

```
while(*ptr_str)
{
    putchar(*ptr_str);
    ptr_str++;
}
printf("\n");
}
```



8.1.6 Ponteiros void

Um ponteiro *void*, como o próprio nome indica (*void* significa vácuo, vazio), por algum motivo, não aponta para um tipo particular. Em geral, é usado com funções, pois aceitará qualquer tipo na sequência.

```
void * p;
```

8.1.7 Manipulando estruturas com ponteiros

É possível acessar e alterar uma estrutura por meio de ponteiros. Tomemos como exemplo o seguinte programa, que mostrará valores iniciais dos componentes de uma estrutura; um deles será alterado em seguida:

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

struct Pessoa
{
    char nome[30];
    int idade;
};

alteracao(struct Pessoa *acesso) // adiciona 20 anos à idade
{
    acesso-> idade += 20;
```



```

}

int main()
{
    struct Pessoa acesso;

    printf("Entre nome: \n");
    gets(acesso.nome);
    printf("Entre idade: \n");
    scanf("%d", &acesso.idade);

    printf("Dados iniciais: \n");
    printf("Nome: %s \n", acesso.nome);
    printf("Idade: %d \n", acesso.idade);

    alteracao(&acesso);

    printf("Dados apos mudan as: \n");
    printf("Nome: %s \n", acesso.nome);
    printf("Idade: %d \n", acesso.idade);
}

```

```

C:\Users\Renato\Desktop\teste_livro.exe
Entre nome:
renato
Entre idade:
50
Dados iniciais:
Nome: renato
Idade: 50
Dados apos mudan as:
Nome: renato
Idade: 70

Process returned 11 (0xB)   execution time : 5.955 s
Press any key to continue.

```

8.1.8 Efici ncia dos ponteiros na Linguagem C

Como dissemos no in cio deste cap tulo, os ponteiros s o respons veis por boa parte da efici ncia caracter stica da Linguagem C.

Consideremos, para ilustrar essa quest o, a c pia de uma *string* em uma outra; o processo tradicional exigiria que cada caractere fosse copiado da estrutura original e escrito na estrutura de destino. Se usarmos ponteiros para tal opera o, teremos:

```

while (*pSOut ++ = *pSIn ++); // o ponteiro ser  incrementado a cada caractere lido

```

Aqui o comando copia porque o ponteiro indica o primeiro elemento de cada *string*. Assim, (**pSOut*)   o conte do do elemento.

O processo termina na cópia de ‘\0’, quando *pSOut será zero e a repetição se encerrará.

8.1.9 Vetores de ponteiros

A linguagem C permite criar vetores de ponteiros, nos quais cada elemento é inicializado, por exemplo, com o endereço de uma *string*:

```
char * mes[] = {"Jan","Fev", ...};
```

Podemos percorrer esses vetores de *strings* por meio de um ponteiro. Vejamos:

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

main()
{
    char *dia[] = {"Domingo", "Segunda", "Terca", "Quarta", "Quinta", "Sexta",
    "Sabado", 0}; // repare o indicador de final de vetor
    char **ptr_dia; /* *dia é um ponteiro para uma string e
    **ptr_dia é um ponteiro para um ponteiro para uma string */

    ptr_dia = dia; /* apontando ptr_dia para o início da matriz dia */

    while(*ptr_dia)
    {
        printf("%s \n",*ptr_dia);
        ptr_dia++;
    }
    system("pause");
}
```

8.2 Alocação dinâmica de memória

Ao declararmos um vetor, alocamos memória para o armazenamento dos dados desse vetor. Se precisarmos alterar seu tamanho, teremos que modificar o código do programa, pois a alocação feita

é estática.

Por exemplo, se quisermos copiar uma *string* em outra, será necessário alocar espaço inicial, que com certeza será maior que o utilizado, o que gera desperdício de recursos (costumamos, por prevenção, alocar mais espaço do que o necessário).

A saída otimizada para esse problema seria alocar a memória *dinamicamente*, ou seja, o programa alocaria a memória necessária *durante a execução*. Para isso, poderemos utilizar as funções *calloc()* e *malloc()*.

A sintaxe de *calloc()* é:

```
* calloc(Quantidade_de_Elementos, tamanho)
```

EXEMPLO:

```
int *pont2 = (int *) calloc(n, sizeof(int)); /* aloca espaço para um vetor de 'n'
números inteiros */
```

Tal função aloca quantidade de memória para um vetor com o número de elementos indicado; cada um deles tem seu tamanho expresso em *bytes*. A função retorna um ponteiro para o início do bloco de memória alocado, ou NULL no caso de erro ou problema.

A sintaxe de *malloc()* é:

```
void * malloc(tamanho)
```

EXEMPLO:

```
int *pont1 = (int *) malloc(sizeof(int)); // aloca espaço para um número inteiro
```

Essa função aloca uma quantidade de *bytes* e retorna um ponteiro para o início da memória alocada, ou NULL caso ocorra erro ou problema.

A memória não mais utilizada deve ser liberada por meio da função *free()*:

```
void free(void *ponteiro)
```

EXEMPLO:

```
free(pont1); // libera o espaço alocado
```

Exemplos de alocação de memória:

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

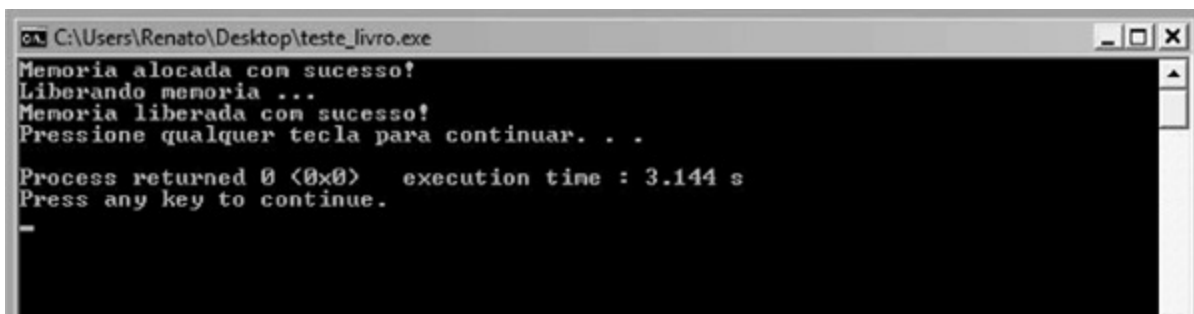
```
main ()
```

```

{
    int *p;
    p=(int *) malloc(sizeof(int));

    if (!p)
    {
        printf ("Memoria Insuficiente! \n");
        exit;
    }
    else
    {
        printf ("Memoria alocada com sucesso! \n");
    }
    system("pause");
}

```



8.3 Alteração do tamanho da memória alocada

Um bloco de memória alocado pode ter seu tamanho alterado pelo emprego da função *realloc()*.

Sua sintaxe é:

```
*realloc(*ponteiro, bytes)
```

Ou seja, a função altera o tamanho do trecho de memória apontado por **ponteiro* para a quantidade de *bytes* adotada.

Vejamos um exemplo de uso de *realloc()*:

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    char *str;
    str = (char *) malloc(50);

    if(str)
        printf("50 bytes alocados! \n");
}

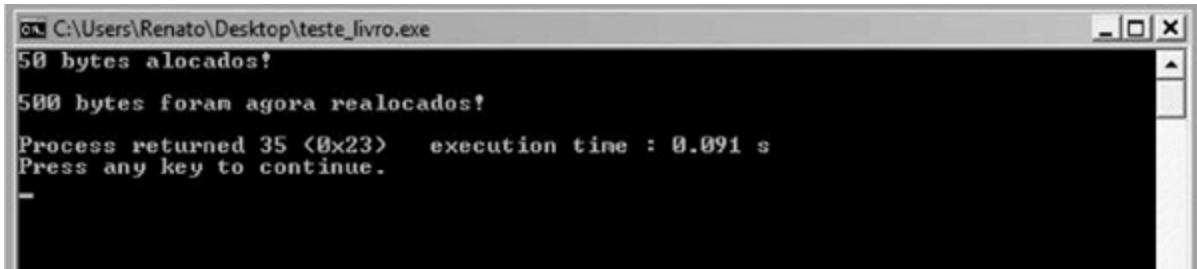
```

```

else
    printf("Erro de alocação de memória! \n");
printf("\n");

realloc(str, 500);
if(str)
    printf("500 bytes foram agora realocados! \n");
else
    printf("Erro de alocação de memória \n");
}

```



A alocação dinâmica de memória é de extrema importância para qualquer estrutura de dados, e não apenas para vetores e *strings* como visto aqui. O assunto será retomado nos próximos capítulos, em especial quando trabalharmos as listas, filas e pilhas.

EXERCÍCIOS

1. Declare um ponteiro para um número inteiro.
2. Direcione o endereço de um número real para um ponteiro, e faça que o ponteiro mostre o valor da variável indiretamente.
3. Qual é a saída do seguinte trecho de código C?

```

int contador = 10, *temp, soma = 0;
temp = &count;
*temp = 20;
temp = &soma;
*temp = contador;
printf("contador = %d, *temp = %d, soma = %d\n", contador, *temp, soma);

```

4. Declare um ponteiro para a *string* "FATEC", chamada *escola*.
5. Considere as seguintes definições e inicializações:

```

char c = 'T', d = 'S';
char *p1 = &c;
char *p2 = &d;
char *p3;

```

Considere, ainda, que o endereço de *c* é 1024, que o endereço de *d* é 2048, e que, por fim, o endereço de *e* é 256.

O que será mostrado na tela quando o código a seguir for executado em sequência?

```
p3 = &d;
printf( ) de *p3?

p3 = p1;
printf( ) de *p3?

*p1 = *p2;
printf( ) de *p1?
```

6. Considere as seguintes definições:

```
int *p;
int i;
int k;
i = 42;
k = i;
p = &i;
```

Após sua execução, qual das seguintes alternativas mudará o valor de *i* para 75?

```
k = 75;
*k = 75;
p = 75;
*p = 75;
```

Duas ou mais das alternativas fazem essa mudança.

7. Considere o vetor `int x[5] = {0, 1, 2, 3, 4}`. Quais serão os valores de *x* após a chamada *troca(x+1, x+4)*?

```
void troca(int *pa, int *pb)
{
    int t;
    t=*pa;  *pa=*pb;  *pb=t;
}
```

8. Quando passamos um vetor para uma função em C, esta alocará um ponteiro para o elemento zero do vetor. Por que a Linguagem C simplesmente não cria uma nova cópia local do vetor, como faz com os números inteiros?

9. Escreva um programa em C que declare e inicialize (da forma que você quiser) um número real, um número inteiro e uma *string*. Ele deverá:

- a) imprimir os endereços e os valores de cada uma dessas variáveis. Lembre-se de que o formato `%p` mostra o formato hexadecimal (base 16) de endereços de memória;
- b) desenhar um diagrama da memória mostrando a localização e valores de cada variável.

10. Crie as variáveis *x* e *y* e os ponteiros *p* e *q*. Atribua o valor 2 a *x*, 8 a *y*, o endereço de *x* para *p*, e o endereço de *y* para *q*. Depois, imprima os seguintes resultados:

- a) o endereço de *x* e o valor de *x*;
- b) o valor de *p* e o valor de **p*;
- c) o endereço de *y* e o valor de *y*;
- d) o valor de *q* e o valor de **q*;

e) o endereço de p ;

f) o endereço de q ;

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7.ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

FEOFILOFF, Paulo. *Algoritmos em Linguagem C*. São Paulo: Campus, 2008.

MIZRAHI, Victorine Viviane. *Treinamento em Linguagem C*. 2.ed. São Paulo: Pearson/ Prentice Hall, 2008.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidiah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

ANEXO: O PODER DAS REDES

Nulla dies sine linea.

PLÍNIO

Simplex ratio veritatis.

CÍCERO

Simplex veri sigillum.

SÊNECA

Em tempos de Internet, meditemos sobre o comportamento explosivo das redes em nossos tempos e a razão pela qual essas redes têm (ao que parece) mudado nossas vidas.

A chamada *Lei de Metcalfe* garante: *a soma de uma rede aumenta com o quadrado do número de seus membros*. O aumento aritmético do número de nós da rede gera um valor incrementado de forma exponencial. A adição de alguns membros à rede pode aumentar dramaticamente o seu valor para todos os membros.

Outra lei básica característica das redes é conhecida como a *Lei dos Retornos Crescentes*. Esta é a nossa mais que conhecida *economia de escala*. Ambas baseiam-se no conceito de *ciclos de feedback* positivo, mas a primeira é alicerçada no poderio de rede. O crescimento das economias de escala é linear; no caso das redes, é exponencial. Assim, o valor explode com o número de componentes, atraindo mais e mais membros, gerando um círculo virtuoso. Isso é contrário à *Lei dos Rendimentos Decrescentes*, característica dos insumos em sistemas produtivos naturais, como o caso da adubação agrícola. Aqui, a resposta das plantas ao aumento dos níveis de fertilizantes tem um máximo, a partir do qual a produção observada começa a decrescer.

Também os processos de comunicação parecem ter sido privilegiados pela inovação trazida

pelos computadores conectados em rede. Essa é a real revolução trazida pelos aparatos digitais de nossos dias. Processos de comunicação, como o sabe qualquer pessoa atuante em uma organização moderna, são mais frágeis do que o processamento de dados automáticos. Douglas Engelbart já se preocupava, na década de 1960, em como melhorar e amplificar as capacidades humanas com o suporte de tecnologia, e também já meditava sobre os formatos e possibilidades de utilização dos computadores e das redes em tais melhoramentos. Imaginou um mundo de comunicação e interação entre seres humanos empenhados em alguma tarefa ou atividade comum. Em tal ambiente de melhoria e desenvolvimento humano, a *inovação* torna-se eixo primordial, e não apenas a otimização de processos.

Estamos nos conectando de forma acelerada nas últimas décadas. Dispositivos são conectados aos milhares, por meio de *microchips*, gerando padrões de emergência que têm sido muito estudados do ponto de vista da criação de uma inteligência coletiva (como a chama Pierre Lévy). Não apenas a Internet tradicional nos tem ligado, mas também o infravermelho e o rádio, criando uma vasta teia de aparatos sem fio (*wireless*), maior até do que a web atual.

Computadores conectados são equivalentes a neurônios estruturados em uma complexa teia neural, característica do cérebro humano. Este conjunto de computadores conectados (no ambiente da Internet, por exemplo), pode ter seu nível de utilidade extremamente incrementado, como mencionado parágrafos atrás. Partes simples, conectadas de forma apropriada, geram resultados complexos.

Finalmente, alguns princípios caracterizam as redes: um propósito unificador, que compartilha foco em resultados desejados e bem determinados; independência dos membros componentes da rede; conexões voluntárias e líderes múltiplos (portanto, uma menor quantidade de chefias): vários líderes podem aumentar a resiliência da rede, pois assim esta não dependerá de decisões lentas e direcionadas.

A metáfora da rede também é utilizada para representar os relacionamentos pessoais e profissionais que cada um de nós deve manter registrados e mapeados. A parceria é um dos modelos de trabalho conjunto mais adotados na atualidade, porque gera efeito de sinergia entre os parceiros, sem demandar grandes esforços de manutenção de conexões formais. Ou seja, podemos nos associar a outros colegas, empresas e organizações sem que uma ligação forte em termos burocráticos e legais tenha que ser mantida.

Listas, pilhas e filas



VISÃO DO CAPÍTULO

Este capítulo trata das estruturas avançadas chamadas de listas, pilhas e filas. Muitas aplicações do dia a dia utilizam essas estruturas para armazenar dados que precisam ser permanentes, bem como aqueles passíveis de processos de busca.

Priorizaremos o entendimento das listas, com exemplos e explicações mais detalhadas; o aluno poderá, então, transferir tais conhecimentos para as outras duas estruturas, se tiver necessidade de usá-las em seu exercício de programador.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- «« Entender os princípios das listas, filas e pilhas;
- «« Implementar listas encadeadas;
- «« Imaginar usos reais para essas estruturas de dados.



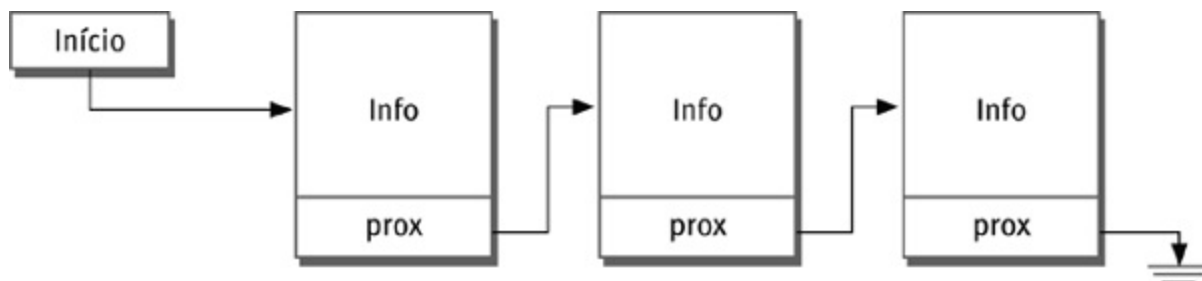
9.1 Listas ligadas (ou encadeadas)

Uma lista ligada (ou encadeada) é uma estrutura de dados linear e dinâmica, formada por elementos que apontam para o próximo da lista. Precisamos ter, portanto, um primeiro elemento, mais um último elemento que aponta para uma célula nula.

Tais listas são estruturas de dados fundamentais para a computação, podendo ser utilizadas na implementação de outras estruturas de dados.

Cada elemento da lista ligada é chamado de *nó*. O nó é composto por variáveis comuns e seus respectivos valores, mais um ponteiro de referência para o próximo nó dentro da lista. Se a lista estiver vazia, esse ponteiro tem valor NULL.

FIGURA 9.1 Exemplo de lista ligada.



A estrutura dessa lista ligada é desenvolvida da seguinte forma:

- em primeiro lugar, criamos a estrutura do nó inicial; esta tem dois membros: uma variável de qualquer tipo (**Info**), e um ponteiro para o próximo nó da lista (**prox**); portanto, para começar uma lista, não basta inserir novas células; é preciso, antes, criar uma base, que será a mesma para a lista toda; esta poderá ser simplesmente um ponteiro para uma lista, ou um ponteiro para uma primeira célula vazia;
- podemos, agora, inserir novos nós no início ou no final da lista.

Vejamos um exemplo:

```
struct Lista
{
    int valor;
    struct Lista *proximo; // ponteiro para a próxima entrada na lista
}
```

Declaramos, então, uma variável inicial do tipo da estrutura recém-criada:

```
struct Lista inicio;
```

Finalmente, alocamos o valor NULL ao ponteiro para o próximo elemento da lista, garantindo que ela esteja vazia. A lista está pronta para ter elementos inseridos, excluídos e percorridos.

Façamos uma inserção manual; como exemplo:

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

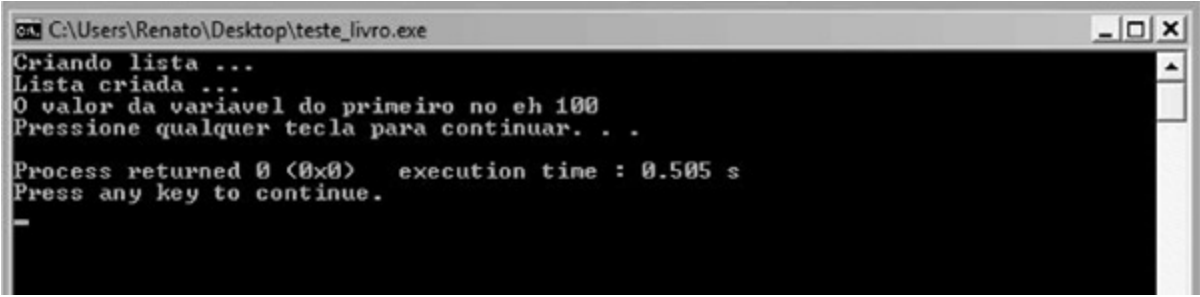
struct Lista
{
    int valor;
    struct Lista * proximo; // ponteiro para a próxima entrada na lista
};

main()
{
```

```

struct Lista * inicio = NULL;
    if(inicio == NULL) // se a lista estiver vazia
    {
        printf("Criando lista ... \n");
        inicio = (struct Lista *) malloc (sizeof(struct Lista)); /* alocamos espaço p
        estrutura */
        printf("Lista criada ... \n");
        if(inicio != NULL) // se a estrutura foi criada
        {
            inicio -> valor = 100;
            inicio -> proximo = NULL; // não aponta para o próximo nó
        }
        printf("O valor da variavel do primeiro no eh %d \n", inicio -> valor);
    }
system("PAUSE");
}

```



```

C:\Users\Renato\Desktop\teste_livro.exe
Criando lista ...
Lista criada ...
O valor da variavel do primeiro no eh 100
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 0.505 s
Press any key to continue.

```

Uma função que poderia realizar a inserção seria algo como:

```

lista_insercao (struct Lista * inicio, int i)
{
    inicio = (struct Lista *) malloc (sizeof(struct Lista));
    inicio -> valor = i;
    inicio -> proximo = NULL;
    return inicio;
}

```

Uma inserção no *início da lista* receberia o valor a ser armazenado em cada nó, e retornaria um ponteiro para o início da lista:

```

struct Lista *insere_inicio(struct Lista *n, int x)
{
    struct Lista *novo;
    if(n == NULL) // se a lista estiver vazia
    {
        n = (struct Lista*)malloc(sizeof(struct Lista));
        n->valor = x;
        n->proximo = NULL; // por ser primeiro nó não deve apontar
        return n;
    }
    else // a lista não está vazia
    {
        novo = (struct Lista*)malloc(sizeof(struct Lista)); // criar novo nó
        novo->valor = x; // recebe o valor
        novo->proximo = n; // o início da lista será o próximo campo do novo nó
    }
}

```

```

        return novo;
    }
}

```

A inserção no *final da lista* ficaria assim:

```

struct Lista *insere_final(struct Lista *n, int x)
{
    struct Lista *novo = (struct Lista*)malloc(sizeof(struct Lista));
    novo->valor = x;
    if(n == NULL) // lista está vazia
    {
        novo->proximo = NULL; // não deve apontar
        return novo; // novo nó será o início da lista
    }
    else // lista não está vazia - vamos ao final para inserir o nó
    {
        struct Lista *temp = n; // criando referência ao primeiro nó
        while(temp->proximo != NULL) // é preciso ir ao último nó
        {
            temp = temp->proximo;
        }
        novo->proximo = NULL;
        temp->proximo = novo; // o último nó apontará para o novo nó
        return n;
    }
}

```

Vimos que para inserir dados (ou removê-los) é necessário ter um ponteiro endereçado ao primeiro elemento, e outro apontado para o final da lista, pois assim as operações serão rapidamente executadas. Se o elemento estiver no meio da lista, será preciso proceder a uma busca pela sua posição específica.

É possível testar se a lista está vazia, ou se o ponteiro para o próximo elemento não aponta para nada, verificando se seus valores são NULL.

Agora, vamos trabalhar um *exemplo completo*:

```

// Programa de lista ligada no tema "bandas"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

struct Banda
{
    char nome[30];
    char cd[30];
    int ano;
    struct Banda *proximo;
};

```

```

struct Banda * cria(void)
{
    return NULL;
}

```

```

struct Banda * insere(struct Banda * l, char * nome, char * cd, int ano)
{
    struct Banda * novo = (struct Banda *) malloc(sizeof(struct Banda));
    strcpy(novo->nome, nome);
    strcpy(novo->cd, cd);
    novo->ano=ano;
    novo->proximo=l;
    return novo;
}

void imprime(struct Banda * l)
{
    struct Banda * p;
    for(p=l; p!=NULL; p=p->proximo)
    {
        printf("Banda = %s \n", p->nome);
        printf("CD = %s \n", p->cd);
        printf("Ano = %d \n", p->ano);
    }
}

int busca(struct Banda * l, char * nome)
{
    struct Banda * p;
    int i=1;
    for(p=l; p!=NULL; p=p->proximo)
    {
        if(strcmp(nome, p->nome) == 0)
        {
            printf("Banda: %s \n", p->nome);
            return i;
        }
        i++;
    }
    return 0;
}

main()
{
    char grupo[30], disco[30];
    int year;
    struct Banda * l;
    char resp='s';
    char procura[30];

    l=cria();

    while(resp!='n')
    {
        printf("Qual eh a banda? \n");
        scanf("%s", grupo);
    }
}

```

```

    fflush(stdin);
    printf("Qual eh o CD? \n");
    scanf("%s", disco);
    fflush(stdin);
    printf("Qual eh o ano? \n");
    scanf("%d", &year);
    fflush(stdin);
    l=insere(l, grupo, disco, year);
    printf("Continua? s/n \n");
    scanf("%c", &resp);
    fflush(stdin);
}

imprime(l);
printf("Qual banda quer procurar? \n");
scanf("%s", procura);
printf("Esta banda esta no elemento %d \n", busca(l, procura));

system("pause");
}

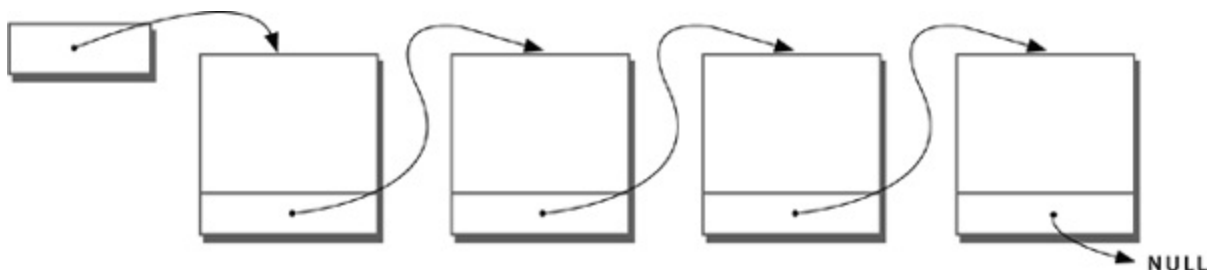
```

9.2 Filas

São estruturas de dados nas quais as inserções são feitas no final e as remoções acontecem no início. As filas representam estruturas de dados importantes, e seus elementos estão organizados dentro de um critério de entrada e saída ao qual chamamos de FIFO (*First In, First Out*), ou seja, o primeiro elemento a entrar é o primeiro a ser retirado (pense em uma fila da vida cotidiana).

As operações básicas implementadas para uma fila são a inicialização da fila, a verificação (que confirma se ela está mesmo vazia), a inserção de um elemento no final e a retirada de um elemento do início da fila.

FIGURA 9.2 Representação de uma fila. A inserção de novos elementos é feita no final da fila, e a retirada é feita no início.



Vejamos um exemplo de fila (extraído de Damas, 2007).¹

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct sPessoa
{
    int Idade;
    char Nome[20+1];
    struct sPessoa *Prox;
} PESSOA;

typedef PESSOA * FILA;

void Inic(FILA * Fila)
{
    *Fila=NULL;
}

void Inserir(FILA * Fila, int Idade, char * Nome)
{
    if(*Fila==NULL)
    {
        *Fila=(FILA)malloc(sizeof(PESSOA));
        if(*Fila==NULL) return;
        (*Fila)->Idade=Idade;
        strcpy((*Fila)->Nome, Nome);
        (**Fila).Prox=NULL;
    }
    else
        Inserir(&(**Fila).Prox, Idade, Nome);
}

main()
{
    FILA F;
    puts("Iniciando fila ... \n");
    Inic(&F);
    puts("Inserindo elemento ... \n");
    Inserir(&F, 10, "Paulo");
    printf("Elemento inserido eh %s \n\n", F->Nome);
    system("PAUSE");
}

```

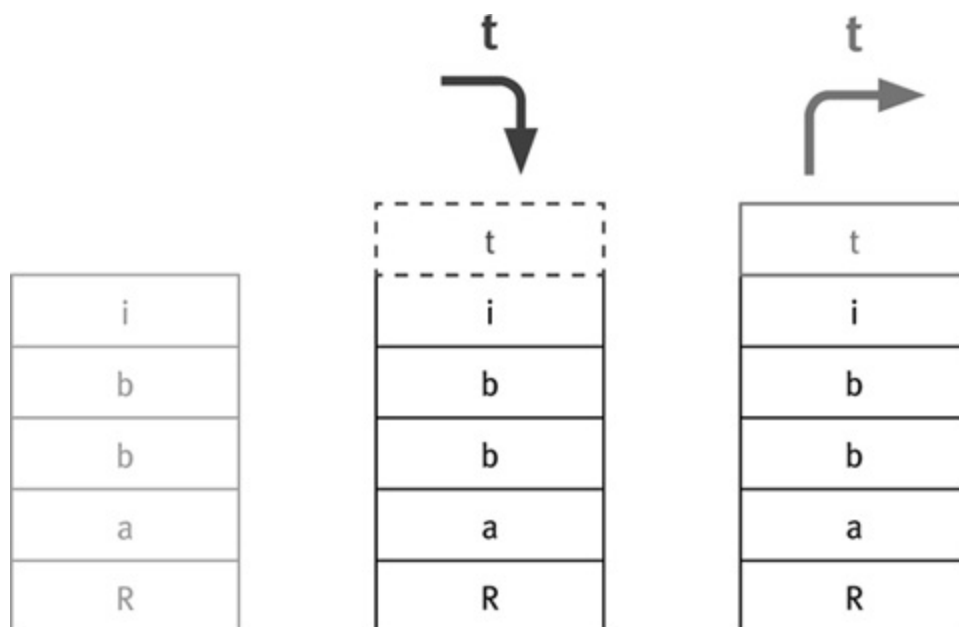
```
C:\Users\Renato\Desktop\teste_livro.exe
Iniciando fila ...
Inserindo elemento ...
Elemento inserido eh Paulo
Pressione qualquer tecla para continuar. . .
Process returned 0 (0x0)   execution time : 0.712 s
Press any key to continue.
-
```

9.3 Pilhas

São estruturas de dados nas quais as inserções e remoções são feitas no início. As pilhas representam estruturas de dados cujos elementos estão organizados dentro de um critério de entrada e saída ao qual chamamos de LIFO (*Last In, First Out*), ou seja, o último elemento a entrar é o primeiro a ser retirado (pense em uma pilha de alguns objetos).

As operações básicas implementadas em uma pilha são a inicialização da pilha, a verificação (que confirma se a pilha está vazia), o retorno do elemento que está no topo, a inserção de um elemento no topo e a retirada de um elemento do topo da pilha.

FIGURA 9.3 Representação de uma pilha. O elemento *t* sofre inicialmente um *push* (inserção na pilha), e depois um *pop* (retirada da pilha).



Vejamos um exemplo de pilha (extraído de Damas, 2007).²

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



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

```
typedef struct sNo
{
    int N;
    struct sNo *Prox;
} NO;
```

```
void Inic(NO ** Pilha)
{
    *Pilha=NULL;
}
```

```
void Push(NO ** Pilha, int Num)
{
    NO * Tmp;
    Tmp=(NO*)malloc(sizeof(NO));
    if(Tmp==NULL) return;
    Tmp->N=Num;
    Tmp->Prox=*Pilha;
    *Pilha=Tmp;
}
```

```
int Empty(NO * Pilha)
{
    return(Pilha==NULL);
}
```

```
void Pop(NO** Pilha)
{
    NO *Tmp=*Pilha;
    if(Empty(*Pilha))
        return;
    *Pilha=(*Pilha)-> Prox;
    free(Tmp);
}
```

```
void Print(NO * Pilha)
{
    if(Empty(Pilha))
        return;
    printf("%d \n", Pilha->N);
    Print(Pilha->Prox);
}
```

```
int Top(NO * Pilha)
{
    if (Empty(Pilha))
        return -1;
    return Pilha->N;
}
```

```
main()
{
```

```

    NO * P;
    Inic(&P);
printf("%s esta vazia \n", Empty(P)?"":"Nao");
Print(P);
puts("Push: 10");
Push(&P, 10);
puts("Push: 20");
Push(&P, 20);
puts("Push: 30");
Push(&P, 30);
printf("%s esta vazia \n", Empty(P)?"":"Nao");
Print(P);
puts("Pop: ");
Pop(&P);
Print(P);
puts("Pop: ");
Pop(&P);
Print(P);
puts("Pop: ");
Pop(&P);
Print(P);
printf("%s esta vazia \n", Empty(P)?"":"Nao");
}

```

```

C:\Users\Renato\Desktop\teste_livro.exe
esta vazia
Push: 10
Push: 20
Push: 30
Nao esta vazia
30
20
10
Pop:
20
10
Pop:
10
Pop:
esta vazia
Process returned 13 (0xD)   execution time : 0.155 s
Press any key to continue.

```

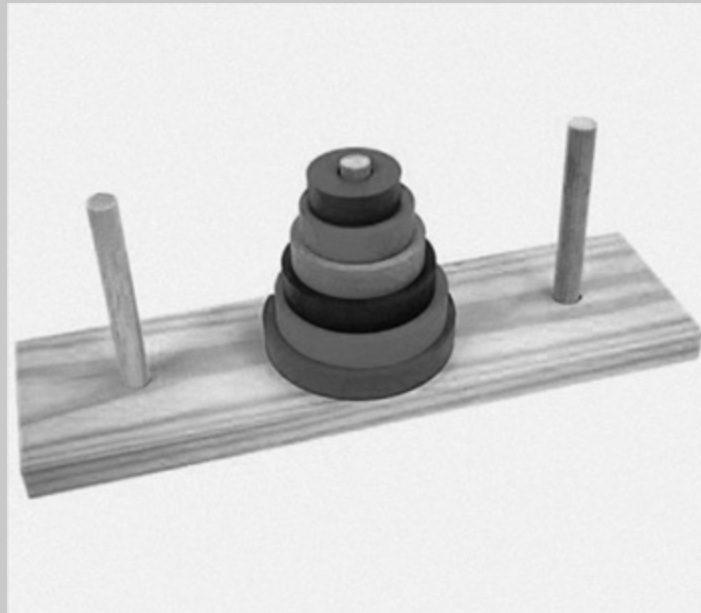
EXERCÍCIOS

1. O jogo *Torres de Hanói* é um desafio bem conhecido, constituído por uma base com três pinos. Em um deles estão dispostos discos empilhados, em ordem crescente de diâmetro, de cima para baixo. O desafio consiste em transferir todos os discos de um pino para outro, qualquer que seja, tomando um dos pinos como armazenamento temporário, mas garantindo que um disco maior nunca fique acima de outro menor. É possível provar por indução que o número mínimo de jogadas que encerra o desafio é $2n - 1$, sendo n o número de discos em uso. Por exemplo, para se ganhar o jogo de 4 discos são necessárias 15 movimentações, para 7 discos são necessárias 127 ações, e assim por diante.

[Tarefa] Analise o jogo *Torres de Hanói* como uma pilha, e pense em suas movimentações de acordo com o conteúdo visto neste capítulo (**Dica:** existem várias simulações do jogo na Internet,

basta procurar por elas com a ajuda de um mecanismo de busca.)

FIGURA 9.4 As Torres de Hanói.



Fonte: tioclebio.blogspot.com

2. Observe uma fila real em algum estabelecimento comercial ou bancário, e tente entendê-la a partir dos conceitos e do código mostrados neste capítulo.
3. Escreva um programa que implemente uma lista ligada semelhante ao exemplo apresentado no capítulo (“bandas”); escolha algum assunto de seu interesse para isso.
4. Execute o código da *fila* deste capítulo, e o altere a fim de entender bem como funciona.
5. Execute o código da *pilha* deste capítulo, e o altere a fim de entender bem como funciona.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7.ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

FEOFILOFF, Paulo. *Algoritmos em Linguagem C*. São Paulo: Campus, 2008.

MIZRAHI, Victorine Viviane. *Treinamento em Linguagem C*. 2.ed. São Paulo: Pearson/ Prentice Hall, 2008.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

ANEXO: GÖDEL E A EXISTÊNCIA DE DEUS

Apresentamos aqui a famosa prova ontológica da existência de Deus, imaginada pelo matemático Kurt Gödel. É um exemplo *sui generis* de uso da lógica para o tratamento de um assunto polêmico, não submetido, em geral, à lógica e à razão. Não deixa de ser brilhante, mesmo que muito difícil de se compreender.

Gödel era uma pessoa muito estranha e reservada, falava pouco e tinha hábitos bastante incomuns. Seu gênio matemático foi reconhecido em vida, após a descoberta pela comunidade acadêmica de seu não menos famoso Teorema de Gödel, no qual demonstra que certas afirmações e provas matemáticas e de lógica não podem ser provadas de forma definitiva.

Conta-se que um dia distribuiu a prova da existência de Deus aos seus colegas de Princeton, mas não se sabe se ele a considerava séria, ou se era apenas um exercício de imaginação:

Prova Matemática da Existência de Deus, de Kurt Gödel³

- *Axioma 1.* (Dicotomia) Uma propriedade é positiva se e somente se sua negação é negativa.
- *Axioma 2.* (Fechamento) Uma propriedade é positiva se necessariamente contém uma propriedade positiva.
- *Teorema 1* Uma propriedade positiva é logicamente consistente (possivelmente tem alguma instância).
- *Definição.* Algo é similar-a-Deus se e somente se possui todas as propriedades positivas.
- *Axioma 3.* Ser similar-a-Deus é uma propriedade positiva.
- *Axioma 4.* Ser uma propriedade positiva é (logicamente) necessário.
- *Definição.* Uma propriedade P é a essência de x se e somente se x tem P e P é necessariamente mínimo.
- *Teorema 2* Se x é similar-a-Deus, então ser similar-a-Deus é a essência de x .
- *Definição.* $NE(x)$: x necessariamente existe se tem uma propriedade essencial.
- *Axioma 5.* Ser NE é ser similar-a-Deus.
- *Teorema 3.* **Necessariamente existe algum x tal que x é similar-a-Deus.**

Arquivos



VISÃO DO CAPÍTULO

Arquivos são importantes em computação pois são a forma de se perpetuar dados em dispositivos de armazenamento, ao contrário da memória de trabalho, que é volátil; além disso, toda a organização dos computadores se baseia no conceito de arquivos, inclusive as metáforas de uso de dados e de sua hierarquia de armazenamento (pastas, diretórios). É fundamental para um programador saber registrar dados no formato de arquivos, bem como conhecer as operações de manipulação destes.

OBJETIVO INSTRUCIONAL

Após estudar o conteúdo e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- «« Entender o armazenamento de dados na forma de arquivos;
- «« Usar os arquivos como forma de registro de dados;
- «« Realizar as operações de manipulação de arquivos.



Ao contrário de outras linguagens, em que os arquivos são vistos como possuidores de uma estrutura interna ou registro associado, um arquivo, em

Linguagem C, é apenas um conjunto de *bytes* colocados uns após os outros de forma sequencial.

Um arquivo é, também, fonte de dados para o programa, podendo servir como entrada de dados (*input*); ou é, também, o destino dos dados gerados pelo programa, ou sua saída de dados (*output*).

Independentemente do periférico usado para a entrada ou saída de dados, a Linguagem C processa os *bytes* por meio de *streams* (conjunto sequencial de caracteres, ou de *bytes*, sem qualquer estrutura interna). As *streams*, portanto, independem do dispositivo utilizado e representam um conceito universal de fluxo de dados.

Cada *stream* está ligada a um arquivo, que pode não corresponder fisicamente a uma estrutura existente no disco, como é o caso do teclado ou da tela do computador.

Todo arquivo, quando aberto em Linguagem C, vai se encaixar em uma das seguintes possibilidades de padrão de *stream*:

```
stdin - a entrada-padrão, em geral o teclado;  
stdout - a saída-padrão, em geral a tela;  
stderr - a saída-padrão das mensagens de erro, em geral a tela;  
stdaux - a porta de comunicação;
```

`stdprn` - a saída para a impressora.

10.1 Operações básicas sobre arquivos

As principais tarefas que podem ser realizadas sobre arquivos são:

- *Abertura*: associamos uma variável ao arquivo com o qual se pretende trabalhar;
- *Utilização*: depois de aberto, podemos manipular o conteúdo do arquivo: ler dados, escrever dados, posicionar ponteiro em dado trecho do arquivo, entre outras tarefas vistas neste capítulo;
- *Fechamento*: devemos fechar os arquivos depois do uso para evitar perdas de dados e do próprio arquivo.

Em Linguagem C, todas essas operações com arquivos são precedidas pela letra *f* (de *file*): *fopen*, *fclose*.

10.2 Funções de manipulação de arquivos

As funções específicas para se trabalhar com arquivos são:

```
fopen( ) - abre um arquivo para trabalho;
fclose( ) - fecha o arquivo;
putc( ) - escreve um caractere no arquivo aberto;
fputc( ) - mesma função de putc( );
getc( ) - lê um caractere do arquivo de trabalho;
fgetc( ) - mesma função de getc( );
fseek( ) - posiciona o ponteiro de arquivo em um byte específico;
rewind( ) - posiciona o ponteiro de arquivo no início deste;
fprintf( ) - idem ao printf na saída-padrão;
fscanf( ) - idem ao scanf na entrada-padrão.
```

O arquivo de cabeçalho *stdio.h* define várias macros para o uso de arquivos, como NULL, FOPEN_MAX, SEEK_SET, EOF, SEEK_CUR e SEEK_END, descritas a seguir:

```
NULL definirá um ponteiro nulo no que diz respeito ao arquivo;
FOPEN_MAX define o número de arquivos que podem ser abertos simultaneamente;
SEEK_SET, SEEK_CUR e SEEK_END são utilizados com a função fseek( ) para se acessar
aleatoriamente determinado arquivo;
EOF retorna -1 quando uma função atinge o final do arquivo.
```

10.3 Trabalhando com arquivos

A partir deste ponto praticaremos diversas operações com arquivos em Linguagem C para que se obtenha um real entendimento do assunto, inclusive em sua relação com o sistema operacional.

10.3.1 Abrindo arquivos

Para se abrir um arquivo, declaramos uma variável do tipo FILE (essa é, na verdade, um ponteiro para FILE – que está definido em *stdio.h* e não é do tipo primitivo):

```
FILE *fp; // ponteiro para FILE
```

E então utilizamos a função *fopen()*:

```
FILE * fopen (const char * filename, const char * mode); /* nome do arquivo e modo de abertura */
```

Podemos testar a abertura de um arquivo da seguinte forma:

```
#include <stdlib.h>
#include <stdlib.h>
main()
{
    FILE * fp;

    char s[100];

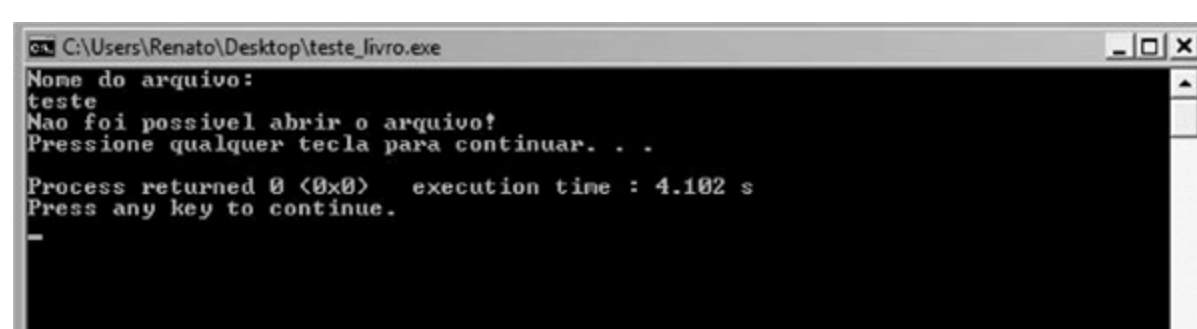
    puts("Nome do arquivo: ");
    gets(s);

    fp = fopen(s, "r");

    if(fp == NULL)
        {printf("Nao foi possivel abrir o arquivo! \n");}
    else
        {printf("Arquivo %s aberto! \n", s);

        fclose(fp);
        }

    system("pause");
}
```



```
C:\Users\Renato\Desktop\teste_livro.exe
None do arquivo:
teste
Nao foi possivel abrir o arquivo!
Pressione qualquer tecla para continuar. . .

Process returned 0 (0x0)   execution time : 4.102 s
Press any key to continue.
-
```

O teste detecta se algum problema ocorreu na abertura do arquivo (arquivo inexistente, disco cheio, arquivo corrompido ou protegido).

10.3.2 *Modos de abertura*

São vários os modos de se abrir um arquivo, em função das exigências do usuário ou do programador.

Assim:

```
"r" - read (se não abrir retorna NULL);  
"w" - write (cria novo arquivo; se não puder criar, retorna NULL);  
"a" - append (se não existir ele o cria).
```

ou

```
"r+" - abre um arquivo-texto para leitura/escrita. Se o arquivo não existir, será  
criado;  
"w+" - abre um arquivo-texto para leitura/escrita. Se o arquivo não existir, será  
criado;  
"a+" - abre um arquivo-texto para leitura/escrita. Se o arquivo não existir, será  
criado.
```

Ou ainda:

```
"rb" - abre um arquivo binário para leitura;  
"wb" - abre um arquivo binário para escrita. Se um arquivo com o mesmo nome existir,  
será sobrescrito;  
"ab" - abre um arquivo binário para anexação. Se o arquivo não existir, será criado.
```

A abertura de um arquivo pode ser feita, portanto, nos formatos de *texto* ou *binário* (o modo padrão é texto). O texto possui os caracteres tradicionais, perceptíveis por nós, mais o espaço em branco, a tabulação e o *newline*. Já o binário contém qualquer caractere da tabela ASCII, inclusive os caracteres de controle e os caracteres especiais sem representação visível (por exemplo, o `\0`).

10.3.3 *Fechando arquivos*

Para se fechar um arquivo, usamos a função *fclose()*, que escreve no disco qualquer dado que ainda possa existir em *buffer*, e o fecha no sistema operacional.

Falhas no fechamento de uma *stream* podem provocar problemas como perda de dados, arquivos corrompidos e erros no programa.

A sintaxe dessa função é:

```
int fclose(FILE * arq);
```

Nessa sintaxe, *arq* é o ponteiro para o arquivo aberto.

Se o fechamento do arquivo ocorrer sem problemas, será retornado o valor zero. Qualquer outro valor indica erro de fechamento.

Passaremos, agora, às operações de manipulação de arquivos.

10.3.4 *Escrevendo no arquivo e apagando o que existia*

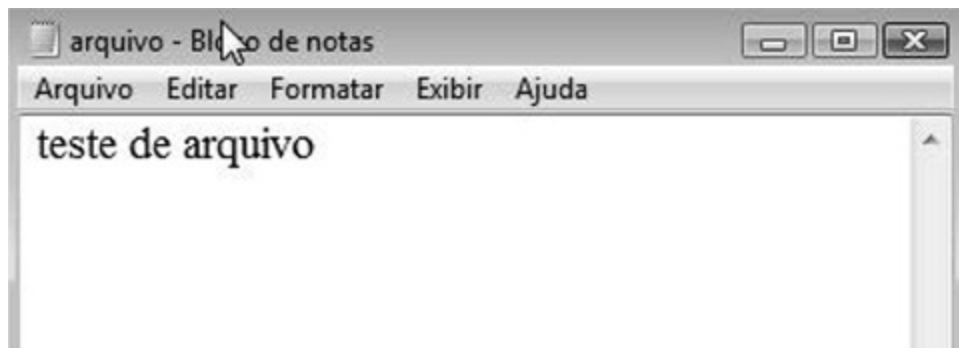
Essa operação sobrescreve o conteúdo já existente no arquivo. Em primeiro lugar, precisamos abrir o arquivo em modo que permita a escrita, e em seguida gravar o texto digitado pelo usuário:

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

main()
{
    FILE * arquivo;
    char mensagem[80];
    printf("Entre o texto: \n");
    gets(mensagem);

    if((arquivo = fopen("arquivo.txt","w")) == NULL) // mode de escrita ativo
    {
        printf("Erro de abertura! \n");
    }
    else
    {
        fprintf(arquivo, "%s \n", mensagem);
        fclose(arquivo);
    }
    system("pause");
}
```

FIGURA 10.1 Arquivo criado pelo exercício.



10.3.5 *Escrevendo no arquivo mas mantendo o que existia*

É a operação de anexação (*append*). Em primeiro lugar, precisamos abrir o arquivo em modo que permita adicionar conteúdo, e depois gravar o texto:

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

```

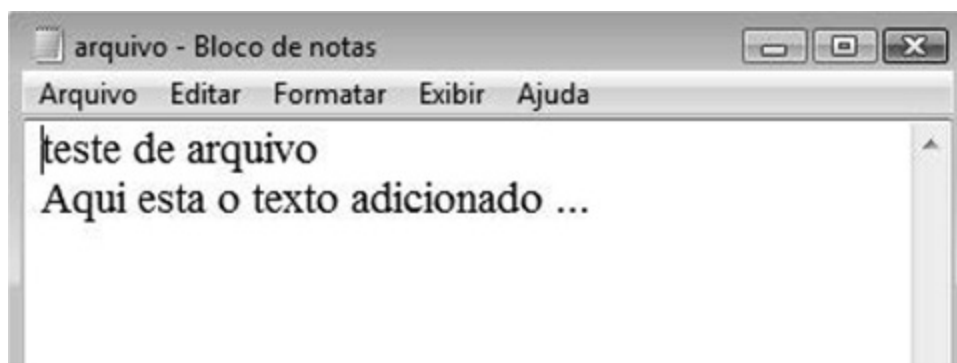
main()
{
    FILE * arquivo;
    char mensagem[80];
    printf("Entre o texto: \n");
    gets(mensagem);

    if((arquivo = fopen("arquivo.txt","a")) == NULL)
    {
        printf("Erro de abertura! \n");
    }
    else
    {
        fprintf(arquivo, "%s \n", mensagem);
        fclose(arquivo);
    }

    system("pause");
}

```

FIGURA 10.2 Arquivo criado pelo exercício, após a adição de texto.



10.3.6 *Escrevendo strings no arquivo*

Esse programa acumula cadeias de caracteres em várias linhas, até que o caractere de nova linha seja encontrado:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main( )
{
    char texto[100];
    FILE *arq;

    if((arq = fopen("teste.txt","w")) == NULL)
    {
        printf("Erro de abertura do arquivo. \n");
        exit(1);
    }
}

```

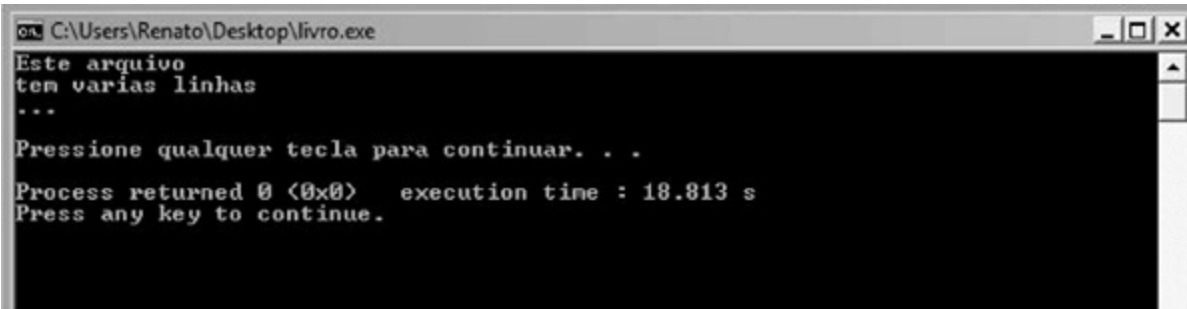
```

}

do
{
    gets(texto);
    strcat(texto, "\n");
    fputs(texto, arq);
} while(*texto != '\n');

fclose(arquivo);
system("pause");
}

```



10.3.7 *Lendo o conteúdo do arquivo*

Vamos, agora, ler o conteúdo de um arquivo já existente. A função *fgets()* fará a leitura a partir de um *buffer* de *bytes*, ou seja, um bloco de dados de determinado tamanho:

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    char buffer[128];
    FILE *arquivo;

    if((arquivo = fopen("arquivo.txt","r")) == NULL)
    {
        printf("Erro de abertura! \n");
    }
    else
    {
        fgets(buffer, 80, arquivo);
        while(!feof(arquivo))
        {
            printf("%s", buffer);
            fgets(buffer, 80, arquivo);
        }
        fclose(arquivo);
    }
    system("pause");
}

```

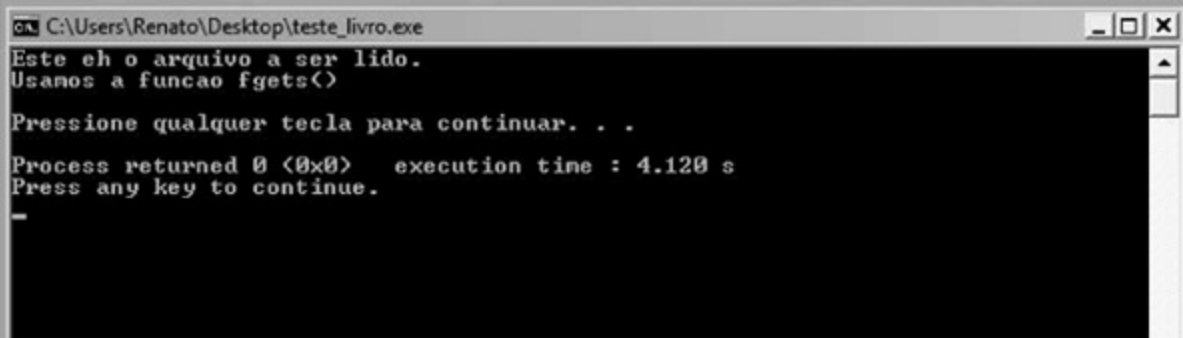
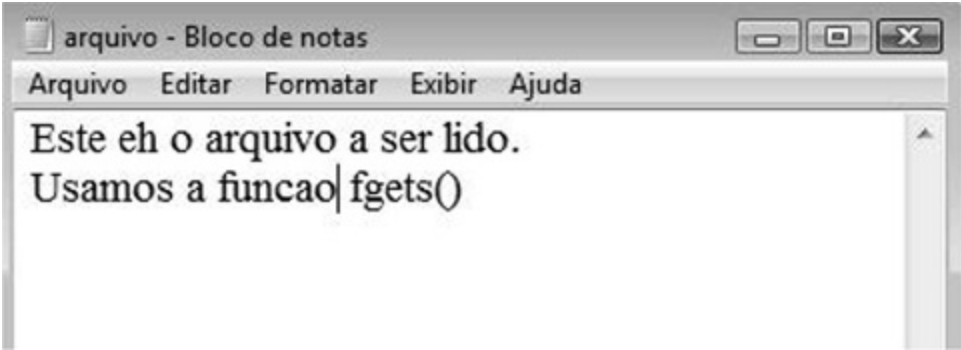


FIGURA 10.3 Arquivo lido pelo exercício, usando fgets().



10.3.8 *Solicitando o nome do arquivo*

Neste exemplo, o programa solicita ao usuário o nome que deseja dar ao arquivo:

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

main() {

    FILE *arquivo;
    char mensagem[80], nome[80];

    printf("Entre o nome do arquivo: ");
    gets(nome);

    printf("Entre a mensagem: ");
    gets(mensagem);

    if (( arquivo=fopen(nome, "w")) == NULL)
    {
        printf("O arquivo não pode ser aberto.\n");
    }
    else
    {
        fprintf(arquivo, "%s\n", mensagem);
        fclose(arquivo);
    }
}
```

```
}
system("pause");
}
```

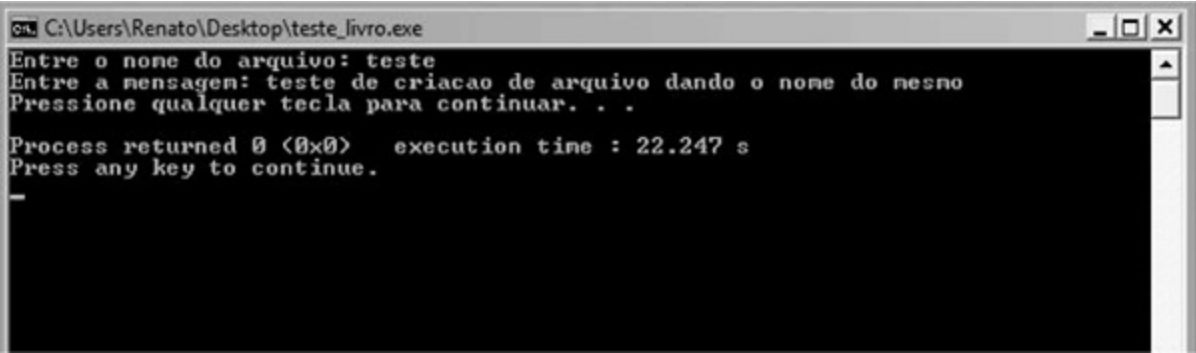
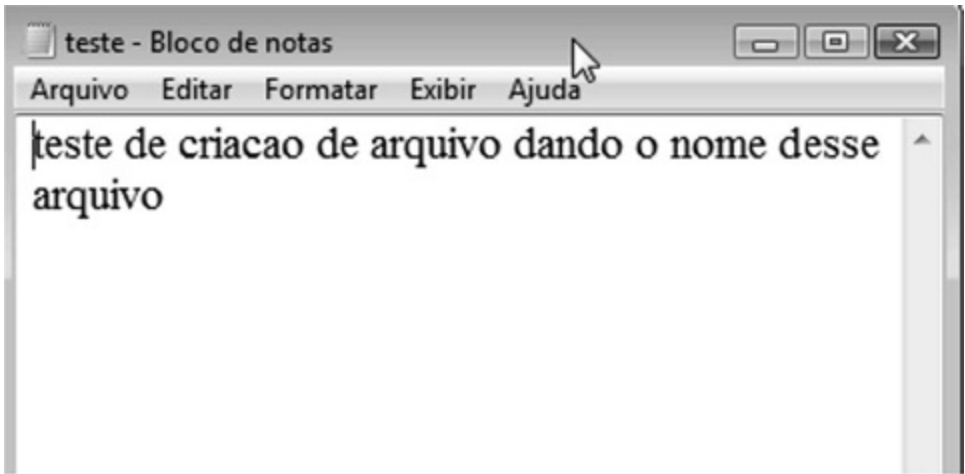


FIGURA 10.4 Arquivo criado pelo exercício de fornecimento do nome desse arquivo.



10.3.9 *Lendo e escrevendo no arquivo caractere a caractere*

As funções *putc()* e *fputc()* escrevem um caractere de cada vez em um arquivo e podem ser utilizadas de forma idêntica (são variações advindas da padronização da Linguagem C). Por exemplo:

```
do
{
    letra = getchar();
    putc(letra, nome_arquivo);
} while(caractere != '#');
```

A leitura do caractere individual, a partir do arquivo, pode ser feita por *getc()* ou *fgetc()*, da seguinte forma:

```
while((caractere=fgetc(fp))!=EOF)
{
```

```
ação  
}
```

10.3.10 Copiando um arquivo

Aqui vamos criar uma cópia de um arquivo; esse programa tem que ser executado na linha de comando do sistema operacional (em ambiente Windows basta executar o comando ‘cmd’ no *Iniciar*; em Unix/Linux, basta abrir uma janela – Terminal):

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

int main(int argc, char *argv[])
{
    FILE *original, *copia;
    int ch;

    if(argc != 3)
    {
        printf("A sintaxe correta eh: \n");
        printf("copiar arq_origem arq_destino \n");
        exit(1);
    }

    original = fopen(argv[1], "rb");
    if (original == NULL)
    {
        printf("Erro de abertura do arquivo. \n");
        exit(2);
    }

    if((copia = fopen(argv[2], "wb")) == NULL) /* usando o modo binário para qualquer arquivo */
    {
        printf("Erro na criação da copia do arquivo \n");
        exit(3);
    }
    while ((ch=fgetc(original)) != EOF)
        fputc(ch, copia);

    fclose(original);
    fclose(copia);

    system("pause");
}
```

FIGURA 10.5 Arquivo de origem, a ser copiado.

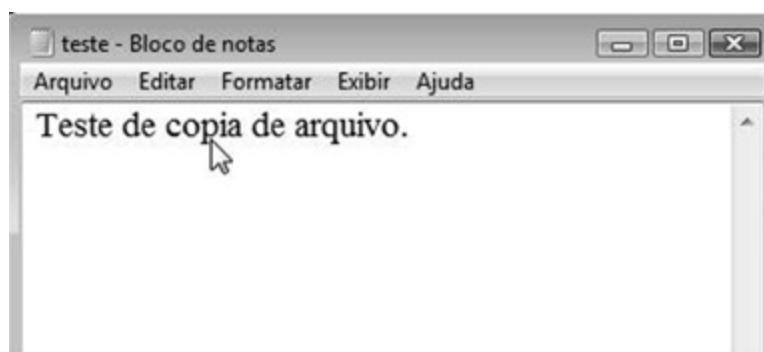
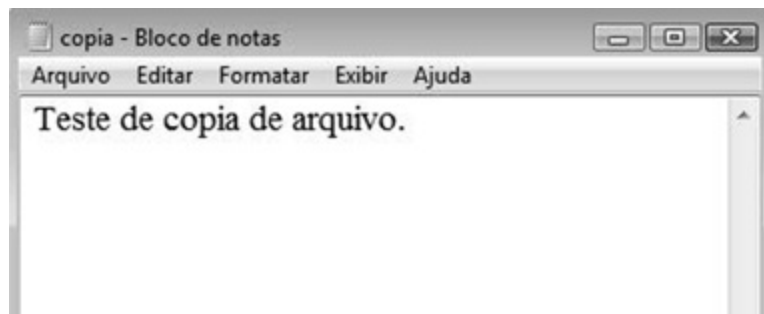


FIGURA 10.6 Arquivo de destino, cópia do original.



10.3.11 *Entrada e saída formatadas*

As funções *fprintf()* e *fscanf()* têm emprego similar a *printf()* e a *scanf()*, embora encaminhem os dados trabalhados diretamente para um arquivo. São, portanto, entrada e saída formatadas.

Vejamos um exemplo de uso:

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

main()
{
    FILE *arq;
    char texto[100];

    if((arq = fopen("teste.txt","w")) == NULL)
    {
        printf("Erro de abertura do arquivo! \n");
        exit(1);
    }

    printf("Entre uma expressao: \n");
    fscanf(stdin, "%s", texto);
    fprintf(arq, "%s", texto);
    fclose(arq);
}
```

```
system("pause");  
}
```

10.3.12 *Buscando determinada posição do arquivo e apontando para o seu início*

Podemos apontar para um local específico dentro de um arquivo pelo emprego de *fseek()*:

```
fseek(FILE, n_bytes, origem);
```

FILE é um ponteiro para um arquivo, *n_bytes* é a quantidade de *bytes* a serem movimentados, e *origem* é o ponto de partida no que diz respeito à posição – notação de macros, vista anteriormente neste capítulo.

Para se atingir o início de um arquivo, utilizamos a função *rewind()*:

```
rewind(FILE);
```

Nesse caso, *FILE* é um ponteiro para um arquivo.

EXEMPLO:

```
fseek(fp, 100, SEEK_SET); // busca o centésimo byte do arquivo  
fseek(fp, -30, SEEK_CUR); // busca 30 bytes atrás da posição atual  
fseek(fp, -10, SEEK_END); // busca o décimo byte anterior ao fim do arquivo  
fseek(fp, 0, SEEK_SET); // busca o início do arquivo  
  
rewind(fp); // busca o início do arquivo
```

Veremos, agora, algumas operações sobre arquivos relacionadas com as chamadas a sistema (*system calls*) do sistema operacional.

10.3.13 *Manipulando erros*

Para determinar se uma operação com arquivo gerou algum tipo de erro, utilizamos a função *ferror()*, da seguinte forma:

```
ferror(FILE);
```

Nesse caso, *FILE* é um ponteiro para um arquivo; a função retornará *verdade* se ocorrer um erro na manipulação do arquivo.

10.3.14 *Apagando um arquivo*

Para eliminar um arquivo, utilizamos a função *remove()*, que tem relação com a chamada de sistema equivalente do sistema operacional:

```
remove(FILE);
```


Nesse caso, FILE é um ponteiro para um arquivo.

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

int main(int argc, char *argv[])
{
    FILE * arquivo;
    char opcao[5];

    if(argc != 2)
    {
        printf("Erro de sintaxe. \n");
        exit(1);
    }

    printf("Deseja apagar o arquivo %s (s/n)?", argv[1]);
    gets(opcao);

    if((*opcao) == 's')
        if(remove(argv[1]))
        {
            printf("Erro de operacao. \n");
            exit(1);
        }
    else
        printf("Arquivo excluído. \n");

return 0;
}
```

10.3.15 *Escrevendo e lendo tipos de dados definidos pelo usuário (estruturas)*

Para ler e gravar estruturas definidas pelo usuário, usamos as funções *fread()* e *fwrite()*. O assunto das estruturas já foi visto anteriormente.

A sintaxe de *fread()* é a seguinte:

```
fread(var, tamanho, quantidade, arq);
```

Nessa sintaxe, *var* é endereço da variável que armazenará os dados lidos do arquivo; *tamanho* é o número de bytes a serem lidos; *quantidade* indica quantas unidades da dimensão de tamanho serão lidas; e *arq* é um ponteiro para o arquivo aberto de trabalho (deve ser aberto em modo binário, para que possamos manipular qualquer tipo de dado).

Já *fwrite()* tem sintaxe similar – cabe somente notar que *var* será o endereço da variável que contém os dados a serem escritos no arquivo.

EXEMPLO:

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

main()
{
    FILE *arq;
    char texto[10]="fatec";
    int alunos=3000;

    if((arq = fopen("teste.txt","wb")) == NULL)
    {
        printf("Erro de abertura do arquivo. \n");
        exit(1);
    }

    fwrite(&texto, sizeof(texto), 1, arq);
    fwrite(&alunos, sizeof(alunos), 1, arq);
    fclose(arq);

    system("pause");
}
```

Vamos, agora, ler o arquivo criado:

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

main()
{
    FILE *arq;
    char texto[10];
    int alunos;

    if((arq = fopen("teste.txt","rb")) == NULL)
    {
        printf("Erro de abertura do arquivo. \n");
        exit(1);
    }

    fread(&texto, sizeof(texto), 1, arq);
    fread(&alunos, sizeof(alunos), 1, arq);

    printf("Escola: %s \n", texto);
    printf("Alunos: %d \n", alunos);
    fclose(arq);
    system("pause");
}
```

Um outro exemplo:

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

```

struct
{
char nome[30];
int idade;
} p;

main()
{
FILE *f;
strcpy(p.nome, "Paulo");
p.idade=25;
f = fopen("teste.dat", "wb");
fwrite(&p, 1, sizeof(p), f);
fclose(f);
system("pause");
}

```

EXERCÍCIOS

1. Escreva um programa que crie um arquivo-texto com alguns caracteres gravados.
2. Escreva um programa que registre, no formato de um arquivo binário, a estrutura definida pelo usuário composta pelas variáveis *nome*, *idade* e *salario*.
3. [Desafio] Crie um “mini” editor de texto, com funções de manipulação de caracteres; poderá estar associado a um menu de opções de trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7.ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

ANEXO: MODELAGEM E SIMULAÇÃO – APRENDENDO SOBRE O MUNDO

You can't think seriously about thinking without thinking about thinking about something.
SEYMOUR PAPERT

Existe outro velho ditado, muito citado nos trabalhos de língua inglesa, que diz algo como “dê a uma

pessoa um martelo, e o mundo todo parecerá um prego”.

Uma das possíveis interpretações dessa frase poderia ser: a forma pela qual enxergamos o mundo é diretamente influenciada pelas ferramentas e meios de que dispomos em determinado momento histórico. Se dermos apenas lápis e papel a um cientista, como suas ferramentas de trabalho, isso o levará a ver o mundo no formato de equações diferenciais. Conclusão direta desse fato é que, caso tenhamos novas ferramentas e meios de trabalho, poderemos apreciar o mundo sob uma nova ótica.

A modelagem, que é a representação da realidade por meio de modelos, é atividade permanente na vida do ser humano. Quando modelamos alguma coisa, fazemos de conta que dominamos essa coisa. Nos damos os poderes de representar sua realidade e simular suas características e funções.

Os computadores também modelam e simulam. Atividades de modelagem computacional podem auxiliar pessoas comuns e gestores de negócios na passagem de modelos mentais centralizados para visões descentralizadas do mundo e dos negócios. Novos *insights* e apreciações inovadoras seriam, então, providos pelo emprego de tais ferramentas e métodos.

Para tal tarefa, há que se adotar alguns princípios centrais, característicos da modelagem descentralizada: encorajar a *construção de modelos* (e não apenas a manipulação dos modelos já existentes); repensar *o que* foi aprendido (e não apenas *como* é aprendido); estudar as possibilidades de *conexão pessoal entre assuntos* (e não apenas as abstrações matemáticas) e, finalmente, focar na *estimulação*, e não apenas na *simulação*.

Esse é um novo tipo de projeto: o *designer* controla as ações das partes, e não mais do todo. Os padrões resultantes não podem ser previstos ou projetados, já que são resultantes de um processo de emergência de comportamentos individuais.

As linguagens de programação e seu emprego são fundamentais para tais ações.

Busca e ordenação



VISÃO DO CAPÍTULO

Os algoritmos de busca e ordenação são fundamentais para a computação, pois é exatamente nesses processos que os computadores têm grande eficiência e desempenho. Boa parte das aplicações e dos sistemas computacionais precisam, em dado momento, procurar por alguma informação ou ordenar os dados. Este capítulo dá ao leitor uma visão geral do assunto.

OBJETIVO INSTRUCIONAL

Após estudar e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- Escolher o melhor método de busca para os seus algoritmos;
- Escolher o melhor método de ordenação para os seus algoritmos;
- Diferenciar entre os diversos métodos de busca e classificação, e saber das vantagens e desvantagens de cada um.



11.1 Algoritmos de busca

11.1.1 Busca linear (sequencial)

Podemos procurar um valor desejado em um vetor por meio da pesquisa sequencial. Esta começa no primeiro elemento da matriz e a percorre até o final, localizando o valor escolhido. É a forma de busca mais simples, porém pouco eficiente.

EXEMPLO:

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
```

```

int numeros[5]={50, 20, 10, 70, 15};

int i, valor;

printf("Qual eh o valor a procurar? \n");

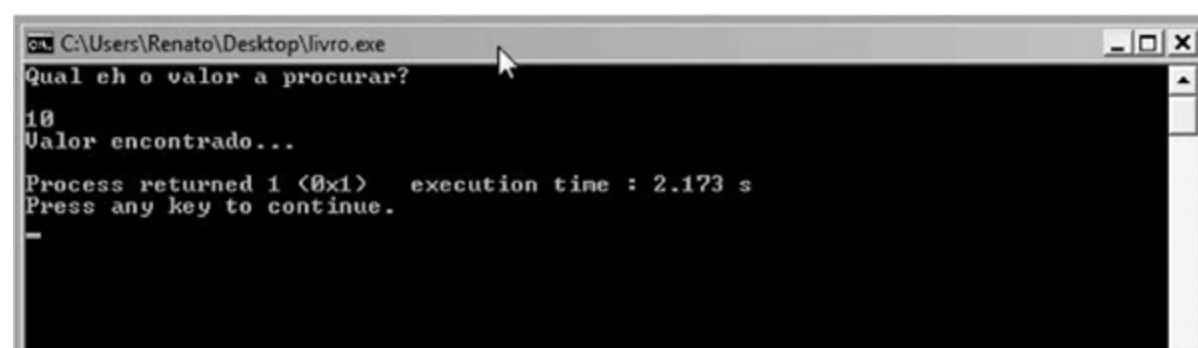
scanf("%d", &valor);

for(i=0; i<4; i++)
{
    if(valor == numeros[i])
    {
        printf("Valor encontrado... \n");
        exit(1);
    }
}

printf("Valor nao encontrado... \n");

system("pause");
}

```



Esse tipo de pesquisa pode ser ineficiente se tivermos um vetor de muitos elementos. Nesse caso, poderíamos utilizar a *busca binária*, como mostrado no próximo item.

11.1.2 *Busca binária*

Ao contrário da busca linear, a busca binária procura otimizar o processo de procura por determinado elemento de um vetor ordenado, quando considera que é possível sempre dividir o conjunto de elementos em duas partes (metade).

Como analogia, podemos citar a busca por determinada palavra em um dicionário: se buscamos a palavra “natureza”, seria uma boa estratégia abrir o dicionário no meio e decidir se a palavra

procurada está antes ou depois desse ponto. No nosso caso, descartaremos a metade inferior do livro, o que é bastante eficiente.

A seguir, tomamos a metade restante e procedemos da mesma forma, o que eliminará o último quarto de páginas do dicionário, nos trazendo mais perto da palavra buscada. Ao final de algumas iterações, localizaremos o termo “natureza”, e cumprimos nossa missão.

Importante: lembrar que o vetor tem que estar ordenado de forma crescente ou decrescente.

A seguir, um exemplo de busca binária:

```
#include <stdio.h>

#include <stdlib.h>

int busca_binaria(int vetor[], int valor, int tamanho)
{
    int achou = 0;

    int alto = tamanho, baixo = 0, meio;

    meio = (alto + baixo)/2;

    while((!achou) && (alto>=baixo))

    {
        printf("Baixo %d Meio %d Alto %d \n", baixo, meio, alto);

        if(valor == vetor[meio])

            achou=1;

        else if (valor < vetor[meio])

            alto=meio - 1;

        else

            baixo=meio + 1;

        meio=(alto+baixo)/2;

    }

    return((achou) ? meio: -1);

}

main()

{
```

```

int vetor[100], i;

for(i=0; i<100; i++)

    vetor[i] = i;

printf("Resultado da busca %d \n", busca_binaria(vetor, 33, 100));

printf("Resultado da busca %d \n", busca_binaria(vetor, 75, 100));

printf("Resultado da busca %d \n", busca_binaria(vetor, 1, 100));

printf("Resultado da busca %d \n", busca_binaria(vetor, 1001, 100));

system("pause");

}

```

```

C:\Users\Renato\Documents\C#4digo-fonte\binary_search.exe
Baixo 0 Meio 24 Alto 49
Baixo 25 Meio 37 Alto 49
Baixo 25 Meio 30 Alto 36
Baixo 31 Meio 33 Alto 36
Resultado da busca 33
Baixo 0 Meio 50 Alto 100
Baixo 51 Meio 75 Alto 100
Resultado da busca 75
Baixo 0 Meio 50 Alto 100
Baixo 0 Meio 24 Alto 49
Baixo 0 Meio 11 Alto 23
Baixo 0 Meio 5 Alto 10
Baixo 0 Meio 2 Alto 4
Baixo 0 Meio 0 Alto 1
Baixo 1 Meio 1 Alto 1
Resultado da busca 1
Baixo 0 Meio 50 Alto 100
Baixo 51 Meio 75 Alto 100
Baixo 76 Meio 88 Alto 100
Baixo 89 Meio 94 Alto 100
Baixo 95 Meio 97 Alto 100
Baixo 98 Meio 99 Alto 100
Baixo 100 Meio 100 Alto 100
Resultado da busca -1
Pressione qualquer tecla para continuar. . .

```

11.2 Algoritmos de ordenação e classificação

11.2.1 Método de ordenação BubbleSort

Por ser o mais simples método de ordenação, o *BubbleSort* não é tão eficiente, e por isso é indicado para vetores de 30 elementos no máximo.

Nesse método, os elementos do vetor são percorridos, e cada par adjacente é comparado; se necessário, o par é ordenado pela troca de posição (*swap*). Essa operação se repete até que o vetor

esteja ordenado.

Vamos explicar com base no seguinte vetor:

```
numero[0] = 80;  
numero[1] = 60;  
numero[1] = 90;  
numero[1] = 75;  
numero[1] = 65;
```

Na primeira passagem, teríamos a inversão dos dois primeiros elementos:

60, 80, 90, 75, 65

A seguir, seriam comparados 80 e 90, que não precisariam ser trocados. Mas 90 seria trocado com 75, e assim por diante:

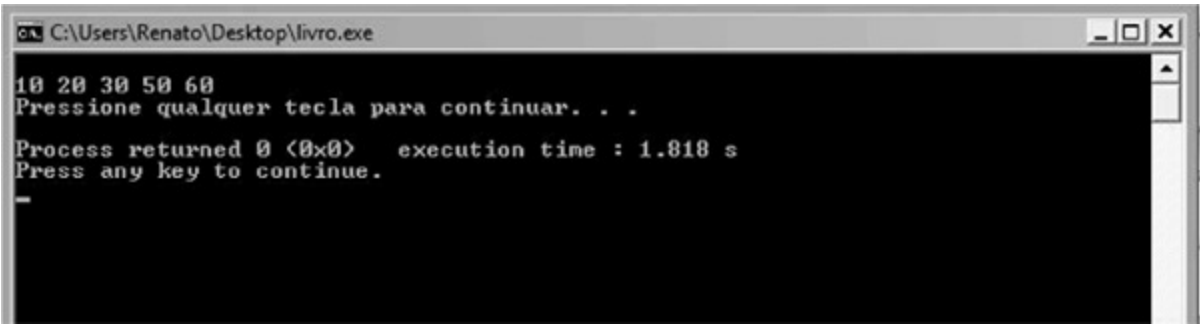
60, 80, **75, 90,** 65
60, 80, 75, **65, 90**

Um novo ciclo de comparações se inicia, dois a dois, até que todos os elementos estejam ordenados, em ordem crescente.

Como exemplo, vamos ordenar um vetor de números inteiros, em ordem crescente, e pelo “método da bolha”:

```
#include <stdio.h>  
#include <stdlib.h>  
  
main()  
{  
int vetor[5] = {50, 20, 30, 10, 60};  
int i;  
int contador;  
int auxiliar;  
int ordenados = 0; /* indica que os elementos adjacentes não estão ordenados */  
while(ordenados == 0)  
{  
ordenados = 1; /* considera todos os elementos ordenados corretamente */  
for(contador=0;contador < 4;contador++)  
{  
  
if(vetor[contador] > vetor[(contador + 1)])  
  
{  
auxiliar = vetor[contador];  
vetor[contador] = vetor[(contador + 1)];  
vetor[(contador + 1)] = auxiliar;  
ordenados = 0; /* força outra passagem no laço while */  
}  
}  
}  
  
/* imprimindo os valores ordenados */
```

```
printf("\n");
for(contador=0;contador < 5;contador++)
    printf("%d ",vetor[contador]);
printf("\n");
system("pause");
}
```



11.2.2 Método de ordenação QuickSort

Utilizado em vetores com muitos elementos, o *QuickSort* é um método muito eficiente.

O algoritmo *QuickSort* inicialmente seleciona o valor posicionado no centro da lista de elementos, ao qual chamaremos *elemento central*. Em seguida, divide o vetor em duas listas menores, separando, em uma delas, os elementos cujos valores são maiores que o valor do elemento central e, na outra lista, os elementos cujos valores são menores que o valor do elemento central. Entra-se, em seguida, em um processo recursivo em cada uma das listas, as quais tornam-se sempre menores, até que o vetor esteja todo ordenado.

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

void quick_sort(int array[ ], int primeiro, int ultimo)
{
    int temp, baixo, alto, separador;
    baixo = primeiro;
    alto = ultimo;
    separador = array[(primeiro + ultimo) / 2];
    do
    {
        while(array[baixo] < separador)
            baixo++;

        while(array[alto] > separador)
            alto--;

        if(baixo <= alto)
```

```

    {

        temp = array[baixo];
        array[baixo++] = array[alto];
        array[alto--] = temp;
    }

} while (baixo <= alto);

if(primeiro < alto)

    quick_sort(array, primeiro, alto);

if(baixo < ultimo)

    quick_sort(array, baixo, ultimo);
}

main()

{

    int valores[100], i;

    for(i=0; i<100; i++)

        valores[i] = rand( ) % 100;

    quick_sort(valores, 0, 99);

    for(i=0; i<100; i++)

        printf("%d ", valores[i]);

}

```

```

C:\Users\Renato\Desktop\livro.exe
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29
29 31 33 34 35 35 36 37 37 38 38 39 40 40 41 41 41 41 42 42 42 44 44 45 46 47 47
48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 7
8 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99
Process returned 2293164 (0x22FDAC) execution time : 0.140 s
Press any key to continue.

```

11.2.3 *Outros tipos de algoritmos de ordenação*

Existem outros algoritmos de ordenação, os quais apenas citaremos, tendo em vista que os dois métodos mostrados ilustram muito bem o problema de localização de elementos dentro de uma estrutura de dados tradicional.

Método SelectionSort

Parecido com o *BubbleSort*; inicia com um elemento (em geral o primeiro) e percorre a estrutura até achar o menor dos valores, que é colocado naquela posição; seleciona, então, um segundo elemento, e busca pelo segundo menor elemento da estrutura, que é então alocado na segunda posição do vetor – e assim por diante, até que o vetor esteja ordenado.

Método ShellSort

Compara elementos separados por determinada distância (*gap*) até que os elementos que ele identifica com a distância atual estejam ordenados. O algoritmo divide então o *gap* em dois, e o processo continua, até a ordenação completa.

Ordenação de strings

Podemos também ordenar vetores de cadeias de caracteres, o que é interessante para listas de nomes e de expressões alfanuméricas.

Como exemplo, eis o algoritmo *BubbleSort* em ação:

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

void bubble_sort(char *array[ ], int size)
{
    char *temp;

    int i, j;

    for(i=0; i < size; i++)
        for(j=0; j < size; j++)
            if(strcmp(array[i], array[j]) < 0)
            {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
}

main()
{
    char *values[ ] = {"AAA", "CCC", "BBB", "EEE", "DDD"};

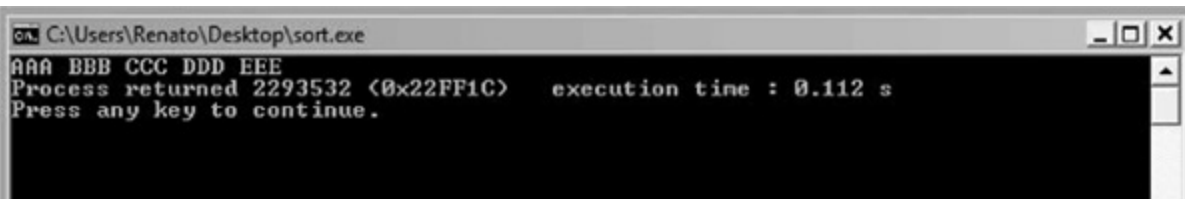
    int i;

    bubble_sort(values, 5);
```

```
for(i=0; i < 5; i++)
```

```
    printf("%s " , values[i]);
```

```
}
```



EXERCÍCIOS

1. O que é ordenação? Por que ela é importante para as aplicações de programação de computadores?
2. Explique os diferentes tipos de ordenação apresentados neste capítulo, refazendo os exercícios com alterações de sua autoria.
3. O que é busca em computação? Por que a busca é importante para as aplicações de programação de computadores?
4. Explique os diferentes tipos de busca apresentados neste capítulo, refazendo os exercícios com alterações de sua autoria.
5. Escreva um programa que ordena (de forma decrescente) dez números inteiros digitados pelo usuário.
6. Escreva um programa que receba os nomes de cinco alunos de uma turma e os liste em ordem alfabética.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à estrutura de dados: com técnicas de programação em C*. 7.ed. São Paulo: Campus/Sociedade Brasileira de Computação (SBC), 2004.

DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.

SCHILDT, Herbert. *C completo e total*. 3.ed. São Paulo: Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson Makron Books, 1995.

Considerações finais

Os temas aqui discutidos podem ser utilizados por você, leitor, como forma de desenvolvimento de novas competências necessárias para sua vida profissional, e para o posicionamento frente ao emprego da tecnologia nos processos diários.

Assuma, então, uma posição de pessoa criativa e inovadora, analise os problemas do dia a dia de forma mais lógica, racional, sistemática. A programação de computadores é única no desenvolvimento de habilidades para se lidar com a solução de problemas, pois trabalha o nível de abstração de nossas mentes, e não apenas o prático e o senso comum.

Desejo-lhe sucesso nesses enfrentamentos. Citando Virgílio, “audaces fortuna juvat” – *a sorte favorece os bravos*. Candidate-se a esse nível de audácia e... *bom trabalho!*

Assuntos complementares

Enumerações

A Linguagem C pode trabalhar tipos enumerados, como estes:

```
enum dias {segunda, terça, quarta, quinta, sexta, sábado, domingo};
```

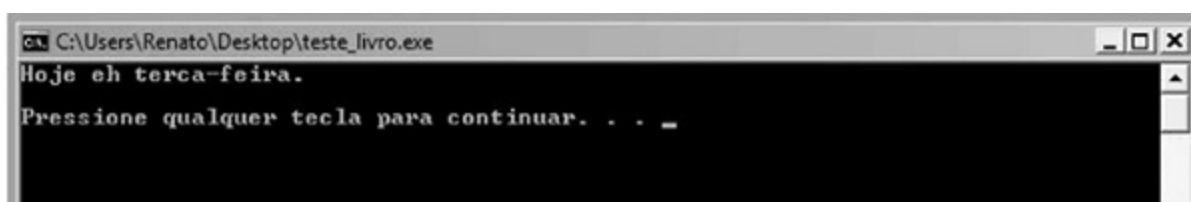
Aqui, passamos a ter um novo conjunto de valores que poderemos utilizar como se fossem constantes da linguagem.

Consideremos o seguinte programa, como ilustração:

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

enum dias {segunda, terça, quarta, quinta, sexta, sábado, domingo};

main()
{
    int a, b;
    a=segunda;
    b=terça;
    a++;
    if(a==terça)
        puts("Hoje eh terça-feira. \n");
    system("pause");
}
```



Repare que o compilador atribuiu valores sequenciais a partir de zero para os dias da semana, em que *segunda* ficou definida como 0, *terça* como 1, e assim por diante.

União

A palavra reservada *union* serve para declarar estruturas especiais. É semelhante às já mencionadas estruturas tradicionais definidas pelo usuário, embora seja diferente no que diz respeito ao armazenamento.

O exemplo a seguir mostra essa diferença:

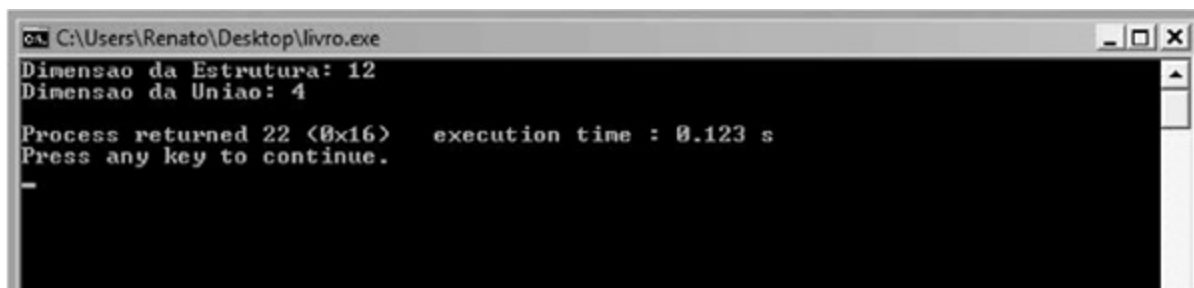
```
#include <stdio.h>
```

```
#include <stdlib.h>

struct Est
{
    char c;
    int n;
    float x;
};

union Uni
{
    char c;
    int n;
    float x;
}

main( )
{
    printf("Dimensao da Estrutura: %d \n", sizeof(struct Est));
    printf("Dimensao da Uniao: %d \n", sizeof(union Uni));
}
```



```
C:\Users\Renato\Desktop\livro.exe
Dimensao da Estrutura: 12
Dimensao da Uniao: 4

Process returned 22 (0x16)   execution time : 0.123 s
Press any key to continue.
-
```

Aplicações

Apresentaremos, a título de fechamento deste livro, alguns problemas reais e ilustrativos que servirão de guia para um estudo mais profundo da Linguagem C em seu potencial de utilização prática.

Um pequeno sistema estatístico

Faz a distribuição dos valores e calcula amplitude e moda.

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

float truncF(float value, int shift)
{
```



```

        return (int)(value * pow(10,shift)) / (float)pow(10,shift);
    }

main()
{
    int i;
    int num=11;
    float dados[]={2.5, 3.0, 1.5, 1.8, 3.2, 1.0, 2.0, 2.0, 2.7, 1.5, 2.8};
    float w;

    printf("Ordenando os dados... \n");

    float auxiliar;
    int ordenados = 0; /* indica que os elementos adjacentes não estão ordenados */
    while(ordenados == 0)
    {
        ordenados = 1; /* considera todos os elementos ordenados corretamente */
        for(i=0; i<num-1; i++)
        {
            if(dados[i] > dados[(i + 1)])
            {
                auxiliar = dados[i];
                dados[i] = dados[(i + 1)];
                dados[(i + 1)] = auxiliar;
                ordenados = 0; /* força outra passagem no laço while */
            }
        }
    }

    /* imprimindo os valores ordenados */

    printf("\n");
    for(i=0; i<num; i++)
        printf("%.1f", dados[i]);
    printf("\n \n \n");

    printf("Menor elemento = %.1f e maior elemento = %.1f \n", dados[0], dados[num-1]);
    printf("Amplitude = %.1f\n", dados[num-1]-dados[0]);
    printf("Numero de intervalos k = %.0f\n", sqrt(num));
    w=truncF((dados[num-1]-dados[0])/sqrt(num), 1);
    printf("Intervalo das classes c = %f \n\n", w);

    printf(" :: Distribuicao dos Dados:: \n\n");
    printf("-----\n");
    printf("Faixa de valores \n");
    printf("-----\n");

    float r[(int)(sqrt(num))][2];

    int c=0;
    r[c][0] = dados[0];
    printf(" %.1f|--", r[c][0]);
    r[c][1] = r[c][0] + w;
    printf("%.1f \n", r[c][1]);

    int cont=0;
    for(i=0; k<num; i++)
    {

```

```

        if(dados[i]<r[c][1])
        {
            cont++;
        }
    }
    printf(" n(i) = %d \t fr(i) = %.1f %%\n", cont, 100*(cont/43.0));

float total=0;
total = cont;

for(c=1; c<=sqrt(num); C++)
{

    r[c][0] = r[c-1][0]+w;
    printf("%.1f \n", r[c][0]);
    r[c][1]=r[c-1][0] + 2*w;
    printf("%.1f \n", r[c][1]);

    int cont=0;
    for(i=0; i<num; i++)
    {

        if(dados[i]<r[c][1] && dados[i]>=r[c][0])
        {
            cont++;
        }

    }
    printf(" n(i) = %d \t fr(i) = %.1f %%\n", cont, 100*(cont/(float)num)); //
    total = total + cont;

}
printf(" ----- \n");
printf(" Total = %.1f \n\n\n", total);

int j, maior=0;
int repet;
int n;

n = sizeof(dados)/sizeof(float);
for(i=0; i<n; i+=1+repet)
{
    repet=0;
    for(j=i+1; j<n; j++)
    {
        if(dados[i] == dados[j])
            repet++;
    }
    printf("Repeticao de %.1f = %d \n", dados[i], repet);
    if(repet>=maior)
        maior=repet;
}
printf("Maior numero de repeticoes = %d \n", maior);

n = sizeof(dados)/sizeof(float);
for(i=0; i<n; i+=1+repet)
{
    repet=0;

```

```

for(j=i+1; j<n; j++)
{
    if(dados[i] == dados[j])
        repet++;
}
if(repet==maior)
{
    printf("%.1f eh moda. \n", dados[i]);
}
}
system("pause");
}

```

Jogo da Velha

O tradicional jogo conhecido por todos.

Deve ser jogado por meio de coordenadas (x, y), variando de 0 a 2; por exemplo, a primeira casa do tabuleiro é (0, 0), e a última é (2, 2); a entrada de dados é feita por meio desses dois números, separados por vírgula.

```

#include <stdio.h>
#include <stdlib.h>

```

```

void inicia(char s[3][3])
{
    int i,j;
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
        {
            s[i][j]=' '
        }
    }
}

```

```

void mostra(char s[3][3])
{
    int i, j;
    for(i=0; i<3; i++)
    {
        printf("\n");
        for(j=0; j<3; j++)
        {
            printf("| %c      s[i][j]);
        }
        printf("\n");
    }
}

```

```

if((s[0][0]=='x' && s[0][1]=='x' && s[0][2]=='x') ||
    (s[1][0]=='x' && s[1][1]=='x' && s[1][2]=='x') ||
    (s[2][0]=='x' && s[2][1]=='x' && s[2][2]=='x') ||
    (s[0][0]=='x' && s[1][0]=='x' && s[2][0]=='x') ||
    (s[0][1]=='x' && s[1][1]=='x' && s[2][1]=='x') ||
    (s[0][2]=='x' && s[1][2]=='x' && s[2][2]=='x') ||
    (s[0][0]=='x' && s[1][1]=='x' && s[2][2]=='x')

```

```

        (s[0][2]=='x' && s[1][1]=='x' && s[2][0]=='x'))
    {
        printf("\n\a\t\tJogador x venceu! \n");
        system("pause");

        exit(1);
    }

    if((s[0][0]=='o' && s[0][1]=='o' && s[0][2]=='o')||
        (s[1][0]=='o' && s[1][1]=='o' && s[1][2]=='o')||
        (s[2][0]=='o' && s[2][1]=='o' && s[2][2]=='o')||
        (s[0][0]=='o' && s[1][0]=='o' && s[2][0]=='o')||
        (s[0][1]=='o' && s[1][1]=='o' && s[2][1]=='o')||
        (s[0][2]=='o' && s[1][2]=='o' && s[2][2]=='o')||
        (s[0][0]=='o' && s[1][1]=='o' && s[2][2]=='o')||
        (s[0][2]=='o' && s[1][1]=='o' && s[2][0]=='o'))
    {
        printf("\n\a\t\tJogador o venceu! \n");
        system("pause");
        exit(1);
    }
    /*if(cont==9)
    {
        printf("Partida empatada! \n");
    }*/
}

main()
{
    char velha[3][3];
    int x, y, z, w;
    inicia(velha);
    while(1)
    {
        printf("\n\n Faca sua jogada (x): \n");
        scanf("%d %d", &x, &y);
        velha[x][y]='x';
        system("cls");
        mostra(velha);
        printf("\n");
        printf("\n\n Faca sua jogada (o): \n");
        scanf("%d %d", &z, &w);
        velha[z][w]='o';
        system("cls");
        mostra(velha);
        printf("\n");
    } ;
}

```

1. Dennis Ritchie, em seu clássico livro de introdução à Linguagem C, escreveu: “the only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: Print the words *hello, world.*” RITCHIE, D. M.; KERNIGHAN, B. W. *The C programming language*. 2.ed. Upper Saddle River: Prentice Hall, 1988.

1. A série *The Big Bang Theory* é muito interessante e divertida, pois mostra o lado *nerd* que algumas pessoas especiais têm! Acaba sendo um pouco do conceito que as pessoas, em geral, têm sobre aqueles que gostam de Tecnologia (e de estudar...). No episódio em questão, chamado *The Friendship Algorithm*, o personagem Sheldon precisa utilizar um novo supercomputador que chegou à universidade onde trabalha. Seu colega Leonard o informa que o não muito querido pesquisador Kripke controla quem pode usar o computador, e dá preferência aos amigos... Sheldon decide, então, *ficar amigo de Kripke*. Para isso, decide adotar um enfoque científico para a ação, qual seja, desenhar um *algoritmo* que seja infalível na tarefa... (Assista ao episódio e saiba o que realmente aconteceu!) Veja mais informações sobre a série no site brasileiro da Warner: www.br.warnerbros.com.

1. Os livros de Donald Ervin Knuth (Stanford University) são clássicos no tema Algoritmos Computacionais. Cf. KNUTH, Donald E. *The art of computer programming*: Vol 1, Fundamentals algorithms. 3.ed. Boston: Addison-Wesley, 1997.

1. DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.
2. DAMAS, Luis. *Linguagem C*. Rio de Janeiro: LTC, 2007.
3. Tradução livre do autor.