

Algoritmos e Lógica de Programação em C

Uma Abordagem Didática

Algoritmos e Lógica de Programação em C

Uma abordagem didática



Silvio do Lago Pereira

Algoritmos e Lógica de Programação em C

Uma abordagem didática

1^a Edição



www.editoraerica.com.br

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

Pereira, Silvio do Lago

Algoritmos e Lógica de Programação em C: Uma abordagem didática / Silvio do Lago Pereira.
-- 1. ed. -- São Paulo : Érica, 2010.

Bibliografia.

ISBN 978-85-365-0966-2

1. Algoritmos 2. C (Linguagem de programação para computadores) 3. Dados – Estruturas
(Ciência da Computação) I. Título.

10.12237

CDD-005.1

Índice para catálogo sistemático:

1. Algoritmos: Computadores: Programação: Processamento de dados 005.1

Copyright © 2010 da Editora Érica Ltda.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida por qualquer meio ou forma sem prévia autorização da Editora Érica. A violação dos direitos autorais é crime estabelecido na Lei nº 9.610/98 e punido pelo Artigo 184 do Código Penal.

Coordenação Editorial: Rosana Arruda da Silva
Capa: Maurício S. de França
Editoração e Finalização: Roseane Gomes Sobral
Marlene Teresa S. Alves
Carla de Oliveira Moraes
Avaliação Técnica: José Augusto N. G. Manzano

O Autor e a Editora acreditam que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia, explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, porventura mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação. Eventuais erratas estarão disponíveis para download no site da Editora Érica.

Conteúdo adaptado ao Novo Acordo Ortográfico da Língua Portuguesa, em execução desde 1º de janeiro de 2009.

A Ilustração de capa e algumas imagens de miolo foram retiradas de <www.shutterstock.com>, empresa com a qual se mantém contrato ativo na data de publicação do livro. Outras foram obtidas da Coleção MasterClips/MasterPhotos® da IMSI, 100 Rowland Way, 3rd floor Novato, CA 94945, USA, e do CorelDRAW X5 e X6, Corel Gallery e Corel Corporation Samples. Copyright® 2013 Editora Érica, Corel Corporation e seus licenciadores. Todos os direitos reservados.

Todos os esforços foram feitos para creditar devidamente os detentores dos direitos das imagens utilizadas neste livro. Eventuais omissões de crédito e copyright não são intencionais e serão devidamente solucionadas nas próximas edições, bastando que seus proprietários contatem os editores.

Seu cadastro é muito importante para nós

Ao preencher e remeter a ficha de cadastro constante no site da Editora Érica, você passará a receber informações sobre nossos lançamentos em sua área de preferência.

Conhecendo melhor os leitores e suas preferências, vamos produzir títulos que atendam suas necessidades.

Contato com o editorial: editorial@editoraerica.com.br

Editora Érica Ltda. | Uma Empresa do Grupo Saraiva

Rua São Gil, 159 - Tatuapé

CEP: 03401-030 - São Paulo - SP

Fone: (11) 2295-3066 - Fax: (11) 2097-4060

www.editoraerica.com.br



Fabricante

Produto: **Pelles C for Windows**

Fabricante: Pelle Orinius

Site: www.pellesc.de

Produto: **Microsoft Windows XP / Microsoft Windows Vista**

Fabricante: Microsoft Corporation

Site: www.microsoft.com

Microsoft Informática Ltda.

Av. Nações Unidas, 12.901 - Torre Norte - 27º andar

CEP: 04578-000 - São Paulo - SP

Fone: (11) 4706-0900

Site: www.microsoft.com.br

Produto: **GCC**

Fabricante: Free Software Foundation, Inc.

Site: <http://gcc.gnu.org>

Produto: **Debian GNU/Linux**

Fabricante: Free Software Foundation, Inc.

Site: <http://www.debian.org>

Requisitos de hardware e de software

Plataforma Windows

- Compilador *Pelles C for Windows*, v. 6.00.4, disponível gratuitamente em <http://www.smorgasbordet.com/pellesc>
- Computador de 32 bits
- Sistema operacional *Microsoft Windows XP* ou *Microsoft Windows Vista*
- Mínimo de 64MB de memória RAM
- Mínimo de 40MB de espaço em disco

Plataforma Unix/Linux

- Compilador *GCC*, v. 4.3.2, disponível gratuitamente em <http://gcc.gnu.org>
- Computador de 32 bits
- Sistema operacional *Debian GNU/Linux*, v. 4.3.2-1.1, disponível gratuitamente em <http://www.debian.org>
- Mínimo de 64MB de memória RAM
- Mínimo de 5GB de espaço em disco



Dedicatória

Dedico este livro à minha mãe,
que sempre dedicou tanto a mim.

*“O melhor presente que vocês me podem dar
é escutar com atenção as minhas palavras.”*

J6, 21:2

Agradecimentos

Aos colegas da Universidade Cidade de São Paulo e da Faculdade de Tecnologia de São Paulo pela oportunidade de discutir algumas das ideias que contribuíram para o desenvolvimento deste livro;

Aos alunos dos cursos tecnológicos de Análise e Desenvolvimento de Sistemas e Gestão de Tecnologia de Informação pela participação nas aulas de Algoritmos e Lógica de Programação, a partir das quais este livro foi elaborado, e para as quais ele é especialmente dirigido;

À Editora Érica que me proporcionou a oportunidade de ver este livro publicado.



Sumário

Capítulo 1 - Introdução.....	15
1.1 Algoritmo	15
1.2 Fluxograma.....	16
1.3 Programa	18
1.4 Criação de programas e compiladores.....	20
1.5 Programação estruturada.....	22
1.6 Exercícios	23
Capítulo 2 - Sequência	25
2.1 Tipos de dados, constantes e variáveis	25
2.2 Operadores, expressões e atribuição.....	26
2.3 Operações de entrada e saída de dados	27
2.4 Estrutura sequencial	30
2.5 A biblioteca de funções matemáticas	32
2.6 Exercícios	33
Capítulo 3 - Seleção Simples	35
3.1 Expressões condicionais	35
3.2 Estrutura de seleção simples	36
3.3 Uso de blocos e omissão de alternativa	38
3.4 Estruturas de seleção simples encaixadas	40
3.5 Exercícios	43
Capítulo 4 - Seleção Múltipla	45
4.1 Estruturas de seleção encadeadas	45
4.2 Estrutura de seleção múltipla.....	48
4.3 Variações do comando switch-case.....	50
4.4 Exercícios	53
Capítulo 5 - Repetição Contada.....	55
5.1 Acumuladores e contadores	55
5.2 Estrutura de repetição contada	56
5.3 Contagem decrescente.....	58
5.4 Estruturas de repetição encaixadas.....	59
5.5 Exercícios	61



Capítulo 6 - Repetição com Precondição.....	63
6.1 Estrutura de repetição com precondição	63
6.2 Repetição com terminação forçada	65
6.3 Simulação de pingue-pongue.....	67
6.4 Exercícios	68
Capítulo 7 - Repetição com Poscondição.....	69
7.1 Estrutura de repetição com poscondição	69
7.2 Consistência de entrada de dados.....	69
7.3 Repetição com confirmação do usuário	72
7.4 Processos orientados por menus	73
7.5 Exercícios	76
Capítulo 8 - Macros e Funções	77
8.1 Macros.....	77
8.2 Funções	80
8.3 Tipos de funções.....	81
8.4 Uso de protótipos	84
8.5 Exercícios	85
Capítulo 9 - Vetores	87
9.1 Armazenamento.....	87
9.2 Vetor com tamanho variável	88
9.3 Vetor como parâmetro de função	89
9.4 Iniciação de vetor com tamanho constante.....	90
9.5 Exercícios	93
Capítulo 10 - Ordenação e Busca.....	95
10.1 Ordenação pelo método da bolha.....	95
10.2 Busca linear	97
10.3 Busca binária	99
10.4 Exercícios.....	101
Capítulo 11 - Strings	103
11.1 Armazenamento	103
11.2 Leitura e exibição de strings	104
11.3 Funções para manipulação de strings.....	105
11.4 A biblioteca string.h.....	108
11.5 Exercícios.....	111



Capítulo 12 - Matrizes	113
12.1 Armazenamento	113
12.2 Ordenação de strings	113
12.3 Jogo da velha.....	116
12.4 Exercícios.....	118
Capítulo 13 - Arquivos de Registros	121
13.1 Registros	121
13.2 Arquivos de registros.....	122
13.3 Uma agenda eletrônica	124
13.4 Exercícios.....	128
Apêndice A - Tabela ASCII.....	129
Apêndice B - O Compilador Pelles C	133
Apêndice C - Adaptação para Unix/Linux	137
Apêndice D - Solução dos Exercícios.....	143
Bibliografia	187
Índice Remissivo	189



Sobre o autor

Silvio do Lago Pereira é professor universitário na área de ciência da computação desde 1990, graduado e licenciado em Tecnologia de Processamento de Dados pela Universidade Estadual Paulista, especialista em Automação Industrial pela Escola de Engenharia Industrial de São José dos Campos, e mestre e doutor em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo.

Atualmente é pesquisador e consultor *ad hoc* do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), com bolsa de produtividade em pesquisa nível PQ-2, assessor *ad hoc* da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), professor doutor da Universidade Cidade de São Paulo (Unicid) e professor Pleno II da Faculdade de Tecnologia de São Paulo (Fatec-SP), ministrando as disciplinas de Algoritmos e Lógica de Programação, Estruturas de Dados, Programação em Lógica e Inteligência Artificial.

Autor do livro Estruturas de Dados Fundamentais - Conceitos e Aplicações, publicado pela Editora Érica.



Prefácio

Ao longo de vários anos ministrando a disciplina de **Algoritmos e Lógica de Programação em C** a calouros de cursos superiores tecnológicos, pude observar a grande dificuldade que iniciantes em programação têm em compreender a natureza dinâmica de um algoritmo. Particularmente, quando se usa pseudocódigo, os alunos demoram a entender que a execução de um algoritmo não ocorre da mesma forma que lemos seu pseudocódigo, ou seja, da primeira à última linha, sequencialmente, sem saltar linhas ou retornar a linhas anteriores. De fato, para iniciantes em programação, o comportamento dinâmico de um algoritmo acaba sendo mascarado pela natureza estática do texto (pseudocódigo) que o descreve. Por outro lado, pelo fato de os cursos tecnológicos terem duração mais curta que aqueles de licenciatura e bacharelado, os seus professores têm menos tempo para introduzir os conceitos necessários para o projeto de algoritmos e sua implementação numa linguagem de programação.

Neste contexto, o uso de fluxograma, em vez de pseudocódigo, tem a vantagem de facilitar a compreensão do funcionamento dos algoritmos para iniciantes. Por exemplo, ao seguir os passos de um algoritmo descrito por um fluxograma, pode-se entender mais facilmente a dinâmica das estruturas de seleção e repetição, já que esta é representada explicitamente pelas setas que ligam os símbolos que compõem o fluxograma.

Visando uma abordagem mais didática, neste livro, o fluxograma é usado como uma ferramenta básica para a introdução dos conceitos de sequência, seleção e repetição. Vale ressaltar que o uso de fluxogramas neste livro não tem o objetivo de documentar programas, mas sim o de facilitar a compreensão da dinâmica dos algoritmos. Assim, após a introdução das estruturas de controle básicas nos capítulos iniciais, os fluxogramas não são mais usados. Além disso, para que o aluno seja iniciado o mais rapidamente possível na programação em C, todos os algoritmos descritos por fluxogramas são também codificados nesta linguagem.

O livro foi elaborado de modo que cada capítulo possa ser coberto em aproximadamente quatro horas de aula teórica e duas horas de aula prática em laboratório, com uma carga horária total de aproximadamente 80 horas.

Os principais objetivos deste livro são:

- Introduzir o conceito de algoritmo computacional.
- Apresentar as técnicas para projeto de algoritmos estruturados usando fluxograma.
- Destacar os comandos para a implementação de algoritmos estruturados em C.
- Apresentar as estruturas básicas de armazenamento e manipulação de dados em C.



Outras características deste livro são:

- Apresenta os conceitos essenciais de programação de forma didática e concisa.
- Mostra exemplos que podem ser diretamente executados em computador.
- Traz uma extensa lista de exercícios resolvidos que podem ser executados com o compilador *Pelles C*, no Windows, ou com o compilador *GCC*, no Unix e no Linux.
- Usa um compilador C bastante atual - *Pelles C for Windows* - padrão ISO, que pode ser obtido gratuitamente em www.smorgasbordet.com/pellesc.

Embora seja especialmente direcionado a alunos iniciantes de cursos tecnológicos da área de informática, este livro é útil para qualquer programador iniciante, de qualquer outro curso.

O autor



```
int main(void) {  
    char m[10][3];  
    ler(m,10);  
    ordenar(m,10);  
}
```

Introdução

1.1 Algoritmo

Vamos iniciar considerando um problema conhecido como *Torres de Hanói*. Trata-se de um quebra-cabeça composto por uma base contendo três torres (A, B e C) e três discos de diâmetros distintos (1, 2 e 3), como ilustrado na Figura 1.1. Neste quebra-cabeça, o objetivo é encontrar uma forma de mover todos os discos da torre A para a torre C, usando a torre B como espaço auxiliar, de modo que:

- apenas um disco seja movido de cada vez;
- nenhum disco seja posicionado sobre outro disco de diâmetro menor;
- os discos sejam imediatamente transferidos de uma torre para outra.

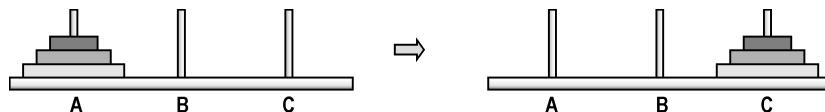


Figura 1.1 - O problema das torres de Hanói.

Há várias soluções possíveis para este problema. Uma delas é a seguinte:

- **1º passo:** mova o disco do topo da torre A para o topo da torre C.
- **2º passo:** mova o disco do topo da torre A para o topo da torre B.
- **3º passo:** mova o disco do topo da torre C para o topo da torre B.
- **4º passo:** mova o disco do topo da torre A para o topo da torre C.
- **5º passo:** mova o disco do topo da torre B para o topo da torre A.
- **6º passo:** mova o disco do topo da torre B para o topo da torre C.
- **7º passo:** mova o disco do topo da torre A para o topo da torre C.

Uma vez definida esta sequência de passos, qualquer pessoa capaz de executá-la é também capaz de solucionar automaticamente o problema. Nenhum raciocínio adicional é necessário, pois a sequência já descreve um procedimento correto para resolver o problema. Basta que a pessoa execute estes passos na ordem indicada.

Um procedimento para resolver um problema, definido por uma sequência finita e ordenada de passos executáveis, é denominado **algoritmo**.



De fato, algoritmos são muito comuns no nosso dia a dia. Alguns exemplos são:

- O manual de instalação de um aparelho de DVD, que descreve passo a passo como devemos proceder para conectar esse aparelho a um televisor.
- Uma receita culinária, que descreve os passos para preparação de um prato.
- Um mapa, que descreve como proceder para chegar a uma localização.

Porém, no contexto da ciência da computação, estamos interessados em definir algoritmos que serão executados por computadores e não por pessoas. Assim, ao definir um algoritmo computacional, precisamos nos restringir a um conjunto bastante limitado de passos (ou operações) que um computador é capaz de executar. Além disso, precisamos de uma notação que permita descrever precisamente estes passos, sem nenhuma ambiguidade.

1.2 Fluxograma

Um **fluxograma** é uma descrição precisa e detalhada de um algoritmo, feita numa notação que combina elementos gráficos e textuais. Na Tabela 1.1, temos a descrição dos principais símbolos usados na composição de fluxogramas.

Tabela 1.1 - Principais símbolos usados em fluxogramas, conforme o padrão ISO 5807.

Símbolo	Descrição
	Terminal (início e final do algoritmo).
	Entrada de dados (via teclado).
	Processamento de dados (cálculos).
	Saída de dados (via vídeo).
	Tomada de decisão (condição).
	Execução de processo predefinido.
	Ponto de conexão.

Num fluxograma, os passos de um algoritmo são representados por símbolos e a ordem de execução desses passos é representada por setas conectando os símbolos. Cada símbolo pode conter uma anotação indicando sua finalidade específica. Por exemplo, quando anotado com a palavra *início*, o símbolo terminal indica o primeiro passo do algoritmo; por outro lado, quando anotado



com a palavra fim, este mesmo símbolo indica que não há mais passos a serem executados.

Antes de projetar um fluxograma que descreve um algoritmo para resolver um problema, é recomendável definir como um computador seguindo esse algoritmo deve interagir com o usuário. Isso é feito por meio da especificação de **telas de execução**, que mostram que dados devem ser fornecidos pelo usuário e que resultados devem ser exibidos pelo computador. A especificação de telas de execução ajuda a ter uma noção mais precisa dos passos necessários para resolver o problema e da ordem em que estes devem ser executados pelo computador.

Como um primeiro exemplo do uso de fluxograma para descrição de algoritmos computacionais considere o Problema 1.1.

Problema 1.1

Dadas as duas notas de um aluno, calcule sua média aritmética e informe a sua situação (aprovado ou reprovado), levando em conta que a média para aprovação deve ser pelo menos 6,0.

Na Figura 1.2, temos dois exemplos de tela de execução para um algoritmo que resolve o Problema 1.1. Neles, os dados digitados pelo usuário são apresentados em negrito e seguidos pelo símbolo '*↵*', que representa a tecla *enter*.



Figura 1.2 - Telas de execução para um algoritmo que resolve o Problema 1.1.

Analisando as telas de execução, podemos perceber que os seguintes passos são necessários para resolver o Problema 1.1:

- **1º passo:** exiba uma mensagem ao usuário, solicitando a digitação das notas.
- **2º passo:** leia os dados digitados pelo usuário e os armazene nas variáveis a e b.
- **3º passo:** calcule a média dos valores em a e b, armazenando o resultado em c.
- **4º passo:** exiba uma mensagem indicando que a média é o valor guardado em c.
- **5º passo:** se o valor da variável c for maior ou igual a 6,0, exiba a mensagem "Aprovado"; senão, exiba a mensagem "Reprovado".

Note que todo dado manipulado pelo computador deve estar armazenado em alguma posição de sua memória. Uma posição de memória contendo um

dado que pode ser alterado pelo algoritmo, denominada **variável**, deve ser representada por um nome iniciando com letra e composto exclusivamente por letras, dígitos e sublinha. Nomes de variáveis devem ser escritos sem aspas.

Na Figura 1.3, temos o fluxograma de um algoritmo que resolve o Problema 1.1. Observe que as operações representadas neste fluxograma correspondem exatamente aos passos descritos anteriormente. Além disso, a execução destes passos deve produzir telas similares àquelas apresentadas na Figura 1.2.

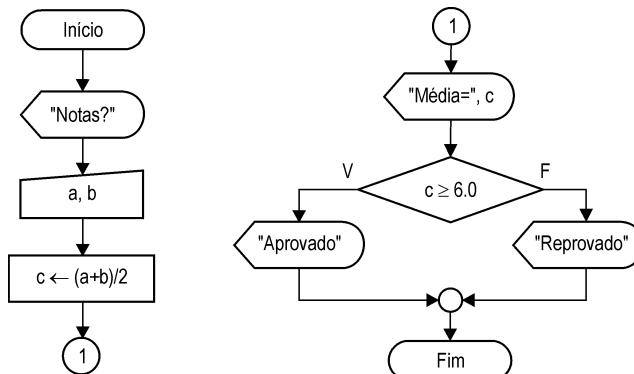


Figura 1.3 - Fluxograma que descreve um algoritmo para resolver o Problema 1.1.

1.3 Programa

Embora um fluxograma seja uma descrição precisa e detalhada de um algoritmo, ele não pode ser diretamente executado pelo computador. Assim, para vermos seu funcionamento num computador, precisamos antes transformá-lo num programa.

Um **programa** é um algoritmo descrito em uma **linguagem de programação**, ou seja, em uma linguagem que o computador seja capaz de interpretar e executar.

Há diversas linguagens de programação que podem ser usadas para a codificação de um algoritmo; por exemplo, podemos usar Pascal, C e Python, entre outras. Neste livro, optamos por usar C devido ao fato de ser uma linguagem cujo conhecimento facilita o aprendizado de diversas outras linguagens derivadas dela e que são bastante requisitadas no mercado, como, por exemplo, C++, Java e PHP.

Na Listagem 1.1, temos o programa em linguagem C correspondente ao fluxograma da Figura 1.3. Note que cada operação no fluxograma é representada por um comando neste programa. Alguns detalhes do programa são explicados a seguir.



```

/* media.c - calcula a media de um aluno */

#include <stdio.h>

int main(void) {
    float a, b, c;

    printf("Notas? ");
    scanf("%f %f", &a, &b);

    c = (a+b)/2;

    printf("Media = %.1f\n", c);

    if( c >= 6.0 ) printf("Aprovado\n");
    else printf("Reprovado\n");

    return 0;
}

```

Listagem 1.1 - Programa C correspondente ao fluxograma da Figura 1.3.

Em linguagem C:

- **Comentários** são ignorados pelo computador e servem apenas para introduzir lembretes para o próprio programador. Em C, há dois tipos de comentário: aqueles delimitados por `/*` e `*/`, que podem se estender por várias linhas, e aqueles iniciados por `//`, que terminam no final da linha. Comentários são opcionais em um programa, porém seu uso é fortemente recomendado.
- **Diretivas** do tipo `#include` servem para incluir bibliotecas de comandos pre-definidos da linguagem. Particularmente, a inclusão da biblioteca de entrada e saída padrão `stdio.h` (*standard input/output header*) é necessária sempre que precisamos usar o comando de entrada padrão `scanf()`, que lê dados digitados no teclado e os armazena em variáveis, e o comando de saída padrão `printf()`, que exibe dados em vídeo. Há diversas outras bibliotecas em C que serão apresentadas nos próximos capítulos.
- A lógica do programa é implementada por uma **função principal**, denominada `main()`. Todo programa em C deve ter uma função `main()`, por onde se inicia a sua execução. Os comandos da função são delimitados por um par de chaves.
- Variáveis devem ser declaradas antes de serem usadas. A **declaração** de uma variável consiste na indicação do **tipo de dados** que será armazenado por ela, bem como do nome usado para identificá-la. Particularmente, a declaração `float a, b, c;` indica que o programa usará três variáveis para armazenamento de números reais (note que variáveis são declaradas apenas no programa; no fluxograma isso não é feito).



- A função `printf()` é usada para **exibição de dados formatados** no vídeo. Por exemplo, a execução de `printf("Média = %.1f\n", c)` faz com que o computador exiba a mensagem "Média = ", seguida do valor da variável `c` (`%.1f` indica um valor do tipo `float` com apenas uma casa decimal) e, no final, mude o cursor para uma nova linha do vídeo (`\n` indica *new line*).
- A função `scanf()` é usada para **leitura de dados formatados** do teclado. Por exemplo, a execução de `scanf("%f %f", &a, &b)` faz com que o computador leia dois valores do tipo `float` e os armazene, respectivamente, nos endereços de memória onde foram criadas as variáveis `a` e `b` (`&a` indica o endereço de `a`).
- Expressões aritméticas são compostas por **constantes, variáveis e operadores** aritméticos (+, -, * e /); o operador de **atribuição** (=) avalia a expressão à sua direita e armazena o resultado desta na variável indicada à sua esquerda.
- A execução da função `main()` termina com a execução do comando `return`.
- Todas as **palavras reservadas**, com significado predefinido na linguagem, devem ser escritas em letras minúsculas (por exemplo, `main`, `float`, `if`, `return`).

1.4 Criação de programas e compiladores

As etapas básicas para a criação de um programa são as seguintes:

- **Análise:** nessa etapa, precisamos compreender o problema em questão, definindo que dados são fornecidos como *entrada*, que *processamento* deve ser efetuado e que informações devem ser apresentadas como *saída* (esta é a etapa em que especificamos as *telas de execução*).
- **Projeto:** nessa etapa, precisamos elaborar um *algoritmo* que descreva, passo a passo, como o computador deve proceder para obter os dados de entrada, processá-los e exibir as informações de saída, de acordo com o que foi definido na etapa de análise (esta é a etapa em que construímos o *fluxograma*).
- **Implementação:** nessa etapa, precisamos codificar um *programa* correspondente ao algoritmo elaborado na etapa de projeto (esta é a etapa em que usamos uma *linguagem de programação*).
- **Teste:** nessa etapa, precisamos *executar* o programa num computador e verificar como ele se comporta para diversos dados de entrada (esta é a etapa em que usamos um *compilador*); as telas de execução especificadas na etapa de análise do problema podem ser usadas para testar o programa e verificar se ele, de fato, apresenta as respostas esperadas.



Um **compilador** é um programa que interpreta os comandos escritos em uma linguagem de programação e os converte em uma forma que o computador seja capaz de executar. Em geral, há vários compiladores diferentes para uma mesma linguagem de programação. Neste livro, usamos o compilador gratuito *Pelles C for Windows*, cuja tela principal é apresentada na Figura 1.4. Entretanto, todos os programas desenvolvidos também podem ser compilados com os compiladores *CC* e *GCC* nos sistemas operacionais Unix e Linux.

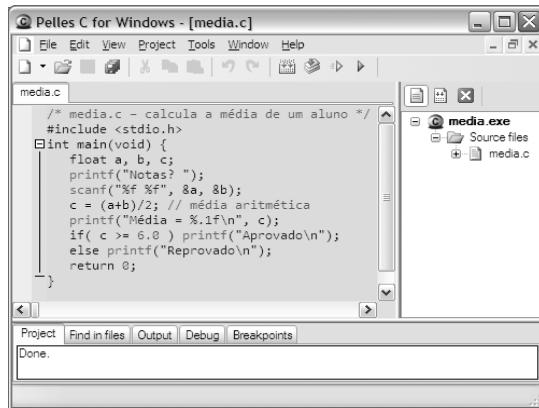


Figura 1.4 - Tela principal do compilador Pelles C for Windows.

Para testar o programa da Listagem 1.1 usando o compilador *Pelles C for Windows*, proceda da seguinte forma:

- **1º passo:** inicie a execução do programa *Pelles C IDE*.
- **2º passo:** no menu deste programa, selecione a opção *File* → *New* → *Project*.
- **3º passo:** na janela *New Projects*, selecione *Win32 Console program (EXE)*; digite o nome de projeto - *media* - e clique no botão *OK*.
- **4º passo:** no menu, selecione a opção *File* → *New* → *Source code*.
- **5º passo:** digite o programa em linguagem C.
- **6º passo:** no menu, selecione a opção *File* → *Save as*; digite o nome do programa - *media.c* - e clique no botão *Salvar*.
- **7º passo:** clique no botão *Sim*, da caixa que apresenta a pergunta "*Do you want to add the file 'media.c' to the current project?*".
- **8º passo:** no menu, clique no botão *Execute* (⇒).
- **9º passo:** forneça os dados de entrada e verifique se as respostas exibidas pelo computador estão de acordo com o esperado.

Para testar o programa da Listagem 1.1 usando o compilador *GCC* em sistema Unix ou Linux, proceda da seguinte forma:

- **1º passo:** use seu editor de textos preferido para digitar e salvar o programa em um arquivo chamado media.c.
- **2º passo:** numa janela de terminal, digite:
% gcc media.c ↴
- **3º passo:** a compilação do arquivo media.c produz um novo arquivo executável chamado a.out. Para executá-lo, digite:
% a.out ↴
- **4º passo:** forneça os dados de entrada e verifique se as respostas exibidas pelo computador estão de acordo com o esperado.

Deste ponto em diante, vamos considerar que o *Pelles C for Windows* é compilador usado para executar os programas desenvolvidos neste livro.

1.5 Programação estruturada

Quando os primeiros fluxogramas começaram a ser desenvolvidos, na década de 1950, ainda não havia regras sobre como construí-los corretamente. Nesta época, praticamente qualquer agrupamento consistente de símbolos era considerado adequado. Porém, com o passar do tempo, ficou claro que um conjunto de regras para a construção de fluxogramas (algoritmos) era necessário. À medida que os programas foram se tornando mais complexos, surgiu a necessidade de *estruturar* a sua lógica, a fim de facilitar sua compreensão, seu desenvolvimento e sua manutenção.

De fato, em meados da década de 1960, Böhm e Jacopini provaram que todo algoritmo computacional pode ser descrito em termos de apenas três padrões de agrupamento de passos. Esses padrões, denominados **estruturas de controle**, são também os componentes básicos a partir dos quais os programas são construídos.

As três estruturas de controle básicas são:

- **Sequência:** essa estrutura permite indicar dois ou mais passos que devem ser executados sequencialmente, na ordem em que são especificados, como ilustrado na Figura 1.5-a.
- **Seleção:** permite indicar dois passos que devem ser executados de forma mutuamente exclusiva, dependendo de uma determinada condição, como ilustrado na Figura 1.5-b.
- **Repetição:** permite indicar um ou mais passos que devem ser executados repetidamente, dependendo de uma determinada condição, como ilustrado na Figura 1.5-c.



Os programas construídos a partir da combinação dessas estruturas de controle básicas são chamados **programas estruturados**. Essas estruturas são estudadas mais detalhadamente nos próximos capítulos. Por enquanto, é suficiente ter uma noção clara da diferença existente entre elas.

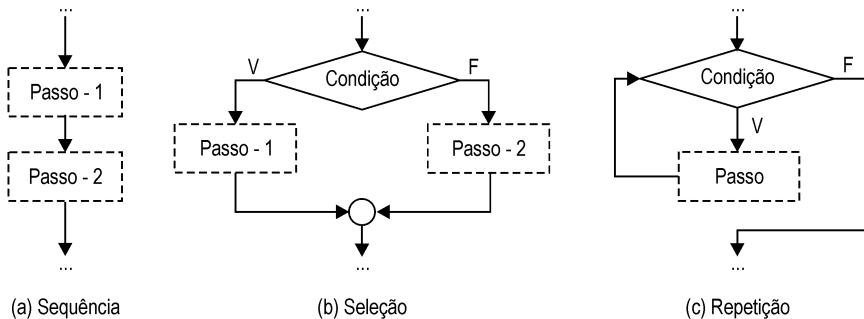


Figura 1.5 - As três estruturas de controle básicas.

1.6 Exercícios

Para cada problema a seguir, elabore um fluxograma, codifique um programa correspondente em linguagem C e teste o programa usando o compilador *Pelles C*.

- 1.1** Dado um número real qualquer, informe qual é o seu dobro.
- 1.2** Dado o total de vendas de um vendedor, calcule a sua comissão. Suponha que a comissão do vendedor seja de 10% do total de vendas.
- 1.3** Dadas as medidas de uma sala em metros (comprimento e largura), informe a sua área em metros quadrados.
- 1.4** Dados um salário e um percentual de reajuste, calcule o salário reajustado. Considere que o percentual de reajuste é dado por um número real entre 0 e 1. Por exemplo, se o reajuste for de 15%, o usuário deve digitar o número 0.15.
- 1.5** Dados o valor da compra e o percentual de desconto, calcule o valor a ser pago. Considere que o percentual de desconto é um número real entre 0 e 1.
- 1.6** Dados um valor em real e a cotação do dólar, converta esse valor em dólares.



Anotações



Sequência

2.1 Tipos de dados, constantes e variáveis

A principal finalidade dos computadores é o processamento de dados e, apesar de os computadores trabalharem internamente apenas com números binários, a maioria das linguagens de programação permite o uso de tipos de dados mais intuitivos. Os **tipos de dados** mais comuns podem ser classificados como na Figura 2.1.

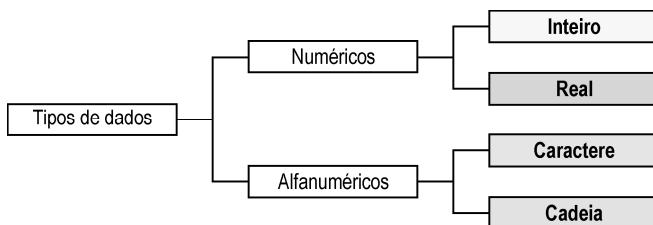


Figura 2.1 - Classificação dos tipos de dados mais comuns em programação.

Para cada tipo de dados há um conjunto específico de operações que podem ser efetuadas. Por exemplo, em C, podemos somar dados dos tipos inteiro ou real, mas não podemos somar dados do tipo cadeia.

Há duas formas distintas de representação de dados em algoritmos:

- **Constantes** representam dados cujos valores permanecem inalterados durante a execução do algoritmo. O tipo de uma constante é definido implicitamente, pela forma como ela é escrita. Em C, por exemplo, `3` é uma constante do tipo inteiro, `3.0` é uma constante do tipo real, `'a'` é uma constante do tipo caractere e `"abc"` é uma constante do tipo cadeia de caracteres (também denominada *string*). Uma constante do tipo caractere, denotada pelo uso de apóstrofos, representa um único símbolo (que pode ser uma letra, um dígito, um sinal de pontuação etc.). Por outro lado, uma constante do tipo cadeia pode representar uma série composta de zero ou mais símbolos. Particularmente, a cadeia vazia é denotada por `""`.



- **Variáveis** representam dados cujos valores podem ser alterados durante a execução do algoritmo. Mais precisamente, uma variável representa uma posição de memória que armazena um dado de um tipo específico. Variáveis são identificadas por nomes iniciando com letra e contendo apenas letras, dígitos e sublinhas (_). Assim, por exemplo, n1, M e salario_bruto são nomes válidos para variáveis; enquanto 1a.nota, média_bimestral e salario-bruto não o são. O tipo de uma variável (int, float, char, ...) deve ser explicitamente declarado quando esta é usada em um programa em C.

2.2 Operadores, expressões e atribuição

Um **operador** é um símbolo usado para criar **expressões**, a partir de constantes e variáveis. Por exemplo, usando o operador aritmético +, a variável x e a constante 2, podemos criar a expressão $x+2$. Na Tabela 2.1, temos os operadores aritméticos usados em fluxogramas e suas respectivas formas de representação em C.

Tabela 2.1 - Operadores aritméticos.

Operação	Fluxograma	Programa em C
Soma	+	+
subtração	-	-
multiplicação	*	*
divisão inteira	div	/
divisão real	/	/
resto da divisão inteira	mod	%

Em C, o tipo do resultado obtido com a avaliação de uma expressão depende do tipo de suas componentes. Se a expressão é composta apenas por valores inteiros, então o seu resultado também será do tipo inteiro; caso contrário, se pelo menos um dos valores envolvidos na expressão for do tipo real, então o resultado de sua avaliação será do tipo real. Assim, por exemplo, a avaliação de $1+2$ resulta no valor inteiro 3, enquanto a avaliação de $1+2.0$ resulta no valor real 3.0.

Entre todas as operações aritméticas, apenas a divisão requer uma explicação adicional. Há duas formas de divisão que o computador é capaz de efetuar: *divisão inteira*, que descarta a parte fracionária do quociente, e *divisão real*, que mantém a parte fracionária do quociente. Por exemplo, num fluxograma, a expressão $7 \text{ div } 2$ representa uma divisão inteira, cujo quociente é 3, enquanto a expressão $7/2$ representa uma divisão real, cujo quociente é 3.5. Em C, porém,



estas duas formas de divisão são indicadas pelo mesmo símbolo (/) e, neste caso, o tipo do resultado obtido depende do tipo dos dados envolvidos no cálculo. Se os dois valores forem inteiros, o resultado da divisão será inteiro; caso contrário, o resultado da divisão será real. Por exemplo, em C, a expressão `7/2` tem valor 3, enquanto as expressões `7.0/2`, `7/2.0` e `7.0/2.0` têm valor 3.5.

Além dos operadores aritméticos, outro operador bastante usado é o operador de **atribuição**, Tabela 2.2, que avalia a expressão existente do seu lado direito (que pode ser uma constante, uma variável ou uma expressão propriamente dita) e atribui o seu valor à variável que aparece do seu lado esquerdo.

Tabela 2.2 - Operador de atribuição.

Operação	Fluxograma	Programa em C
atribuição	←	=

Alguns exemplos de uso do operador de atribuição são:

- `a ← 1`
- `b ← a`
- `c ← 2*a + b/3`

2.3 Operações de entrada e saída de dados

A operação de **entrada de dados** lê valores digitados no teclado e os armazena em posições específicas da memória do computador. Num fluxograma, essa operação é representada como indicado na Figura 2.2. Repare que o símbolo de entrada de dados deve ser anotado com uma *lista-de-variáveis*, ou seja, uma série composta por um ou mais nomes de variáveis distintos e separados por vírgulas.

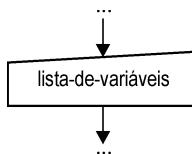


Figura 2.2 - Operação de entrada de dados.

A operação de **saída de dados** exibe valores no vídeo. Num fluxograma, essa operação é representada como indicado na Figura 2.3. Note que o símbolo de saída de dados deve ser anotado com uma *lista-de-valores*, ou seja, uma série de constantes, variáveis ou expressões separadas por vírgulas. Durante a execução dessa operação, as constantes são enviadas diretamente para o vídeo; as variáveis têm seus valores consultados na memória, antes de serem enviados para

o vídeo; e as expressões são avaliadas, antes de seus valores serem enviados para o vídeo.



Figura 2.3 - Operação primitiva de saída de dados.

Em C, o tipo de um dado determina que formato deve ser usado quando ele é lido pelo comando de entrada de dados - `scanf()` - ou exibido pelo comando de saída de dados - `printf()`. Conforme indicado na Tabela 2.3, o formato para o tipo inteiro é "%d"; para o tipo real é "%f"; para o tipo caractere é "%c"; e para o tipo cadeia é "%s".

Tabela 2.3 - Tipos de dados básicos e formatos de leitura e exibição em C.

Tipo de dados	Em C	Formato
caractere	char	%c
inteiro	int	%d
real	float	%f
cadeia	char []	%s

Por exemplo, as declarações em C a seguir criam três variáveis `x`, `y` e `z`, respectivamente, dos tipos `char`, `int` e `float`.

```
char x;
int y;
float z;
```

Uma vez que estas variáveis tenham sido criadas, podemos, por exemplo, executar o comando a seguir:

```
scanf("%c %d %f", &x, &y, &z);
```

Este comando lê três dados digitados pelo usuário (um caractere, um inteiro e um real) e os armazena, nesta ordem, nos endereços de memória em que foram criadas as variáveis `x`, `y` e `z`. Em C, o endereço de uma variável na memória é indicado pelo seu nome prefixado pelo símbolo `&` (chamado **operador de endereço**).

Para exibir os valores destas variáveis, podemos usar o comando:

```
printf("%c %d %f", x, y, z);
```



Note que não usamos o símbolo & prefixando os nomes de variáveis em `printf()`.

A função `printf()` permite também que os campos de exibição sejam formatados. As formatações mais usadas são o preenchimento com zeros à esquerda, para inteiros, e a especificação do número de casas decimais, para reais.

Por exemplo, considere as declarações a seguir, que criam uma variável inteira `a`, com valor inicial 678, e uma variável real `b`, com valor inicial 12.3456:

```
int a = 678;
float b = 12.3456;
```

Executando os comandos a seguir:

```
printf("|%5d|\n", a);      // exibe int em 5 colunas, preenchendo com espaços
printf("|%06d|\n", a);    // exibe int em 6 colunas, preenchendo com zeros
printf("|%7.3f|\n", b);   // exibe float em 7 colunas, com 3 casas decimais
printf("|%.1f|\n", b);    // exibe float com 1 casa decimal
```

Obtemos as seguintes saídas:

678
000678
12.346
12.3

Note que, ao serem exibidos, valores reais são arredondados quando sua primeira casa decimal desprezada é igual ou superior a 5. Por este motivo, quando usamos o formato "%.`3f`", o número 12.3456 é arredondado para 12.346. Por outro lado, com o formato "%.`1f`", o número 12.3456 é exibido como 12.3, sem arredondamento.

Além de formatos de exibição, a cadeia de formatação do `printf()` pode conter também **caracteres de controle**. A saída de um caractere de controle nem sempre causa a exibição de um símbolo no vídeo. De fato, como indicado na Tabela 2.4, a saída de alguns caracteres de controle produz apenas o efeito de som ou de movimentação do cursor.

Por exemplo, quando o comando:

```
printf("\nTESTE\aN");
```

é executado, o cursor é movido para uma nova linha, a palavra TESTE é exibida e, em seguida, o alarme é soado (produzindo o som de um *beep*).



Tabela 2.4 - Caracteres de controle e seus efeitos.

Controle	Efeito
\a	(alarm) soa o alarme do terminal
\b	(back space) muda o cursor para coluna anterior
\n	(new line) muda o cursor para próxima linha
\r	(return) muda o cursor para o início da linha
\t	(tab) muda o cursor para a próxima marca de tabulação
\'	exibe um apóstrofo
\"	exibe uma aspa
\\\	exibe uma barra invertida

O tipo cadeia será visto com mais detalhes no Capítulo 11, porém para uma noção inicial sobre o uso desse tipo de dados, considere a declaração a seguir:

```
char s[31];
```

Esta declaração cria uma variável do tipo cadeia, com capacidade de armazenar até 30 caracteres (a necessidade da posição adicional será explicada em breve). Para ler uma palavra e armazená-la nessa variável, podemos usar o comando:

```
scanf("%s", s); // variável do tipo cadeia não requer o uso de &
```

Para exibir a cadeia armazenada em s, podemos usar o comando:

```
printf("%s", s);
```

Além destes tipos, C também oferece outros tipos de dados que não são essenciais num curso introdutório e que, portanto, não são abordados neste livro.

2.4 Estrutura sequencial

A estrutura mais simples que um algoritmo pode ter é conhecida como **sequencial**. Nessa estrutura os passos do algoritmo são executados, um a um, na ordem em que são especificados. Embora essa estrutura seja muito simples, ela é fundamental para a construção de todo e qualquer programa.

Normalmente, a estrutura sequencial mais básica é composta por:

- **Entrada de dados**, que lê dados digitados pelo usuário e os armazena em posições de memória representadas por variáveis.
- **Processamento**, que calcula expressões compostas por constantes e variáveis e armazena seus resultados em outras variáveis.
- **Saída de dados**, que exibe no vídeo os resultados dos cálculos efetuados.



O Problema 2.1 mostra um caso em que a estrutura sequencial é necessária.

Problema 2.1

Dadas as coordenadas de dois pontos no plano cartesiano, informe a distância entre eles.

Na Figura 2.4 temos os conceitos matemáticos necessários para resolver este problema. Observe que, a partir de dois pontos indicados no plano cartesiano (P_1 e P_2), podemos desenhar um triângulo retângulo cujas medidas dos catetos a e b são dadas pelas diferenças entre as ordenadas e abscissas desses pontos e cuja medida da hipotenusa c , dada pelo Teorema de Pitágoras, é justamente a distância entre os pontos considerados. Na Figura 2.5 temos uma tela de execução que ilustra o funcionamento pretendido para o algoritmo que será desenvolvido. O fluxograma que descreve esse algoritmo é apresentado na Figura 2.6 e o programa correspondente na Listagem 2.1.

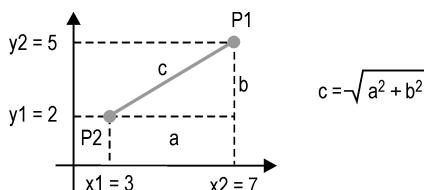


Figura 2.4 - Cálculo da distância entre dois pontos P_1 e P_2 .

P1? 3 2 ↴
P2? 7 5 ↴
Distância = 5.0

Figura 2.5 - Tela de execução para um algoritmo que resolve o Problema 2.1.

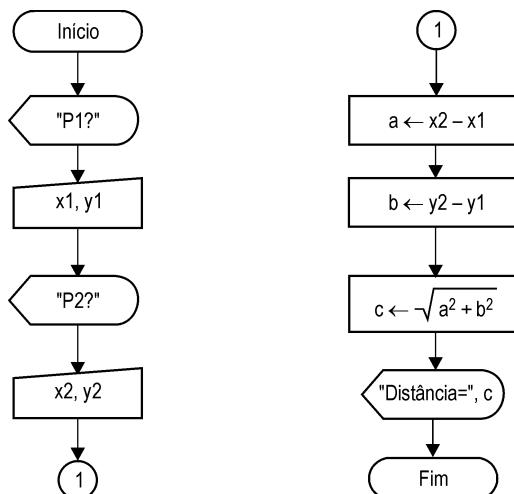


Figura 2.6 - Fluxograma que descreve um algoritmo para o Problema 2.1.



Como vemos na Figura 2.6, a solução para o Problema 2.1 consiste em uma sequência de passos que pode ser dividida em três seções distintas. Primeiramente os dados são solicitados ao usuário, lidos e armazenados em variáveis; em seguida são processados, de modo que a resposta do problema seja obtida; e finalmente a resposta obtida é exibida no vídeo.

2.5 A biblioteca de funções matemáticas

Como vemos na Figura 2.6, não há problema em empregar fórmulas compostas por funções matemáticas em um fluxograma (por exemplo, expoente sobreescrito e sinal de raiz), porém para escrever essas fórmulas num programa em linguagem C, precisamos lembrar que:

- A inclusão da **biblioteca matemática** `math.h` é necessária sempre que usamos funções matemáticas (note que o uso de operadores aritméticos não requer a inclusão dessa biblioteca).
- A função `sqrt()` é usada para calcular a **raiz quadrada** de um valor; assim, por exemplo, `sqrt(x)` representa a raiz quadrada de `x`.
- A função `pow()` é usada para calcular a **potência** de um valor elevado a outro; assim, por exemplo, `pow(x,n)` representa a potência de `x` elevado a `n`.

Além destas duas funções matemáticas - `sqrt()` e `pow()` - há diversas outras funções úteis que podem ser consultadas no sistema de ajuda do compilador Pelles C. Basta pressionar a combinação de teclas `CTRL+F1` e pesquisar a palavra "math".

Um programa que usa a biblioteca `math.h`, correspondente ao fluxograma da Figura 2.6, é apresentado na Listagem 2.1.

```
/* dist.c - calcula a distancia entre dois pontos */
#include <stdio.h>
#include <math.h>

int main(void) {
    float x1, y1, x2, y2, a, b, c;

    printf("P1? ");
    scanf("%f %f", &x1, &y1);
    printf("P2? ");
    scanf("%f %f", &x2, &y2);

    a = x2 - x1;
    b = y2 - y1;
    c = sqrt( pow(a,2) + pow(b,2) );

    printf("Distancia = %.1f\n", c);
    return 0;
}
```

Listagem 2.1 - Programa C correspondente ao fluxograma da Figura 2.6.



Uma observação importante sobre o compilador *GCC* dos sistemas Unix e Linux é que para usar a biblioteca matemática com ele, é preciso indicar a opção `-lm` (*library math*), conforme a seguir:

```
% gcc -lm dist.c ↴
```

2.6 Exercícios

Para cada problema a seguir, elabore um fluxograma, codifique um programa correspondente em linguagem C e teste esse programa usando o compilador *Pelles C*.

- 2.1** Dada uma distância percorrida (em quilômetros), bem como o total de combustível gasto (em litros), informe o consumo médio do veículo.
- 2.2** Dadas as medidas de uma sala em metros (comprimento e largura), bem como o preço do metro quadrado de carpete, informe o custo total para forrar o piso da sala.
- 2.3** O índice de massa corpórea (IMC) de uma pessoa é igual ao seu peso (em quilogramas) dividido pelo quadrado de sua altura (em metros). Dados o peso e a altura de uma pessoa, informe o valor de seu IMC.
- 2.4** Dado o tamanho de um arquivo (em bits), bem como a velocidade da conexão (em bits por segundo), informe o tempo necessário para *download* do arquivo.
- 2.5** Dados um capital C, uma taxa de juros mensal fixa J e um período de aplicação em meses M, informe o montante F no final do período ($F = C * (1 + J/100)^M$).



Anotações



```
int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
}
```

Seleção Simples

3.1 Expressões condicionais

Uma **expressão condicional** é uma expressão cujo valor pode ser *falso* ou *verdadeiro*. Em C, falso é representado por 0 e verdadeiro é representado por 1.

Uma **condição simples** é uma expressão condicional cujo valor é dado por um **operador relacional**, isto é, um operador que efetua uma comparação, Tabela 3.1.

Por exemplo, em C, as expressões $(a+1 > 2*b)$ e $(2*c == 6)$ são condições simples. Supondo $a=1$ e $b=2$, podemos concluir que o valor da expressão $(a+1 > 2*b)$ é 0, ou seja, falso, pois neste caso $a+1$ não é maior que $2*b$. Por outro lado, supondo $c=3$, podemos concluir que o valor da expressão $(2*c == 6)$ é 1, ou seja, verdadeiro, pois neste caso $2*c$ é igual a 6.

Tabela 3.1 - Operadores relacionais.

Operação	Fluxograma	Programa em C
igual	=	==
diferente	≠	!=
menor	<	<
menor ou igual	≤	<=
maior	>	>
maior ou igual	≥	>=

Em C, operadores relacionais não podem ser combinados numa mesma condição simples. Para entender o porquê, suponha que temos $x=5$, $y=5$ e $z=5$. Neste caso, deveríamos esperar que a expressão $x==y==z$ fosse verdadeira, porém devido à forma como C avalia os operadores relacionais, esta expressão é falsa. Isso acontece porque o valor da expressão $x==y==z$ é determinado em duas etapas. Primeiramente se avalia $x==y$, que resulta em verdadeiro (1), depois esse resultado é comparado com o valor de z , ou seja, $1==5$, que resulta em falso (0). Assim, para obtermos o resultado correto, precisamos usar uma condição composta.



Uma **condição composta** é uma expressão condicional formada por duas ou mais condições simples ligadas por **operadores lógicos**, Tabela 3.2.

Tabela 3.2 - Operadores lógicos.

Operação	Fluxograma	Programa em C
não	não	!
e	e	&&
ou	ou	

Quando avaliados, tanto os operadores relacionais quanto os operadores lógicos, resultam num valor falso (0) ou verdadeiro (1). Os resultados dos operadores lógicos são definidos conforme indicado na Tabela 3.3.

Tabela 3.3 - Resultados dos operadores lógicos em C.

Expressão	Valor	Expressão	Valor	Expressão	Valor
<code>!0</code>	1	<code>0 && 0</code>	0	<code>0 0</code>	0
<code>!1</code>	0	<code>0 && 1</code>	0	<code>0 1</code>	1
		<code>1 && 0</code>	0	<code>1 0</code>	1
		<code>1 && 1</code>	1	<code>1 1</code>	1

Por exemplo, em C, a expressão `(a+1>2*b || 2*c==6)` é uma condição composta. Supondo `a=1`, `b=2` e `c=3`, podemos concluir que o valor desta expressão é 1, ou seja, verdadeiro, pois neste caso `(a+1>2*b)` vale 0, `(2*c==6)` vale 1 e `(0 || 1)` dá 1.

O uso de operadores lógicos é necessário sempre que precisamos representar uma condição contendo mais que um operador relacional. Por exemplo, ainda que `x`, `y` e `z` sejam todas iguais a 5, a expressão `x==y==z` é avaliada como falsa. Para evitar este problema, em vez de `x==y==z`, devemos escrever `x==y && y==z`. Neste caso, como `x==y` e `y==z` valem 1, a expressão `x==y && y==z` equivale a `1 && 1`, que dá 1.

Numa expressão condicional, os operadores aritméticos são avaliados antes dos operadores relacionais que são avaliados antes dos lógicos.

3.2 Estrutura de seleção simples

A estrutura de **seleção simples**, apresentada na Figura 3.1, serve para selecionar e executar apenas um entre dois comandos alternativos. Para tanto, uma condição é avaliada e:

- se ela for verdadeira, então apenas o primeiro comando é executado;
- senão, apenas o segundo comando é executado.



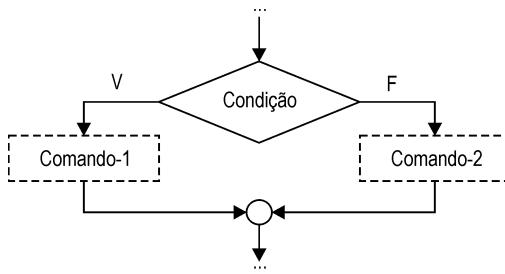


Figura 3.1 - Estrutura de seleção simples.

Como ilustrado na Figura 3.1, apenas um dos comandos indicados na estrutura de seleção simples pode ser executado, porque, quando seguimos para a esquerda, não temos mais a possibilidade de passar pelo comando da direita e vice-versa.

Em C, a estrutura de seleção simples é codificada como indicado na Figura 3.2.

```
if( condição ) comando-1;
else comando-2;
```

Figura 3.2 - Padrão de codificação da estrutura de seleção simples em C.

Vamos usar a estrutura de seleção simples para resolver o Problema 3.1.

Problema 3.1

O índice de massa corpórea (IMC) de uma pessoa é igual ao seu peso (em quilogramas) dividido pelo quadrado de sua altura (em metros). A pessoa é considerada obesa quando seu IMC é superior a 30. Dados o peso e a altura de uma pessoa, informe se ela está obesa.

Na Figura 3.3 temos dois exemplos de tela de execução do programa que será desenvolvido. O fluxograma para esse programa é apresentado na Figura 3.4 e o código correspondente em linguagem C é apresentado na Listagem 3.1.

Peso? 86.2 ↴ Altura? 1.65 ↴ IMC = 31.66 Obeso	Peso? 74.7 ↴ Altura? 1.73 ↴ IMC = 24.96 Normal
--	---

Figura 3.3 - Telas de execução para um programa que resolve o Problema 3.1.

É fácil perceber que esse programa precisa selecionar um entre os dois diagnósticos mutuamente exclusivos (normal ou obeso) para exibir ao usuário. Portanto, ao projetar o seu fluxograma, precisamos usar a estrutura de seleção simples.



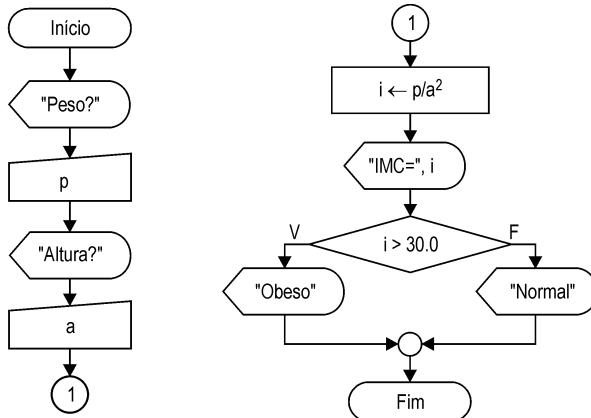


Figura 3.4 - Fluxograma que descreve um algoritmo para o Problema 3.1.

```
/* imc.c - calcula indice de massa corporea */

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

int main(void) {
    float p, a, i;

    printf("Peso? ");
    scanf("%f", &p);
    printf("Altura? ");
    scanf("%f", &a);

    i = p / pow(a, 2);

    printf("IMC = %.2f\n", i);

    if( i > 30.0 ) printf("Obeso\n");
    else printf("Normal\n");

    return 0;
}
```

Listagem 3.1 - Programa C correspondente ao fluxograma da Figura 3.4.

3.3 Uso de blocos e omissão de alternativa

Embora a estrutura de seleção simples na Figura 3.1 apresente exatamente um comando a ser executado em cada caso (V ou F), há situações em que:

- *Temos de executar mais de um comando no caso de a condição ser verdadeira e/ou no caso de a condição ser falsa; ou*
- *Não temos nenhum comando para executar quando a condição for falsa.*



Nestas situações, não há nada de especial com relação à forma do fluxograma, mas há alguns detalhes que devemos observar ao codificá-lo em C.

Em C, quando temos mais de um comando a ser executado para um determinado valor de condição (v ou f), devemos agrupar esses comandos em um **bloco** delimitado por chaves, conforme indicado na Figura 3.5.

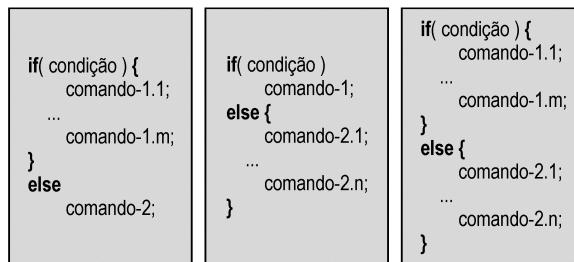


Figura 3.5 - Padrões de codificação da estrutura de seleção simples com blocos em C.

Quando não há nenhum comando a ser executado no caso de a condição ser falsa, devemos omitir a parte `else` do comando `if`, como indicado na Figura 3.6.



Figura 3.6 - Padrões de codificação da estrutura de seleção simples com omissão de else.

O Problema 3.2 ilustra um caso em que precisamos usar estas duas características da linguagem C, ou seja, *uso de bloco* e *omissão de alternativa*.

Problema 3.2

Numa empresa, paga-se R\$19,50 por hora trabalhada e desconta-se 25% de imposto de renda dos salários superiores a R\$ 2.500,00. Dado o número de horas trabalhadas por um funcionário dessa empresa, informe o valor do desconto de IR (se houver) e o valor do salário a ser pago ao funcionário.

Na Figura 3.7 temos dois exemplos de tela de execução do programa que será desenvolvido. O fluxograma para esse programa é apresentado na Figura 3.8 e o código correspondente em linguagem C na Listagem 3.2.



Figura 3.7 - Telas de execução para um programa que resolve o Problema 3.2.



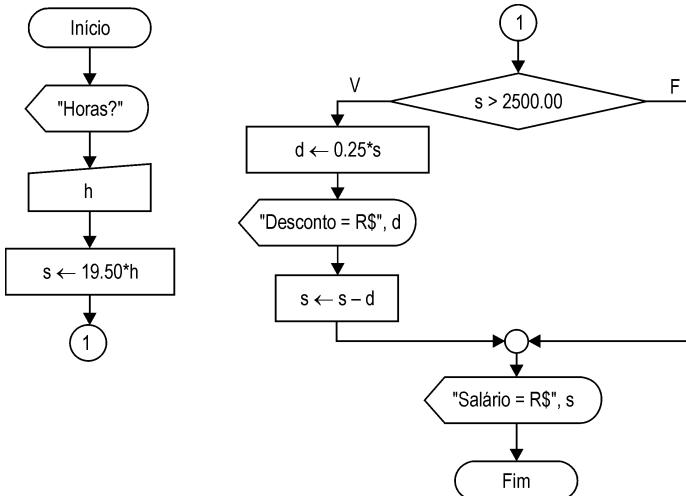


Figura 3.8 - Fluxograma que descreve um algoritmo para o Problema 3.2.

```

/* salario.c - calcula o salario pago a um funcionario */
#include <stdio.h>

int main(void) {
    float h, s, d;

    printf("Horas? ");
    scanf("%f", &h);

    s = 19.50*h;

    if( s > 2500.00 ) {
        d = 0.25*s;
        printf("Desconto = R$ %.2f\n", d);
        s = s - d;
    }

    printf("Salario = R$ %.2f\n", s);
    return 0;
}

```

Listagem 3.2 - Programa C correspondente ao fluxograma da Figura 3.8.

3.4 Estruturas de seleção simples encaixadas

Uma estrutura de seleção simples pode ser encaixada em outra, tanto no lado verdadeiro quanto no lado falso, para formar padrões mais complexos. O Problema 3.3 mostra um caso em que isso é necessário.



Problema 3.3

Numa universidade, a aprovação requer no máximo 40 faltas e no mínimo média 6.0. Se o aluno estourar em faltas, ele é reprovado diretamente; caso contrário, se ele não alcançar a média mínima, ele fica para recuperação. Dada a média de um aluno, bem como seu número de faltas, informe a sua situação.

Na Figura 3.9 e na Listagem 3.3 temos, respectivamente, um fluxograma e o programa correspondente que resolve o Problema 3.3. Note que, na lógica descrita, apesar de sabermos que o aluno não está reprovado por falta, precisamos de mais uma decisão para determinar se ele está aprovado ou para recuperação.

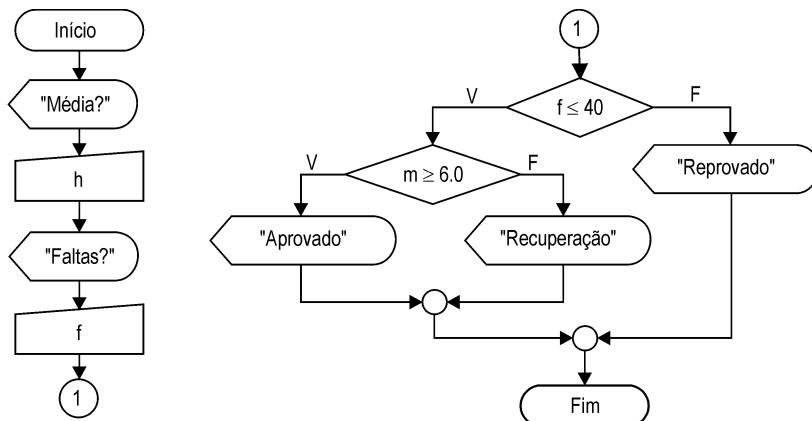


Figura 3.9 - Fluxograma que descreve um algoritmo para o Problema 3.3.

```
/* sit1.c - determina a situacao de um aluno */
#include <stdio.h>

int main(void) {
    float m;
    int f;

    printf("Media? ");
    scanf("%f", &m);
    printf("Faltas? ");
    scanf("%d", &f);

    if( f <= 40 )
        if( m >= 6.0 ) printf("Aprovado\n");
        else printf("Recuperacao\n");
    else printf("Reprovado\n");

    return 0;
}
```

Listagem 3.3 - Programa C correspondente ao fluxograma da Figura 3.9.



A forma como as estruturas de seleção são encaixadas depende das condições e da ordem em que elas são verificadas. Alterando as condições, ou a ordem em que elas são avaliadas, altera-se também a estrutura do fluxograma.

Por exemplo, na Figura 3.10 temos outro fluxograma que também resolve corretamente o Problema 3.3. Neste caso, porém, como usamos uma condição diferente na estrutura de seleção principal, a segunda estrutura de seleção teve de ser encaixada na parte falsa da primeira. Compare os fluxogramas das Figuras 3.9 e 3.10 e observe como essa pequena mudança tornou a lógica mais simples e mais fácil de ser entendida.

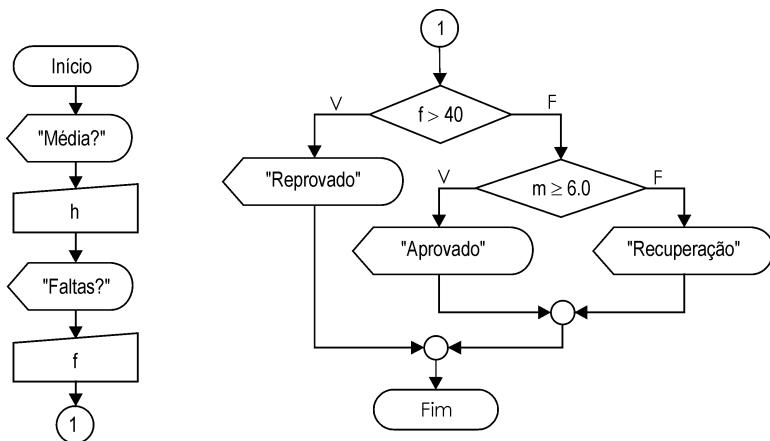


Figura 3.10 - Fluxograma que descreve um algoritmo para o Problema 3.3.

O programa correspondente ao fluxograma da Figura 3.10 é apresentado na Listagem 3.4.

```

/* sit2.c - determina a situacao de um aluno */
#include <stdio.h>

int main(void) {
    float m;
    int f;

    printf("Media? ");
    scanf("%f", &m);
    printf("Faltas? ");
    scanf("%d", &f);
    if( f > 40 ) printf("Reprovado\n");
    else if( m >= 6.0 ) printf("Aprovado\n");
    else printf("Recuperacao\n");
    return 0;
}

```

Listagem 3.4 - Programa C correspondente ao fluxograma da Figura 3.10.



3.5 Exercícios

Para cada problema a seguir, elabore um fluxograma, codifique um programa correspondente em linguagem C e teste o programa usando o compilador *Pelles C*.

- 3.1** Numa papelaria, até 100 folhas, a cópia custa R\$ 0,25, e acima de 100 folhas custa R\$ 0,20. Dado o total de cópias, informe o total a ser pago.
- 3.2** Numa fábrica, uma máquina precisa de manutenção sempre que o número de peças defeituosas supera 10% da produção. Dados o total de peças produzidas e o total de peças defeituosas, informe se a máquina precisa de manutenção.
- 3.3** Dada a idade de um nadador, informe a sua categoria: *infantil* (até 10 anos), *juvenil* (até 17 anos) ou *sênior* (acima de 17 anos).
- 3.4** Dados três números, verifique se eles podem ser medidas de um triângulo e, se puderem, classifique o triângulo como *equilátero*, *isóscele* ou *escaleno*.
- 3.5** Dados três números distintos, exiba-os em ordem crescente.



Anotações



```
int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
}
```

Seleção Múltipla

4.1 Estruturas de seleção encadeadas

Dizemos que duas estruturas de seleção simples estão **encadeadas** se uma delas está *encaixada* no lado falso da outra, conforme indicado na Figura 4.1.

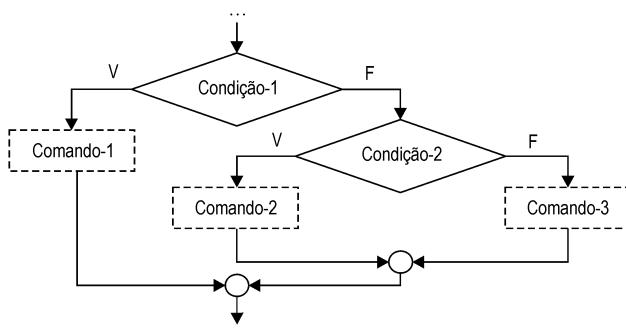


Figura 4.1 - Estruturas de seleção encadeadas.

Estruturas de seleção encadeadas são úteis quando precisamos escolher *apenas um* entre vários comandos alternativos. Como vemos na Figura 4.1, numa estrutura de seleção encadeada, cada *comando* é associado a uma *condição*, que deve ser verdadeira para que ele seja executado; exceto o último comando à direita, que é executado sempre que nenhuma das condições anteriores é verdadeira.

O Problema 4.1 mostra um caso em que estruturas encadeadas são úteis.

Problema 4.1

Dado um número, informe se ele é positivo, negativo ou nulo.

Na Figura 4.2, temos alguns exemplos de tela de execução para este problema.

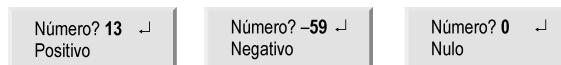


Figura 4.2 - Telas de execução para um programa que resolve o Problema 4.1.

O fluxograma para resolver o Problema 4.1 é apresentado na Figura 4.3 e o código correspondente em linguagem C é apresentado na Listagem 4.1. Observe que se um número não é maior que 0, nem menor que 0, ele só pode ser igual a 0.

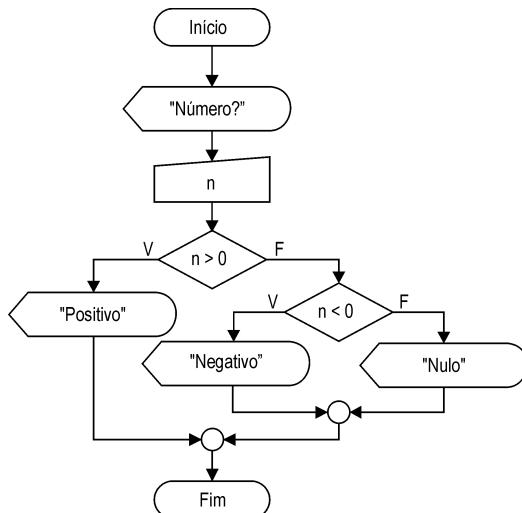


Figura 4.3 - Fluxograma que descreve um algoritmo para o Problema 4.1.

```
/* sinal.c - informa o sinal de um numero inteiro */

#include <stdio.h>

int main(void) {
    int n;

    printf("Numero? ");
    scanf("%d", &n);

    if( n > 0 ) printf("Positivo\n");
    else if( n < 0 ) printf("Negativo\n");
    else printf("Nulo\n");

    return 0;
}
```

Listagem 4.1 - Programa C correspondente ao fluxograma da Figura 4.3.

O Problema 4.2 mostra mais um caso em que estruturas encadeadas são úteis. Na Figura 4.4 temos alguns exemplos de tela de execução para este problema.



Problema 4.2

Dada uma letra maiúscula, representando o sexo de uma pessoa, informe o sexo correspondente por extenso.



Figura 4.4 - Telas de execução para um programa que resolve o Problema 4.2.

O fluxograma para resolver o Problema 4.2 é apresentado na Figura 4.5 e o código correspondente em linguagem C encontra-se na Listagem 4.2.

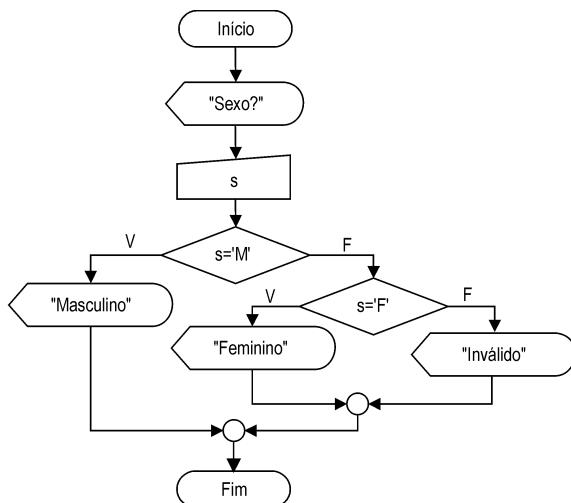


Figura 4.5 - Fluxograma que descreve um algoritmo para o Problema 4.2.

```
/* sexol.c - informa o sexo por extenso */
#include <stdio.h>

int main(void) {
    char s;

    printf("Sexo? ");
    scanf("%c", &s);

    if( s=='M' ) printf("Masculino\n");
    else if( s=='F' ) printf("Feminino\n");
    else printf("Invalido\n");

    return 0;
}
```

Listagem 4.2 - Programa C correspondente ao fluxograma da Figura 4.5, usando if-else.



Note que o programa da Listagem 4.2 só apresenta o sexo por extenso quando o usuário digita o caractere 'M' ou 'F'. Quando o caractere digitado é 'm', 'f', ou outro qualquer, o programa exibe a palavra `Inválido`. Isso acontece porque o computador representa caracteres usando códigos ASCII (Apêndice A). Por exemplo, o código ASCII do caractere 'M' é 77 e o código ASCII do caractere 'm' é 109. Portanto, quando usamos a expressão `s=='M'`, estamos, na verdade, comparando o valor `s` com 77. Por outro lado, quando o comando `scanf("%c", &s)` é executado, se o usuário digita o caractere 'm', a variável `s` fica valendo 109, mas como 109 é diferente de 77, a expressão `s=='M'` é avaliada como falsa. Ou seja, o computador trata 'm' e 'M' como letras distintas, já que seus códigos são distintos. Uma solução simples para este problema é usar a condição `s=='m' || s=='M'`. Uma versão melhorada do programa da Listagem 4.2, baseada nesta ideia, é apresentada na Listagem 4.3.

```
/* sexo2.c - informa o sexo por extenso */
#include <stdio.h>
int main(void) {
    char s;

    printf("Sexo? ");
    scanf("%c", &s);

    if( s=='m' || s=='M' ) printf("Masculino\n");
    else if( s=='f' || s=='F' ) printf("Feminino\n");
    else printf("Invalido\n");

    return 0;
}
```

Listagem 4.3 - Programa da Listagem 4.2 melhorado.

4.2 Estrutura de seleção múltipla

Uma **estrutura de seleção múltipla** é composta de uma série de estruturas de seleção simples encadeadas, em que observamos as seguintes propriedades:

- Todas as condições nas decisões são de *igualdade*.
- Todas as condições comparam uma *mesma expressão* a uma *constante*.
- Todas as constantes consideradas são de tipo *inteiro* ou *caractere*.

Em C, podemos codificar uma estrutura de seleção múltipla usando o comando `if-else`, porém, usando o comando `switch-case`, podemos obter um código mais claro e conciso. A estrutura básica do comando `switch-case` é exibida na Figura 4.6.



```

switch( expressão ) {
    case constante-1: comando-1; break;
    case constante-2: comando-2; break;
    ...
    default: comando-n; break;
}

```

Figura 4.6 - Padrão de codificação da estrutura de seleção múltipla em C.

Ao encontrar o comando `switch-case`, o computador avalia a *expressão* indicada dentro do par de parênteses que segue a palavra `switch` e executa apenas o *comando* associado ao caso cuja *constante* tem valor igual ao desta expressão. Se nenhum dos casos tem uma constante com o mesmo valor da expressão, então o comando indicado no caso `default` é executado.

Embora o padrão apresentado na Figura 4.6 indique apenas um *comando* em cada caso, na verdade podemos ter vários comandos num mesmo caso (sem a necessidade de agrupá-los entre chaves). Além disso, quando um caso é selecionado para execução, a execução do `switch-case` só é interrompida quando o comando `break` é encontrado. Nesse momento, a execução do programa prossegue no primeiro comando existente após o bloco `switch-case`.

Vale ressaltar que toda estrutura de seleção encadeada pode ser codificada com `if-else`, todavia apenas aquelas que apresentam as propriedades indicadas no início desta seção podem ser codificadas com o uso de `switch-case`.

Por exemplo, usando `switch-case` para codificar o algoritmo da Figura 4.5, obtemos o programa da Listagem 4.4. Compare esse programa com aquele da Listagem 4.2 e veja como a lógica ficou mais clara e simples.

```

/* sexo3.c - informa o sexo por extenso */
#include <stdio.h>

int main(void) {
    char s;

    printf("Sexo? ");
    scanf("%c", &s);

    switch( s ) {
        case 'M': printf("Masculino\n"); break;
        case 'F': printf("Feminino\n"); break;
        default : printf("Invalido\n"); break;
    }

    return 0;
}

```

Listagem 4.4 - Programa C correspondente ao fluxograma da Figura 4.5, usando `switch-case`.



Podemos também melhorar o programa da Listagem 4.4 para que ele funcione corretamente mesmo quando o usuário digitar caracteres minúsculos. Isso pode ser feito com a ajuda da função `toupper()`, da biblioteca `ctype.h`, que converte minúscula em maiúscula. Quando o computador encontra a expressão `toupper(s)`, caso `s` seja uma letra minúscula, em vez de usar esta letra, ele usa a letra maiúscula correspondente; caso contrário, ele usa o próprio valor de `s`. A Listagem 4.5 mostra como melhorar o programa da Listagem 4.4 usando essa função.

```
/* sexo4.c - informa o sexo por extenso */

#include <stdio.h>
#include <ctype.h>

int main(void) {
    char s;

    printf("Sexo? ");
    scanf("%c", &s);

    switch( toupper(s) ) {
        case 'M': printf("Masculino\n"); break;
        case 'F': printf("Feminino\n"); break;
        default : printf("Invalido\n"); break;
    }

    return 0;
}
```

Listagem 4.5 - Programa da Listagem 4.4 melhorado com o uso da função `toupper()`.

Note que em `ctype.h` existe também uma função, chamada `tolower()`, que converte maiúscula em minúscula.

4.3 Variações do comando switch-case

No comando `switch-case`, tanto o `break` quanto o caso `default` são opcionais:

- Se um caso é selecionado para execução e não existe um `break` no seu final, então, após a sua execução, a execução continua no caso seguinte.
- Se o caso `default` é omitido, e nenhum dos demais casos tem constante igual ao valor da expressão, então nada é feito e a execução do programa prossegue no primeiro comando após o bloco `switch-case`.

O Problema 4.3 mostra um caso em que podemos usar estas características do comando de seleção múltipla. O fluxograma para ele é apresentado na Figura 4.8.



Problema 4.3

Dado um número inteiro representando a placa de um veículo, informe em que dia da semana esse veículo está no rodízio.



Figura 4.7 - Telas de execução para um programa que resolve o Problema 4.3.

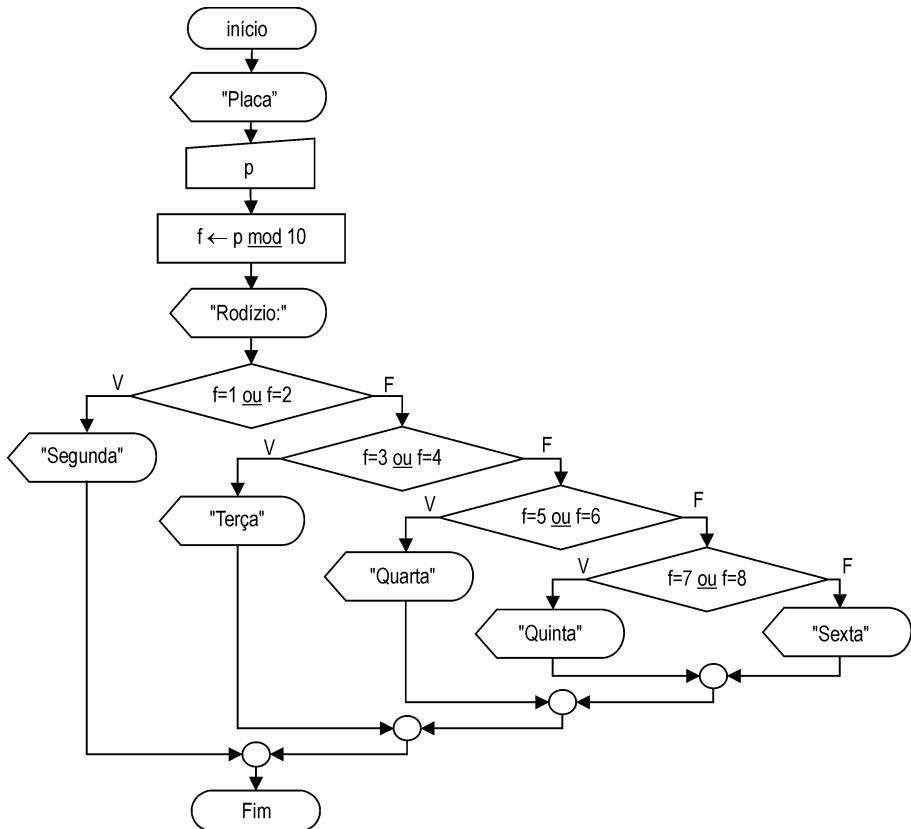


Figura 4.8 - Fluxograma que descreve um algoritmo para o Problema 4.3.

Embora o usuário digite um número de placa composto de quatro dígitos, para determinar o dia do rodízio correspondente, o algoritmo precisa verificar apenas o dígito final desse número. Para isso, basta dividir o número da placa por 10 e tomar o resto dessa divisão, pois sempre que dividimos um número por 10, o resto da divisão é igual ao último dígito do número. No fluxograma, a operação que dá o resto de uma divisão inteira é indicada pela palavra `mod` e, em C, é indicada pelo símbolo `%`.



Em C, a codificação do algoritmo descrito pelo fluxograma da Figura 4.8 pode ser feita de duas formas distintas. Na Listagem 4.6 temos uma versão em que usamos o comando `if-else`, e na Listagem 4.7 temos uma versão em que usamos o comando `switch-case`. Note que esta distinção é feita apenas na codificação.

```
/* rodizio1.c - informa o dia do rodizio de um veiculo */

#include <stdio.h>

int main(void) {
    int p, f;

    printf("Placa? ");
    scanf("%d", &p);

    f = p % 10;

    printf("Rodizio: ");

    if( f==1 || f==2 ) printf("segunda\n");
    else if( f==3 || f==4 ) printf("terca\n");
        else if( f==5 || f==6 ) printf("quarta\n");
            else if( f==7 || f==8 ) printf("quinta\n");
                else printf("sexta\n");

    return 0;
}
```

Listagem 4.6 - Programa usando `if-else`, correspondente ao fluxograma da Figura 4.8.

```
/* rodizio2.c - informa o dia do rodizio de um veiculo */

#include <stdio.h>

int main(void) {
    int p, f;

    printf("Placa? ");
    scanf("%d", &p);

    f = p % 10;

    printf("Rodizio: ");

    switch( f ) {
        case 1: case 2: printf("segunda\n"); break;
        case 3: case 4: printf("terca\n"); break;
        case 5: case 6: printf("quarta\n"); break;
        case 7: case 8: printf("quinta\n"); break;
        case 9: case 0: printf("sexta\n"); break;
    }

    return 0;
}
```

Listagem 4.7 - Programa usando `switch-case`, correspondente ao fluxograma da Figura 4.8.



4.4 Exercícios

Para cada problema a seguir, elabore um fluxograma, codifique um programa correspondente em linguagem C e teste o programa usando o compilador *Pelles C*.

- 4.1** Dada uma letra (s, c, d ou v), informe o estado civil correspondente por extenso (*Solteiro*, *Casado*, *Divorciado* ou *Viúvo*).
- 4.2** Dado um número inteiro indicando uma operação num caixa eletrônico, informe a opção correspondente: 1 – *saldo*, 2 – *extrato*, 3 – *saque*, 4 – *sair*.
- 4.3** Dados dois números reais e um caractere (+, -, *, /) representando uma operação a ser efetuada com eles, calcule e informe o resultado da operação.
- 4.4** O perfil de uma pessoa é dado pelo seu ano de nascimento. Por exemplo, se o ano é 1987, calculamos a soma $19+87$, dividimos seu resultado (106) por 5 e pegamos o resto (1). Este resto indica o perfil da pessoa: 0 - *tímido*, 1 - *sonhador*, 2 - *paquerador*, 3 - *atraente*, 4 - *irresistível*. Dado o ano de nascimento de uma pessoa, informe qual é o seu perfil.
- 4.5** Dados o sexo e a altura de uma pessoa, determine seu peso ideal, de acordo com as fórmulas a seguir:
 - para homens o peso ideal é $72.7 \cdot \text{altura} - 58$
 - para mulheres o peso ideal é $62.1 \cdot \text{altura} - 44.7$



Anotações



```
int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
}
```

Repetição Contada

5.1 Acumuladores e contadores

Um **acumulador** é uma variável que ocorre em ambos os lados de uma atribuição e que, antes de ser usada pela primeira vez, é iniciada com um valor específico. Por exemplo, efetuando a atribuição $a \leftarrow 3$, iniciamos a variável a com o valor 3; então, quando a atribuição $a \leftarrow a + 2$ é executada, essa variável passa a ter o valor 5. Note que, se o valor inicial de a não é definido pela primeira atribuição, não é possível determinar o seu valor após a execução da segunda atribuição.

Pelo fato de operações com acumuladores serem tão comuns em programação, C oferece um conjunto de operadores **aritméticos de atribuição**, Tabela 5.1, que permitem escrever expressões com acumuladores de uma forma mais compacta. Por exemplo, usando o operador aritmético de atribuição $+=$, a operação de atribuição $a \leftarrow a + 2$ pode ser codificada como $a += 2$ e a operação de atribuição $a \leftarrow a + 2 * b$ pode ser codificada como $a += 2 * b$.

Tabela 5.1 - Operadores aritméticos de atribuição em C.

Operação	Fluxograma	Programa em C
soma	$a \leftarrow a + \text{expr}$	$a += \text{expr}$
subtração	$a \leftarrow a - \text{expr}$	$a -= \text{expr}$
multiplicação	$a \leftarrow a * \text{expr}$	$a *= \text{expr}$
divisão real	$a \leftarrow a / \text{expr}$	$a /= \text{expr}$
divisão inteira	$a \leftarrow a \text{ div } \text{expr}$	$a /= \text{expr}$
resto da divisão	$a \leftarrow a \text{ mod } \text{expr}$	$a \%=\text{expr}$

Um **contador** é um tipo de acumulador cujo valor aumenta, ou diminui, de 1 em 1. A Tabela 5.2 mostra os operadores em C para lidar com contadores.

Tabela 5.2 - Operadores de incremento e decremento em C.

Operação	Fluxograma	Programa em C
incremento	$c \leftarrow c + 1$	$c++$
decremento	$c \leftarrow c - 1$	$c--$



Operadores de incremento e decremento podem ser usados na forma prefixa (`++c` e `--c`) ou posfixa (`c++` e `c--`). Na forma prefixa, o valor da variável é modificado e depois usado; na forma posfixa, o valor da variável é usado e depois modificado. Assim, por exemplo, se `c` vale 3, a operação `d = ++c` armazena 4 em `d`; enquanto a operação `d = c++` armazena 3 em `d`. Em ambos os casos, o valor final de `c` é 4.

5.2 Estrutura de repetição contada

A estrutura de **repetição contada**, apresentada na Figura 5.1, serve para repetir a execução de um comando por um determinado número de vezes. Para saber quando o total de repetições desejadas já foi atingido, a estrutura de repetição contada usa um *contador*. Nesta estrutura:

- Primeiramente o contador é *iniciado* com um valor específico.
- Depois o contador é *testado*: se o total de repetições ainda não foi atingido, a repetição continua; caso contrário, ela termina.
- A cada nova repetição o contador é *alterado*.

A estrutura de repetição contada é codificada conforme indicado na Figura 5.2. Note que para executar mais de um comando, é preciso usar um par de chaves.

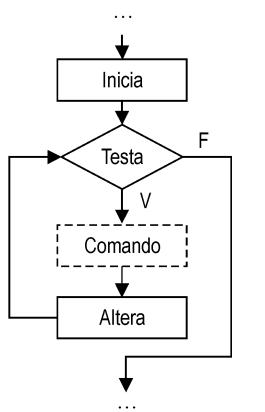


Figura 5.1 - Estrutura de repetição contada.

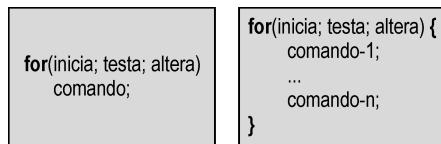


Figura 5.2 - Padrão de codificação da estrutura de repetição contada em C.

O Problema 5.1 mostra um caso em que a estrutura de repetição contada é útil para produzir uma saída em vídeo, composta de várias linhas. Na Figura 5.3 temos um exemplo de tela de execução para este problema.



Problema 5.1

Dado um número inteiro, exiba a sua tabuada.

O fluxograma para resolver o Problema 5.1 é apresentado na Figura 5.4 e o código correspondente em linguagem C é mostrado na Listagem 5.1.

```
Número? 5
Tabuada do 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

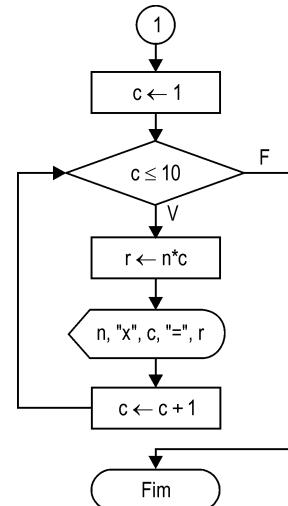
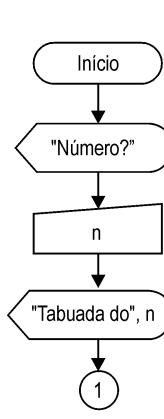


Figura 5.3 - Tela de execução para um programa que resolve o Problema 5.1.

Figura 5.4 - Fluxograma que descreve um algoritmo para o Problema 5.2.

```
/* tabuada.c - exibe a tabuada de um numero inteiro */
#include <stdio.h>

int main(void) {
    int n, c, r;

    printf("Número? ");
    scanf("%d", &n);

    printf("\nTabuada do %d\n\n", n);

    for(c=1 ; c<=10 ; c++) {
        r = n*c;
        printf("%d x %2d = %3d\n", n, c, r);
    }

    return 0;
}
```

Listagem 5.1 - Programa C correspondente ao fluxograma da Figura 5.4.



5.3 Contagem decrescente

A estrutura de repetição contada também pode ser usada com um contador que, em vez de aumentar de 1 em 1, diminui de 1 em 1.

O Problema 5.2 mostra um caso em que podemos usar **contagem decrescente**. Um exemplo de tela de execução para este problema é apresentado na Figura 5.5.

Problema 5.2

Dado um número n , exiba uma contagem regressiva de n até 0.

O fluxograma para resolver o Problema 5.2 é apresentado na Figura 5.6 e o código correspondente em linguagem C é mostrado na Listagem 5.2.

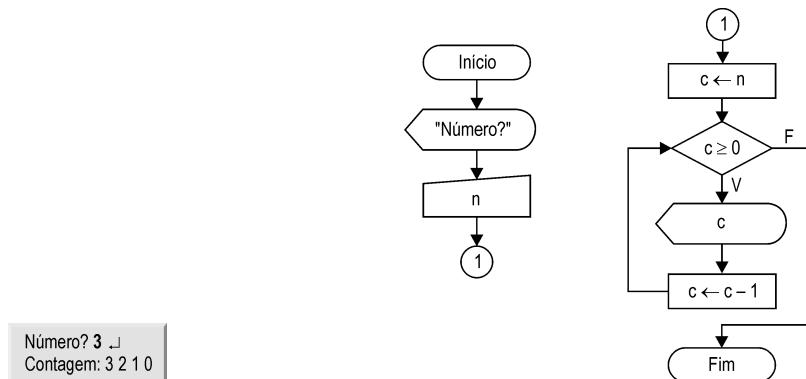


Figura 5.5 - Tela de execução para um programa que resolve o Problema 5.2.

Figura 5.6 - Fluxograma que descreve um algoritmo para o Problema 5.2.

```
/* regr.c - exibe uma contagem regressiva */
#include <stdio.h>
int main(void) {
    int n, c;
    printf("Número? ");
    scanf("%d", &n);
    printf("Contagem: ");
    for(c=n ; c>=0 ; c--)
        printf("%d ", c);
    return 0;
}
```

Listagem 5.2 - Programa C correspondente ao fluxograma da Figura 5.6.

5.4 Estruturas de repetição encaixadas

Estruturas de repetição contada podem ser encaixadas para formar padrões de repetição mais complexos. O Problema 5.3 mostra um caso em que isso é necessário e, na Figura 5.7, temos um exemplo de tela de execução para esse problema.

Problema 5.3

Dado um número natural n , desenhe um quadrado $n \times n$.

O fluxograma para resolver o Problema 5.3 é apresentado na Figura 5.8 e o código correspondente em linguagem C é indicado na Listagem 5.3. Note que, neste algoritmo, o comando que muda o cursor para uma nova linha (indicado pelo símbolo de processo predefinido) é fundamental para a obtenção do resultado desejado e, portanto, deve constar no fluxograma.

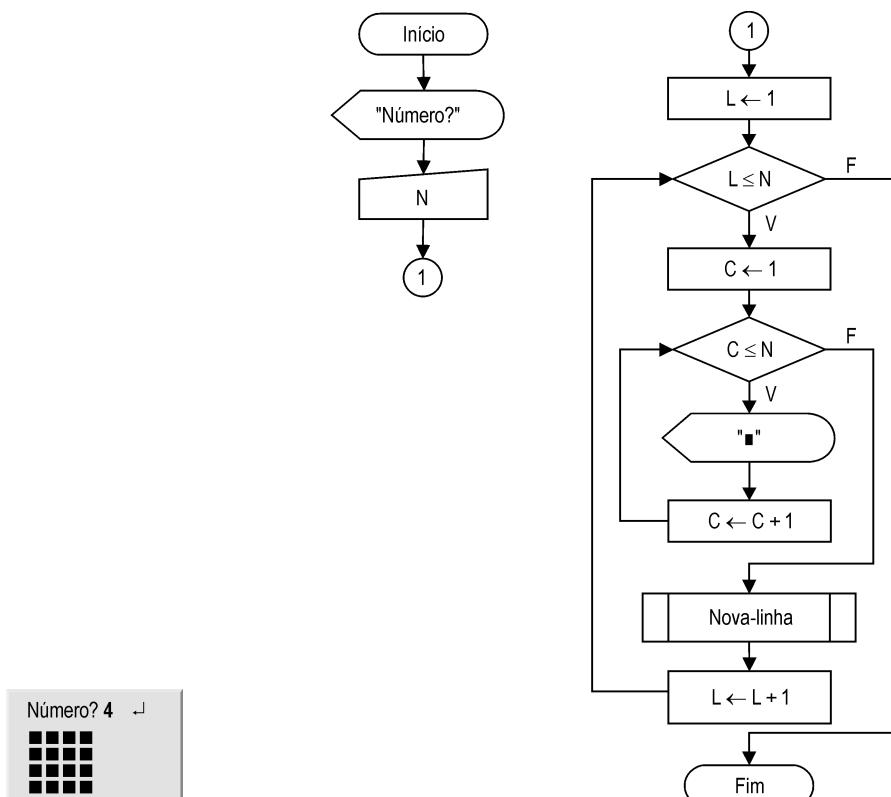


Figura 5.7 - Tela de execução para um programa que resolve o Problema 5.3.

Figura 5.8 - Fluxograma que descreve um algoritmo para o Problema 5.3.



```

/* quadrado.c - desenha um quadrado */

#include <stdio.h>

int main(void) {
    int N, L, C;

    printf("Número? ");
    scanf("%d", &N);

    for(L=1; L<=N; L++) {
        for(C=1; C<=N; C++)
            printf("%c", 219);
        printf("\n");
    }

    return 0;
}

```

Listagem 5.3 - Programa (para Pelles C) correspondente ao fluxograma da Figura 5.8.

Para exibir o caractere '■', cujo código ASCII é 219, o programa da Listagem 5.3 usa o comando `printf()` com o formato `%c`. Isso faz com que o valor 219 seja exibido na forma de caractere em vez de número. De fato, com o formato adequado, `printf()` é capaz de converter números em caracteres e vice-versa. Essa conversão é feita, automaticamente, com base na tabela ASCII. Por exemplo, executando o comando `printf("%d", 'A')`, exibimos o número 65, que é o código ASCII do caractere 'A'; inversamente, executando `printf("%c", 65)`, exibimos o caractere 'A', cujo código ASCII é 65.

Um aspecto interessante do programa na Listagem 5.3 é que, a cada exibição do caractere '■', as variáveis `L` e `C` indicam as coordenadas (linha e coluna) da posição em que ele é exibido dentro do quadrado. Isso indica que podemos alterar facilmente a lógica desse programa para que, em vez de um quadrado, seja exibido um padrão semelhante a um tabuleiro de xadrez, como na Figura 5.9. Basta observar que, quando a soma das coordenadas de uma posição é par, o caractere é exibido em uma cor e, quando a soma é ímpar, o caractere é exibido em outra cor.

Para selecionar a cor em que os caracteres são exibidos no vídeo, podemos usar a função `_textcolor(coro)`, da biblioteca `conio.h`. As cores são representadas por números inteiros entre 0 e 15, cada um correspondendo a uma cor distinta. Note que, segundo a convenção adotada pelo compilador *Pelles C*, todas as funções que não são padrão em C têm seus nomes iniciando com um caractere de sublinha. Isso alerta para o fato de que, para ser compilado com outro

	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

Figura 5.9 - Um tabuleiro de xadrez e a relação entre a soma das coordenadas e cores.



compilador, o programa pode precisar de alguma adaptação (consulte o Apêndice C para ter mais detalhes).

A versão do programa que exibe o tabuleiro de xadrez é apresentada na Listagem 5.4. Observe que, quando a soma das coordenadas de uma posição do quadradão é par, a expressão $(L+C) \% 2$ vale 0 e, quando é ímpar, a expressão vale 1. Portanto, usando o comando `_textcolor((L+C)%2+1)`, selecionamos a cor 1 para as posições com soma par e a cor 2 para as posições com soma ímpar. Além disso, nesse programa usamos a função `putchar()`, da biblioteca `stdio.h`, para exibir o caractere correspondente ao código ASCII 219. Essa função serve exclusivamente para a exibição de caracteres, podendo receber como parâmetro tanto um caractere quanto um inteiro (neste caso, o inteiro é transformado no caractere correspondente).

```
/* xadrez.c - desenha um tabuleiro de xadrez */

#include <stdio.h>
#include <conio.h>

int main(void) {
    int N, L, C;

    printf("Número? ");
    scanf("%d", &N);

    for(L=1; L<=N; L++) {
        for(C=1; C<=N; C++) {
            _textcolor((L+C)%2+1);
            putchar(219);
        }
        printf("\n");
    }

    return 0;
}
```

Listagem 5.4 - Programa (para Pelles C) que exibe um tabuleiro de xadrez.

Apenas como curiosidade, vale lembrar que, além de `_textcolor()`, que define a cor do texto, há também uma função chamada `_textbackground()`, que define a cor do fundo sobre o qual o texto é escrito. Então, outra forma de apresentar o tabuleiro de xadrez seria, em vez de exibir o caractere 219, alternando-se a cor do texto, exibir um espaço em branco (ASCII 32), alternando-se a cor do fundo.

5.5 Exercícios

Para cada problema a seguir, elabore um fluxograma, codifique um programa correspondente em linguagem C e teste o programa usando o compilador *Pelles C*.



- 5.1** Dadas as notas dos alunos de uma turma, informe a média da turma. O programa deve funcionar como indicado a seguir:

```
Total de alunos? 4 ↴
1a nota? 7 ↴
2a nota? 4 ↴
3a nota? 8 ↴
4a nota? 6 ↴
Média da turma = 6.3
```

- 5.2** Dadas as idades dos pacientes de uma clínica, informe a idade daquele mais idoso. Considere que todas as idades são distintas e que o número de pacientes é informado pelo usuário, no momento da execução do programa.

- 5.3** Dados um capital, uma taxa de juros mensal e um período em meses, informe o montante ao final de cada mês. O programa deve funcionar como a seguir:

```
Capital? 100.00 ↴
Juros? 10 ↴
Período? 3 ↴
1º mês: R$ 110.00
2º mês: R$ 121.00
3º mês: R$ 133.10
```

- 5.4** Numa eleição há três candidatos, identificados como A, B e C. Dados os votos dos eleitores, informe o resultado da eleição, conforme exemplificado a seguir:

```
Total de eleitores? 5 ↴
1º voto? B ↴
2º voto? A ↴
3º voto? D ↴
4º voto? B ↴
5º voto? E ↴
Resultado
A.....: 1
B.....: 2
C.....: 0
Nulos...: 2
```

- 5.5** Exibir um relógio digital, cujo mostrador varia de 00:00 a 23:59. **Dica:** para exibir o relógio fixo em um ponto da tela, use o comando `_gotoxy(coluna, linha)`, da biblioteca `conio.h`, em que `coluna` é um número inteiro entre 1 e 80 e `linha` é um número inteiro entre 1 e 25; para dar uma pausa entre a exibição de um horário e do próximo, use o comando `_sleep(segundos)`, da biblioteca `time.h`. Consulte o Apêndice C, caso você deseje usar essas funções no Unix/Linux.



```
int main(void) {
    char m[10][3];
    ler(m,10);
    ordenar(m,10);
```

Repetição com Precondição

6.1 Estrutura de repetição com precondição

A estrutura de **repetição com precondição**, na Figura 6.1, serve para executar um comando, repetidamente, enquanto uma determinada condição for verdadeira. Observe que, como a condição é avaliada *antes* de o comando ser executado, se ela for inicialmente falsa, o *comando* dentro da repetição jamais é executado.

A estrutura de repetição com precondição é codificada como na Figura 6.2.

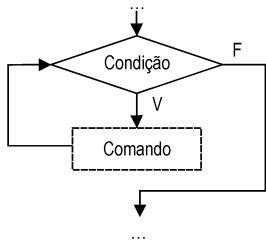


Figura 6.1 - Estrutura de repetição com precondição.

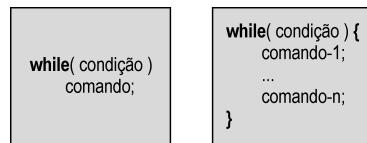


Figura 6.2 - Padrão de codificação da estrutura de repetição com precondição em C.

O Problema 6.1 ilustra um caso em que podemos usar a estrutura de repetição com precondição. Uma tela de execução correspondente é exibida na Figura 6.3.

Problema 6.1

Dado um número inteiro positivo, exiba seus dígitos, Figura 6.1.

O fluxograma para resolver o Problema 6.1 é apresentado na Figura 6.4 e o código correspondente em C é apresentado na Listagem 6.1. Note que o dígito mais à direita de um número é sempre igual ao resto de sua divisão inteira por 10.

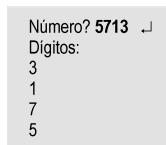


Figura 6.3 - Tela de execução para um programa que resolve o Problema 6.1.



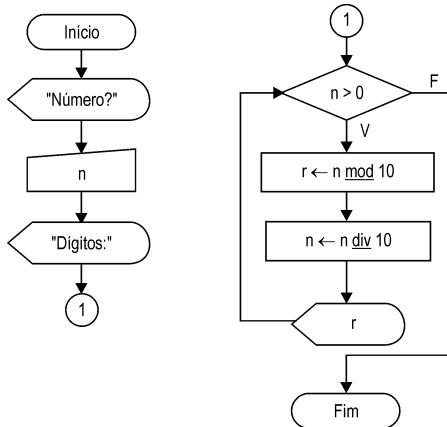


Figura 6.4 - Fluxograma que descreve um algoritmo para o Problema 6.1.

```
/* digitos.c - exibe os dígitos de um número inteiro positivo */
#include <stdio.h>
int main(void) {
    int n, r;
    printf("Número? ");
    scanf("%d", &n);
    printf("Dígitos:\n");
    while( n > 0 ) {
        r = n % 10;
        n = n / 10;
        printf("%d\n", r);
    }
    return 0;
}
```

Listagem 6.1 - Programa C correspondente ao fluxograma da Figura 6.4.

Podemos adaptar a lógica apresentada na Figura 6.4 para resolver o Problema 6.2. Na Figura 6.5, temos um exemplo de tela de execução para este problema.

Problema 6.2

Num banco, as contas são identificadas por um *número de conta* com *dígito verificador*. Esse dígito verificador é calculado do seguinte modo: primeiramente somamos todos os dígitos do número de conta, depois dividimos a soma por 10 e tomamos o resto. Por exemplo, se o número de conta for 5713, temos a soma $3+1+7+5 = 16$; dividindo 16 por 10, temos o resto 6, que é seu dígito verificador. Dado um número de conta, informe o dígito verificador correspondente.



O fluxograma para resolver o Problema 6.2 é apresentado na Figura 6.6 e o código correspondente em linguagem C é apresentado na Listagem 6.2.

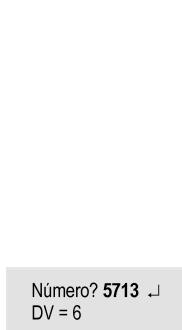


Figura 6.5 - Tela de execução para um programa que resolve o Problema 6.2.

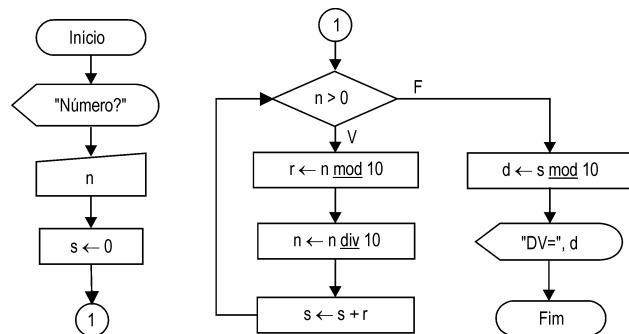


Figura 6.6 - Fluxograma que descreve um algoritmo para o Problema 6.1.

```

/*
 * digver.c - calcula digito verificador */
#include <stdio.h>

int main(void) {
    int n , s, r, d;
    printf("Número? ");
    scanf("%d", &n);
    s = 0;
    while( n > 0 ) {
        r = n % 10;
        n /= 10;           // equivalente a n = n / 10
        s += r;            // equivalente a s = s + r
    }
    d = s % 10;
    printf("DV = %d\n", d);
    return 0;
}
    
```

Listagem 6.2 - Programa C correspondente ao fluxograma da Figura 6.6.

6.2 Repetição com terminação forçada

Em C, quando usamos o comando `while(1) {...}`, criamos uma repetição cuja condição de repetição é sempre verdadeira, ou seja, uma **repetição infinita**; porém com o comando `break`, podemos forçar o término de qualquer repetição.



Repetição com terminação forçada não é um padrão convencional de programação estruturada e, sendo assim, devemos evitar o seu uso quando construímos fluxogramas a serem implementados em outras linguagens.

O Problema 6.3 apresenta uma situação em que este padrão pode ser útil. Uma tela de execução correspondente é exibida na Figura 6.7.

Problema 6.3

Dada uma sequência de números positivos, representando os valores dos itens de uma compra, informe o total a ser pago. Considere que a sequência termina quando aparece o primeiro número não positivo, Figura 6.7.

O fluxograma para resolver o Problema 6.3 é apresentado na Figura 6.8 e o programa correspondente em linguagem C é exibido na Listagem 6.3. Observe que, embora tenhamos usado o comando `while` para codificar esse programa, a condição de parada da repetição não é uma *precondição* propriamente dita, já que é avaliada no *meio* do laço de repetição e não no seu ínicio.

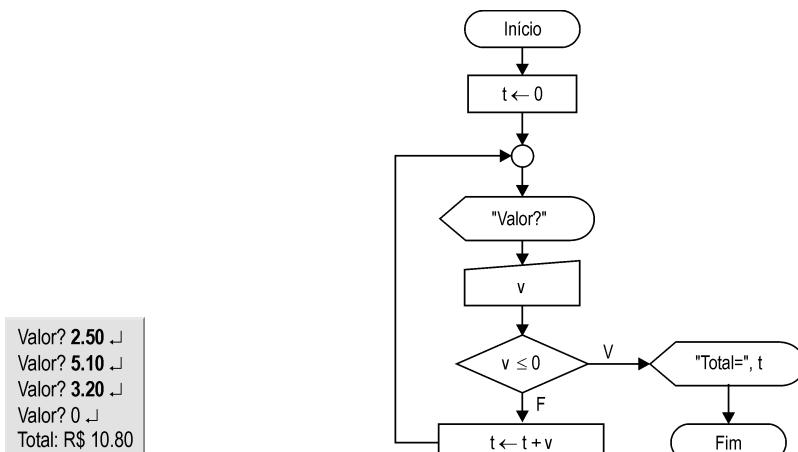


Figura 6.7 - Tela de execução para um programa que resolve o Problema 6.3.

Figura 6.8 - Fluxograma que descreve um algoritmo para o Problema 6.3.

Observe também que o uso da estrutura de repetição contada (Capítulo 5) não seria apropriado para resolver este problema, pois não sabemos de antemão quantos valores serão digitados antes que o valor não positivo seja fornecido.

```

/*
 * compras.c - calcula o total a ser pago por uma compra */
#include <stdio.h>
int main(void) {
    float t=0, v;
  
```



```

while( 1 ) {
    printf("Valor? ");
    scanf("%f",&v);
    if( v<=0 ) break; // forca termino da repeticao
    t += v;
}

printf("\nTotal: R$ %.2f\n", t);

}

```

Listagem 6.3 - Programa correspondente ao fluxograma da Figura 6.8.

6.3 Simulação de pingue-pongue

Nesta seção, vamos usar a estrutura de repetição com precondição para criar um programa que simula o movimento de uma bola de pingue-pongue, conforme ilustrado na Figura 6.9, enquanto o usuário não pressionar uma tecla.

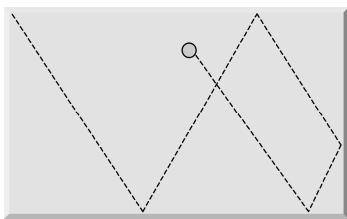


Figura 6.9 - Tela de execução para um programa que simula uma bola de pingue-pongue.

Para entender a lógica desse programa é preciso lembrar que janela de execução padrão do sistema tem 25 linhas e 80 colunas. As linhas são numeradas a partir de 1, de cima para baixo, e as colunas são numeradas a partir de 1, da esquerda para a direita. A posição da bola dentro da janela é indicada pelas variáveis x e y , sendo que x indica a coluna e y indica a linha da janela em que a bola se encontra. Inicialmente essas variáveis têm o valor 1, ou seja, a bola inicia no canto esquerdo superior da janela.

Para simular o movimento da bola, basta criar uma repetição dentro da qual alteramos os valores de x e y . Incrementando o valor de x , fazemos a bola se deslocar da esquerda para a direita e decrementando o valor de x , fazemos a bola se deslocar no sentido contrário. Analogamente, incrementando o valor de y , fazemos a bola se deslocar de cima para baixo e decrementando o valor de y , fazemos a bola se deslocar no sentido contrário. Para indicar o sentido do deslocamento em x , vamos usar a variável dx e para indicar o sentido do deslocamento em y , vamos usar a variável dy . Inicialmente essas variáveis têm o valor 1, ou seja, a bola começa se deslocando de cima para baixo, da esquerda para a direita. A cada vez que a bola bate na borda esquerda ou direita da tela, o sinal da variável dx é invertido (para que o movimento seja invertido).



Analogamente, quando a bola bate na borda superior ou inferior, o valor da variável `dy` é invertido.

O programa C que simula o movimento de uma bola de pingue-pongue, apresentado na Listagem 6.4, utiliza as seguintes funções da biblioteca `conio.h`:

- `_gotoxy(x,y)`, que posiciona o cursor na coluna `x` da linha `y` da tela.
- `_kbhit()`, que devolve verdadeiro se alguma tecla foi pressionada pelo usuário e falso em caso contrário.

```
/* ping pong.c - simula movimento de pingue-pongue */

#include <stdio.h>
#include <conio.h>

int main(void) {
    int x=1, y=1, dx=1, dy=1, i;

    while( ! kbhit() ) {
        gotoxy(x,y);
        putchar('O');
        for(i=0; i<40000000; i++); // causa uma pausa
        gotoxy(x,y);
        putchar(' ');
        x += dx;
        y += dy;
        if( x==1 || x==80) dx = -dx;    // inverte movimento horizontal
        if( y==1 || y==24) dy = -dy;    // inverte movimento vertical
    }

    return 0;
}
```

Listagem 6.4 - Programa (para Pelles C) que simula o movimento de uma bola de pingue-pongue.

6.4 Exercícios

- 6.1 O algoritmo apresentado na Figura 6.6, baseado em *soma simples*, não detecta erros de digitação causados por permutação de dígitos. Uma versão mais robusta pode ser obtida usando *soma ponderada*, em que cada dígito é multiplicado por sua posição. Implemente essa nova versão do algoritmo.
- 6.2 Numa fábrica há dois alarmes: um deles dispara a cada x horas e o outro, a cada y horas. Codifique um programa em C que, dados os valores de x e y , informe qual o tempo mínimo necessário para que os dois alarmes disparem simultaneamente. Considere que x e y são números inteiros positivos.
- 6.3 Um número natural é denominado **triangular** se é igual à soma dos n primeiros números naturais consecutivos, a partir de 1. Por exemplo, 1, 3, 6, 10, 15, ... são triangulares. Dado um natural $n \geq 1$, informe se ele é ou não triangular.



```
int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
```

Repetição com Poscondição

7.1 Estrutura de repetição com poscondição

A estrutura de **repetição com poscondição**, na Figura 7.1, serve para executar um comando, repetidamente, até que uma determinada condição se torne *falsa*. Essa estrutura é usada quando não sabemos de antemão quantas vezes o comando deve ser repetido, mas precisamos que ele seja executado pelo menos uma vez.

A estrutura de repetição com precondição é codificada como na Figura 7.2.

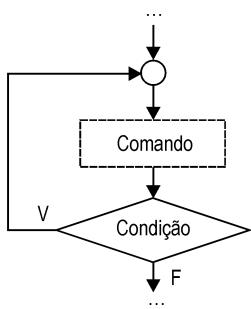


Figura 7.1 - Estrutura de repetição com poscondição.

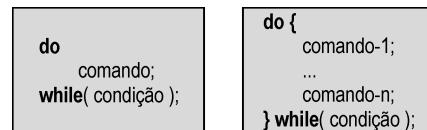


Figura 7.2 - Padrão de codificação da estrutura de repetição com poscondição em C.

Em geral, a estrutura de repetição com poscondição é usada para:

- Garantir *consistência* de entrada de dados.
- Repetir um processo com *confirmação do usuário*.
- Implementar processos orientados por *menus*.

7.2 Consistência de entrada de dados

Para executar corretamente, os programas precisam que o usuário forneça dados **consistentes**. Por exemplo, um programa que calcula a raiz quadrada de um número fornecido pelo usuário não pode executar corretamente se esse número for negativo, pois como não existe raiz quadrada de negativos, sua



execução terminará com **erro fatal** (ou seja, será abortada pelo sistema). Neste caso, em vez de aceitar qualquer valor digitado pelo usuário, o programa deve repetir a entrada de dados até que ela seja *consistente*. O Problema 7.1 ilustra esta situação.

Problema 7.1

Dado um número real não negativo, informe sua raiz quadrada.

A tela de execução, o fluxograma e o código em linguagem C para o Problema 7.1 são apresentados, respectivamente, nas Figuras 7.3 e 7.4 e na Listagem 7.1.

```
Digite um número não negativo: -5 ↵
Digite um número não negativo: -17 ↵
Digite um número não negativo: 9 ↵
A raiz quadrada de 9.0 é 3.0
```

Figura 7.3 - Tela de execução para um programa que resolve o Problema 7.1.

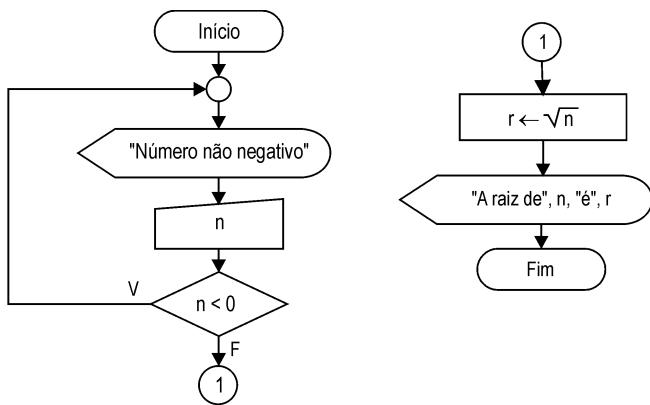


Figura 7.4 - Fluxograma que descreve um algoritmo para o Problema 7.2.

```
/*
 * raiz.c - exibe a raiz de um numero nao negativo */
#include <stdio.h>
#include <math.h>

int main(void) {
    float n, r;
    do {
        printf("Digite um numero nao negativo: ");
        scanf("%f", &n);
    } while( n<0 );
    r = sqrt(n);
    printf("\nA raiz quadrada de %.1f e %.1f\n", n, r);
    return 0;
}
```

Listagem 7.1 - Programa C correspondente ao fluxograma da Figura 7.4.



Note que o uso de estrutura de repetição com precondição não é apropriado para consistência de dados. Para entender o porquê, analise o fluxograma na Figura 7.5. Repare que, quando a condição $n < 0$ é avaliada, o usuário ainda não forneceu o valor de n , portanto não temos como saber se ela é ou não verdadeira.

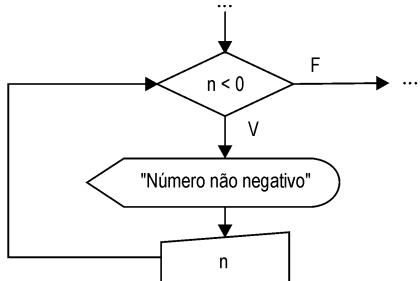


Figura 7.5 - Uso inadequado da repetição com precondição para consistência de dados.

Outra situação em que podemos usar repetição com poscondição para realizar consistência de dados é ilustrada pelo Problema 7.2, cuja tela de execução é exibida na Figura 7.6. A solução para este problema é apresentada na Figura 7.7 e na Listagem 7.2.

Problema 7.2

Dado um número inteiro entre 1 e 10, exiba a sua tabuada:

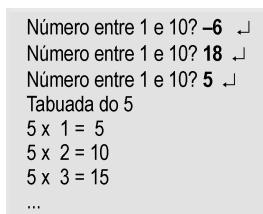


Figura 7.6 - Telas de execução para um programa que resolve o Problema 7.2.

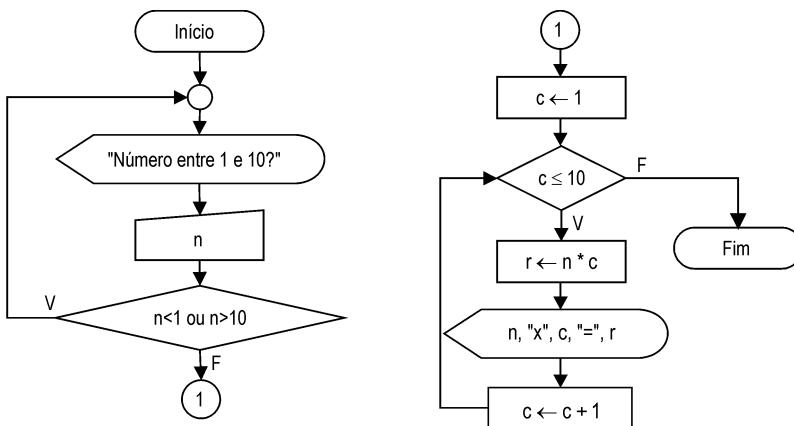


Figura 7.7 - Fluxograma que descreve um algoritmo para o Problema 7.2.



```

/* tabuada.c - exibe a tabuada de um numero inteiro entre 1 e 10 */

#include <stdio.h>

int main(void) {
    int n, c, r;

    do {
        printf("Número entre 1 e 10? ");
        scanf("%d", &n);
    } while( n<1 || n>10 );

    printf("\nTabuada do %d\n\n", n);

    for( c=1 ; c<=10 ; c++ ) {
        r = n*c;
        printf("%d x %2d = %3d\n", n, c, r);
    }

    return 0;
}

```

Listagem 7.2 - Programa C correspondente ao fluxograma da Figura 7.7.

7.3 Repetição com confirmação do usuário

A repetição com **confirmação do usuário** consiste num padrão em que um processo é executado e, ao final, o usuário é indagado se deseja continuar ou não. Trata-se de um padrão bastante genérico, conforme ilustrado na Figura 7.8.

O programa na Listagem 7.3 mostra o uso deste padrão. Nesse programa, a função `getchar()`, da biblioteca `stdio.h`, é usada para ler um caractere, representando a resposta do usuário, e a função `toupper()`, da biblioteca `ctype.h`, é usada para converter esse caractere em maiúscula antes de compará-lo a '`N`'. O formato `%*c`, em `scanf()`, é necessário para descartar o *enter* digitado ao final da entrada do número; sem isso, a função `getchar()` não vai parar para ler a resposta do usuário. A função `_clrscr()`, definida em `conio.h`, é usada para limpar a tela a cada nova repetição.

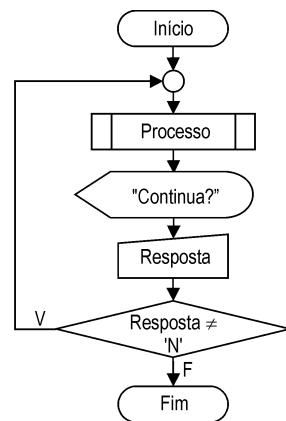


Figura 7.8 - Padrão de repetição com confirmação do usuário.

```

/* tabuada.c - exibe a tabuada de um numero inteiro entre 1 e 10 */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>

int main(void) {
    int n, c, r;

    do {
        clrscr();
        do {
            printf("Numero entre 1 e 10? ");
            scanf("%d%c", &n);
        } while( n<1 || n>10 );
        printf("\nTabuada do %d\n\n", n);
        for(c=1; c<=10; c++) {
            r = n*c;
            printf("%d x %2d = %3d\n", n, c, r);
        }
        printf("\nContinua (s/n)? ");
    } while( toupper(getchar()) !='N' );

    return 0;
}

```

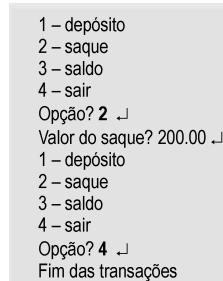
Listagem 7.3 - Programa (para Pelles C) que usa o padrão apresentado na Figura 7.8.

7.4 Processos orientados por menus

Num processo orientado por menu, uma série de opções é apresentada e, conforme a opção escolhida pelo usuário, uma ação correspondente é executada. Em seguida, o menu de opções é reapresentado e o processo se repete até que o usuário decida finalizar a sua execução. O Problema 7.3 ilustra esse tipo de aplicação, cuja tela de execução correspondente é apresentada na Figura 7.9.

Problema 7.3

Simular o funcionamento de um caixa eletrônico, que oferece as seguintes opções ao cliente: 1 - *depósito*, 2 - *saque*, 3 - *saldo* e 4 - *sair*. Suponha que o saldo inicial do cliente seja de R\$ 1.000,00 e que o saldo pode ficar negativo.



```

1 - depósito
2 - saque
3 - saldo
4 - sair
Opção? 2 ↵
Valor do saque? 200.00 ↵
1 - depósito
2 - saque
3 - saldo
4 - sair
Opção? 4 ↵
Fim das transações

```

Figura 7.9 - Tela de execução para um programa que resolve o Problema 7.3.



O fluxograma para resolver o Problema 7.3 é apresentado na Figura 7.10 e o código correspondente em linguagem C é mostrado na Listagem 7.4.

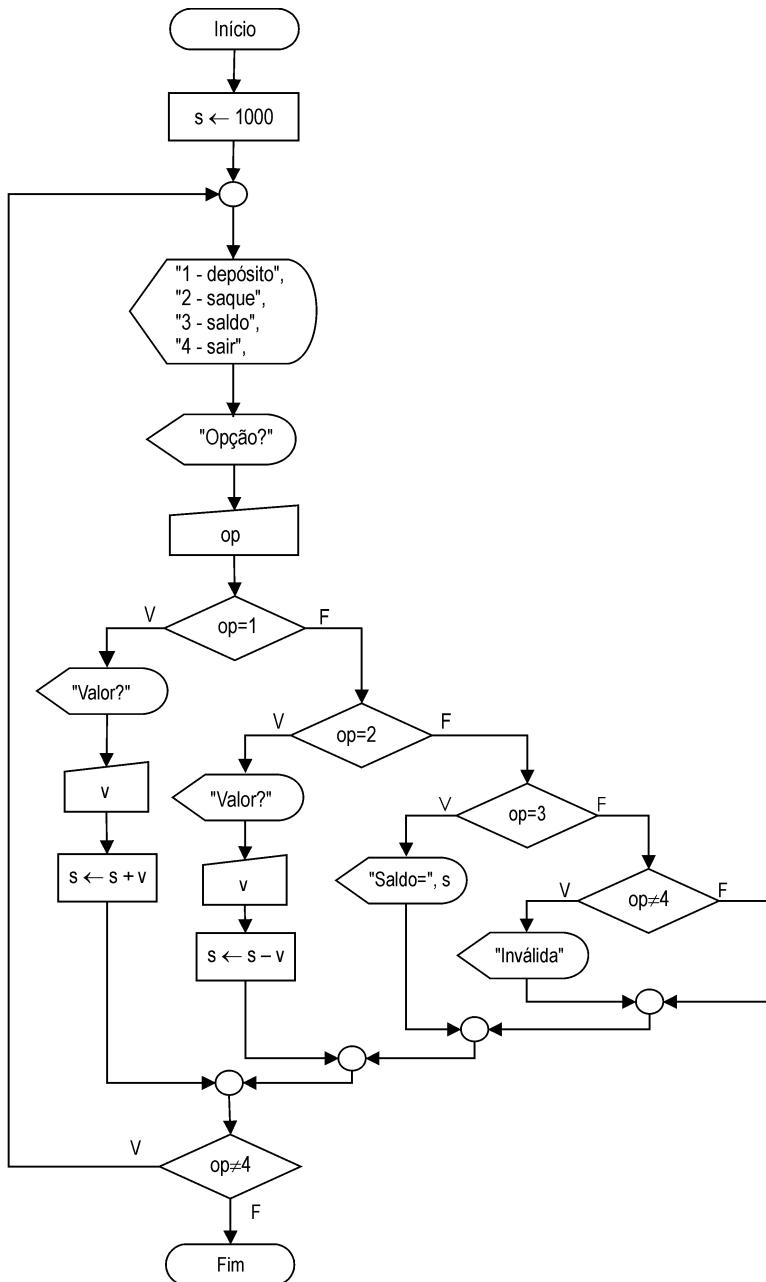


Figura 7.10 - Fluxograma que descreve um algoritmo para o Problema 7.3.



O programa da Listagem 7.4 usa a função `puts()` para exibir as opções do menu. Essa função, que faz parte da biblioteca `stdio.h`, é especializada para a exibição de strings e tem a vantagem de mudar o cursor para uma nova linha, automaticamente, depois de exibir a string. Evidentemente, o mesmo efeito pode ser obtido com o uso da função `printf()`, porém com `puts()` a sintaxe fica mais simples. Vale ressaltar que a função `puts()` só serve para a exibição de strings, não podendo ser usada para a exibição de valores de outros tipos de dados (`char`, `int` e `float`).

```
/* caixa.c - simula o funcionamento de um caixa eletronico */
#include <stdio.h>

int main(void) {
    float s = 1000.00, v;
    int op;

    do {
        puts("\n1 - deposito");
        puts("2 - saque");
        puts("3 - saldo");
        puts("4 - sair");
        printf("\nOpcao? ");
        scanf("%d", &op);
        switch( op ) {
            case 1 : printf("\nValor do deposito? ");
                       scanf("%f", &v);
                       s += v;
                       break;
            case 2 : printf("\nValor do saque? ");
                       scanf("%f", &v);
                       s -= v;
                       break;
            case 3 : printf("\nSaldo atual = R$ %.2f\n", s);
                       break;
            default: if( op!=4 ) puts("\nOpcao invalida!");
        }
    } while( op!= 4 );

    puts("Fim das transacoes");
    return 0;
}
```

Listagem 7.4 - Programa C que resolve o Problema 7.3.



7.5 Exercícios

Para cada problema a seguir, elabore um fluxograma, codifique um programa correspondente em linguagem C e teste o programa usando o compilador *Pelles C*.

7.1 Dadas as duas notas de um aluno, informe a sua média. Seu programa deve forçar o usuário a digitar notas na faixa de 0 a 10.

7.2 Modifique a lógica elaborada para o exercício anterior de modo que, ao final de cada execução, o usuário tenha a opção de repetir o processo.

7.3 Crie um programa que exiba um menu com as seguintes opções:

1 - somar

2 - subtrair

3 - multiplicar

4 - dividir

5 - sair

Após a escolha da opção, o usuário deve fornecer dois números e o programa deve mostrar o resultado da operação.

7.4 Refaça o programa da Listagem 6.3 (no Capítulo 6), usando o comando `do-while` em vez do comando `while`.

7.5 Escolha um programa que você tenha feito anteriormente e altere-o de modo que, ao final, o usuário tenha a opção de repetir o processo.



```
int main(void){  
    char m[10][31];  
    ler(m,10);  
    ordenar(m,10);  
}
```

Macros e Funções

8.1 Macros

O processo de criação de um programa executável pode ser descrito pelo esquema na Figura 8.1. Primeiramente o texto do programa em C, chamado **código-fonte**, passa pelo **pré-processador**, que faz algumas modificações nele e o encaminha ao **compilador**. O compilador, por sua vez, traduz esse código-fonte para uma forma intermediária, mais próxima daquela que o computador é capaz de executar, chamada **código-objeto**. Finalmente, o código-objeto é ligado às funções de biblioteca, pelo **linkeditor**, e transformado num programa **executável**. Dependendo do sistema operacional e do compilador utilizados, esse processo pode ter algumas variações. Por exemplo, quando usamos o compilador GCC no Unix/Linux, o programa executável criado é chamado `a.out`; mas o processo é basicamente o mesmo.

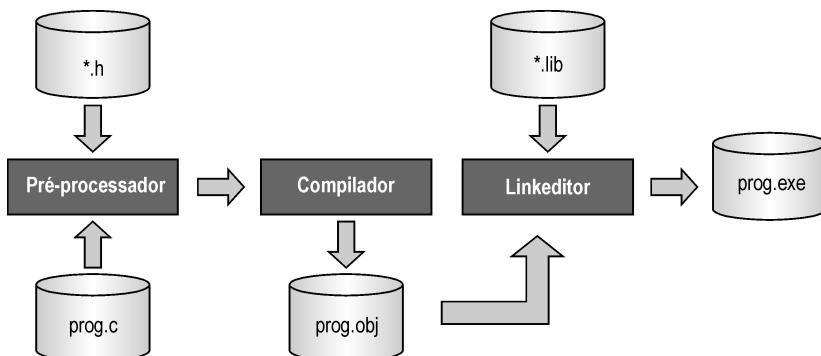


Figura 8.1 - Processo de criação de programa executável em C.

O pré-processador é capaz de realizar várias alterações interessantes no código-fonte de um programa, antes de enviá-lo ao compilador. Essas alterações podem ser especificadas por meio de **diretivas** de preprocessamento embutidas no código-fonte do programa. Entre as diversas diretivas existentes, vamos considerar apenas as duas mais importantes: `#include` e `#define`.

A diretiva `#define` pode ser usada de duas formas:

```
#define identificador texto
#define macro(parâmetros) texto parametrizado
```

Na primeira forma, durante o processamento, todas as ocorrências do *identificador* no código-fonte são substituídas pelo *texto* correspondente. Na segunda forma, os valores na chamada da macro são casados com os respectivos *parâmetros* em sua definição e todas as suas ocorrências são substituídas pelo *texto parametrizado*.

A finalidade básica da diretiva `#define` é aumentar a legibilidade e facilitar a manutenção do programa, definindo nomes simbólicos para constantes numéricas. Por exemplo, em vez de usar diretamente o valor 3.14 num programa, podemos definir esse número como PI e usar este nome para referenciá-lo:

```
#define PI 3.14
```

A vantagem é que, se mais tarde precisarmos aumentar a precisão desse valor, substituindo-o por 3.141593, mesmo que ele tenha sido usado dezenas de vezes no programa, basta alterarmos o `#define`.

Vale ressaltar que a diretiva `#define` pode ser usada para realizar qualquer tipo de substituição, não apenas numérica, como vemos no programa da Listagem 8.1.

```
/* portugues.c */
#include <stdio.h>

#define algoritmo      int main(void)
#define inicio          {
#define fim             return 0; }
#define exiba(mensagem) puts(mensagem)

algoritmo
inicio
    exiba("Oi!");
fim
```

Listagem 8.1 - Uso da diretiva `#define`, com e sem parâmetros.

As macros criadas com `#define` também são úteis para representar expressões numa forma mais simples. Por exemplo, para decidirmos se um número inteiro *n* é par, precisamos dividi-lo por 2 e comparar o resto da divisão com 2, ou seja, precisamos avaliar a expressão $n \% 2 == 0$. Então, se esta for uma operação muito usada em um programa, podemos definir a macro:

```
#define par(n) ((n)%2==0)
```



```
#define par(n) ((n)%2==0)
```

Assim, quando precisarmos saber se um valor *x* é par, basta escrever `par(x)`. Analogamente, supondo que frequentemente precisamos verificar se um dado valor está dentro de um determinado intervalo, podemos definir a macro:

```
#define entre(v,min,max) ((v)>=(min) && (v)<=(max))
```

Assim, quando precisarmos saber se uma variável *x* tem valor entre 1 e 10 e se *y* tem valor entre -1.5 e +1.5, basta escrever `entre(x,1,10)` e `entre(y,-1.5,+1.5)` respectivamente. Note como o uso de macros aumenta a legibilidade do programa.

A diretiva `#include` pode ser usada de duas formas:

```
#include <arquivo_do_compilador>
#include "arquivo_do_programador"
```

Essa diretiva, quando executada pelo pré-processador, faz com que uma cópia do arquivo indicado seja incluída no programa. A primeira forma é usada com arquivos do próprio compilador (por exemplo, `stdio.h`), enquanto a segunda forma é usada com arquivos criados pelo programador. Por exemplo, podemos criar um arquivo com as definições da Listagem 8.2 e salvá-lo com o nome `port.h`. Então, quando quisermos usar essas definições, basta adicionar uma diretiva `#include`, conforme indicado na Listagem 8.3.

```
/* port.h */
#define algoritmo      int main(void)
#define inicio         {
#define fim            return 0; }
#define exiba(mensagem) puts(mensagem)
```

Listagem 8.2 - Arquivo de cabeçalho criado pelo programador.

```
/* portugues2.c */
#include <stdio.h>
#include "port.h"

algoritmo
inicio
    exiba("Oi!");
fim
```

Listagem 8.3 - Uso da diretiva `#include`, nas duas formas possíveis.



8.2 Funções

Essencialmente, uma **função** é um bloco de código cuja execução visa atingir um objetivo específico. Em C, há funções predefinidas, como, por exemplo, `_clrscr()` e `sqrt()`, bem como funções definidas pelo próprio programador, como, por exemplo, `main()`. Na Figura 8.2, temos o padrão que foi estabelecido pela ISO (*International Organization for Standardization*) para criação de funções em C.

```
tipo nome( parâmetros ){
    variáveis locais;
    comandos;
}
```

Figura 8.2 - Padrão ISO para criação de funções em C.

Neste padrão:

- **tipo**: refere-se ao tipo do valor devolvido como resposta ao final da execução da função (deve ser `void` se não houver um valor a ser devolvido).
- **nome**: identifica a função e permite que ela seja referenciada no programa.
- **parâmetros**: é uma lista de variáveis representando dados de entrada, necessários para a execução da função (deve ser `void` se não houver dados de entrada).
- **variáveis locais**: são variáveis criadas quando a função entra em execução e destruídas quando a execução termina (acessíveis apenas dentro da função).
- **comandos**: são passos a serem executados pela função.

Para ser executada, uma função precisa ser chamada, *direta* ou *indiretamente*, pela função principal `main()`. Por exemplo, no programa da Listagem 8.4, a função `main()` chama *diretamente* a função `f()` que, por sua vez, chama a função `g()`, ou seja, `main()` chama *indiretamente* a função `g()`. Como `h()` não é chamada por `main()`, nem *direta* nem *indiretamente*, essa função não será executada. Portanto, a saída produzida por esse programa será `M1 F1 G F2 M2`.

```
#include <stdio.h>

void h(void)      { puts("H"); }
void g(void)      { puts("G"); }
void f(void)      { puts("F1"); g(); puts("F2"); }
int main(void)   { puts("M1"); f(); puts("M2"); return 0; }
```

Listagem 8.4 - Um programa composto por várias funções.



8.3 Tipos de funções

Ao criarmos uma função, a primeira tarefa é definir o seu tipo. Para isso, é preciso determinar se, quando chamarmos essa função, o objetivo será dar uma *ordem* ou fazer uma *pergunta*. Se o objetivo for dar uma ordem, a execução da função deve produzir um *efeito*; caso contrário, se o objetivo for fazer uma pergunta, então sua execução deve devolver uma *resposta*.

Em C, uma função cuja execução produz apenas um efeito deve ser definida com o tipo `void`; por outro lado, uma função cuja execução devolve resposta deve ser definida com o tipo dessa resposta (por exemplo, `char`, `int` ou `float`).

Para tornar estes conceitos mais claros, vamos analisar os tipos de algumas funções predefinidas em C:

- `_clrscr()`: uma chamada a esta função corresponde à ordem “*limpe a tela*”, portanto essa função deve ser definida com o tipo `void`.
- `toupper('a')`: uma chamada a esta função corresponde à pergunta “*qual a maiúscula da letra ‘a’?*”; como a resposta esperada para esta pergunta é um caractere, esta função pode ser criada com o tipo `char`.
- `_kbhit()`: uma chamada a esta função corresponde à pergunta “*alguma tecla foi pressionada?*”; como a resposta esperada para esta pergunta é verdadeiro (1) ou falso (0), esta função deve ser definida com o tipo `int`.
- `sqrt(2)`: uma chamada a esta função corresponde à pergunta “*qual a raiz quadrada de 2?*”; como a resposta desta pergunta é um número real, ela deve ser definida com o tipo `float`.

O Problema 8.1 ilustra um caso em que precisamos criar funções do tipo `void`, ou seja, funções que não devolvem resposta.

Problema 8.1

Crie a função `retangulo(x1, y1, x2, y2, c)`, que exibe um retângulo cujo canto *esquerdo superior* tem coordenadas (x_1, y_1) , cujo canto *direito inferior* tem coordenadas (x_2, y_2) e cuja cor é `c`. Em seguida, faça um programa para exibir um retângulo vermelho no vídeo, conforme ilustrado na Figura 8.3.

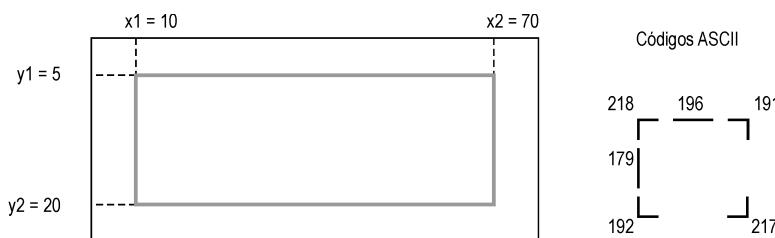


Figura 8.3 - Vídeo exibindo um retângulo vermelho com cantos (10,5) e (70,20).

Um programa que resolve o Problema 8.1 é apresentado na Listagem 8.5. Nesse programa, além da função `retangulo()`, criamos também a função `exibe(c,x,y)`, que exibe o caractere de código ASCII `c` na coluna `x` da linha `y` do vídeo. Ambas as funções são do tipo `void`. De fato, toda função cujo único objetivo é exibir dados em vídeo deve ser definida como `void`.

```
/* retangulo.c - exibe um retângulo vermelho no vídeo */

#include <stdio.h>
#include <conio.h>

void exibe(int c, int x, int y) {
    gotoxy(x,y);
    putchar(c);
}

void retangulo(int x1, int y1, int x2, int y2, int c) {
    int x, y;
    textcolor(c);
    exibe(218,x1,y1); exibe(191,x2,y1);
    exibe(192,x1,y2); exibe(217,x2,y2);
    for(x=x1+1; x<=x2-1; x++) {
        exibe(196,x,y1);
        exibe(196,x,y2);
    }
    for(y=y1+1; y<=y2-1; y++) {
        exibe(179,x1,y);
        exibe(179,x2,y);
    }
    textcolor(7);
}

int main(void) {
    retangulo(10,5,70,20,4); // 4 = vermelho
    gotoxy(1,25);

    return 0;
}
```

Listagem 8.5 - Programa (para Pelles C) que resolve o Problema 8.1.

Note que, ainda que todos os parâmetros de uma função sejam do mesmo tipo, como acontece nas funções `exibe()` e `retangulo()`, este tipo deve ser repetido a cada novo parâmetro declarado.

O Problema 8.2 ilustra um caso em que precisamos de funções que devolvem respostas. Essas funções não podem ser definidas com o tipo `void`.

Problema 8.2

Numa disciplina são dadas duas provas e dois trabalhos, mas a média é calculada considerando-se apenas a maior nota de prova e a maior nota de trabalho. Dadas as quatro notas de um aluno, informe a sua média. Crie e use as



funções `maior()`, que determina o maior entre dois números, e `media()`, que calcula a média aritmética de dois números.

Um programa que resolve o Problema 8.2 é apresentado na Listagem 8.6.

```
/* media.c - exibe a media de um aluno */
#include <stdio.h>

float maior(float a, float b) {
    if( a>b ) return a;
    else return b;
}

float media(float a, float b) {
    float c;
    c = (a+b)/2;
    return c;
}

int main(void) {
    float p1, p2, t1, t2, m;

    printf("Provas? ");
    scanf("%f %f", &p1, &p2);
    printf("Trabalhos? ");
    scanf("%f %f", &t1, &t2);

    m = media( maior(p1,p2), maior(t1,t2) );
    printf("\nMedia = %.1f\n", m);

    return 0;
}
```

Listagem 8.6 - Programa que resolve o Problema 8.2.

Note que o comando `return` é usado para definir o valor que deve ser devolvido como resposta pela função. Esse valor, que pode ser uma constante, uma variável ou uma expressão, deve ser indicado logo após a palavra `return`. Além disso, ao ser executado, o comando `return` termina imediatamente a execução da função, fazendo com que a execução do programa continue no comando que aparece logo após o ponto em que a função foi chamada.

Vale ressaltar que o comando `return` também pode ser usado em uma função do tipo `void`. Neste caso, porém, ele não pode ter um valor associado e sua execução serve apenas para interromper a execução da função, sem devolver resposta.



8.4 Uso de protótipos

De acordo com as regras da linguagem C, uma função só pode chamar outra função que tenha sido previamente *definida* ou *declarada* no programa. Assim, por exemplo, se invertermos a ordem em que as funções estão definidas nos programas das Listagens 8.4, 8.5 e 8.6, esses programas não poderão ser compilados corretamente. Porém, se declararmos as funções que serão usadas num programa (adicionando seus protótipos), então essas funções podem ser definidas em qualquer ordem no código-fonte desse programa.

Um **protótipo** nada mais é que uma declaração da interface de uma função, ou seja, uma declaração que especifica o *tipo*, o *nome* e a lista de *parâmetros* que a função precisa para ser corretamente executada.

A solução do Problema 8.3, na Listagem 8.7, mostra o uso de protótipos.

Problema 8.3

Dada uma data, se ela for válida, informe a data do dia seguinte.

```
/* data.c - valida data e exibe data do dia seguinte */

#include <stdio.h>

// prototipos das funcoes

int valida(int d, int m, int a);
int ultimoDia(int m, int a);
int bissexto(int a);
void exibeDiaSeguinte(int d, int m, int a);

// definicoes das funcoes

int main(void) {
    int d, m, a;

    printf("Data? ");
    scanf("%d/%d/%d", &d, &m, &a);

    if( valida(d,m,a) )
        exibeDiaSeguinte(d,m,a);
    else
        puts("Data inválida! ");

    return 0;
}

int valida(int d, int m, int a) {
    if( a<0 ) return 0;
    if( m<1 || m>12 ) return 0;
    if( d<1 || d> ultimoDia(m,a) ) return 0;
    return 1;
}

int ultimoDia(int m, int a) {
```



```

    if( m==2 ) return 28 + bissexto(a);
    if( m==4 || m==6 || m==9 || m==11 ) return 30;
    return 31;
}

int bissexto(int a) {
    return (a%4==0 && a%100!=0) || (a%400==0);
}

void exibeDiaSeguinte(int d, int m, int a) {
    if( d<ultimoDia(m,a) ) d++;
    else {
        d=1;
        m++;
        if( m==13 ) {
            m=1;
            a++;
        }
    }
    printf("\nData do dia seguinte: %02d/%02d/%d\n", d, m, a);
}

```

Listagem 8.7 - Programa que resolve o Problema 8.3.

8.5 Exercícios

Para cada exercício a seguir, defina a função solicitada e crie um programa para testar seu funcionamento usando o compilador *Pelles C*.

- 8.1** Codifique uma função que receba um número natural n como entrada e exiba uma contagem regressiva de n até 0.
- 8.2** Codifique uma função que receba um número inteiro como entrada e exiba o dia da semana correspondente por extenso (1 – *domingo*, 2 – *segunda*, ...).
- 8.3** Codifique uma função que receba um número natural como entrada e determine se ele é par ou não.
- 8.4** Codifique uma função que calcula a média ponderada de duas notas. Essa função deve receber quatro dados de entrada, sendo as notas e seus respectivos pesos. Os pesos devem ser valores entre 0 e 1, com soma igual a 1.
- 8.5** O *fatorial* de um número natural n é definido pelo produto $1 \times 2 \times 3 \times \dots \times n$. Codifique uma função que receba um número natural e determine o seu fatorial.
- 8.6** Com base no programa da Listagem 8.7, crie um programa que leia uma data e exiba a data do dia anterior à data lida.



Anotações



```
int main(void) {
    char m[10][3];
    ler(m,10);
    ordenar(m,10);
}
```

Vetores

9.1 Armazenamento

Um **vetor** é um tipo de variável capaz de armazenar uma coleção de dados do mesmo tipo. Cada um dos dados armazenados num vetor, denominado **item**, é identificado por um número natural, a partir de 0, denominado **índice**.

Para indicar que uma variável é um vetor, devemos declará-la com o sufixo `[n]`, sendo `n` um valor inteiro positivo que estabelece o tamanho do vetor. Por exemplo, a declaração:

```
char v[3];
```

cria um vetor com três posições, cada uma das quais capaz de armazenar um caractere. Na verdade, com esta única declaração, criamos as variáveis `v[0]`, `v[1]` e `v[2]`, conforme ilustrado na Figura 9.1. Note que, como a indexação inicia-se em 0, o último item de um vetor de tamanho `n` é armazenado na posição `n-1`.

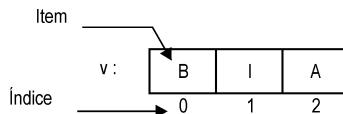


Figura 9.1 - Representação de um vetor na memória do computador.

O Problema 9.1 ilustra uma situação em que precisamos usar um vetor. O programa que resolve este problema é apresentado na Listagem 9.1 e sua tela de execução é apresentada na Figura 9.2.

Problema 9.1

Leia uma sequência de cinco números e exiba-a em ordem inversa.

```
1º número? 32 ↴
2º número? 76 ↴
3º número? 12 ↴
4º número? 99 ↴
5º número? 27 ↴
Ordem inversa: 27 99 12 76 32
```

Figura 9.2 - Tela de execução para um programa que resolve o Problema 9.1.



A lógica desse programa é simples. Basta ir lendo os números e guardando nas posições do vetor, da esquerda para a direita; em seguida, após o completo preenchimento do vetor, os itens são acessados da direita para a esquerda e exibidos.

```
/* inv5.c - exibe sequencia de 5 numeros em ordem inversa */
#include <stdio.h>

int main(void) {
    int v[5], i;

    for(i=0; i<5; i++) {
        printf("%do. numero? ", i+1);
        scanf("%d", &v[i]);
    }

    printf("\nOrdem inversa: ");

    for(i=4; i>=0; i--)
        printf("%d ", v[i]);
}

return 0;
}
```

Listagem 9.1 - Programa que resolve o Problema 9.1.

9.2 Vetor com tamanho variável

De acordo com o padrão ISO, o tamanho de um vetor também pode ser indicado por uma variável. O Problema 9.2 ilustra uma situação em que isso é necessário. O programa que resolve este problema é apresentado na Listagem 9.2 e sua tela de execução é mostrada na Figura 9.3.

Problema 9.2

Leia uma sequência de n números e exiba-a em ordem inversa.

```
Tamanho da sequência? 3 ↴
1º número? 92 ↴
2º número? 47 ↴
3º número? 88 ↴
Ordem inversa: 88 47 92
```

Figura 9.3 - Tela de execução para um programa que resolve o Problema 9.2.



```

/* invn.c - exibe sequência de n numeros em ordem inversa */
#include <stdio.h>
int main(void) {
    int n, i;

    printf("Tamanho da sequencia? ");
    scanf("%d", &n);

    int v[n]; // cria vetor com tamanho igual ao valor de n

    for(i=0; i<n; i++) {
        printf("%do. numero? ", i+1);
        scanf("%d", &v[i]);
    }

    printf("\nSequencia invertida: ");
    for(i=n-1; i>=0; i--)
        printf("%d ", v[i]);

    return 0;
}

```

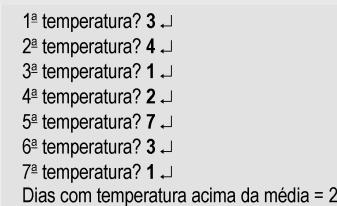
Listagem 9.2 - Programa que resolve o Problema 9.2.

9.3 Vetor como parâmetro de função

Um vetor pode ser passado como parâmetro a uma função. Neste caso, o parâmetro correspondente na função torna-se uma **referência** para o vetor original e qualquer modificação feita nessa referência afeta também o vetor original. O Problema 9.3 ilustra uma situação em que a passagem de vetor por referência é útil.

Problema 9.3

Dadas as temperaturas registradas diariamente, durante uma semana, informe em quantos dias a temperatura ficou acima da média.



```

1ª temperatura? 3 ↵
2ª temperatura? 4 ↵
3ª temperatura? 1 ↵
4ª temperatura? 2 ↵
5ª temperatura? 7 ↵
6ª temperatura? 3 ↵
7ª temperatura? 1 ↵
Dias com temperatura acima da média = 2

```

Figura 9.4 - Tela de execução para um programa que resolve o Problema 9.3.



```

/* temp.c - contabiliza dias mais quentes da semana */
#include <stdio.h>
void preencher(float t[7]) {
    int i;

    for(i=0; i<7; i++) {
        printf("%da. temperatura? ", i+1);
        scanf("%f", &t[i]);
    }
}

float media(float t[7]) {
    int i;
    float s=0;

    for(i=0; i<7; i++) s += t[i];
    return s/7;
}

int conta(float t[7], float m) {
    int i, d=0;

    for(i=0; i<7; i++)
        if( t[i]>m )
            d++;

    return d;
}

int main(void) {
    float t[7], m;

    preencher(t);
    m = media(t);
    printf("\nDias com temperatura acima da media = %d\n", conta(t,m));
    return 0;
}

```

Listagem 9.3 - Programa que resolve o Problema 9.3.

9.4 Iniciação de vetor com tamanho constante

Um vetor de tamanho *constante* também pode ser iniciado ao ser declarado. Neste caso, os valores iniciais do vetor devem ser indicados entre chaves e separados por vírgulas. Por exemplo, a declaração:

```
char vogais[5] = {'a', 'e', 'i', 'o', 'u'};
```

cria um vetor chamado *vogais*, que armazena as letras 'a', 'e', 'i', 'o' e 'u', nesta ordem; ou seja, a variável *vogais[0]* vale 'a', *vogais[1]* vale 'e', *vogais[2]* vale 'i' e assim por diante. Analogamente, a declaração:

```
float moedas[6] = {1.00, 0.50, 0.25, 0.10, 0.05, 0.01};
```



cria um vetor chamado `moedas`, que armazena os números reais 1.00, 0.50, 0.25, 0.10, 0.05 e 0.01, nesta ordem.

Se a quantidade de valores iniciais na declaração de um vetor for *menor* que o tamanho deste, as demais posições do vetor são automaticamente zeradas. Assim, por exemplo, a declaração:

```
int vetor[4] = {10, 20};
```

cria um vetor cujos dois primeiros itens são 10 e 20 e cujos últimos dois itens são iguais a zero. Por outro lado, se a quantidade de valores iniciais é *maior* que o tamanho do vetor, uma mensagem de erro é exibida pelo compilador. Vale ressaltar que vetores de tamanho variável não podem ser iniciados.

O Problema 9.4 ilustra uma situação em que a iniciação de vetor de tamanho constante é útil.

Problema 9.4

Modifique o programa da Listagem 9.3 para que ele exiba um gráfico de barras mostrando a variação da temperatura durante a semana. Nesse gráfico, o tamanho de cada barra deve ser proporcional à temperatura que esta representa. Ademais, as barras devem ser coloridas com *azul*, *vermelho* ou *verde*, respectivamente, para temperaturas abaixo, acima ou igual à média.

```
1a temperatura? 3 ↵
2a temperatura? 4 ↵
3a temperatura? 1 ↵
4a temperatura? 2 ↵
5a temperatura? 7 ↵
6a temperatura? 3 ↵
7a temperatura? 1 ↵
Gráfico de variação da temperatura:
D: ***
S: ****
T: *
Q: **
S: *****
S: ***
S: *
```

Figura 9.5 - Tela de execução para um programa que resolve o Problema 9.4.

O programa que resolve o Problema 9.4 é apresentado na Listagem 9.4 e sua tela de execução é apresentada na Figura 9.5. Nesse programa, a iniciação de vetor é útil para armazenar a sequência de letras representando os dias da semana.



```

/* graf.c - exibe grafico de barras para variacao de temperatura */

#include <stdio.h>
#include <conio.h>

void preencher(float t[7]) {
    int i;

    for(i=0; i<7; i++) {
        printf("%da. temperatura? ", i+1);
        scanf("%f", &t[i]);
    }
}

float media(float t[7]) {
    int i;
    float s=0;

    for(i=0; i<7; i++) s += t[i];
    return s/7;
}

void barra(float t, int c) {
    int i;

    _textcolor(c);
    for(i=1; i<=t; i++)
        printf("*");
    _textcolor(7); // branco
}

void grafico(float t[7], float m) {
    char d[7] = {'D', 'S', 'T', 'Q', 'Q', 'S', 'S'};
    int i, c;

    printf("\nGrafico de variacao da temperatura:\n");

    for(i=0; i<7; i++) {
        printf("\n%c: ", d[i]);
        if( t[i]<m ) c = 1; // azul
        else if( t[i]>m ) c = 4; // vermelho
        else c = 2; // verde
        barra(t[i],c);
    }
    puts("\n");
}

int main(void) {
    float t[7], m;

    preencher(t);
    m = media(t);
    grafico(t,m);

    return 0;
}

```

Listagem 9.4 - Programa (para Pelles C) que resolve o Problema 9.4.



9.5 Exercícios

- 9.1** Crie uma função para preencher um vetor com dez números inteiros.
- 9.2** Crie uma função que recebe um vetor contendo dez números inteiros e devolve como resposta o valor do maior número armazenado no vetor.
- 9.3** Usando as funções definidas nos Exercícios 9.1 e 9.2, crie um programa que solicita ao usuário a digitação de dez números inteiros e, em seguida, informa qual é o maior entre os números fornecidos pelo usuário.
- 9.4** O programa a seguir *parece* ser capaz de adivinhar o pensamento do usuário. Teste esse programa e tente descobrir como ele funciona. No programa, a função `time()`, definida em `time.h`, é usada para determinar o número de segundos que se passaram desde a meia-noite do dia 1º de janeiro de 1970. O valor é passado à função `srand()` para que ela inicie o gerador de números aleatórios. Depois disso, cada chamada feita à função `rand()` resulta num número aleatório entre 0 e 1073741823. As funções `srand()` e `rand()` são definidas na biblioteca `stdlib.h`.

```
/* telepatia.c - adivinha o que o usuario esta pensando */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void) {
    int s[14] = {33,34,35,36,37,38,40,41,42,43,47,60,61,64};
    char r;
    int i;
    srand(time(NULL));
    r = s[rand()%14];
    puts("Pense em um numero de dois digitos (ex. 35)");
    puts("Subtraia deste numero a soma de seus digitos(ex. 35-8=27)");
    puts("Procure na tabela abaixo o simbolo a direita do resultado");
    puts("Concentre-se neste simbolo e pressione a tecla <enter>...\n");
    for(i=99; i>=0; i--) {
        printf("%02d", i);
        if( i%9==0 ) printf("%c ",r);
        else printf("%c ", s[rand()%14]);
        if( i%10==0 ) printf("\n");
    }
    getchar();
    printf("\nO simbolo em que voce se concentrou e %c\n\n", r);
    return 0;
}
```



Anotações



```

int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
}
  
```

Ordenação e Busca

10.1 Ordenação pelo método da bolha

O problema de **ordenação** consiste em organizar os itens de um vetor, inicialmente dispostos em ordem aleatória, de modo que eles fiquem em ordem crescente. Por exemplo, após a ordenação do vetor $v[5] = \{46, 50, 38, 19, 27\}$, devemos ter $v[5] = \{19, 27, 38, 46, 50\}$, conforme ilustrado na Figura 10.1.



Figura 10.1 - O processo de ordenação de vetores.

Há diversos métodos que podem ser usados para resolver este problema. Um dos métodos mais simples é conhecido como **método da bolha** (ou *bubble sort*). A estratégia básica desse método de ordenação consiste em comparar itens consecutivos do vetor e, caso estejam fora de ordem, permutá-los. Procedendo desta forma, após a primeira passagem completa pelo vetor, da esquerda para a direita, podemos garantir que o maior item terá sido deslocado para a última posição do vetor. Na Tabela 10.1, temos um exemplo que mostra como o maior item existente no vetor $v[5] = \{46, 50, 38, 19, 27\}$ é deslocado para sua última posição.

Tabela 10.1 - Comparações e trocas efetuadas na primeira passagem pelo vetor.

Comparação	$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$
1 ^a	46	50	38	19	27
2 ^a	46	50	38	19	27
3 ^a	46	38	50	19	27
4 ^a	46	38	19	50	27
-	46	38	19	27	50



Como podemos observar na Tabela 10.1, após a primeira passagem pelo vetor, restam apenas quatro itens a ordenar. De modo geral, a cada passagem pelo vetor, chamada **etapa de ordenação**, um item é deslocado para sua posição definitiva. Sendo assim, num vetor com n itens a serem ordenados, após a primeira etapa restarão apenas $n-1$ itens a serem ordenados; após a segunda, restarão apenas $n-2$ itens e assim sucessivamente, até que reste apenas 1 item (nesse ponto, concluímos que o vetor está completamente ordenado).

O processo que ordena completamente o vetor $v[5] = \{46, 50, 38, 19, 27\}$ é apresentado na Tabela 10.2. Nesse processo usamos a variável i para controlar o número de etapas de ordenação e a variável j para indicar a posição do primeiro item do par a ser comparado em cada passo de uma etapa. Note que, na última etapa, o vetor já está ordenado, mas uma última comparação precisa ser feita para que isso seja confirmado.

Tabela 10.2 - Processo de ordenação de um vetor usando o método da bolha.

i	j	v[0]	v[1]	v[2]	v[3]	v[4]
1	0	46	50	38	19	27
	1	46	50	38	19	27
	2	46	38	50	19	27
	3	46	38	19	50	27
		46	38	19	27	50
2	0	46	38	19	27	50
	1	38	46	19	27	50
	2	38	19	46	27	50
		38	19	27	46	50
3	0	38	19	27	46	50
	1	19	38	27	46	50
		19	27	38	46	50
4	0	19	27	38	46	50
		19	27	38	46	50

Analisando a Tabela 10.2, vemos que para ordenar um vetor com n itens são necessárias $n-1$ etapas de ordenação. Sendo assim, o contador i deve iniciar com o valor 1 e aumentar até o valor $n-1$. Além disso, em cada etapa i , o contador j deve iniciar com o valor 0 e aumentar até o valor $n-i-1$. Portanto, para codificar esse método de ordenação, precisamos usar dois comandos `for` e um comando `if` encadeados. Um programa para ordenação de vetores é apresentado na Listagem 10.1.



```

/* bolha.c - ordena um vetor usando o metodo da bolha */

#include <stdio.h>

void preenche(int v[], int n);
void ordena(int v[], int n);
void exibe(int v[], int n);

int main(void) {
    int v[5];

    preenche(v, 5);
    ordena(v, 5);
    exibe(v, 5);

    return 0;
}

void preenche(int v[], int n) {
    int i;

    for(i=0; i<n; i++) {
        printf("%do. item? ", i+1);
        scanf("%d", &v[i]);
    }
}

void ordena(int v[], int n) {
    int i, j;

    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if( v[j]>v[j+1] ) {
                int x = v[j];
                v[j] = v[j+1];
                v[j+1] = x;
            }
}

void exibe(int v[], int n) {
    int i;

    for(i=0; i<n; i++)
        printf("%d\n", v[i]);
}

```

Listagem 10.1 - Programa que ordena um vetor usando o método da bolha.

10.2 Busca linear

A **busca** é um processo que determina se um item x está armazenado num vetor v . O Problema 10.1 ilustra uma situação em que a operação de busca é necessária.



Problema 10.1

Num certo sistema, as senhas dos usuários são representadas pelo vetor $U[9] = \{28, 56, 81, 39, 77, 92, 45, 19, 63\}$. Faça um programa que leia uma senha e verifique se ela é válida ou não.

Há dois métodos de busca que podem ser usados para resolver este problema. O primeiro deles, denominado **busca linear**, não faz nenhuma suposição a respeito da ordem em que os itens estão dispostos no vetor; o segundo, denominado **busca binária**, supõe que os itens estão dispostos em ordem crescente.

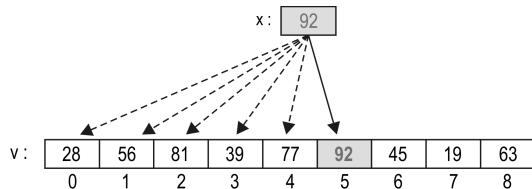


Figura 10.2 - Estratégia empregada pela busca linear para encontrar o item x no vetor v .

A **busca linear**, o método de busca mais simples que existe, consiste em examinar cada um dos itens do vetor, até que o item desejado seja encontrado ou, então, até que todos os itens do vetor tenham sido examinados, conforme a Figura 10.2.

Um programa que resolve o Problema 10.1, usando o método de busca linear, é apresentado na Listagem 10.2.

```
/* linear.c - usa busca linear para validar uma senha */
#include <stdio.h>

int pertence(int x, int v[], int n);

int main(void) {
    int U[9] = {28, 56, 81, 39, 77, 92, 45, 19, 63};
    int x;

    while( 1 ) {
        printf("Senha? ");
        scanf("%d", &x);
        if( pertence(x,U,9) ) break; // finaliza a execucao do while
        puts("Senha invalida!");
    }

    puts("Senha valida!");

    return 0;
}

int pertence(int x, int v[], int n) { // busca linear
    int i;

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

```



```

        if( x==v[i] )
            return 1;

        return 0;
    }

```

Listagem 10.2 - Programa que valida uma senha usando busca linear.

Embora a busca linear seja muito simples, quando o vetor é muito grande, ela se torna extremamente demorada. Neste caso, uma alternativa mais eficiente seria ordenar o vetor e usar o método de busca binária.

10.3 Busca binária

O método de **busca binária** é semelhante àquele empregado quando procuramos uma palavra num dicionário. Primeiramente examinamos uma página no meio do dicionário; se tivermos a sorte de encontrar a palavra desejada nessa página, a busca termina; senão, verificamos se a palavra procurada deveria ocorrer antes ou depois da página que examinamos e continuamos, usando a mesma estratégia, a procurá-la na primeira ou na segunda parte do dicionário.

Suponha que desejamos encontrar um item x num vetor ordenado v , contendo n itens, e que $v[m]$ seja um item aproximadamente no meio de v . Então há três possibilidades:

- $x == v[m]$: neste caso, o problema está resolvido.
- $x < v[m]$: neste caso, devemos continuar procurando na primeira metade de v .
- $x > v[m]$: neste caso, devemos continuar procurando na segunda metade de v .

Caso a busca deva continuar, procedemos exatamente da mesma maneira, ou seja, examinamos o item no meio da metade considerada e, se ele ainda não for aquele desejado, continuamos procurando-o na primeira ou na segunda metade dessa metade. Esse processo continua até que o item seja encontrado, como na Tabela 10.3, ou até que não existam mais itens a serem examinados, como na Tabela 10.4.

Tabela 10.3 - Funcionamento da busca binária quando o item x pertence ao vetor v .

Passo	x	i	f	m	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
1º	63	0	8	4	14	27	39	46	55	63	71	80	92
2º	63	5	8	6						63	71	80	92
3º	63	5	5	5						63			



Na Tabela 10.3, a variável x indica o item procurado e as variáveis i , f e m indicam, respectivamente, o *início*, o *fim* e o *meio* do intervalo de busca considerado em cada passo. No primeiro passo, temos $i=0$, $f=8$ e $m=4$, ou seja, o intervalo de busca é $0..8$ e o item a ser examinado é $v[4]$. Como $x>v[4]$, a busca deve continuar na segunda metade do intervalo $0..8$. Indicamos esse fato atribuindo à variável i o valor 5. No segundo passo, o intervalo é $5..8$ e o item a ser examinado é $v[6]$. Como $x<v[6]$, a busca deve continuar na primeira metade do intervalo $5..8$. Para isso, temos de atribuir o valor 5 à variável f . No terceiro passo, o intervalo é $5..5$, ou seja, resta um único item a ser examinado, porém como $x == v[5]$, a busca termina com sucesso.

Para entender o que acontece quando o item desejado não está no vetor, analise o exemplo da Tabela 10.4. Como a cada passo o vetor reduz-se aproximadamente à metade, é evidente que a busca binária sempre termina, ainda que o item desejado não esteja no vetor. Nesse caso, porém, ela só termina quando não há mais itens a examinar, ou seja, quando o intervalo de busca $i..f$ considerado fica *vazio*. Essa situação é detectada quando o índice i fica maior que o índice f .

Tabela 10.4 - Funcionamento da busca binária quando o item x não pertence ao vetor v .

Passo	x	i	f	m	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
1º	30	0	8	4	14	27	39	46	55	63	71	80	92
2º	30	0	3	1	14	27	39	46					
3º	30	2	3	2			39	46					
4º	30	2	1	?									

Ao contrário da busca linear, que funciona independentemente da ordem dos itens no vetor, a busca binária só funciona corretamente quando o vetor está ordenado, o que pode ser uma desvantagem. Porém, à medida que o tamanho do vetor aumenta, o número máximo de comparações feitas pela busca binária torna-se extremamente menor que aquele feito pela busca linear. Então, se o vetor é muito grande, e a busca é uma operação muito frequente na aplicação, esse aumento de eficiência na busca deve compensar o custo da ordenação do vetor.

Na Listagem 10.3 temos uma solução para o Problema 10.1 que emprega busca binária em vez de busca linear. Nesse programa, como os itens são conhecidos, eles já foram digitados em ordem crescente. Porém, poderíamos ter digitado os itens em qualquer ordem e usado a função `ordena()`, definida na Listagem 10.1, para ordená-los antes de realizar a busca binária.



```

/* binaria.c - usa busca binaria para validar uma senha */
#include <stdio.h>

int pertence(int x, int v[], int n);

int main(void) {
    int U[9] = {19, 28, 39, 45, 56, 63, 77, 81, 92};
    int x;

    while( 1 ) {
        printf("Senha? ");
        scanf("%d", &x);
        if( pertence(x,U,9) ) break;
        puts("Senha invalida!");
    }

    puts("Senha valida!");

    return 0;
}

int pertence(int x, int v[], int n) { // busca binaria
    int i=0, f=n-1, m;

    while( i<=f ) {
        m = (i+f)/2;
        if( x==v[m] ) return 1;
        if( x<v[m] ) f = m-1;
        else i = m+1;
    }

    return 0;
}

```

Listagem 10.3 - Programa que valida uma senha usando busca binária.

10.4 Exercícios

- 10.1** Digite e teste o programa da Listagem 10.1. O que acontece quando o vetor contém itens repetidos? Mesmo assim o programa funciona?
- 10.2** Altere o programa da Listagem 10.1 para que ele ordene o vetor de forma decrescente. Dica: *basta alterar a comparação que é feita na função ordena()*.
- 10.3** Um concurso público teve n candidatos. Dada a nota de cada um deles (suponha que as notas sejam todas distintas e que haja pelo menos três notas), apresente as três melhores notas em ordem decrescente.



- 10.4** Altere o programa da Listagem 10.3, considerando que há também um vetor A com cinco senhas de administradores de sistema. Dada uma senha, o programa deve verificar se ela é uma senha de usuário, de administrador ou inválida.
- 10.5** Refaça o programa da Listagem 10.3 para que as senhas válidas sejam cadastradas pelo usuário, numa ordem aleatória, e o próprio programa ordene essas senhas antes de realizar a busca binária.



```
int main(void) {
    char m[10][3];
    ler(m,10);
    ordenar(m,10);
}
```

Strings

11.1 Armazenamento

Em C, uma **string**, ou **cadeia**, é uma sequência de caracteres finalizada por um byte nulo, representado por '\0'. Note que o caractere '\0' (ASCII 0) é diferente do caractere '0' (ASCII 48). Assim, por exemplo, a execução do comando `puts("ABCDE")` causa a exibição do texto ABCODE, mas a execução de `puts("ABC\0DE")` apresenta apenas os caracteres ABC.

Strings constantes devem ser indicadas entre aspas e podem conter zero ou mais caracteres. Nas strings constantes, o byte nulo é adicionado automaticamente pelo compilador. Assim, por exemplo, a string "ABC" é armazenada na memória conforme indicado na Figura 11.1.

A	B	C	\0
0	1	2	3

Figura 11.1 - Representação interna da string constante "ABC".

Strings variáveis são, na verdade, vetores de caracteres. Assim, por exemplo, para armazenar uma string com até sete caracteres, precisamos criar um vetor com oito posições:

```
char s[8];
```

Note que esta posição adicional é necessária para que haja espaço suficiente para o armazenamento do byte nulo. Ademais, o espaço alocado para uma variável string não precisa ser completamente usado. Por exemplo, na Figura 11.2 podemos ver como a variável `s` se modifica quando alteramos a palavra que ela guarda. Inicialmente, a variável `s` guarda a palavra "MARIANA"; quando mudamos seu valor para "EVA", os valores das posições 4 a 7 são tratados como "lixo", já que agora a string termina na posição 3.

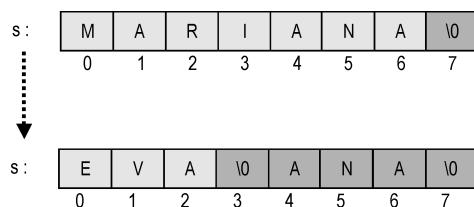


Figura 11.2 - Sucessivos estados da variável s na memória.



11.2 Leitura e exibição de strings

Para ler e exibir variáveis do tipo string, podemos usar `scanf()` e `printf()`, respectivamente, com o formato `%s`. O Problema 11.1, cuja solução é apresentada na Listagem 11.1, ilustra uma situação em que precisamos ler e exibir uma string. Note que, como o nome de um vetor representa seu endereço na memória, ele não deve ser prefixado com o operador `&`, ao ser passado como parâmetro a `scanf()`.

Problema 11.1

Pergunte o nome do usuário e diga-lhe olá.

```
/* oi.c - cumprimenta o usuario */

#include <stdio.h>

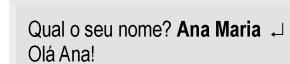
int main(void) {
    char n[31];

    printf("Qual o seu nome? ");
    scanf("%s", n); // nao devemos usar &
    printf("\nOlá %s!\n", n);

    return 0;
}
```

Listagem 11.1 - Programa que cumprimenta o usuário.

Um inconveniente deste programa é que ele só lê strings que não contenham espaço, pois quando um espaço é digitado, a função `scanf()` encerra imediatamente a leitura da string que estava sendo lida. Veja um exemplo na Figura 11.3.



Qual o seu nome? Ana Maria ↴
Olá Ana!

Figura 11.3 - Inconveniente existente com o programa que cumprimenta o usuário.

Assim, se precisarmos ler strings que podem ter espaços em branco, é preferível usar a função `gets()`, específica para esta finalidade, como vemos na Listagem 11.2.

```
/* oi.c - cumprimenta o usuario - versao 2 */

#include <stdio.h>

int main(void) {
    char n[31];
```



```
    printf("Qual o seu nome? ");
    gets( n );
    printf("\nOla %s!\n", n );
    return 0;
}
```

Listagem 11.2 - Programa que cumprimenta o usuário.

Para exibir apenas uma string, constante ou variável, podemos também usar a função `puts()`, em vez de `printf()`. As funções `gets()` e `puts()` fazem parte da biblioteca padrão `stdio.h`.

11.3 Funções para manipulação de strings

Usando a definição de string, podemos criar funções para a sua manipulação. Por exemplo, o Problema 11.2, cuja solução é apresentada na Listagem 11.3, ilustra um caso em que precisamos criar uma função para determinar o comprimento de uma string. Essa função, que denominamos `strlen()`, percorre a string `s`, contando seus caracteres, até encontrar o caractere '`\0`'. Ao encontrar esse caractere, ela devolve o valor do contador como resposta.

Problema 11.2

Dada uma mensagem com no máximo 80 caracteres, exiba-a centralizada na linha 13 do vídeo.

```
/* centraliza.c - centraliza uma mensagem no video */
#include <stdio.h>
#include <conio.h>

int strlen(char s[]) {
    int i=0;

    while( s[i]!='\0' ) i++;
    return i;
}

void centraliza(char m[], int y) {
    gotoxy((80-strlen(m))/2 + 1,y);
    puts(m);
}

int main(void) {
    char m[128];

    while( 1 ) {
        printf("Digite uma mensagem com ate 80 posicoes: ");

```



```

    gets( m );
    if( strlen(m) <= 80 ) break;
    puts("Mensagem muito longa!");
}

clrscr();
centraliza(m,13);
getchar();

return 0;
}

```

Listagem 11.3 - Programa (para Pelles C) que exibe uma mensagem centralizada no vídeo.

A lógica para centralizar a mensagem é simples. Considerando que uma linha do vídeo tem 80 colunas, podemos usar a expressão `(80-strlen(m))/2` para calcular quantos espaços devemos ter à esquerda de `m`, para que ela fique centralizada numa linha `y`. Então, somando 1 a esse valor, obtemos a coluna a partir da qual `m` deve ser exibida em `y`. Para posicionar o cursor no ponto da tela em que a mensagem deve aparecer, usamos a função `_gotoxy()`, da biblioteca `conio.h`.

O Problema 11.3, cuja solução está na Listagem 11.4, ilustra outro caso em que é útil podermos criar as próprias funções para manipular strings.

Problema 11.3

Um espião descobriu que seu inimigo codifica mensagens secretas substituindo as letras 'A' a 'Y' pelas respectivas letras seguintes e a letra 'Z' por 'A'. Crie um programa que o ajude a cifrar e decifrar mensagens usando esta ideia.

```

/* espiao.c - cifra e decifra mensagens secretas */

#include <stdio.h>
#include <ctype.h>

#define entre(v,min,max) ((v)>=(min) && (v)<=(max))

void cifra(char m[]) {
    int i;

    for(i=0; m[i]!='\0'; i++)
        if( entre(m[i], 'A', 'Y') ) m[i]++;
        else if( m[i]== 'Z' ) m[i]= 'A';
}

void decifra(char m[]) {
    int i;

    for(i=0; m[i]!='\0'; i++)
        if( entre(m[i], 'B', 'Z') ) m[i]--;
        else if( m[i]== 'A' ) m[i]= 'Z';
}

```



```

int main(void) {
    char m[128], r;
    int op;

    do {
        printf("\nMensagem em maiusculas? ");
        gets( m );
        printf("\n1 - Cifrar\n2 - Decifrar\n");
        scanf("%d%c", &op);
        switch( op ) {
            case 1: cifra(m); break;
            case 2: decifra(m); break;
        }
        printf("\nResultado: %s\n", m);
        printf("\nContinua? (s/n) ");
        scanf("%c%c", &r);
    } while( toupper(r) != 'N' );

    return 0;
}

```

Listagem 11.4 - Programa que cifra e decifra mensagens secretas.

No programa da Listagem 11.4, a instrução `m[i]++` substitui o caractere em `m[i]` pelo seu sucessor na tabela ASCII. Analogamente, `m[i]--` substitui o caractere em `m[i]` pelo seu predecessor. A macro `entre()` é usada para testar se o caractere `m[i]` está dentro do intervalo desejado, antes de convertê-lo em seu sucessor ou em seu predecessor. Note que, se uma opção inválida é escolhida pelo usuário, nenhuma conversão é feita no texto digitado por ele. Uma tela de execução para esse programa é apresentada na Figura 11.4.

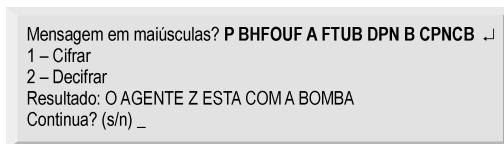


Figura 11.4 - Exemplo de tela de execução para o programa da Listagem 11.4.

Um problema existente com o programa da Listagem 11.4 é que ele só funciona corretamente quando o texto digitado pelo usuário é composto apenas por letras maiúsculas. Quando há letras minúsculas, essas letras não são substituídas. Então, para resolver o problema, podemos criar uma função para conversão de strings em maiúsculas e usá-la para converter o texto digitado, antes de cífrá-lo ou decífrá-lo.

Para converter uma string em maiúsculas, basta percorrer a string, aplicando a função `toupper()` a cada um de seus caracteres, como indicado na Listagem 11.5. Observe que, como o caractere '`\0`' tem código ASCII 0 (que representa falso em C), quando a variável `s[i]` se torna igual a '`\0`', a condição do comando `for` é avaliada como falsa, fazendo com que a repetição seja finalizada.



```

void strupr(char s[]) {
    int i;

    for(i=0; s[i]; i++)
        s[i] = toupper(s);
}

```

Listagem 11.5 - Função para converter uma string em maiúsculas.

11.4 A biblioteca string.h

Embora possamos criar nossas próprias funções para strings, sempre que possível, é preferível usar aquelas já disponíveis na biblioteca `string.h`. Há dezenas de funções nessa biblioteca e, de fato, a função `strlen()` apresentada na Listagem 11.3 é uma delas. Na Tabela 11.1 temos outras funções interessantes dessa biblioteca.

Tabela 11.1 - Principais funções da biblioteca `string.h`.

Função	Finalidade
<code>strlen(s)</code>	Determina o comprimento da string <code>s</code>
<code>strcat(a, b)</code>	Concatena as strings <code>a</code> e <code>b</code>
<code>strcpy(a, b)</code>	Copia o valor da string <code>b</code> para a string <code>a</code>
<code>strcmp(a, b)</code>	Compara as strings <code>a</code> e <code>b</code> , lexicograficamente, e devolve: <ul style="list-style-type: none"> ■ <i>nulo</i> se as strings <code>a</code> e <code>b</code> são iguais ■ <i>negativo</i> se a string <code>a</code> é menor que a string <code>b</code> ■ <i>positivo</i> se a string <code>a</code> é maior que a string <code>b</code>

A função `strcat(s1, s2)` serve para anexar a string `s2` ao final de `s1`. Ao usar essa função, certifique-se de que `s1` tenha espaço suficiente para armazenar a string resultante da concatenação. Na Listagem 11.6 temos um exemplo de uso dessa função.

```

/* anexa.c - anexa uma string ao final de outra */

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

int main(void) {
    char n[128], s[128];

    printf("Nome? ");
    gets(n);
    printf("Sobrenome? ");
    gets(s);

    strcat(n, " ");
    strcat(n, s);

```

```

    printf("Nome completo: %s\n", n);
    return 0;
}

```

Listagem 11.6 - Exemplo de uso da função de concatenação.

Para entender a necessidade da função `strcmp()`, analise o programa na Listagem 11.7. Intuitivamente, esperamos que ele exiba a palavra `iguais`, mas, na verdade, ele exibe a palavra `diferentes`. Isso acontece porque, em C, o nome de um vetor representa o seu endereço na memória, portanto, quando escrevemos `a==b`, estamos comparando o endereço do vetor `a` ao endereço do vetor `b` (e não os seus conteúdos). Evidentemente, como esses vetores são variáveis distintas, seus endereços também o são, o que justifica a saída exibida pelo programa.

```

/* comp.c - comparacao inadequada de strings */
#include <stdio.h>

int main(void) {
    char a[] = "um";
    char b[] = "um";

    if( a==b ) puts("iguais");
    else puts("diferentes");

    return 0;
}

```

Listagem 11.7 - Comparação inadequada de strings.

Assim, para que o programa da Listagem 11.7 funcione corretamente, em vez de `a==b`, devemos escrever `strcmp(a,b)==0`. De modo geral, se `a` e `b` são duas strings, em vez de compará-las diretamente usando um operador relacional, devemos comparar o resultado devolvido por `strcmp(a,b)` com 0. O operador usado nessa comparação deve ser o mesmo que seria usado diretamente com `a` e `b`, se isso fosse válido em C. Por exemplo, para verificar se `a` é maior que `b`, devemos escrever `strcmp(a,b)>0` e para verificar se `a` é diferente de `b`, devemos escrever `strcmp(a,b)!=0`.

Para entender melhor o funcionamento de `strcmp()`, analise o código na Listagem 11.8. Observe que essa função percorre as strings `a` e `b`, enquanto suas posições correspondentes têm o mesmo caractere e este é diferente de `'\0'`; quando o percurso termina, a função devolve como resposta a diferença entre os códigos ASCII dos últimos dois caracteres comparados (`a[i]-b[i]`). Então, se o percurso termina porque dois caracteres distintos foram encontrados em posições correspondentes de `a` e `b`, e o código de `a[i]` é menor que o código de `b[i]`, a função devolve um valor negativo; caso contrário, se `a[i]` é maior que o código de `b[i]`, a função devolve um valor positivo. Por outro lado, se o percurso termi-



na porque o caractere encontrado em ambas as strings é igual a '\0', então a diferença $a[i] - b[i]$ dá 0, o que indica que as strings comparadas são iguais.

```
int strcmp(char a[], char b[]) {
    int i = 0;

    while( a[i]==b[i] && a[i]!='\0' ) i++;
    return a[i]-b[i];
}
```

Listagem 11.8 - Função de comparação de strings.

O Problema 11.4, cuja solução é apresentada na Listagem 11.9, ilustra um caso em que a função de comparação de strings é útil.

Problema 11.4

Dadas duas palavras quaisquer, exiba-as em ordem alfabética.

```
/* alfabetica.c - exibe duas palavras em ordem alfabetica */

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

int main(void) {
    char n1[128], n2[128];

    printf("Primeira palavra? ");
    gets(n1);
    printf("Segunda palavra? ");
    gets(n2);

    printf("Ordem alfabetica:\n");

    if( strcmp(n1,n2)<0 ) printf("%s\n%s\n", n1, n2);
    else printf("%s\n%s\n", n2, n1);

    return 0;
}
```

Listagem 11.9 - Programa que exibe duas palavras em ordem alfabética.

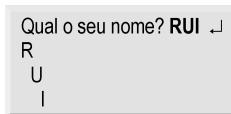
Vale ressaltar que a necessidade da função `strcpy(a,b)`, que copia a string `b` para `a`, é justificada do mesmo modo que no caso da função `strcmp()`. Observe que quando escrevemos `a=b`, o compilador C entende que queremos alterar o endereço do vetor `a`, e como isso não é permitido, ele indica um erro de compilação. Portanto, para que a atribuição seja feita corretamente, precisamos copiar os caracteres de `b`, um a um, para `a`. Esta é justamente a tarefa da função `strcpy()`.



11.5 Exercícios

11.1 O programa da Listagem 11.3 define a função `strlen()`, que está disponível em `string.h`. Execute-o para ver seu funcionamento e depois altere-o para que ele use a versão predefinida dessa função.

11.2 Crie uma função que receba uma string como parâmetro e exiba os caracteres dessa string em diagonal. Em seguida, crie também um programa principal para testar a sua função, conforme exemplificado a seguir:

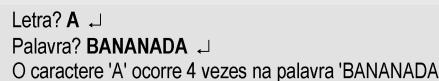


11.3 A função `strcmp()` diferencia maiúsculas de minúsculas. Assim, por exemplo, ela considera que "Ana" e "ANA" são strings distintas. Usando `tolower()`, que converte um caractere em minúscula, crie a função `strcmpi()`, que ignora a diferença entre maiúsculas e minúsculas. Baseie-se no código na Listagem 11.9.

11.4 Dados dois nomes, informe se eles são iguais ou, caso não sejam, exiba-os em ordem alfabética decrescente. Seu programa **não deve** diferenciar maiúsculas de minúsculas. Por exemplo, se forem digitadas as strings `ANA` e `Ana`, seu programa deve informar que os nomes são iguais.

11.5 Faça um programa que solicite uma senha ao usuário e, caso a senha fornecida seja a palavra `ABRACADABRA`, apresente a mensagem `ok`. Seu programa **deve** diferenciar maiúsculas de minúsculas.

11.6 Crie uma função que recebe um caractere `c` e uma string `s` como parâmetros e informa quantas vezes o caractere `c` ocorre na string `s`. Em seguida, crie um programa principal para testar a sua função, conforme exemplificado a seguir:



Anotações



```
int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
}
```

Matrizes

12.1 Armazenamento

Em C, uma **matriz** é um vetor cujos itens são também vetores. Uma matriz pode ter qualquer número de dimensões, mas as matrizes bidimensionais são mais usadas. Por exemplo, para criar uma matriz bidimensional de inteiros, com cinco linhas e quatro colunas, declaramos:

```
int m[5] [4];
```

Se for desejado iniciar essa matriz com valores específicos, podemos declarar:

```
int m[5] [4] = { {5, 9, 3, 0},
                 {0, 2, 7, -1},
                 {6, -3, 5, 0},
                 {3, 5, 4, 8},
                 {1, 2, -6, 9} };
```

Com esta declaração, criamos uma variável `m` conforme ilustrado na Figura 12.1.

<code>m</code>	0	1	2	3
0	5	9	3	0
1	0	2	7	-1
2	6	-3	5	0
3	3	5	4	8
4	1	2	-6	9

Figura 12.1 - Representação interna de uma matriz bidimensional 5x4.

12.2 Ordenação de strings

Como vimos no Capítulo 11, uma string é representada por um vetor de caracteres. Portanto, para armazenar uma coleção de strings, precisamos de um vetor cujos itens sejam vetores, isto é, uma matriz bidimensional. Neste caso, o número de strings a serem armazenadas define o total de linhas nessa matriz e o comprimento máximo das strings define o total de colunas.



Por exemplo, para representar a coleção de strings formada pelos itens "DENIS", "CAIO", "ANA", "EVA" e "BEATRIZ", podemos declarar:

```
char m[5][8] = {"DENIS", "CAIO", "ANA", "EVA", "BEATRIZ"};
```

Com esta declaração criamos uma variável conforme ilustrado na Figura 12.2.

m[1]	0	1	2	3	4	5	6	7
0	D	E	N	I	S	\0		
1	C	A	I	O	\0			
2	A	N	A	\0				
3	E	V	A	\0				
4	B	E	A	T	R	I	Z	\0

Figura 12.2 - Representação interna de um vetor contendo os nomes de cinco pessoas.

Apesar de termos declarado a matriz `m` com duas dimensões, quando escrevemos `m`, sem nenhuma dimensão, estamos nos referindo a toda a matriz; quando escrevemos `m[i]`, com apenas uma dimensão, estamos nos referindo à linha `i` da matriz e, finalmente, quando escrevemos `m[i][j]`, estamos nos referirmos ao item que se encontra na linha `i` e coluna `j` da matriz.

Como exemplo do uso de vetores de strings, vamos considerar o Problema 12.1.

Problema 12.1

Faça um programa para ler cinco nomes e exibi-los em ordem alfabética. Considere que cada nome pode ter no máximo 30 caracteres.

Para resolver este problema, vamos criar três funções:

- `ler()`, que lê `n` strings de até 30 caracteres cada e as armazena numa matriz.
- `ordenar()`, que ordena uma matriz contendo `n` strings de até 30 caracteres.
- `exibir()`, que exibe as `n` strings armazenadas numa matriz.

Destas três funções, a única que requer atenção especial é aquela de ordenação. Como C não permite que strings sejam diretamente atribuídas ou comparadas entre si, precisamos adaptar o algoritmo de ordenação visto na Seção 10.1, usando a função `strcpy()` para copiar uma string para outra e a função `strcmp()` para comparar duas strings.

Uma solução para o Problema 12.1 é apresentada na Listagem 12.1 e um exemplo de tela de execução é apresentado na Figura 12.3.



```

1º nome? Denis ↴
2º nome? Caio ↴
3º nome? Ana ↴
4º nome? Eva ↴
5º nome? Beatriz ↴
Ordem alfabetica
Ana
Beatriz
Caio
Denis
Eva

```

Figura 12.3 - Tela de execução para um programa que resolve o Problema 12.1.

```

/* alfa.c - ordena em ordem alfabetica */

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

void ler(char m[] [31], int n) {
    int i;

    for(i=0; i<n; i++) {
        printf("%do. nome? ", i+1);
        gets(m[i]);
    }
}

void ordenar(char m[] [31], int n) {
    int i, j;

    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if( strcmp(m[j],m[j+1])>0 ) {
                char x[31];
                strcpy(x,m[j]);
                strcpy(m[j],m[j+1]);
                strcpy(m[j+1],x);
            }
}

void exibir(char m[] [31], int n) {
    int i;

    for(i=0; i<n; i++)
        printf("%do. nome: %s\n", i+1, m[i]);
}

int main(void) {
    char m[5] [31];

    ler(m,5);
    ordenar(m,5);
    puts("\nOrdem alfabetica:\n");
    exibir(m,5);

    return 0;
}

```

Listagem 12.1 - Programa que exibe uma lista de nomes em ordem alfabética.



12.3 Jogo da velha

Nesta seção, vamos usar uma matriz bidimensional para criar um programa que joga o jogo da velha com o usuário. No início da execução, o programa pede que o usuário escolha par ou ímpar. Se o usuário quiser par, deve digitar 0; se quiser ímpar, deve digitar 1. Em seguida, um número é sorteado pelo computador e se esse número for igual ao escolhido pelo usuário, ele começa o jogo; caso contrário, o computador começa. A partir daí, as jogadas se alternam entre os adversários (usuário e computador) até que um deles vença ou que haja empate.

O tabuleiro do jogo é representado por uma matriz bidimensional de caracteres `t[3][3]` e, além de `main()`, o programa é composto por mais cinco funções:

- `limpar(t)`, que atribui espaços em branco a todas as posições de `t`.
- `mostrar(t)`, que mostra o estado do jogo no vídeo, conforme os valores que estão armazenados em `t`, como ilustrado na Figura 12.4.
- `usuário(t)`, que efetua a jogada do usuário; esta função pergunta ao usuário em que posição ele quer jogar e marca seu símbolo ('x') nessa posição.
- `computador(t)`, que efetua a jogada do computador; essa função sorteia uma posição aleatória de `t` e marca o símbolo do computador ('o') nessa posição.
- `vencedor(t)`, que verifica se há três símbolos iguais a 'x' ou a 'o' alinhados na mesma linha, coluna ou diagonal. Se houver, ela devolve o símbolo que está alinhado (indicando o vencedor); caso contrário, ela devolve um espaço em branco (indicando que até o momento não houve vencedor).

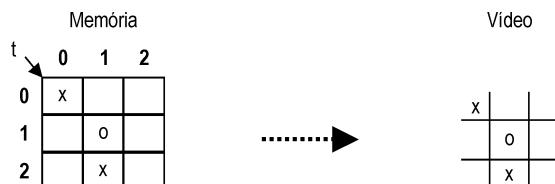


Figura 12.4 - Resultado da exibição da matriz `t[3][3]` no vídeo do computador.

Para gerar números aleatórios, usamos duas funções da biblioteca `stdlib.h`:

- `srand(s)`, que inicia o gerador de números aleatórios usando como semente o número inteiro `s`. Para que a sequência gerada seja distinta a cada nova execução do programa, usamos a função `time()`, da biblioteca `time.h`, para definir a hora do sistema como semente.
- `rand() % n`, que produz um número inteiro aleatório no intervalo de 0 a `n-1`.



O programa completo é apresentado na Listagem 12.2.

```
/* velha.c - jogo da velha */

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

void limpar(char t[3][3]);
void mostrar(char t[3][3]);
void usuario(char t[3][3]);
void computador(char t[3][3]);
char vencedor(char t[3][3]);

int main(void) {
    char t[3][3], v;
    int j=0, e, s;

    printf("par (0) ou impar (1)?");
    scanf("%d", &e);
    srand(time(NULL));
    s=rand()%2;
    limpar(t);

    do {
        mostrar(t);
        if(e==s) usuario(t);
        else computador(t);
        j++;
        s = !s;
        v=vencedor(t);
    } while(j<9 && v==' ');

    mostrar(t);

    switch(v) {
        case 'x': puts("\n\nVoce ganhou"); break;
        case 'o': puts("\n\nEu ganhei"); break;
        case ' ': puts("\n\nEmpatamos");
    }

    return 0;
}

void limpar(char t[3][3]) {
    int i, j;

    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            t[i][j]=' ';
}

void mostrar(char t[3][3]) {
    int i;

    clrscr();

    for(i=0;i<3;i++) {
        printf("\n %c | %c ", t[i][0], t[i][1], t[i][2]);
        if(i<2) printf("\n---+---+---");
    }
}
```

```

void usuario(char t[3][3]) {
    int L, C;

    do {
        printf("\n\nPosição?");
        scanf("%d,%d", &L, &C);
    } while(L<0 || L>2 || C<0 || C>2 || t[L][C]!=' ');

    t[L][C]='x';
}

void computador(char t[3][3]) {
    int L, C;

    do {
        L = rand()%3;
        C = rand()%3;
    } while(t[L][C]!=' ');

    t[L][C]='o';
}

char vencedor(char t[3][3]) {
    int i;

    for(i=0; i<3; i++) {
        if( t[i][0]==t[i][1] && t[i][1]==t[i][2] && t[i][0]!=' ')
            return t[i][0];
        if( t[0][i]==t[1][i] && t[1][i]==t[2][i] && t[0][i]!=' ')
            return t[0][i];
    }

    if( t[0][0]==t[1][1] && t[1][1]==t[2][2] && t[0][0]!=' ')
        return t[0][0];

    if( t[0][2]==t[1][1] && t[1][1]==t[2][0] && t[0][2]!=' ')
        return t[0][2];
    }
    return ' ';
}

```

Listagem 12.2 - Programa (para Pelles C) para jogar o jogo da velha com o usuário.

12.4 Exercícios

- 12.1** Teste o programa da Listagem 12.1. Em seguida, modifique-o de modo que ele leia dez nomes de até 30 posições e os exiba em ordem alfabética decrescente.
- 12.2** Teste o programa da Listagem 12.2.
- 12.3** Ao testar o programa do jogo da velha, você deve ter percebido que, como o computador escolhe sua posição de jogada aleatoriamente, é muito fácil ganhar dele. Para tornar o jogo mais desafiador, podemos fazer uma alteração que torna bastante difícil ganhar do computador. A alteração consiste em modificar a função `computador()` de modo que ela implemente a seguinte estratégia:



- 1º [ataque] Encontre duas letras 'o' alinhadas com um espaço em branco e preencha esse espaço em branco com uma terceira letra 'o'. Neste caso, o computador vence e o jogo está terminado.
- 2º [defesa] Se não for possível vencer o jogo, o computador deve evitar que o usuário o vença. Para isso é necessário encontrar duas letras 'x' alinhadas com um espaço em branco e preencher esse espaço em branco com uma letra 'o'. Neste caso, o computador não vence o jogo, mas tem a chance de vencê-lo na sua próxima jogada.
- 3º [aleatório] Se nem o computador nem o usuário estão em condições de vencer o jogo, então o computador pode jogar em qualquer posição livre.

Modifique o programa da Listagem 12.2 para funcionar desta maneira.

- 12.4** Usando uma matriz bidimensional, crie um programa para implementar um jogo de *campo minado*. Para preencher as posições da matriz, use números aleatórios de modo que 0 represente uma posição livre e 1 represente uma posição com bomba. O usuário vai escolhendo as posições (sem ver o que há nelas) e, caso ele acerte todas as posições livres, ele ganha o jogo. O usuário deve ter a chance de cometer até três erros, no máximo. Se ele cometer mais de três erros, ele perde o jogo. Durante o jogo, o programa deve apresentar no vídeo a matriz que representa o campo minado: posições livres já escolhidas devem ser marcadas com o símbolo '•' (ASCII 2), posições com bombas já escolhidas devem ser marcadas com o símbolo '☼' (ASCII 15) e posições ainda não escolhidas devem ser marcadas com o símbolo '?' (ASCII 63), conforme ilustrado a seguir:

Vídeo

?	?	?	●	?
?	●	?	?	☼
☼	?	?	●	?
?	?	●	?	?



Anotações



```
int main(void) {  
    char m[10][31];  
    ler(m,10);  
    ordenar(m,10);
```

Arquivos de Registros

13.1 Registros

Um **registro** é um tipo de variável capaz de armazenar uma coleção de dados, ou **campos**, que podem ser de tipos distintos. Antes de criarmos um registro, precisamos definir o seu tipo. Em C, registros são também denominados **estruturas**, por isso usamos a palavra reservada `struct` para definir novos tipos de registros.

Na Listagem 13.1 vemos como definir um tipo de registro para armazenar uma data composta de três campos do tipo `int`, denominados, `dia`, `mes` e `ano`.

```
typedef struct {  
    int dia;  
    int mes;  
    int ano;  
} data;           // nome do tipo de registro
```

Listagem 13.1 - Definição de um tipo de registro para armazenar uma data.

Uma vez definido um tipo de registro, ele pode ser usado para declarar variáveis, como exemplificado na Listagem 13.2.

```
data a;          // cria uma variável do tipo data  
data b = {26,11,2009}; // cria e inicia variável do tipo data
```

Listagem 13.2 - Declaração e iniciação de variáveis do tipo data.

Por exemplo, para ler uma data e armazená-la na variável `a`, podemos executar o comando:

```
scanf("%d/%d/%d", &a.dia, &a.mes, &a.ano);
```

Para exibir a data armazenada na variável `b`, podemos executar o comando:

```
printf("%02d/%02d/%d", b.dia, b.mes, b.ano);
```



Também podemos definir um tipo de registro contendo campos que são do tipo vetor ou registro. Na Listagem 13.3 temos um exemplo de especificação de registro em que isso acontece.

```
typedef struct {
    char nome[31]; // campo do tipo vetor
    float salario;
    data admissao; // campo do tipo registro
} func;
```

Listagem 13.3 - Definição de um tipo de registro para armazenar dados de um funcionário.

Note que a definição do tipo `func` só pode ser feita depois da definição do tipo `data`. Além disso, se `f` é uma variável do tipo `func`, podemos, por exemplo, executar o comando a seguir para exibir a data de admissão do funcionário:

```
printf("%02d/%02d/%d", f.admissao.dia,f.admissao.mes,f.admissao.ano);
```

Para exibir apenas a primeira letra do nome do funcionário, podemos executar o comando:

```
printf("%c", f.nome[0]);
```

Os registros são especialmente úteis quando precisamos manter dados armazenados em arquivos em disco, como veremos a seguir.

13.2 Arquivos de registros

Um **arquivo de registros** é uma coleção de registros mantida em disco. Conceitualmente, um arquivo é similar a um vetor, porém seu tamanho não precisa ser definido *a priori* e pode aumentar enquanto houver espaço disponível em disco.

A grande vantagem no uso de arquivos é que, como eles são mantidos em disco, os dados armazenados por eles não são perdidos quando o programa termina sua execução (diferentemente do que ocorre com vetores, que são mantidos na memória principal). A desvantagem é que o acesso a disco é muito mais demorado do que o acesso à memória principal e, consequentemente, o uso de arquivos torna a execução do programa mais lenta.

Para um acesso a disco mais eficiente e seguro, em geral, os sistemas operacionais usam uma área de memória principal, chamada **buffer**, para controlar a transferência de dados da memória para o disco e vice-versa, como indicado na Figura 13.1. Quando um programa envia dados a um arquivo, eles são temporariamente armazenados no buffer; então, quando o buffer fica cheio,



o sistema operacional automaticamente descarrega esses dados em disco. Por outro lado, quando um programa recebe dados de um arquivo, na verdade, ele lê os dados que estão no buffer e, sempre que o buffer fica vazio, o sistema operacional o preenche novamente com dados do arquivo. O uso do buffer permite diminuir o número de acessos a disco e, consequentemente, aumentar a velocidade de execução do programa. Além disso, como os programas não podem acessar o disco diretamente (apenas por meio do buffer), o sistema operacional garante um nível maior de proteção aos dados em disco.

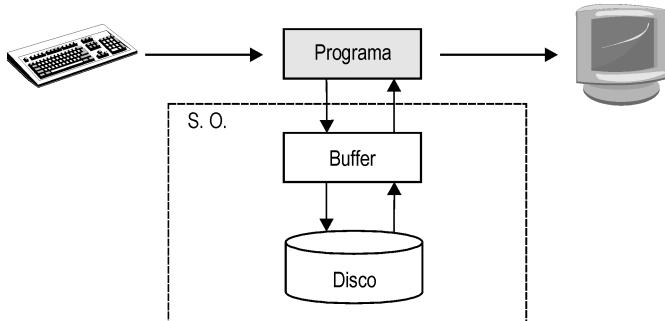


Figura 13.1 - Gravação e leitura de arquivos usando um buffer.

Para usar um arquivo, isto é, para ter acesso ao seu buffer, precisamos declarar uma variável do tipo `FILE *`, definido em `stdio.h`. Em seguida, devemos abrir o arquivo e associá-lo a essa variável. Uma vez que isso tenha sido feito, podemos gravar ou ler registros do arquivo e, ao final, devemos fechá-lo. A Tabela 13.1 apresenta os principais comandos para manipulação de arquivos de registros, supondo que `a` é uma variável do tipo `FILE *`.

Tabela 13.1 - Principais comandos para manipulação de arquivos de registros.

Comando	Finalidade
<code>a = fopen("n", "m");</code>	Abre o arquivo de nome " <code>n</code> ", no modo " <code>m</code> ", que pode ser: <ul style="list-style-type: none"> ▪ "wb": cria arquivo vazio para gravação (<i>write</i>) ▪ "ab": abre o arquivo para inclusão (<i>append</i>) ▪ "rb": abre o arquivo para leitura (<i>read</i>) Após a abertura, o arquivo fica sendo <i>apontado</i> pela variável de arquivo <code>a</code> . Caso o arquivo não possa ser aberto, a variável <code>a</code> fica com valor <code>NULL</code> .
<code>fwrite(&r, sizeof(reg), 1, a);</code>	Grava um registro <code>r</code> , do tipo <code>reg</code> , no arquivo apontado por <code>a</code> .
<code>fread(&r, sizeof(reg), 1, a);</code>	Lê um registro <code>r</code> , do tipo <code>reg</code> , do arquivo apontado por <code>a</code> .



Comando	Finalidade
<code>feof(a)</code>	Informa se o final do arquivo apontado por <code>a</code> foi detectado na última operação realizada, devolvendo um valor verdadeiro ou falso.
<code>fclose(a)</code>	Fecha o arquivo apontado pela variável <code>a</code> .
<code>remove ("nome")</code>	Exclui do disco o arquivo chamado "nome".
<code>rename ("velho", "novo")</code>	Renomeia o arquivo chamado "velho" como "novo".

13.3 Uma agenda eletrônica

Para entendermos como os comandos apresentados na Tabela 13.1 funcionam, vamos criar um programa para manter uma agenda eletrônica em um arquivo em disco e permitir que o usuário faça as seguintes operações com a agenda: *incluir* um novo registro, *listar* todos os registros já incluídos, *consultar* a agenda em busca de um registro particular e *excluir* um determinado registro da agenda.

Na Listagem 13.4a temos a primeira parte desse programa. Inicialmente, o programa apresenta um menu com as opções que o usuário pode escolher e, de acordo com a opção escolhida por ele, chama uma função específica para manipulação do arquivo - `incluir()`, `listar()`, `consultar()` ou `excluir()`. A função `exit()`, da biblioteca `stdlib.h`, é usada para interromper a execução do programa.

```
/* agenda.c - mantém uma agenda eletrônica num arquivo em disco */

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

typedef struct {
    char nome[31];
    char fone[21];
} pessoa;

int menu(void);
void incluir(void);
void listar(void);
void consultar(void);
void excluir(void);

int main(void) {
    while( 1 ) {
        switch( menu() ) {
            case 1: incluir();    break;
            case 2: listar();    break;
            case 3: consultar(); break;
            case 4: excluir();   break;
            case 5: exit(0);     // sai do programa
        }
    }
}
```



```

        }
    }

    return 0;
}

int menu(void) {
    int op;

    puts("\n1 - Incluir");
    puts("2 - Listar");
    puts("3 - Consultar");
    puts("4 - Excluir");
    puts("5 - Sair");
    printf("\nOpcão? ");
    scanf("%d%c", &op);

    return op;
}

```

Listagem 13.4a - Programa que mantém uma agenda eletrônica num arquivo em disco.

A função `incluir()`, na Listagem 13.4b, abre o arquivo para inclusão (modo "ab"), lê os dados a serem incluídos e os armazena num registro do tipo `pessoa` (que está na memória), transfere esses dados para o disco, usando a função `fwrite()`, e fecha o arquivo. Assim, a cada vez que essa função é chamada, um novo registro é adicionado ao final do arquivo. Note que a abertura para inclusão assume como posição *corrente* a primeira posição livre no final do arquivo e, fechando o arquivo, garantimos que o registro é efetivamente gravado em disco.

```

/* inclui um novo registro no final agenda */

void incluir(void) {
    FILE *s;
    pessoa p;

    s = fopen("agenda.dat", "ab");

    if( s==NULL ) {
        puts("erro fatal: o arquivo não pode ser aberto!");
        exit(1);
    }

    printf("\nNome? ");
    gets(p.nome);
    printf("Fone? ");
    gets(p.fone);
    fwrite(&p,sizeof(pessoa),1,s);
    printf("\nRegistro gravado!\n");
    fclose(s);
}

```

Listagem 13.4b - Função para inclusão de registros no arquivo.



A função `listar()`, apresentada na Listagem 13.4c, abre o arquivo para leitura (modo "rb") e enquanto o final do arquivo não é detectado pela função `feof()`, lê um registro do arquivo e exibe seus campos no vídeo. Ao final, o arquivo é fechado. A abertura para leitura assume como posição *corrente* a primeira posição existente no início do arquivo.

```
/* lista todos os registros da agenda */
void listar(void) {
    FILE *s;
    pessoa p;

    s = fopen("agenda.dat", "rb");

    if( s==NULL ) {
        puts("erro fatal: o arquivo nao pode ser aberto!");
        exit(2);
    }

    while( 1 ) {
        fread(&p,sizeof(pessoa),1,s);
        if( feof(s) ) break;
        printf("\n%s - %s",p.nome,p.fone);

    }
    printf("\n\n");
    fclose(s);
}
```

Listagem 13.4c - Função para listar os registros do arquivo.

A função `consultar()`, na Listagem 13.4d, é semelhante à função `listar()`, todavia ela exibe apenas os telefones correspondentes ao nome indicado pelo usuário, no momento da consulta.

A cada registro lido do arquivo, a função `consultar()` compara o nome digitado pelo usuário com aquele armazenado no registro lido e, para garantir que essa comparação ignore a diferença entre maiúsculas e minúsculas, ela usa a função `strcmpl()`.

```
/* consulta um nome na agenda */

int strcmpl(char a[], char b[]){
    int i = 0;

    while( toupper(a[i])==toupper(b[i]) && a[i]!='\0' )
        i++;

    return a[i]-b[i];
}

void consultar(void) {
    FILE *s;
    pessoa p;
    char n[31];
    int t=0;
```



```

s = fopen("agenda.dat", "rb");

if( s==NULL ) {
    puts("erro fatal: o arquivo nao pode ser aberto!");
    exit(3);
}

printf("\nNome? ");
gets(n);

while( 1 ) {
    fread(&p,sizeof(pessoa),1,s);
    if( feof(s) ) break;
    if( strcasecmp(n,p.nome)==0 ) {
        printf("\nFone: %s",p.fone);
        t++;
    }
}

printf("\nRegistro(s) encontrado(s): %d\n", t);
fclose(s);
}

```

Listagem 13.4d - Função para consultar os registros do arquivo.

Finalmente, a função excluir(), na Listagem 13.4e, exclui todos os registros contendo o nome indicado pelo usuário. Ela renomeia o arquivo atual com a extensão .bak (arquivo de *backup*) e faz uma cópia desse arquivo (com extensão .dat), eliminando os registros que contêm o nome indicado. Para ignorar a diferença entre maiúsculas e minúsculas a função excluir() usa a função strcasecmp(), já definida na Listagem 13.4d. Note que, a cada chamada da função excluir(), o arquivo corrente é transformado num arquivo de backup e o arquivo criado pode ter registros a menos. Então, caso um registro seja excluído por engano, podemos usar o arquivo de backup para restabelecer a agenda à sua configuração anterior.

```

/* exclui um registro da agenda */

void excluir(void) {
    FILE *e, *s;
    pessoa p;
    char n[31];
    int t = 0;

    remove("agenda.bak");
    rename("agenda.dat", "agenda.bak");

    e = fopen("agenda.bak", "rb");

    if( e==NULL ) {
        puts("erro fatal: o arquivo nao pode ser aberto!");
        exit(4);
    }

    s = fopen("agenda.dat", "wb");

    if( s==NULL ) {
        puts("erro fatal: o arquivo nao pode ser criado!");

```



```

        exit(5);
    }

printf("\nNome? ");
gets(n);

while( 1 ) {
    fread(&p,sizeof(pessoa),1,e);
    if( feof(e) ) break;
    if( strcmpi(n,p.nome)!=0 ) fwrite(&p,sizeof(pessoa),1,s);
    else t++;
}

printf("Registro(s) excluido(s): %d\n\n", t);
fclose(e);
fclose(s);
}

```

Listagem 13.4e - Função para exclusão de registros do arquivo.

13.4 Exercícios

- 13.1 Teste o programa da Listagem 13.4.
- 13.2 Altere o programa da Listagem 13.4 para que ele mantenha, além do nome e do telefone, a data de aniversário das pessoas (apenas dia e mês).
- 13.3 Com base no programa desenvolvido no exercício anterior, crie um programa para manter um cadastro de funcionários de uma empresa. Para cada funcionário armazene seu *nome*, seu *salário* e sua data de *admissão*. Adicione também uma opção para, dada uma faixa salarial, listar apenas os funcionários que recebem salário dentro dessa faixa.
- 13.4 Crie um programa para manter um cadastro de livros. Para cada livro armazene seu *título*, *autor*, *ano* de publicação e *preço*. Adicione também opções para listar todos os livros de um determinado autor, listar todos os livros dentro de uma faixa de preço, listar todos os livros publicados num determinado ano.



```
int main(void) {
    char m[10][31];
    ler(m,10);
    ordenar(m,10);
}
```

Tabela ASCII

ASCII é uma sigla para *American Standard Code for Information Interchange*. A tabela ASCII padrão apresenta 128 caracteres, numerados de 0 a 127, dos quais os 32 primeiros são códigos de controle (não têm representação gráfica). Por exemplo, a exibição do código de controle 7 soa o alarme (BEL), enquanto a exibição do código de controle 10 (LF) faz o cursor avançar para uma nova linha.

ASCII	Caractere	ASCII	Caractere	ASCII	Caractere	ASCII	Caractere
0	NUL	32	SP	64	@	96	'
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(72	H	104	h
9	TAB	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Há também tabelas ASCII estendidas (códigos de 128 a 255), que são dependentes do sistema operacional. A tabela estendida a seguir é usada em sistemas derivados do DOS, como é o caso do Windows.



Execute o programa a seguir para ver a tabela ASCII usada no seu computador.

```
/* ascii.c - exibe a tabela ASCII especifica do seu computador */

#include <stdio.h>

int main(void) {
    int c;

    for(c=0; c<=255; c++)
        printf("(%d,%c) ", c, c);

    return 0;
}
```



Anotações



O Compilador Pelles C

B.1 Instalação

O compilador *Pelles C for Windows* é um ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*) que pode ser obtido gratuitamente em www.smorgasbordet.com/pellessc.

Para instalar esse compilador, basta fazer o download do programa *setup.exe* e executá-lo em seu computador. Durante a instalação, uma pasta denominada *Pelles C Projects* será automaticamente criada dentro da pasta *Meus Documentos*. Posteriormente, todos os programas criados pelo *Pelles C* serão gravados nessa pasta.

Após a instalação, clique no botão *Iniciar* e selecione *Todos os programas* → *Pelles C for Windows* → *Pelles C IDE*. Você verá, então, a tela inicial da Figura B.1.



Figura B.1 - Tela inicial do compilador Pelles C for Windows.

B.2 Criação de programas

Para criar um programa usando o compilador *Pelles C for Windows*, você deve proceder da seguinte forma:

- **1º passo:** inicie a execução do programa *Pelles C IDE*.
- **2º passo:** no menu desse programa, selecione a opção *File* → *New* → *Project*.
- **3º passo:** na janela *New Projects*, selecione *Win32 Console program (EXE)*, digite o nome de projeto e clique no botão *OK*.
- **4º passo:** no menu, selecione a opção *File* → *New* → *Source code*.
- **5º passo:** digite o programa em linguagem C.
- **6º passo:** no menu, selecione a opção *File* → *Save as*, digite o nome de programa (que pode ser o mesmo dado ao projeto) e clique no botão *Salvar*.
- **7º passo:** clique no botão *Sim*, da caixa que apresenta a pergunta “*Do you want to add the file ‘...’ to the current project?*”.
- **8º passo:** no menu, clique no botão *Execute* (▷).

B.3 Sistema de ajuda

O sistema de ajuda do compilador pode ser acionado, a qualquer instante, pelo pressionamento da tecla F1. A tela inicial desse sistema é exibida na Figura B.2.

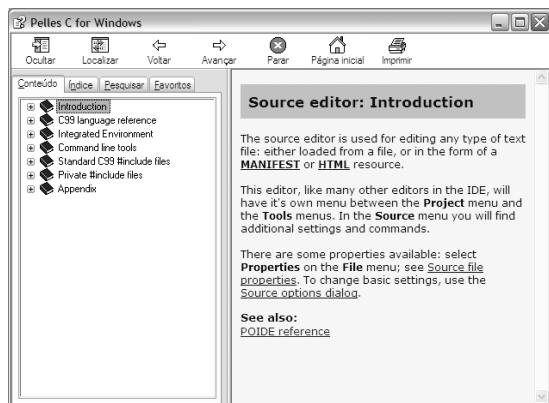


Figura B.2 - Tela inicial do sistema de ajuda do compilador Pelles C for Windows.

Você pode navegar nessa tela, expandindo os itens da lista que aparece na aba *Conteúdo* ou então pesquisando nas abas *Índice* e *Pesquisar*. Outra alternativa é marcar no editor de textos do sistema, Figura B.1, a palavra específica sobre a qual você deseja obter informações e pressionar a tecla F1.



B.4 Rastreamento da execução de um programa

Você também pode usar o sistema de Debug para rastrear a execução de um programa, executando-o passo a passo, e observando como suas variáveis são alteradas durante a execução.

Após ter digitado e salvado seu programa, selecione a opção *Project*, no menu principal, Figura B.1, e clique em *Project options*. Depois clique na aba *Compiler* e defina *Debug information* como *Full*. Em seguida, clique na aba *Linker* e defina *Debug Information* como *CodeView format*.

Para executar o programa no modo de rastreamento, clique no botão *Go/Debug* (▶) e depois no botão *Step into* (⇨). Na aba *Watch* (na parte inferior direita da janela do compilador), clique com o botão direito e escolha *Add*. Selecione as variáveis do programa cujas alterações você deseja acompanhar e clique no botão *OK*. A partir daí, a cada vez que a tecla F10 é pressionada, uma instrução do programa é executada e você pode ver na aba *Watch* como a execução dessa instrução afeta os valores das variáveis.

Para acompanhar a evolução da execução do programa e, ao mesmo tempo, ver a sua tela de execução, redimensione as janelas *Pelles C for Windows* e *Console Program output* de modo que ambas possam ser vistas simultaneamente. A cada entrada de dados, feita na janela *Console Program output*, não se esqueça de voltar para a janela *Pelles C for Windows* antes de executar a próxima instrução.

Se você desejar concluir o rastreamento, antes de chegar ao final da execução do programa, pressione a tecla F5.

Na Figura B.3 temos um exemplo da tela de rastreamento.

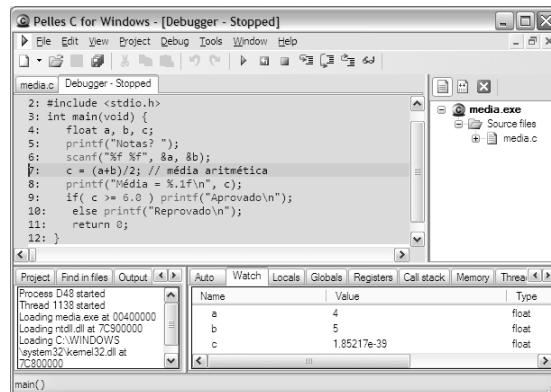


Figura B.3 - Tela do sistema de rastreamento da execução de um programa.



Anotações



```
int main(void) {  
    char m[10][31];  
    ler(m,10);  
    ordenar(m,10);
```

Adaptação para Unix/Linux

C.1 Necessidade de adaptação

O compilador *Pelles C* oferece algumas funções que, apesar de serem muito úteis, não fazem parte do padrão ISO C99 e não estão disponíveis no compilador *GCC*, usado em Unix/Linux. Assim, antes de compilar um programa que usa uma dessas funções, com *GCC*, precisamos fazer algumas adaptações em seu código-fonte.

Em *Pelles C*, todas as funções que não são definidas pelo padrão ISO têm nomes iniciando com um caractere de sublinha. Neste livro, procuramos minimizar o uso dessas funções, de modo a aumentar a portabilidade dos programas apresentados, porém, em alguns programas, o uso dessas funções foi imprescindível. Mais precisamente, usamos apenas as funções `_clrscr()`, `_gotoxy()`, `_kbhit()`, `_textcolor()` e `_textbackground()`, da biblioteca `conio.h`, e `_sleep()`, da biblioteca `time.h`. A seguir, mostramos as adaptações necessárias para que os programas que usam essas funções possam ser compilados e executados nos sistemas Unix e Linux.

C.2 As bibliotecas `time.h` e `unistd.h`

No compilador *Pelles C*, a função `_sleep()` é declarada em `time.h`. Por outro lado, em *GCC*, essa função é chamada `sleep()` e é declarada em `unistd.h`. Assim, para executar em Unix/Linux um programa que usa a função `_sleep()`, basta mudar o seu nome para `sleep()` e adicionar a diretiva `#include <unistd.h>`.

C.3 A biblioteca `conio.h`

As definições na Listagem C.1 implementam as adaptações necessárias para que os programas deste livro que usam a biblioteca `conio.h` possam ser compilados com *GCC* (que não oferece essa biblioteca).



```

/* conio.h - para Unix/Linux */

#define kbhit()          0
#define clrscr()         printf("%c[2J%c[1;1H", 27, 27)
#define gotoxy(x,y)     printf("%c[%d;%dH", 27, y, x)
#define textcolor(c)     printf("%c[%3dm", 27, c)
#define textbackground(c) printf("%c[%4dm", 27, c)
#define normalvideo()   printf("%c[0m", 27)

```

Listagem C.1 - Implementação de conio.h, para uso com o compilador GCC.

Destas adaptações, a única que não produz efeito equivalente ao observado em *Pelles C* é aquela para `_kbhit()`. Em *Pelles C*, essa função devolve 0 se nenhuma tecla foi pressionada e 1 em caso contrário. Por exemplo, em *Pelles C*, a execução de

```
while( !_kbhit() ) puts("teste");
```

termina assim que uma tecla qualquer é pressionada. Na adaptação proposta para Unix/Linux, uma chamada a `_kbhit()` sempre resulta em 0, portanto para finalizar a execução dessa repetição, será necessário pressionar a combinação de telas `CTRL-C`. De qualquer forma, isso permite que os programas deste livro que usam `_kbhit()` sejam compilados corretamente com o compilador *GCC*.

As demais adaptações propostas são baseadas no fato de que os sistemas Unix e Linux usam terminais compatíveis com o padrão VT100, que são capazes de executar comandos representados por **sequências de escape**, isto é, sequências cujos dois primeiros caracteres são `ESC` e '`[`' e cujos demais caracteres representam o código de uma ação.

Quando o terminal recebe uma sequência de escape, primeiramente ele verifica seu código. Então, caso esse código seja reconhecido, em vez de exibir a sequência, ele executa a ação correspondente. A listagem completa de códigos válidos é apresentada no manual do VT100, disponível no endereço <http://vt100.net/docs/vt100-ug>. Por exemplo, de acordo com esse manual, para limpar a tela do terminal, basta enviar-lhe a sequência `ESC`, '`[`', '`2`' e '`J`'. Em C, isso pode ser feito com o comando:

```
printf("%c[2J", 27);
```

Note que, como o caractere `ESC` não tem representação gráfica, para enviá-lo ao terminal, devemos enviar o seu código ASCII (27) no formato de caractere (%c).

Para mover o cursor para uma posição específica da tela, basta enviar ao terminal uma sequência da forma `ESC[<lin>;<col>H`. Então, se quisermos limpar a tela e mover o cursor para o seu canto esquerdo superior, basta executar o comando

```
printf("%c[2J%c[1;1H", 27, 27);
```



Para selecionar a cor em que o texto é exibido na tela, precisamos enviar ao terminal uma sequência da forma `ESC[3<cor>m` e para selecionar a cor do fundo sobre o qual o texto é exibido, precisamos enviar uma sequência da forma `ESC[4<cor>m`. Os valores permitidos para `<cor>` são apresentados na Tabela C.1.

Tabela C.1 - Representação de cores no terminal VT100.

Cor	preto	vermelho	verde	amarelo	azul	rosa	ciano	branco
Número	0	1	2	3	4	5	6	7

Para usar as definições do arquivo `conio.h`, Listagem C.1, com o compilador *GCC*, basta colocar esse arquivo no mesmo subdiretório em que se encontra o código-fonte do programa e trocar a diretiva `#include <conio.h>` por `#include "conio.h"`.

C.4 Programas adaptados para Unix/Linux

Apenas oito programas deste livro precisam de modificações para serem compilados com o *GCC*. Para os programas das Listagens 7.3, 11.3 e 12.2 a única modificação necessária consiste em substituir `<conio.h>` por `"conio.h"`. Para os programas das Figuras 5.3, 5.4, 6.4, 8.5 e 9.4 são necessárias outras pequenas alterações, conforme indicado nas Listagens C.2, C.3, C.4, C.5 e C.6 respectivamente.

```
/* quadrado.c - desenha um quadrado */

#include <stdio.h>
#include "conio.h"

int main(void) {
    int N, L, C;

    printf("Número? ");
    scanf("%d", &N);

    textbackground(6);

    for(L=1; L<=N; L++) {
        for(C=1; C<=N; C++)
            printf(" ");
        printf("\n");
    }

    normalvideo();

    return 0;
}
```

Listagem C.2 - Versão para Unix/Linux do programa da Figura 5.3.



```

/* xadrez.c - desenha um tabuleiro de xadrez */

#include <stdio.h>
#include "conio.h"

int main(void) {
    int N, L, C;

    printf("Número? ");
    scanf("%d", &N);

    for(L=1; L<=N; L++) {
        for(C=1; C<=N; C++) {
            textbackground((L+C)%2+1);
            printf(" ");
        }
        printf("\n");
    }

    normalvideo();

    return 0;
}

```

Listagem C.3 - Versão para Unix/Linux do programa da Figura 5.4.

```

/* ping_pong.c - simula movimento de pingue-pongue */

#include <stdio.h>
#include "conio.h"

int main(void) {
    int x=1, y=1, dx=1, dy=1, i;

    clrscr();

    while( ! kbhit() ) {
        gotoxy(x,y);
        printf("O\n");

        for(i=0; i<40000000; i++); // causa uma pausa

        gotoxy(x,y);
        printf(" \n");

        x += dx;
        y += dy;

        if( x==1 || x==80) dx = -dx;    // inverte movimento horizontal
        if( y==1 || y==25) dy = -dy;    // inverte movimento vertical
    }

    return 0;
}

```

Listagem C.4 - Versão para Unix/Linux do programa da Figura 6.4.



```

/* retangulo.c - exibe um retangulo vermelho no video */

#include <stdio.h>
#include "conio.h"

void exibe(int c, int x, int y) {
    gotoxy(x,y);
    putchar(c);
}

void retangulo(int x1, int y1, int x2, int y2, int c) {
    int x, y;

    textbackground(c);

    for(x=x1; x<=x2; x++) {
        exibe(32,x,y1);
        exibe(32,x,y2);
    }

    for(y=y1; y<=y2; y++) {
        exibe(32,x1,y);
        exibe(32,x2,y);
    }

    normalvideo();
}

int main(void) {
    clrscr();
    retangulo(10,5,70,20,1); // 1 = vermelho
    gotoxy(1,25);

    return 0;
}

```

Listagem C.5 - Versão para Unix/Linux do programa da Figura 8.5.

```

/* graf.c - exibe grafico de barras para variacao de temperatura */

#include <stdio.h>
#include "conio.h"

void preencher(float t[7]) {
    int i;

    for(i=0; i<7; i++) {
        printf("%da. temperatura? ", i+1);
        scanf("%f", &t[i]);
    }
}

float media(float t[7]) {
    int i;
    float s=0;

    for(i=0; i<7; i++)
        s += t[i];
}

```



```

        return s/7;
    }

void barra(float t, int c) {
    int i;

    textcolor(c);

    for(i=1; i<=t; i++)
        printf("*");

    normalvideo();
}

void grafico(float t[7], float m) {
    char d[7] = {'D', 'S', 'T', 'Q', 'Q', 'S', 'S'};
    int i, c;

    printf("\nGrafico de variacao da temperatura:\n");

    for(i=0; i<7; i++) {
        printf("\n%c: ", d[i]);
        if( t[i]<m ) c = 4;      // azul
        else if( t[i]>m ) c = 1; // vermelho
        else c = 2;              // verde
        barra(t[i],c);
    }
    puts("\n");
}

int main(void) {
    float t[7], m;

    preencher(t);
    m = media(t);
    grafico(t,m);

    return 0;
}

```

Listagem C.6 - Versão para Unix/Linux do programa da Figura 9.4.

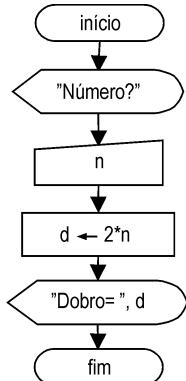


```
int main(void) {
    char m[10][3];
    ler(m,10);
    ordenar(m,10);
```

Solução dos Exercícios

Capítulo 1

1.1



```
/* ex-1-1.c */
#include <stdio.h>
int main(void) {
    float n, d;

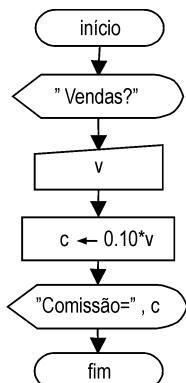
    printf("Número? ");
    scanf("%f", &n);

    d = 2*n;

    printf("Dobro = %.1f\n", d);

    return 0;
}
```

1.2



```
/* ex-1-2.c */
#include <stdio.h>
int main(void) {
    float v, c;

    printf("Vendas? ");
    scanf("%f", &v);

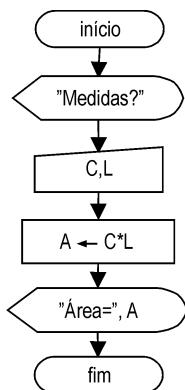
    c = 0.10*v;

    printf("Comissão = R$ %.2f\n", c);

    return 0;
}
```



1.3



```
/* ex-1-3.c */
#include <stdio.h>

int main(void) {
    float C, L, A;

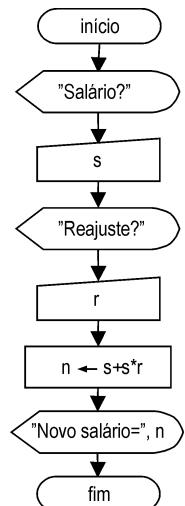
    printf("Medidas? ");
    scanf("%f %f", &L, &C);

    A = C*L;

    printf("Área = %.1f\n", A);

    return 0;
}
```

1.4



```
/* ex-1-4.c */
#include <stdio.h>

int main(void) {
    float s, r, n;

    printf("Salário? ");
    scanf("%f", &s);
    printf("Reajuste? ");
    scanf("%f", &r);

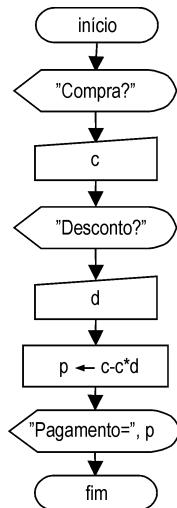
    n = s + s*r;

    printf("Novo salário = %.2f\n", n);

    return 0;
}
```



1.5



```
/* ex-1-5.c */
#include <stdio.h>

int main(void) {
    float c, d, p;

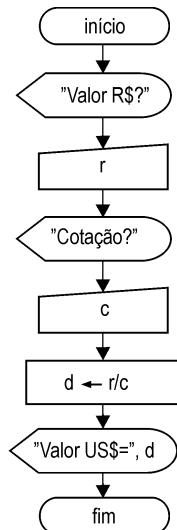
    printf("Compra? ");
    scanf("%f", &c);
    printf("Desconto? ");
    scanf("%f", &d);

    p = c - c*d;

    printf("Pagamento = %.2f\n", p);

    return 0;
}
```

1.6



```
/* ex-1-6.c */
#include <stdio.h>

int main(void) {
    float r, c, d;

    printf("Valor R$? ");
    scanf("%f", &r);
    printf("Cotacao? ");
    scanf("%f", &c);

    d = r/c;

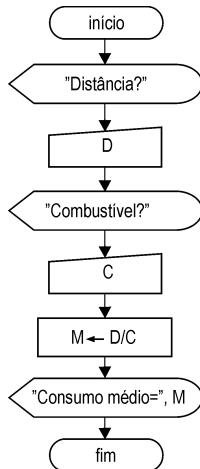
    printf("Valor US$ = %.2f\n", d);

    return 0;
}
```



Capítulo 2

2.1



```
/* ex-2-1.c */
#include <stdio.h>

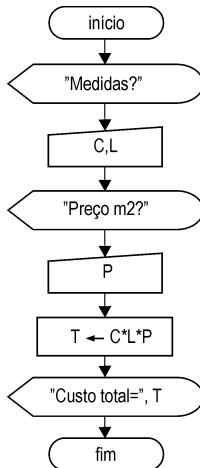
int main(void) {
    float D, C, M;

    printf("Distancia? ");
    scanf("%f", &D);
    printf("Combustivel? ");
    scanf("%f", &C);

    M = D/C;

    printf("Consumo medio = %.1f Km/l\n", M);
    return 0;
}
```

2.2



```
/* ex-2-2.c */
#include <stdio.h>

int main(void) {
    float C, L, P, T;

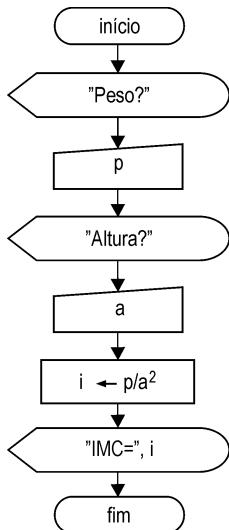
    printf("Medidas? ");
    scanf("%f %f", &L, &C);
    printf("Preco m2? ");
    scanf("%f", &P);

    T = C*L*P;

    printf("Custo total = R$ %.2f\n", T);
    return 0;
}
```



2.3



```
/* ex-2-3.c */
#include <stdio.h>
#include <math.h>

int main(void) {
    float p, a, i;

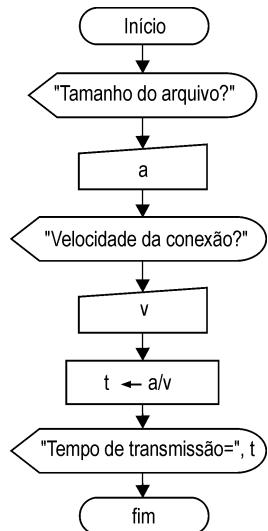
    printf("Peso? ");
    scanf("%f", &p);
    printf("Altura? ");
    scanf("%f", &a);

    i = p/pow(a,2);

    printf("IMC = %.1f\n", i);

    return 0;
}
```

2.4



```
/* ex-2-4.c */
#include <stdio.h>

int main(void) {
    int a;
    float v, t;

    printf("Tamanho do arquivo? ");
    scanf("%d", &a);
    printf("Velocidade da conexao? ");
    scanf("%f", &v);

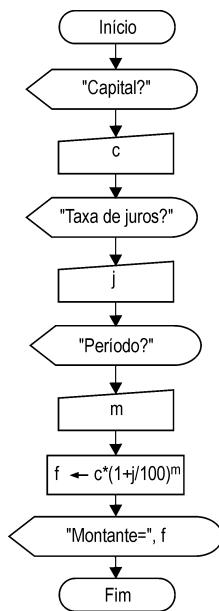
    t = a/v;

    printf("Tempo de transmissao = %.1fs\n", t);

    return 0;
}
```



2.5



```

/* ex-2-5.c */
#include <stdio.h>
#include <math.h>

int main(void) {
    float c, j, f;
    int m;

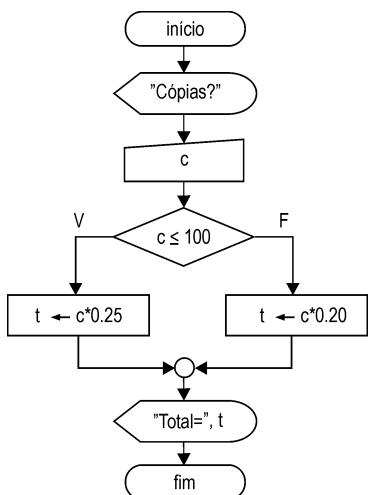
    printf("Capital? ");
    scanf("%f", &c);
    printf("Taxa de juros? ");
    scanf("%f", &j);
    printf("Período? ");
    scanf("%d", &m);

    f = c * pow(1+j/100, m);

    printf("Montante = R$ %.2f\n", f);
    return 0;
}
  
```

Capítulo 3

3.1



```

/* ex-3-1.c */
#include <stdio.h>

int main(void) {
    int c;
    float t;

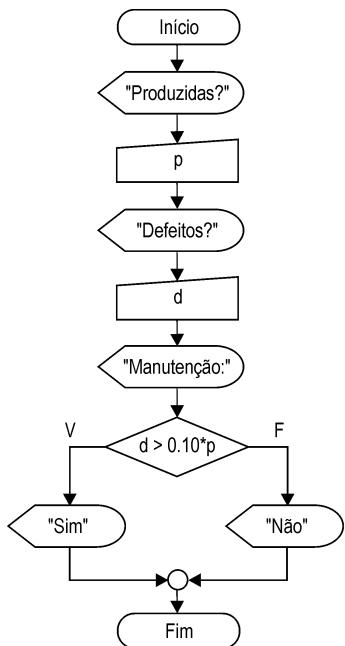
    printf("Copias? ");
    scanf("%d", &c);

    if( c<=100 ) t = c*0.25;
    else t = c*0.20;

    printf("Total = R$ %.2f\n", t);
    return 0;
}
  
```



3.2



```

/* ex-3-2.c */
#include <stdio.h>

int main(void) {
    int p, d;

    printf("Pecas produzidas? ");
    scanf("%d", &p);
    printf("Pecas defeituosas? ");
    scanf("%d", &d);

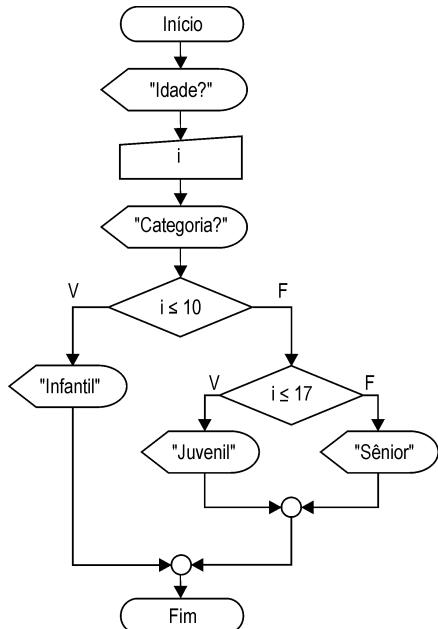
    printf("Manutencao: ");

    if( d>0.10*p ) printf("sim\n");
    else printf("nao\n");

    return 0;
}

```

3.3



```

/* ex-3-3.c */
#include <stdio.h>

int main(void) {
    int i;

    printf("Idade? ");
    scanf("%d", &i);

    printf("Categoria: ");

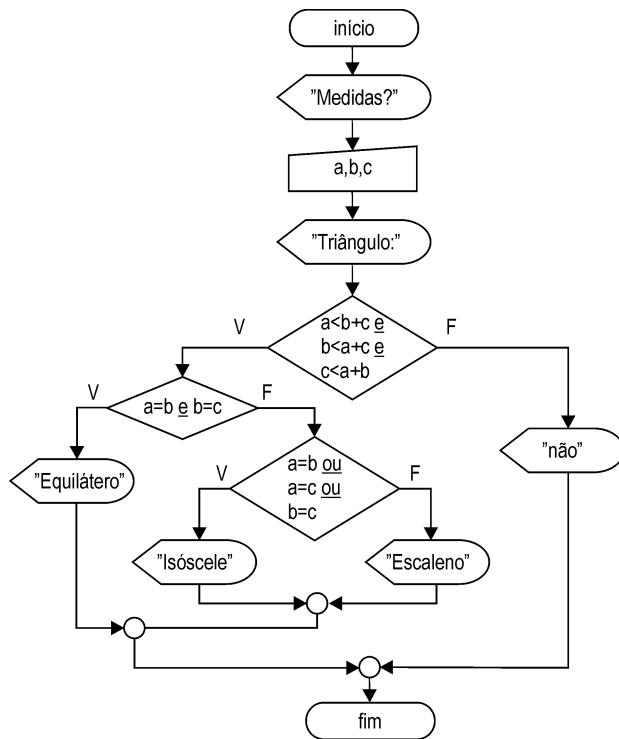
    if( i<=10 )
        printf("infantil\n");
    else
        if( i<=17 )
            printf("juvenil\n");
        else
            printf("senior\n");

    return 0;
}

```



3.4



```

/*
 * ex-3-4.c */
#include <stdio.h>

int main(void) {
    float a, b, c;

    printf("Medidas? ");
    scanf("%f %f %f", &a, &b, &c);

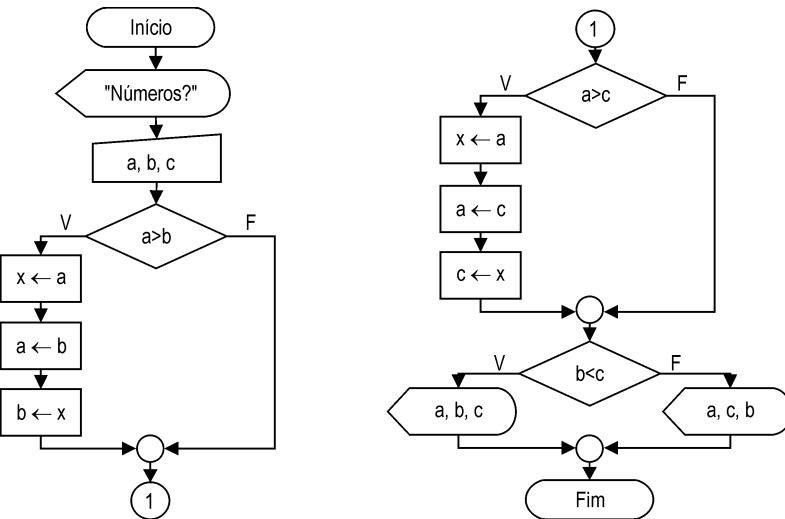
    printf("Triangulo: ");

    if( a<b+c && b<a+c && c<a+b )
        if( a==b && b==c ) printf("equilatero\n");
        else if( a==b || a==c || b==c ) printf("isoscele\n");
        else printf("escaleno\n");
    else printf("nao\n");

    return 0;
}
    
```



3.5



```
/*
 * ex-3-5.c */
#include <stdio.h>

int main(void) {
    float a, b, c, x;

    printf("Numeros? ");
    scanf("%f %f %f", &a, &b, &c);

    if( a>b ) {
        x = a;
        a = b;
        b = x;
    }

    if( a>c ) {
        x = a;
        a = c;
        c = x;
    }

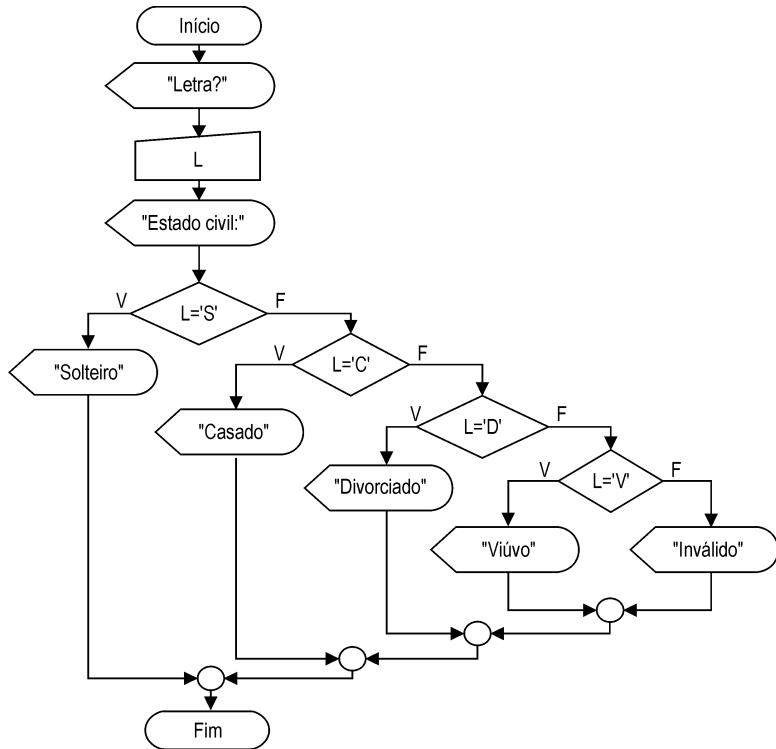
    if( b<c ) printf("Ordem: %.1f, %.1f, %.1f\n",a,b,c);
    else printf("Ordem: %.1f, %.1f, %.1f\n",a,c,b);

    return 0;
}
```



Capítulo 4

4.1



```
/* ex-4-1.c */
#include <stdio.h>
#include <ctype.h>

int main(void) {
    char L;

    printf("Letra? ");
    scanf("%c", &L);

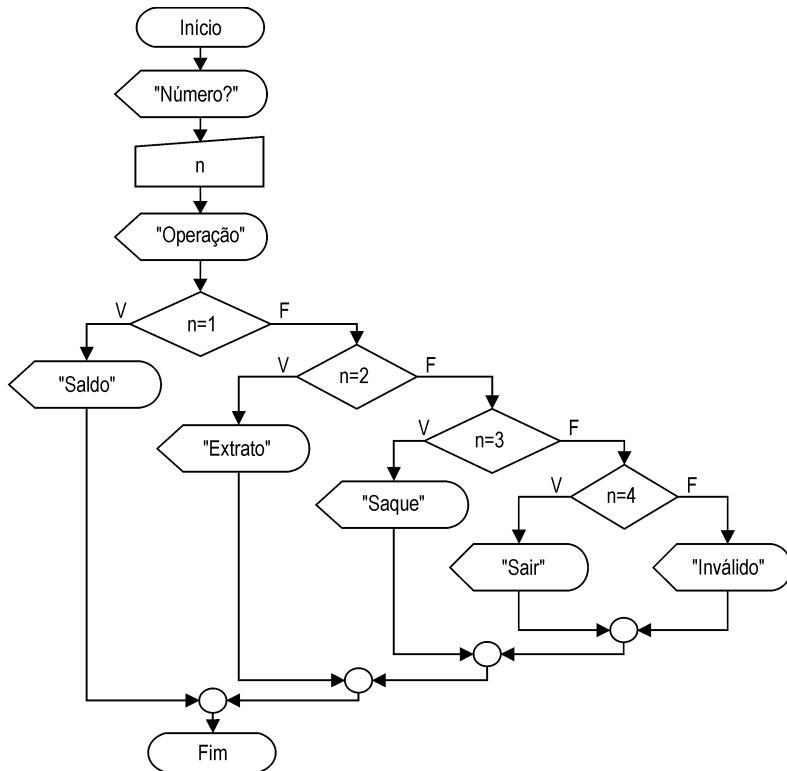
    printf("Estado civil: ");

    switch( toupper(L) ) {
        case 'S': printf("solteiro\n"); break;
        case 'C': printf("casado\n"); break;
        case 'D': printf("divorciado\n"); break;
        case 'V': printf("viuvo\n"); break;
        default : printf("invalido\n");
    }

    return 0;
}
```



4.2



```

/* ex-4-2.c */
#include <stdio.h>

int main(void) {
    int n;

    printf("Número? ");
    scanf("%d",&n);

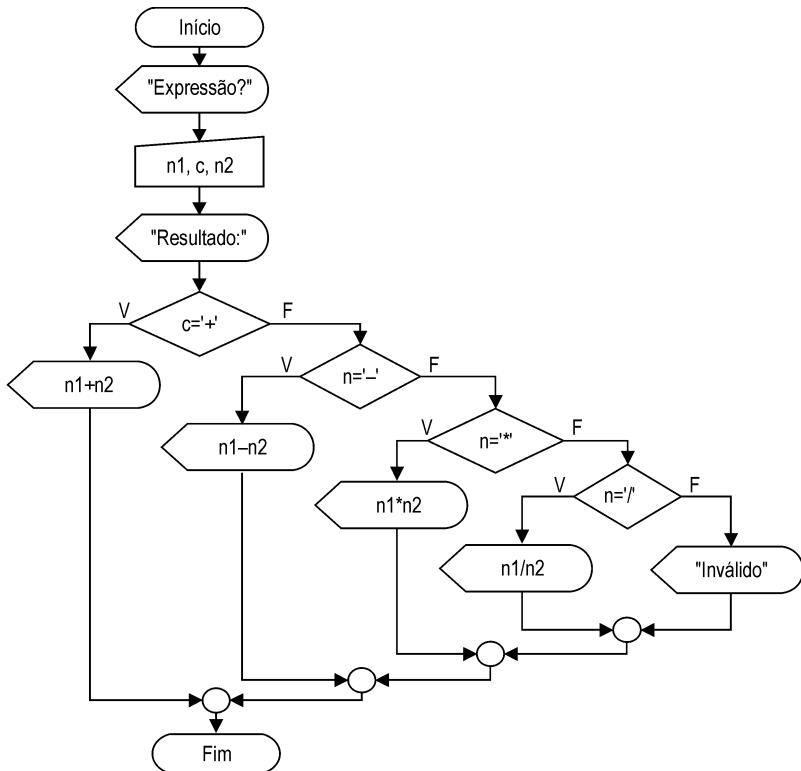
    printf("Operação: ");

    switch( n ) {
        case 1: printf("saldo\n"); break;
        case 2: printf("extrato\n"); break;
        case 3: printf("saque\n"); break;
        case 4: printf("sair\n"); break;
        default: printf("inválido\n");
    }

    return 0;
}
  
```



4.3



```

/*
 * ex-4-3.c *
 */

#include <stdio.h>

int main(void) {
    float n1, n2;
    char c;

    printf("Expressão? ");
    scanf("%f %c %f", &n1, &c, &n2);

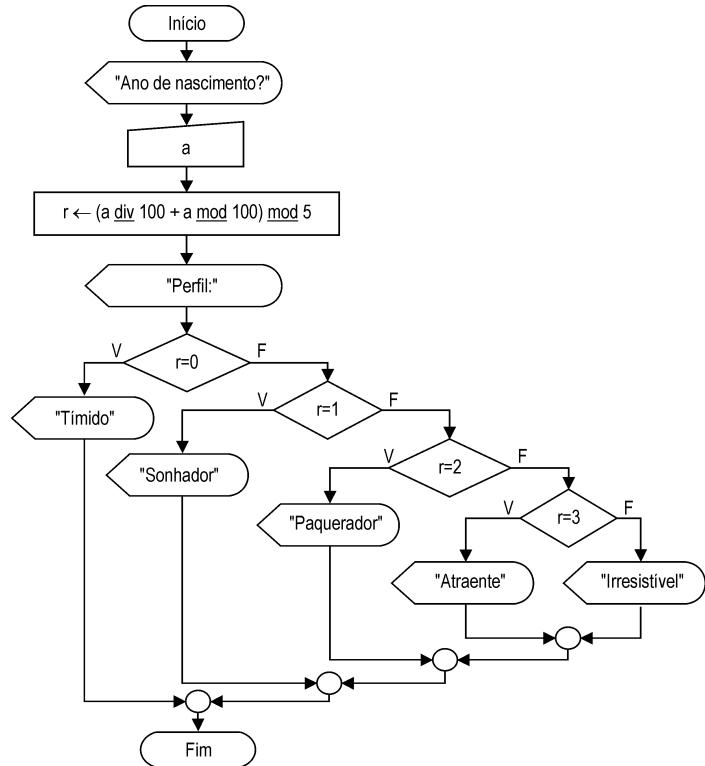
    printf("Resultado: ");

    switch( c ) {
        case '+': printf("%.1f\n", n1+n2); break;
        case '-': printf("%.1f\n", n1-n2); break;
        case '*': printf("%.1f\n", n1*n2); break;
        case '/': printf("%.1f\n", n1/n2); break;
        default : printf("invalido");
    }

    return 0;
}
  
```



4.4



```

/* ex-4-4.c */
#include <stdio.h>

int main(void) {
    int a, r;

    printf("Ano de nascimento? ");
    scanf("%d", &a);

    r = (a/100 + a%100) % 5;

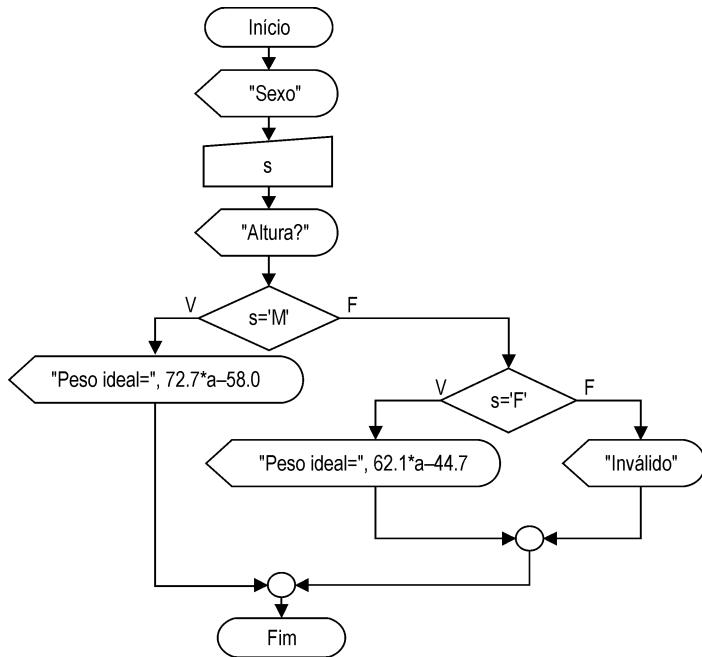
    printf("Perfil: ");

    switch( r ) {
        case 0: printf("timido\n"); break;
        case 1: printf("sonhador\n"); break;
        case 2: printf("paquerador\n"); break;
        case 3: printf("atraente\n"); break;
        case 4: printf("irresistivel\n"); break;
    }

    return 0;
}
  
```



4.5



```

/*
 * ex-4-5.c *
 */

#include <stdio.h>
#include <ctype.h>

int main(void) {
    char s;
    float a;

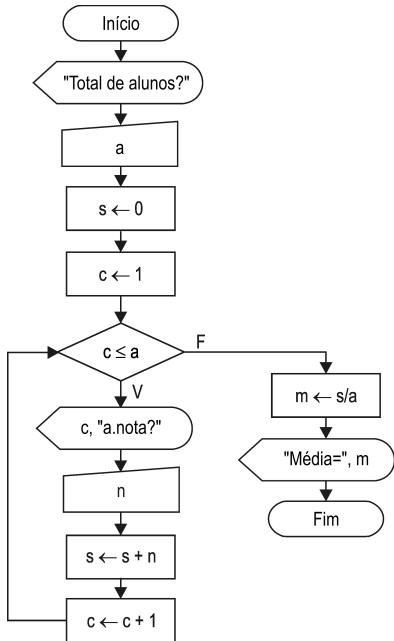
    printf("Sexo? ");
    scanf("%c",&s);
    printf("Altura? ");
    scanf("%f",&a);

    switch( toupper(s) ) {
        case 'M': printf("Peso ideal = %.1f\n",72.7*a-58.0); break;
        case 'F': printf("Peso ideal = %.1f\n",62.1*a-44.7); break;
        default : printf("Invalido\n");
    }

    return 0;
}
  
```

Capítulo 5

5.1



```

/* ex-5-1.c */
#include <stdio.h>

int main(void) {
    int a, c;
    float s, n, m;

    printf("Total de alunos? ");
    scanf("%d", &a);

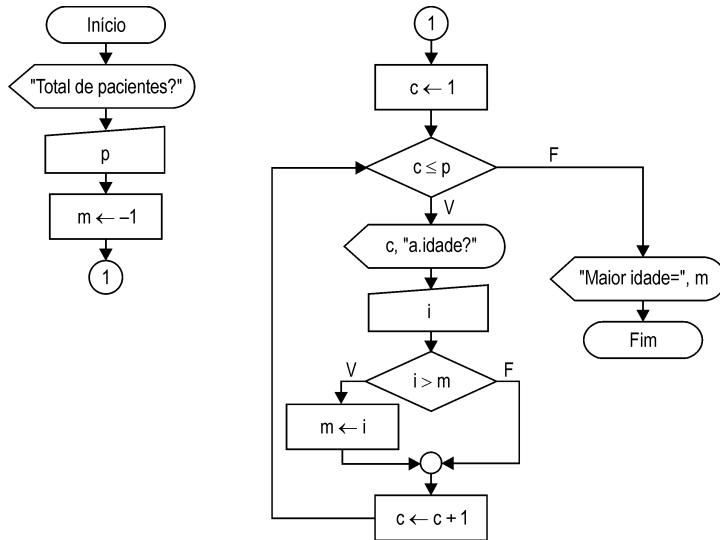
    s = 0;

    for(c=1; c≤a; c++) {
        printf("%da. Nota? ", c);
        scanf("%f", &n);
        s = s + n;
    }

    m = s/a;
    printf("Media = %.1f\n", m);

    return 0;
}
  
```

5.2

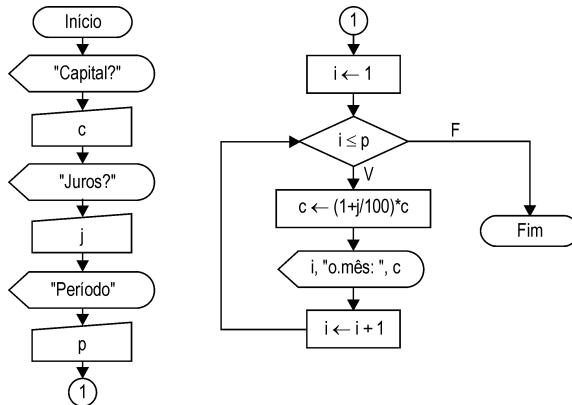


```

/* ex-5-2.c */
#include <stdio.h>
int main(void) {
    int p, m, c, i;
    printf("Total de pacientes? ");
    scanf("%d", &p);
    m = -1;
    for(c=1; c<=p; c++) {
        printf("%da. idade? ", c);
        scanf("%d", &i);
        if( i>m ) m = i;
    }
    printf("Maior idade = %d\n", m);
    return 0;
}

```

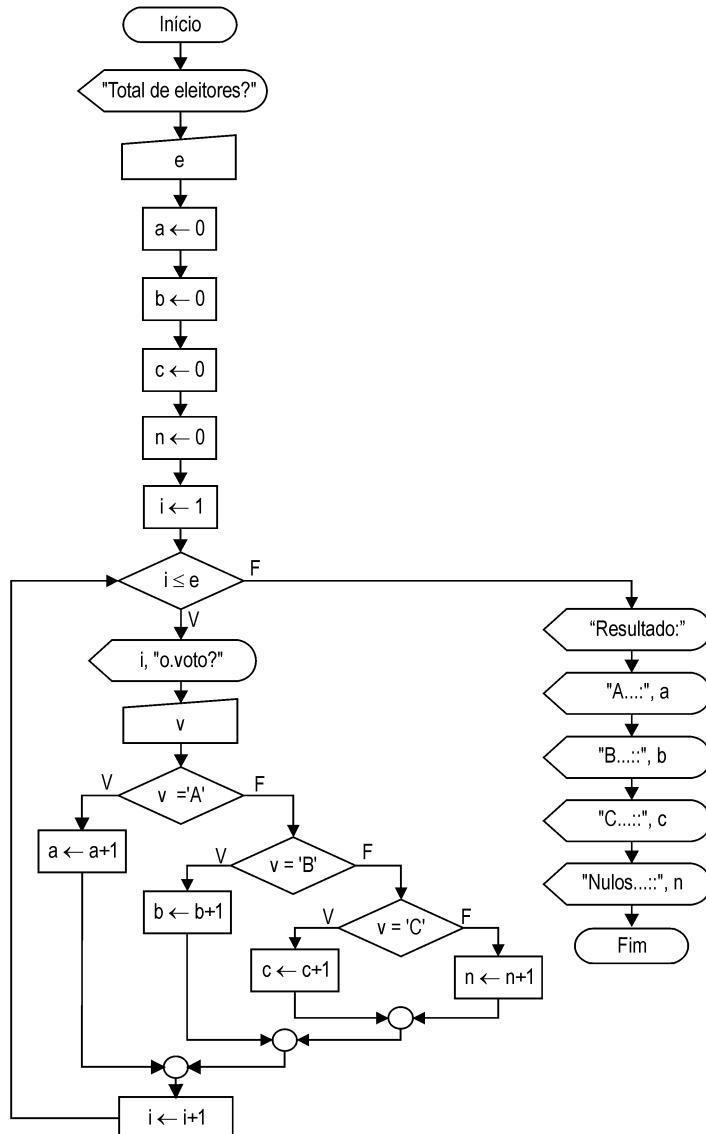
5.3



```

/* ex-5-3.c */
#include <stdio.h>
int main(void) {
    float c, j;
    int p, i;
    printf("Capital? ");
    scanf("%f", &c);
    printf("Juros? ");
    scanf("%f", &j);
    printf("Período? ");
    scanf("%d", &p);
    for(i=1; i<=p; i++) {
        c = (1+j/100)*c;
        printf("%do. mes: R$ %.2f\n", i, c);
    }
    return 0;
}

```



```

/*
 * ex-5-4.c */
#include <stdio.h>
#include <ctype.h>
int main(void) {
    int e, a, b, c, n, i;
    char v;

    printf("Total de eleitores? ");
    scanf("%d", &e);
    a = 0;
    b = 0;

```

```

c = 0;
n = 0;

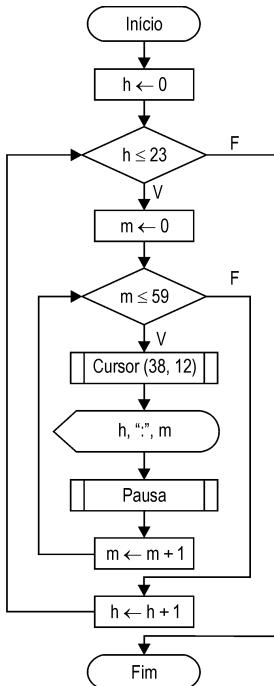
for(i=1; i<=e; i++) {
    printf("%do. voto? ", i);
    scanf("%*c%c", &v); // %*c descarta enter digitado na ultima entrada
    switch( toupper(v) ) {
        case 'A': a++; break;
        case 'B': b++; break;
        case 'C': c++; break;
        default : n++;
    }
}

printf("\nResultado:\n");
printf("A.....: %d\n", a);
printf("B.....: %d\n", b);
printf("C.....: %d\n", c);
printf("Nulo...: %d\n", n);

return 0;
}

```

5.5



```

/* ex-5-5w.c - versao para Windows - Pelles C */

#include <stdio.h>
#include <conio.h>
#include <time.h>

int main(void) {
    int h, m;

```

```

for(h=0; h<=23; h++) {
    for(m=0; m<=59; m++) {
        gotoxy(38,12);
        printf("%02d:%02d", h, m);
        sleep(1);
    }
}
return 0;
}

```

```

/* ex-5-5u.c - versao para Unix/Linux - GCC */

#include <stdio.h>
#include "conio.h"
#include <unistd.h>

int main(void) {
    int h, m;

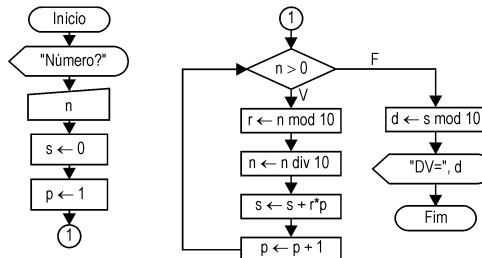
    clrscr();

    for(h=0; h<=23; h++) {
        for(m=0; m<=59; m++) {
            gotoxy(38,12);
            printf("%02d:%02d\n", h, m);
            sleep(1);
        }
    }
    return 0;
}

```

Capítulo 6

6.1



```

/* ex-6-1.c */
#include <stdio.h>

int main(void) {
    int n, s, p, r, d;

    printf("Número? ");
    scanf("%d", &n);

    s = 0;
    p = 1;
}

```

```

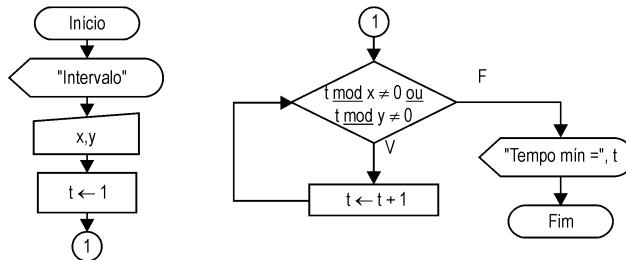
while( n>0 ) {
    r = n%10;
    n = n/10;
    s = s + r*p;
    p++;
}

d = s%10;
printf("DV = %d\n", d);

return 0;
}

```

6.2



```

/* ex-6-2.c */
#include <stdio.h>

int main(void) {
    int x, y, t;

    printf("Intervalos? ");
    scanf("%d %d", &x, &y);

    t = 1;

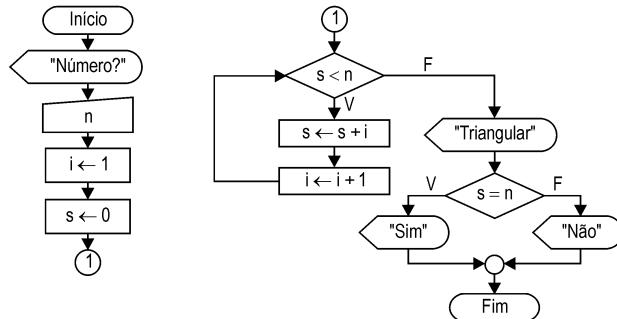
    while( t%x!=0 || t%y!=0 ) t++;

    printf("Tempo min = %d\n", t);

    return 0;
}

```

6.3



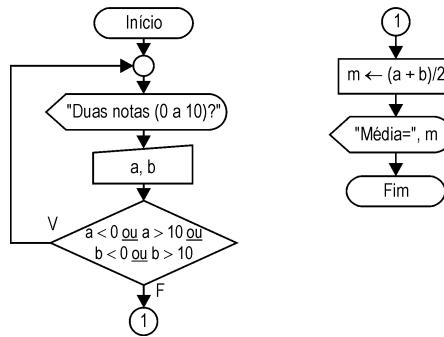
```

/* ex-6-3.c */
#include <stdio.h>
int main(void) {
    int n, i, s;
    printf("Número? ");
    scanf("%d", &n);
    i = 1;
    s = 0;
    while( s<n ) {
        s = s + i;
        i++;
    }
    printf("Triangular: ");
    if( s==n ) printf("sim\n");
    else printf("não\n");
}
return 0;

```

Capítulo 7

7.1



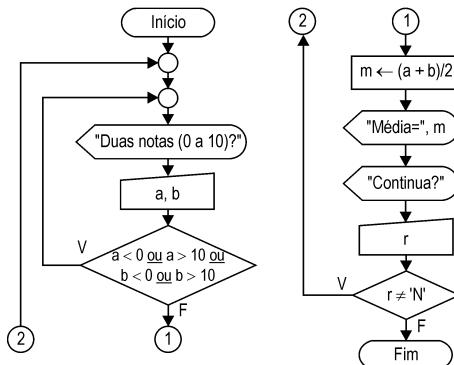
```

/* ex-7-1.c */
#include <stdio.h>
int main(void) {
    float a, b, m;
    do {
        printf("Duas notas (0 a 10)? ");
        scanf("%f %f", &a, &b);
    } while( a<0 || a>10 || b<0 || b>10 );
    m = (a+b)/2;
    printf("Media = %.1f\n", m);
}
return 0;

```



7.2



```

/*
 * ex-7-2.c */
#include <stdio.h>
#include <ctype.h>

int main(void) {
    float a, b, m;
    char r;

    do {
        do {
            printf("Duas notas (0 a 10)? ");
            scanf("%f %f", &a, &b);
        } while( a<0 || a>10 || b<0 || b>10 );

        m = (a+b)/2;
        printf("Media = %.1f\n", m);

        printf("Continua? ");
        scanf("%*c%c", &r); // %*c descarta o enter
    } while( toupper(r) != 'N' );

    return 0;
}
  
```

7.3

```

/*
 * ex-7-3.c */
#include <stdio.h>
#include <ctype.h>

int main(void) {
    float a, b;
    int op;

    do {
        puts("\n1 - somar");
        puts("2 - subtrair");
        puts("3 - multiplicar");
        puts("4 - dividir");
        puts("5 - sair");

        printf("Opcão? ");
        scanf("%d", &op);

        if( op>=1 && op<=4 ) {
  
```

```

        printf("Valores? ");
        scanf("%f %f", &a, &b);
        switch( op ) {
            case 1: printf("Resultado = %.1f\n", a+b); break;
            case 2: printf("Resultado = %.1f\n", a-b); break;
            case 3: printf("Resultado = %.1f\n", a*b); break;
            case 4: printf("Resultado = %.1f\n", a/b); break;
        }
    } while( op!=5 );
}

return 0;
}

```

7.4

```

/* ex-7-4.c */
#include <stdio.h>

int main(void) {
    float t=0, v;

    do {
        printf("Valor? ");
        scanf("%f",&v);
        if( v<=0 ) break; // termina a repeticao
        t += v;
    } while( 1 );

    printf("\nTotal: R$ %.2f\n", t);
    return 0;
}

```

7.5 Veja a resposta do Exercício 7.2.

Capítulo 8

8.1

```

/* ex-8-1.c */
#include <stdio.h>

void contagemRegressiva(int n) {
    int i;

    for(i=n; i>=0; i--)
        printf("%d\n", i);
}

int main(void) {
    int n;

    printf("Número? ");
    scanf("%d",&n);
    contagemRegressiva(n);

    return 0;
}

```



8.2

```
/* ex-8-2.c */
#include <stdio.h>

void diaDaSemana(int n) {
    switch( n ) {
        case 1: printf("domingo\n"); break;
        case 2: printf("segunda\n"); break;
        case 3: printf("terca\n"); break;
        case 4: printf("quarta\n"); break;
        case 5: printf("quinta\n"); break;
        case 6: printf("sexta\n"); break;
        case 7: printf("sabado\n"); break;
        default: printf("invalido\n");
    }
}

int main(void) {

    int n;
    printf("Numero? ");
    scanf("%d", &n);
    diaDaSemana(n);

    return 0;
}
```

8.3

```
/* ex-8-3.c */
#include <stdio.h>

int par(int n) {
    if( n%2==0 ) return 1; else return 0;
}

int main(void) {
    int n;

    printf("Numero? ");
    scanf("%d", &n);

    if( par(n) ) printf("Este numero e par!\n");
    else printf("Este numero e impar!\n");

    return 0;
}
```

8.4

```
/* ex-8-4.c */
#include <stdio.h>

float mediap(float n1, float n2, float p1, float p2) {
    return n1*p1 + n2*p2;
}

int main(void) {
    float n1, n2, p1, p2;

    printf("Notas? ");
    scanf("%f %f", &n1, &n2);
    printf("Pesos? ");
```



```

    scanf("%f %f", &p1, &p2);
    printf("Media ponderada: %.1f\n", mediap(n1,n2,p1,p2) );
    return 0;
}

```

8.5

```

/* ex-8-5.c */
#include <stdio.h>

int fat(int n) {
    int f = 1;
    while( n>1 ) { f *= n; n--; }
    return f;
}

int main(void) {
    int n;

    printf("Número? ");
    scanf("%d", &n);
    printf("O fatorial de %d é %d\n", n, fat(n));

    return 0;
}

```

8.6

```

/* ex-8-6.c */
#include <stdio.h>

int bissexto(int a) {
    return (a%4==0 && a%100!=0) || (a%400==0);
}

int ultimoDia(int m, int a) {
    if( m==2 ) return 28 + bissexto(a);
    if( m==4 || m==6 || m==9 || m==11 ) return 30;
    return 31;
}

int valida(int d, int m, int a) {
    if( a<0 ) return 0;
    if( m<1 || m>12 ) return 0;
    if( d<1 || d>ultimoDia(m,a) ) return 0;
    return 1;
}

void exibeDiaAnterior (int d, int m, int a) {
    if( d>1 ) d--;
    else {
        m--;
        if( m==0 ) {
            m=12;
            a--;
        }
        d= ultimoDia(m,a);
    }
    printf("\nData do dia anterior: %02d/%02d/%d\n", d, m, a);
}

```



```

int main(void) {
    int d, m, a;

    printf("Data? ");
    scanf("%d/%d/%d", &d, &m, &a);
    if( valida(d,m,a) ) exibeDiaAnterior (d,m,a);
    else puts("Data invalida! ");

    return 0;
}

```

Capítulo 9

9.1

```

/* ex-9-1.c */

void preencher(int v[10]) {
    int i;

    for(i=0; i<10; i++) {
        printf("%do. numero? ", i+1);
        scanf("%d", &v[i]);
    }
}

```

9.2

```

/* ex-9-2.c */

int maior(int v[10]) {
    int i, m = v[0];

    for(i=1; i<10; i++)
        if( m<v[i] )
            m = v[i];

    return m;
}

```

9.3

```

/* ex-9-3.c */

#include <stdio.h>

void preencher(int v[10]) {
    int i;

    for(i=0; i<10; i++) {
        printf("%do. numero? ", i+1);
        scanf("%d", &v[i]);
    }
}

int maior(int v[10]) {
    int i, m = v[0];

    for(i=1; i<10; i++)
        if( m<v[i] )
            m = v[i];
}

```



```

        return m;
    }

int main(void) {
    int v[10];

    preencher(v);
    printf("Maior valor no vetor = %d\n", maior(v) );
    return 0;
}

```

- 9.4** Para qualquer número de dois dígitos, ao subtrairmos a soma de seus dígitos dele mesmo, sempre obtemos um resultado que é múltiplo de 9. Por exemplo, para 78 a soma de seus dígitos é 15 e 78-15 resulta em 63, que é um múltiplo de 9 ($7 \times 9 = 63$). O programa usa essa propriedade dos números para causar a impressão de que ele adivinha o símbolo no qual o usuário está pensando; porém, se você observar a tabela que ele apresenta, verá que em todas as posições correspondentes aos múltiplos de 9 há sempre o mesmo símbolo (escollhido aleatoriamente a cada execução). Assim, não importa em que número você pense, o programa já sabe a que símbolo ele irá corresponder.

Capítulo 10

- 10.1** Sim, mesmo com itens repetidos, o algoritmo *bubble sort* funciona corretamente. Neste caso, os itens repetidos aparecem agrupados na sequência ordenada.

- 10.2** Veja a resposta do Exercício 10.3.

10.3

```

/* ex-10-3.c */

#include <stdio.h>

void preenche(int v[], int n) {
    int i;

    for(i=0; i<n; i++) {
        printf("%do. item? ", i+1);
        scanf("%d", &v[i]);
    }
}

void ordena(int v[], int n) {
    int i, j;

    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)

            if( v[j]<v[j+1] ) { // <== ordena decrescente
                int x = v[j];
                v[j] = v[j+1];
                v[j+1] = x;
            }
}

```



```

void exibe(int v[], int n) {
    int i;
    for(i=0; i<n; i++)
        printf("%d\n", v[i]);
}

int main(void) {
    int n;

    printf("Total de candidatos? ");
    scanf("%d", &n);

    int v[n];           // cria vetor dinamico
    preenche(v,n);    // preenche com n dados
    ordena(v,n);      // ordena decrescente
    printf("\nTres melhores notas:\n");
    exibe(v,3);        // exibe os tres primeiros dados

    return 0;
}

```

10.4

```

/* ex-10-4.c */

#include <stdio.h>

int pertence(int x, int v[], int n) { // busca binaria
    int i=0, f=n-1, m;

    while( i<=f ) {
        m = (i+f)/2;
        if( x==v[m] ) return 1;
        if( x<v[m] ) f = m-1;
        else i = m+1;
    }

    return 0;
}

int main(void) {
    int A[5] = {10, 20, 30, 40, 50};
    int U[9] = {19, 28, 39, 45, 56, 63, 77, 81, 92};
    int x;

    while( 1 ) {
        printf("\nSenha? ");
        scanf("%d", &x);
        if( pertence(x,A,5) ) { printf("Administrador: "); break; }
        else if( pertence(x,U,9) ) { printf("Usuario: "); break; }
        puts("Senha invalida!");
    }

    puts("senha valida!\n");

    return 0;
}

```



10.5

```
/* ex-10-5.c */
#include <stdio.h>

void preenche(int v[], int n) {
    int i;

    for(i=0; i<n; i++) {
        printf("%da. senha? ", i+1);
        scanf("%d", &v[i]);
    }
}

void ordena(int v[], int n) {
    int i, j;

    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++) {
            if( v[j]>v[j+1] ) {
                int x = v[j];
                v[j] = v[j+1];
                v[j+1] = x;
            }
        }
}

int pertence(int x, int v[], int n) {
    int i=0, f=n-1, m;

    while( i<=f ) {
        m = (i+f)/2;
        if( x==v[m] ) return 1;
        if( x<v[m] ) f = m-1;
        else i = m+1;
    }

    return 0;
}

int main(void) {
    int A[5], U[9], x;

    printf("Cadastro de administradores:\n");
    preenche(A,5);
    printf("\nCadastro de usuarios:");
    preenche(U,9);
    ordena(A,5);
    ordena(U,9);

    while( 1 ) {
        printf("\nSenha? ");
        scanf("%d", &x);

        if( pertence(x,A,5) ) { printf("Administrador: "); break; }
        else if( pertence(x,U,9) ) { printf("Usuario: "); break; }

        puts("Senha invalida!");
    }

    puts("senha valida!\n");
    return 0;
}
```



Capítulo 11

11.1

```
/* ex-11-1w.c - versao para Windows - Pelles C */

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

void centraliza(char m[], int y) {
    gotoxy((80-strlen(m))/2 + 1,y);
    puts(m);
}

int main(void) {
    char m[128];

    while( 1 ) {
        printf("Digite uma mensagem com ate 80 posicoes: ");
        gets( m );
        if( strlen(m) <= 80 ) break;
        puts("Mensagem muito longa!");
    }

    clrscr();
    centraliza(m,12);
    getch();

    return 0;
}
```

```
/* ex-11-1u.c - versao para Unix/Linux - GCC */

#include <stdio.h>
#include "conio.h"
#include <string.h>

void centraliza(char m[], int y) {
    gotoxy((80-strlen(m))/2 + 1,y);
    puts(m);
}

int main(void) {
    char m[128];

    while( 1 ) {
        printf("Digite uma mensagem com ate 80 posicoes: ");
        gets( m );
        if( strlen(m) <= 80 ) break;
        puts("Mensagem muito longa!");
    }

    clrscr();
    centraliza(m,12);
    getch();

    return 0;
}
```



11.2

```
/* ex-11-2.c */
#include <stdio.h>
#include <string.h>

void espacos(int n) {
    int i;

    for(i=1; i<=n; i++)
        printf(" ");
}

int main(void) {
    char n[26];
    int i;

    printf("Qual o seu nome? ");
    gets(n);

    for(i=0; n[i]!='\0'; i++) {
        espacos(i);
        printf("%c\n",n[i]);
    }

    return 0;
}
```

11.3

```
/* ex-11-3.c */
int strcmpi(char a[], char b[]) {
    int i=0;

    while( tolower(a[i])==tolower(b[i]) && a[i]!='\0')
        i++;

    return a[i]-b[i];
}
```

11.4

```
/* ex-11-4.c */
#include <stdio.h>
#include <ctype.h>

int strcmpi(char a[], char b[]) {
    int i=0;

    while( tolower(a[i])==tolower(b[i]) && a[i]!='\0')
        i++;

    return a[i]-b[i];
}

int main(void) {
    char n1[100], n2[100];

    printf("1o. nome? ");
    gets(n1);
    printf("2o. nome? ");
    gets(n2);

    if( strcmpi(n1,n2)>0 ) printf("%s\n%s\n",n1,n2);
    else if( strcmpi(n1,n2)<0 ) printf("%s\n%s\n",n2,n1);
}
```



```
    else printf("Os nomes sao iguais\n");
}
}
```

11.5

```
/* ex-11-5.c */
#include <stdio.h>
#include <string.h>

int main(void) {
    char s[100];

    printf("Senha? ");
    gets(s);

    if( strcmp(s, "ABRACADABRA")==0 ) printf("Ok\n");
    else printf("Senha incorreta\n");

    return 0;
}
```

11.6

```
/* ex-11-6.c */
#include <stdio.h>
#include <string.h>

int ocorrencias(char c, char s[]) {
    int i, t=0;

    for(i=0; s[i]; i++)
        if( s[i]==c )
            t++;

    return t;
}

int main(void) {
    char c, p[100];

    printf("Letra? ");
    scanf("%c%c", &c);
    printf("Palavra? ");
    gets(p);

    printf("O caractere '%c' ocorre %d vezes em '%s'\n",
           c, ocorrencias(c,p),p);

    return 0;
}
```

Capítulo 12

12.1

```
/* ex-12-1.c */
#include <stdio.h>
#include <string.h>

void ler(char m[] [31], int n) {
```



```

int i;

for(i=0; i<n; i++) {
    printf("%do. nome? ", i+1);
    gets(m[i]);
}

void ordenar(char m[][31], int n) {
    int i, j;

    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if( strcmp(m[j],m[j+1])<0 ) { // <== alteracao
                char x[31];
                strcpy(x,m[j]);
                strcpy(m[j],m[j+1]);
                strcpy(m[j+1],x);
            }
}

void exibir(char m[][31], int n) {
    int i;

    for(i=0; i<n; i++)
        printf("%do. nome: %s\n", i+1, m[i]);
}

int main(void) {
    char m[10][31];

    ler(m,10);
    ordenar(m,10);
    puts("\nOrdem alfabetica descrescente:\n");
    exibir(m,10);

    return 0;
}

```

12.2 Note o comportamento aleatório do computador, ao realizar suas jogadas.

12.3

```

/* ex-12-3.c */

int acha2(char t[3][3], char s) {
    int i, j;

    for(i=0; i<3; i++) {
        for(j=0; j<3; j++) {
            if( t[i][j]==' ' && t[i][(j+1)%3]==t[i][(j+2)%3] &&
                t[i][(j+1)%3]==s) {
                t[i][j]='o';
                return 1;
            }
            if( t[j][i]==' ' && t[(j+1)%3][i]==t[(j+2)%3][i] &&
                t[(j+1)%3][i]==s) {
                t[j][i]='o';
                return 1;
            }
        }
        if( t[i][i]==' ' && t[(i+1)%3][(i+1)%3]==t[(i+2)%3][(i+2)%3] &&
            t[(i+1)%3][(i+1)%3]==s) {
            t[i][i]='o';
            return 1;
        }
    }
}
```



```

        if( t[i][2-i]==' ' &&
            t[(i+1)%3][2-(i+1)%3]==t[(i+2)%3][2-(i+2)%3] &&
            t[(i+1)%3][2-(i+1)%3]==s) {
                t[i][2-i]='o';
                return 1;
            }
        }
        return 0;
    }

void computador(char t[3][3]) {
    int L, C;

    if( acha2(t,'o') || // ataca tentando achar dois 'o's alinhados
        acha2(t,'x') ) // defende tentando achar dois 'x's alinhados
        return;           // se conseguir uma das duas coisas, termina jogada

    do {
        L = rand()%3;      // senao, joga em posicao aleatoria
        C = rand()%3;
    } while(t[L][C]!=' ');

    t[L][C]='o';
}

```

12.4

```

/* ex-12-4w.c c - versao para Windows - Pelles C */

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

int iniciar(char c[4][4]) {
    int i, j, b=0;

    for(i=0; i<4; i++)
        for(j=0; j<4; j++)
            if( rand()%3==0 ) { c[i][j]=1; b++; }
            else c[i][j]=0;

    return b;
}

void mostrar(char c[4][4]) {
    int i, j;

    _clrscr();

    for(i=0; i<4; i++) {
        for(j=0; j<4; j++) {
            switch( c[i][j] ) {
                case -2: _textcolor(10); printf("%c ",2); break;
                case -15: _textcolor(12); printf("%c ",15); break;
                default : _textcolor(9); printf(" ? ");
            }
            printf("\n");
        }
        _textcolor(7);
    }
}

int jogar(char c[4][4]) {
    int i, j;

```

```

do {
    printf("\n\nPosicao? ");
    scanf("%d,%d", &i, &j);
} while(i<0 || i>3 || j<0 || j>3 || c[i][j]<0);

if( c[i][j]==0 ) { c[i][j]= -2; return 0; }
else { c[i][j]=-15; return 1; }
}

int main(void) {
    char c[4][4];
    int j=0, e=0, b;

    srand(time(NULL));

    b = iniciar(c);
    printf("Existem %d bombas no campo!\n", b);

    _sleep(3);

    do {
        mostrar(c);
        e += jogar(c);
        j++;
    } while(j<16-b+e && e<=3);

    mostrar(c);

    if( e<=3 && j==16-b+e) printf("Voce ganhou!\n");
    else printf("Voce perdeu!\n");

    return 0;
}

```

```

/* ex-12-4u.c c - versao para Unix/Linux - GCC */

#include <stdio.h>
#include "conio.h"
#include <stdlib.h>
#include <time.h>

void limpar(char t[3][3]);
void mostrar(char t[3][3]);
void usuario(char t[3][3]);
void computador(char t[3][3]);
char vencedor(char t[3][3]);

int main(void) {
    char t[3][3], v;
    int j=0, e, s;

    printf("par(0) ou impar(1)?");
    scanf("%d", &e);
    srand(time(NULL));
    s=rand()%2;
    limpar(t);

    do {
        mostrar(t);
        if(e==s) usuario(t);
        else computador(t);
        j++;
        s = !s;
        v=vencedor(t);
    } while(j<9 && v==' ');

    mostrar(t);
    switch(v) {

```



```

        case 'x': puts("\n\nVoce ganhou"); break;
        case 'o': puts("\n\nEu ganhei"); break;
        case ' ': puts("\n\nEmpatamos");
    }
    return 0;
}

void limpar(char t[3][3]) {
    int i, j;
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            t[i][j]=' ';
}

void mostrar(char t[3][3]) {
    int i;
    clrscr();
    for(i=0;i<3;i++) {
        printf("\n %c | %c ", t[i][0], t[i][1], t[i][2]);
        if(i<2) printf("\n---+---+---");
    }
}

void usuario(char t[3][3]) {
    int L, C;
    do {
        printf("\n\nPosicao?");
        scanf("%d,%d", &L, &C);
    } while(L<0 || L>2 || C<0 || C>2 || t[L][C]!=' ');
    t[L][C]='x';
}

void computador(char t[3][3]) {
    int L, C;
    do {
        L = rand()%3;
        C = rand()%3;
    } while(t[L][C]!=' ');
    t[L][C]='o';
}

char vencedor(char t[3][3]) {
    int i;
    for(i=0; i<3; i++) {
        if( t[i][0]==t[i][1] && t[i][1]==t[i][2] && t[i][0]!=' ')
            return t[i][0];
        if( t[0][i]==t[1][i] && t[1][i]==t[2][i] && t[0][i]!=' ')
            return t[0][i];
    }

    if( t[0][0]==t[1][1] && t[1][1]==t[2][2] && t[0][0]!=' ')
        return t[0][0];
    if( t[0][2]==t[1][1] && t[1][1]==t[2][0] && t[0][2]!=' ')
        return t[0][2];
    return ' ';
}

```



Capítulo 13

13.2

```
/* ex-13-2.c */

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

typedef struct {
    int dia;
    int mes;
} data;

typedef struct {
    char nome[31];
    char fone[21];
    data aniv;
} pessoa;

int menu(void);
void incluir(void);
void listar(void);
void consultar(void);
void excluir(void);

int main(void) {
    while( 1 ) {
        switch( menu() ) {
            case 1: incluir(); break;
            case 2: listar(); break;
            case 3: consultar(); break;
            case 4: excluir(); break;
            case 5: exit(0); // sai do programa
        }
    }
    return 0;
}

int menu(void) {
    int op;
    puts("\n1 - Incluir");
    puts("2 - Listar");
    puts("3 - Consultar");
    puts("4 - Excluir");
    puts("5 - Sair");
    printf("\nOpcão? ");
    scanf("%d%c", &op);

    return op;
}

void incluir(void) {
    FILE *s;
    pessoa p;

    s = fopen("agenda.dat", "ab");
    if( s==NULL ) {
        puts("erro fatal: o arquivo não pode ser aberto!");
        exit(1);
    }

    printf("\nNome? ");
    gets(p.nome);
    printf("Fone? ");
}
```



```

gets(p.fone);
printf("Aniv? ");
scanf("%d/%d%c",&p.aniv.dia,&p.aniv.mes);
fwrite(&p,sizeof(pessoa),1,s);
printf("\nRegistro gravado!\n");
fclose(s);
}

void listar(void) {
FILE *s;
pessoa p;

s = fopen("agenda.dat","rb");
if( s==NULL ) {
    puts("erro fatal: o arquivo nao pode ser aberto!");
    exit(2);
}

while( 1 ) {
    fread(&p,sizeof(pessoa),1,s);
    if( feof(s) ) break;
    printf("\n%s - %s - %02d/%02d", p.nome,p.fone,
           p.aniv.dia,p.aniv.mes);
}

printf("\n\n");
fclose(s);
}

int strcmpi(char a[], char b[]){
int i = 0;

while( toupper(a[i])==toupper(b[i]) && a[i]!='\0' )
    i++;

return a[i]-b[i];
}

void consultar(void) {
FILE *s;
pessoa p;
char n[31];
int t=0;

s = fopen("agenda.dat","rb");
if( s==NULL ) {
    puts("erro fatal: o arquivo nao pode ser aberto!");
    exit(3);
}

printf("\nNome? ");
gets(n);

while( 1 ) {
    fread(&p,sizeof(pessoa),1,s);
    if( feof(s) ) break;
    if( strcmpi(n,p.nome)==0 ) {
        printf("\nFone: %s",p.fone);
        printf("\nAniv: %02d/%02d\n",p.aniv.dia,p.aniv.mes);
        t++;
    }
}

printf("Registro(s) encontrado(s): %d\n", t);
fclose(s);
}

void excluir(void) {
FILE *e,*s;
pessoa p;

```



```

char n[31];
int t = 0;

remove("agenda.bak");
rename("agenda.dat","agenda.bak");

e = fopen("agenda.bak","rb");
if( e==NULL ) {
    puts("erro fatal: o arquivo nao pode ser aberto!");
    exit(4);
}

s = fopen("agenda.dat","wb");
if( s==NULL ) {
    puts("erro fatal: o arquivo nao pode ser criado!");
    exit(5);
}

printf("\nNome? ");
gets(n);

while( 1 ) {
    fread(&p,sizeof(pessoa),1,e);
    if( feof(e) ) break;
    if( strncmp(n,p.nome)!=0 ) fwrite(&p,sizeof(pessoa),1,s);
    else t++;
}

printf("Registro(s) excluido(s): %d\n", t);
fclose(e);
fclose(s);
}

```

13.3

```

/* ex-13-3.c */

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

typedef struct {
    int dia;
    int mes;
    int ano;
} data;

typedef struct {
    char nome[31];
    float salario;
    data adm;
} func;

int menu(void) {
    int op;

    puts("\n1 - Incluir");
    puts("2 - Listar");
    puts("3 - Listar faixa");
    puts("4 - Consultar");
    puts("5 - Excluir");
    puts("6 - Sair");
    printf("\nOpcão? ");
    scanf("%d%c",&op);

    return op;
}
void incluir(void) {

```



```

FILE *s;
func f;

s = fopen("cadfunc.dat", "ab");
if( s==NULL ) {
    puts("erro!");
    exit(2);
}

printf("\nNome? ");
gets(f.nome);
printf("Salario? ");
scanf("%f",&f.salario);
printf("Admissao? ");
scanf("%d/%d/%d%c",&f.adm.dia,&f.adm.mes,&f.adm.ano);
fwrite(&f,sizeof(func),1,s);
fclose(s);
}

void listar(void) {
FILE *s;
func f;

s = fopen("cadfunc.dat","rb");
if( s==NULL ) {
    puts("erro!");
    exit(4);
}

while( 1 ) {
    fread(&f,sizeof(func),1,s);
    if( feof(s) ) break;
    printf("\n%s - %10.2f - %02d/%02d",
           f.nome, f.salario, f.adm.dia, f.adm.mes,f.adm.ano);
}

printf("\n\n");
fclose(s);
}

void listarFaixa(void) {
FILE *s;
func f;
float min, max;

s = fopen("cadfunc.dat", "rb");
if( s==NULL ) {
    puts("erro!");
    exit(4);
}

printf("Faixa salarial? (min,max) ");
scanf("%f,%f",&min,&max);

while( 1 ) {
    fread(&f,sizeof(func),1,s);
    if( feof(s) ) break;
    if( f.salario>=min && f.salario<=max )
        printf("\n%s - %10.2f - %02d/%02d",
               f.nome,f.salario, f.adm.dia, f.adm.mes,f.adm.ano);
}

printf("\n\n");
fclose(s);
}

void consultar(void) {
FILE *s;
func f;
char n[31];
int t=0;

```



```

s = fopen("cadfunc.dat", "rb");
if( s==NULL ) {
    puts("erro!");
    exit(3);
}

printf("\nNome? ");
gets(n);

while( 1 ) {
    fread(&f,sizeof(func),1,s);
    if( feof(s) ) break;
    if( strcmp(n,f.nome)==0 ) {
        printf("\nSalario.: %.2f",f.salario);
        printf("\nAdmissao: %02d/%02d/%d\n",
               f.adm.dia, f.adm.mes, f.adm.ano);
        t++;
    }
}
printf("Foram encontrados %d registros!\n\n", t);
fclose(s);
}

void excluir(void) {
FILE *e, *s;
func f;
char n[31];

remove("cadfunc.bak");
rename("cadfunc.dat","cadfunc.bak");

e = fopen("cadfunc.bak", "rb");
if( e==NULL ) {
    puts("erro!");
    exit(5);
}

s = fopen("cadfunc.dat", "wb");
if( s==NULL ) {
    puts("erro!");
    exit(6);
}

printf("\nNome? ");
gets(n);

while( 1 ) {
    fread(&f,sizeof(func),1,e);
    if( feof(e) ) break;
    if( strcmp(n,f.nome)!=0 ) fwrite(&f,sizeof(func),1,s);
}
fclose(e);
fclose(s);
}

int main(void) {
while( 1 ) {
    switch( menu() ) {
        case 1: incluir(); break;
        case 2: listar(); break;
        case 3: listarFaixa(); break;
        case 4: consultar(); break;
        case 5: excluir(); break;
        case 6: return 0;
    }
}
return 0;
}

```



13.4

```
/* ex-13-4.c */

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

typedef struct {
    char titulo[31];
    char autor[31];
    int ano;
    float preco;
} livro;

int menu(void) {
    int op;

    puts("\n1 - Incluir");
    puts("2 - Listar por autor");
    puts("3 - Listar por faixa de preço");
    puts("4 - Listar por ano de publicação");
    puts("5 - Consultar por título");
    puts("6 - Excluir");
    puts("7 - Sair");
    printf("\nOpção? ");
    scanf("%d%c", &op);

    return op;
}

void incluir(void) {
    FILE *s;
    livro l;

    s = fopen("cadlivros.dat", "ab");
    if( s==NULL ) {
        puts("erro!");
        exit(2);
    }

    printf("\nTítulo? "); gets(l.titulo);
    printf("Autor.? "); gets(l.autor);
    printf("Ano...? "); scanf("%d", &l.ano);
    printf("Preço.? "); scanf("%f%c", &l.preco);
    fwrite(&l, sizeof(livro), 1, s);
    fclose(s);
}

int strcmpi(char a[], char b[]) {
    int i=0;

    while( tolower(a[i])==tolower(b[i]) && a[i]!='\0' )
        i++;

    return a[i]-b[i];
}

void listarAutor(void) {
    FILE *s;
    livro l;
    char a[100];
    int t=0;

    s = fopen("cadlivros.dat", "rb");
    if( s==NULL ) {
        puts("erro!");
        exit(4);
    }

    printf("\nAutor? ");
    gets(a);
```



```

        while( 1 ) {
            fread(&l,sizeof(livro),1,s);
            if( feof(s) ) break;
            if( strcmp(a,l.autor)==0 ) {
                printf("%s - %d - %.2f\n", l.titulo, l.ano, l.preco);
                t++;
            }
        }

        printf("Registro(s) encontrado(s) : %d\n",t);
        fclose(s);
    }

void listarFaixa(void) {
    FILE *s;
    livro l;
    float min, max;
    int t=0;

    s = fopen("cadlivros.dat","rb");
    if( s==NULL ) {
        puts("erro!");
        exit(4);
    }

    printf("Faixa de preco? (min,max) ");
    scanf("%f,%f",&min,&max);

    while( 1 ) {
        fread(&l,sizeof(livro),1,s);
        if( feof(s) ) break;
        if( l.preco>=min && l.preco<=max ) {
            printf("%s - %s - %d - %.2f\n", l.titulo, l.autor,
                   l.ano, l.preco);
            t++;
        }
    }

    printf("Registro(s) encontrado(s) : %d\n",t);
    fclose(s);
}

void listarAno(void) {
    FILE *s;
    livro l;
    int a, t=0;

    s = fopen("cadlivros.dat","rb");
    if( s==NULL ) {
        puts("erro!");
        exit(4);
    }

    printf("\nAno? ");
    scanf("%d%c",&a);

    while( 1 ) {
        fread(&l,sizeof(livro),1,s);
        if( feof(s) ) break;
        if( a==l.ano )
            printf("%s - %s - %.2f\n", l.titulo, l.autor, l.preco);
    }

    printf("Registro(s) encontrado(s) : %d\n",t);
    fclose(s);
}

void consultarTitulo(void) {
    FILE *s;
    livro l;
    char n[31];
    int t=0;

    s = fopen("cadlivros.dat","rb");
    if( s==NULL ) {

```



```

        puts("erro!");
        exit(3);
    }

    printf("\nTitulo? ");
    gets(n);

    while( 1 ) {
        fread(&l,sizeof(livro),1,s);
        if( feof(s) ) break;
        if( strcmp(n,l.titulo)==0 ) {
            printf("\nAutor.: %s",l.autor);
            printf("\nAno...: %d",l.ano);
            printf("\nPreco.: %.2f\n",l.preco);
            t++;
        }
    }

    printf("Foram encontrados %d registros!\n\n", t);
    fclose(s);
}

void excluir(void) {
    FILE *e, *s;
    livro l;
    char n[31];
    int t=0;

    remove("cadlivros.bak");
    rename("cadlivros.dat","cadlivros.bak");

    e = fopen("cadlivros.bak","rb");
    if( e==NULL ) {
        puts("erro!");
        exit(5);
    }

    s = fopen("cadlivros.dat","wb");
    if( s==NULL ) {
        puts("erro!");
        exit(6);
    }

    printf("\nTitulo? ");
    gets(n);

    while( 1 ) {
        fread(&l,sizeof(livro),1,e);
        if( feof(e) ) break;
        if( strcmp(n,l.titulo)!=0 ) fwrite(&l,sizeof(livro),1,s);
        else t++;
    }

    printf("Registro(s) excluido(s): %d\n",t);
    fclose(e);
    fclose(s);
}

int main(void) {
    while( 1 ) {
        switch( menu() ) {
            case 1: incluir();           break;
            case 2: listarAutor();       break;
            case 3: listarFaixa();       break;
            case 4: listarAno();         break;
            case 5: consultarTitulo();   break;
            case 6: excluir();          break;
            case 7: return 0;
        }
    }
    return 0;
}

```



Bibliografia

Algoritmos

AZEREDO, P. A. **Métodos de Classificação de Dados e Análise de Suas Complexidades**. Rio de Janeiro: Ed. Campus, 1996.

BÖHM, C. and Jacopini, G. **Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules**. Communications of the ACM 9 (5), p. 366-371, 1966.

BROOKSHEAR, J. Glenn. **Ciência da Computação: Uma Visão Abrangente**. Porto Alegre: Bookman, 2000.

FARRER, H. et al. **Algoritmos Estruturados**. Rio de Janeiro: Guanabara S.A., 1989.

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de Programação: A Construção de Algoritmos e Estruturas de Dados**. São Paulo: Pearson Education, 1999.

SALIBA, W. L. C. **Técnicas de Programação: Uma Abordagem Estruturada**. São Paulo: Makron Books, 1992.

Linguagem C

HANCOCK, L.; KRIEGER, M. **Manual de Linguagem C**. Rio de Janeiro: Campus, 1985.

KERNIGHAN, B. W.; RICTHIE, D. M. **A Linguagem de Programação C: Padrão ANSI**. Rio de Janeiro: Ed. Campus, 1990.

MIZRHAHI, V. V. **Treinamento em Linguagem C: Módulo I**. São Paulo: Makron Books, 1990.

MIZRHAHI, V. V. **Treinamento em Linguagem C: Módulo II**. São Paulo: Makron Books, 1990.

PLAUGER, P.J.; BRODIE J. **C: Guia de Referência Básica**. São Paulo: McGraw-Hill, 1991.

SCHILD'T, H. **C Completo e Total**. 3.ed. São Paulo: Makron Books, 1996.

Padrões

ANSI - American National Standards Institute. **ANSI X3.6: Flowchart Symbols and their Usage in Information Processing**. USA, 1970.

DEC - Digital Equipment Corporation. **VT100: User Guide**. 3.ed. USA, 1981.

ISO - International Organization for Standardization. **ISO 5807: Information Processing - Documentation Symbols and Conventions; Program and System Flowcharts**, USA, 1985.

ISO - International Organization for Standardization. **ISO/IEC 9899: Programming Languages - C**, USA, 1999.



Marcas registradas

Todos os nomes registrados, marcas registradas ou direitos de uso citados neste livro pertencem aos seus respectivos proprietários.



Índice remissivo

A

Algoritmo, 15
Arquivo, 122
 consultar, 126
 excluir, 127
 FILE *, 123
 incluir, 125
 listar, 126
ASCII, 48
 estendido, 130
 padrão, 129
Atribuição, 27

B

Biblioteca
 conio.h, 62, 68, 137
 ctype.h, 50
 math.h, 32
 stdio.h, 19
 stdlib.h, 93, 116
 string.h, 108
 time.h, 62, 93
Bloco, 39

C

Caractere de controle, 29
Comando
 break, 50, 65
 do-while, 69
 for, 56
 if-else, 37
 return, 83
 switch-case, 49
 while, 63
Comentário, 19
Compilador, 21, 77
 CC, 21
 GCC, 21, 33
 Pelles C, 21, 133
Condição
 composta, 36
 simples, 35
Constante, 25

D

Dados
 entrada, 27
 saída, 27
Diretiva, 19
 #define, 77
 #include, 77
Divisão
 inteira, 26
 real, 26

E

Estrutura, 121
 de controle, 22
 encadeada, 45
 encaixada, 40, 59
 repetição com poscondição, 69
 repetição com precondição, 63
 repetição contada, 56
 repetição infinita, 65
 seleção múltipla, 48
 seleção simples, 36
 sequencial, 30
Expressão, 26
 condicional, 35

F

Fluxograma, 16
Formato
 %*c, 72
 %c, 28
 %d, 28
 %f, 28
 %s, 28, 103
Função
 clrscr(), 72
 gotoxy(), 62, 68
 kbhit(), 68
 normalvideo(), 137
 sleep(), 62, 137
 textbackground(), 61
 textcolor(), 60
 definição, 80
 fclose(), 123
 feof(), 123



fopen(), 123
fread(), 123
fwrite(), 123
getchar(), 72
gets(), 104
main(), 19
parâmetro, 80
pow(), 32
printf(), 29
protótipo, 84
putchar(), 61
puts(), 75, 105
rand(), 93, 116
remove(), 123
rename(), 123
scanf(), 28
sleep(), 137
sqrt(), 32
srand(), 93, 116
strcat(), 108
strcmp(), 108
strcpy(), 108
strlen(), 108
time(), 93
tipo, 80
tolower(), 50
toupper(), 50

L

Linguagem de programação, 18
Linkeditor, 77

M

Matriz, 113
jogo da velha, 116

O

Operador, 26
aritmético, 26
aritmético de atribuição, 55
atribuição, 27
endereço, 28
lógico, 36
relacional, 35
seleção de campo, 121

P

Palavra reservada, 20
Preprocessador, 77

Programa, 18
análise, 20
código executável, 77
código-fonte, 77
código-objeto, 77
estruturado, 23
implementação, 20
projeto, 20
rastreamento, 135
teste, 20

R

Registro, 121
struct, 121

S

String, 102
constante, 103
ordenação, 114
variável, 103

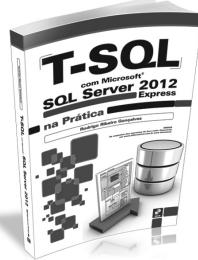
T

Tela de execução, 17
Terminal
sequência de escape, 138
VT100, 138
Tipo de dados
cadeia, 102
char, 28
FILE *, 123
float, 28
int, 28
void, 80

V

Variável, 18, 26
acumulador, 55
contador, 55
declaração, 19
nome, 26
tipo de dados, 19, 25
Vetor, 87
busca binária, 99
busca linear, 97
índice, 87
iniciação, 90
ordenação, 95
referência, 89
tamanho variável, 88



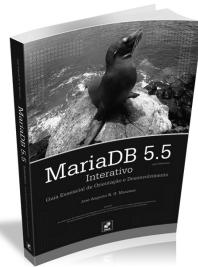


T-SQL com Microsoft SQL Server 2012 Express na Prática

Autor: Rodrigo Ribeiro Gonçalves

Código: 4537 • 120 páginas • Formato: 17 x 24 cm • ISBN: 978-85-365-0453-7 • EAN: 9788536504537

Esta obra serve como referência para o desenvolvimento de banco de dados com o Microsoft SQL Server 2012 Express, para profissionais, estudantes e interessados na área de Tecnologia da Informação. De forma didática, com exemplos e exercícios práticos, descreve técnicas de programação com a linguagem SQL, ensinando a instalar e utilizar softwares relacionados. Explica como criar e alterar tabelas, procedimentos, funções, views e triggers; utilizar o SQL dinâmico; trabalhar com datas, chaves primárias e estrangeiras, JOINS e strings. Oferece noções de administração de banco de dados, como rotinas de backup e restauração. Recomenda-se que o leitor tenha noções de programação e teoria de banco de dados.



MariaDB 5.5 - Interativo: Guia Essencial de Orientação e Desenvolvimento - para Windows

Autor: José Augusto N. G. Manzano

Código: 4155 • 256 páginas • Formato: 17,5 x 24,5 cm • ISBN: 978-85-365-0415-5 • EAN: 9788536504155

Apresenta ao iniciante na administração de bancos de dados os recursos do programa gerenciador MariaDB versão 5.5, de forma simples e didática. Destaca os conceitos básicos, o surgimento da linguagem de consulta estruturada SQL; aquisição e instalação do programa; manuseio, criação e remoção de bancos de dados.

Descreve como criar tabelas e realizar consultas; operadores aritméticos; uso de funções; agrupamentos, uniões, junções e visualizações de dados; utilização de índices; chaves primária e estrangeira; stored procedures; triggers e functions.



Integração de Dados na Prática

Técnicas de ETL para Business Intelligence com Microsoft Integration Services 2012

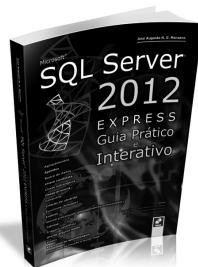
Autor: Rodrigo Ribeiro Gonçalves

Código: 4094 • 160 páginas • Formato: 17 x 24 cm • ISBN: 978-85-365-0409-4 • EAN: 9788536504094

Entender conceitos como Data Warehouse, Business Intelligence, ETL (extração, transformação e carga) e os jargões da indústria é essencial para quem quer trabalhar com integração de dados. Neste sentido, o livro foi escrito, explicando também tabelas fato e dimensão e a importância das dimensões que mudam lentamente, chamadas de Slowly Changing Dimensions.

Apresenta o Microsoft Integration Services 2012 e o Solution Explorer, introduzindo controles mais avançados como Control Flow e Data Flow, variáveis e arquivos de configuração. Para que o leitor entenda melhor a ferramenta, um pequeno projeto de Business Intelligence é desenvolvido, sendo criados três bancos de dados básicos. Conforme o projeto de ETL é montado, integrações são realizadas: Import/Export Wizard, arquivos texto, Microsoft SQL Server 2012, Data Conversion, Derived Column, Conditional Split, Merge e o MultiCast, além de mostrar como utilizar o Data Viewer para analisar a execução dos pacotes.

É indicado a analistas de sistemas, administradores de banco de dados e desenvolvedores de software com conhecimentos básicos de banco de dados e SQL para criação de tabelas e consultas, assim como inserts e updates.



Microsoft SQL Server 2012 Express - Guia Prático e Interativo

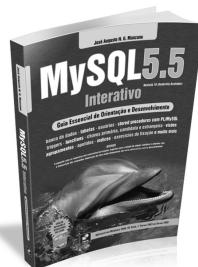
Autor: José Augusto N. G. Manzano

Código: 414A • 208 páginas • Formato: 17 x 24 cm • ISBN: 978-85-365-0414-8 • EAN: 978853654148

Estudantes e profissionais da área encontram nesta obra os principais recursos do programa gerenciador Microsoft SQL Server 2012 Express.

Descreve o surgimento da linguagem de consulta estruturada SQL; aquisição e instalação do programa; manuseio, criação e remoção de bancos de dados; tabelas e consultas; uso de funções; agrupamentos, uniões, junções e visualizações de dados; utilização de índices; chaves primária e estrangeira; stored procedures; triggers e functions.

Contempla também uma série de atividades para facilitar o aprendizado.



MySQL 5.5 - Interativo - Guia Essencial de Orientação e Desenvolvimento

Autor: José Augusto N. G. Manzano

Código: 3851 • 240 páginas • Formato: 17,5 x 24,5 cm • ISBN: 978-85-365-0385-1 • EAN: 9788536503851

Esta obra apresenta ao leitor iniciante na prática de administração de bancos de dados o programa gerenciador Oracle MySQL 5.5. Abrange o surgimento da linguagem de consulta estruturada SQL; aquisição e instalação do programa; manuseio, criação e remoção de bancos de dados; tabelas e consultas; uso de funções; agrupamentos, uniões, junções e visualizações de dados; utilização de índices; chaves primária e estrangeira; stored procedures; triggers e functions.

O conteúdo segue a mesma estrutura didática do livro MySQL 5.1 - Interativo - Guia Básico de Orientação e Desenvolvimento, porém as telas, os comandos e os textos foram adaptados à versão 5.5 do programa.



PostgreSQL 8.3.0 - Interativo: Guia de Orientação e Desenvolvimento

Autor: José Augusto N. G. Manzano

Código: 1987 • 240 páginas • Formato: 17,5 x 24,5 cm • ISBN: 978-85-365-0198-7 • EAN: 9788536501987

Objetiva, didática e interativa, esta obra fornece um estudo do programa gerenciador de banco de dados PostgreSQL em sua versão 8.3.0, para Windows.

Apresenta o histórico da linguagem de consulta estruturada SQL e do PostgreSQL. Descreve a aquisição e instalação do programa, manuseio, criação e remoção de bancos de dados, tabelas, consultas, uso de funções, agrupamentos, uniões, junções e visualização de dados, gerenciamento de usuários, chaves primária e estrangeira, rotinas armazenadas (funções e gatilhos) e ferramentas de administração.

Possui uma série de atividades de manuseio do programa e exercícios de fixação e é indispensável aos iniciantes na administração de banco de dados.



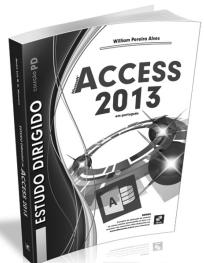
Oracle Database 10g Express Edition - Guia de Instalação, Configuração e Administração com Implementação PL/SQL Relacional e Objeto-Relacional

Autor: Robson Soares Silva

Código: 1628 • 240 páginas • Formato: 17 x 24 cm • ISBN: 978-85-365-0162-8 • EAN: 9788536501628

O livro detalha os principais recursos desse poderoso banco de dados, como a aquisição e instalação do programa, criação de tabelas, chaves primária e estrangeira, consultas SQL, utilização de índices, usuários do sistema, criação de usuários, concessão e restrição de direitos de acesso, programação PL/SQL, procedures, functions, packages, triggers, implementação objeto-relacional, criação de aplicações, acesso remoto, backup e restore, flashback, níveis de certificação e conexão Oracle com o Java.

Possui também diversos exercícios propostos, além de um simulado com questões para certificação.



Estudo Dirigido de Microsoft Access 2013

Autor: William Pereira Alves

Código: 4605 • 272 páginas • Formato: 17,5 x 24,5 cm • ISBN: 978-85-365-0460-5 • EAN: 9788536504605

Com estilo agradável, este livro trata dos principais recursos do Access 2013 para o desenvolvimento de aplicações de banco de dados. O aprendizado é complementado com exercícios e exemplos, incluindo um projeto de aplicativo.

Conceitos e fundamentos de bancos de dados relacionais abrem o estudo. Em seguida, explicam-se criação e alteração de tabelas; manipulação de registros do banco de dados (inserção, alteração e exclusão); criação, alteração e execução de consulta; construção e uso de formulários de entrada de dados, relatórios e etiquetas de endereçamento; elaboração de macros e desenvolvimento de rotinas em Visual Basic for Applications (VBA), para automatizar tarefas. Ao final, ensina-se como trocar informações do Access 2013 com o Word 2013 e o Excel 2013.

A leitura é recomendada a estudantes, profissionais e demais interessados na área.



Banco de Dados - Projeto e Implementação - Edição Revisada

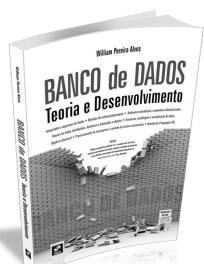
Autor: Felipe Nery Rodrigues Machado

Código: 0190 • 400 páginas • Formato: 17 x 24 cm • ISBN: 978-85-365-0019-5 • EAN: 9788536500195

Com uma apresentação diferenciada, utilizando metodologia e conceitos embasados na execução prática de projetos de banco de dados, o livro traz a experiência do autor, com uma linguagem simples e objetiva, que permite a compreensão das técnicas de forma gradativa e efetiva.

Destaca aspectos conceituais, a orientação à gestão de negócios e aborda a utilização de álgebra relacional, mapeamento OO → ER no projeto de banco de dados, comandos básicos da linguagem SQL ANSI, explanados com exemplos e estudos de caso.

A segunda reimpressão da segunda edição foi revisada e atualizada para Microsoft SQL 2008 e Oracle 10g.



Banco de Dados - Teoria e Desenvolvimento

Autor: William Pereira Alves

Código: 2557 • 288 páginas • Formato: 17,5 x 24,5 cm • ISBN: 978-85-365-0255-7 • EAN: 9788536502557

Didático e interativo, este livro aborda os principais conceitos e fundamentos de bancos de dados, como os tipos de acesso a arquivos, estruturas de dados (listas lineares e árvore), métodos de ordenação (Bubble Sort, QuickSort e Shell), arquitetura e organização, álgebra relacional, uso de índices, modelagem e normalização de dados, os cuidados necessários à implementação de um banco de dados, conceito e estrutura de Data Warehouse (OLAP, OLTP, Data Mining e Data Mart), bancos de dados distribuídos, dedutivos, hierárquicos, de rede e orientados a objetos (ODL e OQL), segurança e proteção com níveis de acesso dos usuários, processamento de transações e princípios da linguagem SQL.