

NOVA EDIÇÃO ATUALIZADA

C++ COMO PROGRAMAR

— 5^a Edição —

Antecipando
INTRODUÇÃO
A CLASSES
+ OBJETOS
COM UML™ 2

RING!



INTRODUÇÃO ANTECIPADA A CLASSES/OBJETOS/POO

- CLASSES, OBJETOS, ENCAPSULAMENTO
- HERANÇA, POLIMORFISMO
- ESTUDOS DE CASO POO INTEGRADOS:
TIME, GRADEBOOK, EMPLOYEE

FUNDAMENTOS

- HISTÓRIA, HARDWARE, SOFTWARE
- E/S DE FLUXO, TIPOS, OPERADORES
- INSTRUÇÕES DE CONTROLE, FUNÇÕES
- ARRAYS, VETORES
- PONTEIROS, REFERÊNCIAS
- CLASSE STRING, STRINGS NO ESTILO C
- SOBRECARGA DE OPERADOR
- EXCEÇÕES, ARQUIVOS
- PROGRAMAÇÃO WEB
- MANIPULAÇÃO DE BIT E CARACTERES
- DEPURADORES GNU™ C++/VISUAL C++®

ESTRUTURAS DE DADOS

- RECURSAO, PESQUISA, CLASSIFICAÇÃO
- LISTAS, FILAS, PILHAS, ÁRVORES
- TEMPLATES
- STANDARD TEMPLATE LIBRARY:
CONTÊINERES, ITERADORES E ALGORITMOS

ESTUDO DE CASO OOD/UML™ 2 ATM (OPCIONAL)

- DETERMINAR CLASSES, ATRIBUTOS, ESTADOS, ATIVIDADES, OPERAÇÕES, COLABORAÇÕES
- DIAGRAMAS: CASO DE USO, CLASSE, ESTADO, ATIVIDADE, COMUNICAÇÃO, SEQUÊNCIA

DEITEL
DEITEL

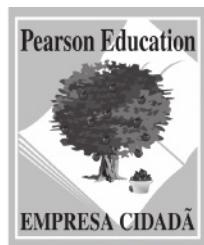
PEARSON

DEITEL®



O CD inclui as versões de exemplo de código aberto (free-code) em C++ do livro, links para compiladores e ferramentas de desenvolvimento em C++, centrais de recursos na Web, incluindo referências gerais, tutoriais, FAQs, newsgroups e informações sobre STL.

C COMO ++ PROGRAMAR — 5^a Edição —



C++ COMO PROGRAMAR — 5^a Edição —

H. M. Deitel
Deitel & Associates, Inc.

P. J. Deitel
Deitel & Associates, Inc.

Tradução
Edson Furmankiewicz

Revisão técnica
Fábio Luis Picelli Lucchini
Bacharel em Ciência da Computação pela Faculdade de Tecnologia Padre Anchieta
Mestre em Engenharia da Computação pela Unicamp (FEEC)
Professor Universitário da Faculdade de Tecnologia Padre Anchieta

PEARSON

abdr 
ASSOCIAÇÃO
BRASILEIRA
DE EDITORES
REPRODUTORES
Respeite o direito autoral

© 2006 by Pearson Education do Brasil

© 2006 by Pearson Education, Inc.

Tradução autorizada a partir da edição original em inglês C++, How to Program 5th edition de Deitel, Harvey; Deitel, Paul, publicada pela Pearson Education, Inc., sob o selo Pearson Prentice Hall.

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

Gerente editorial: Roger Trimer

Editora sênior: Sabrina Cairo

Editora de desenvolvimento: Marileide Gomes

Editora de texto: Fábio Pessotti

Preparação: Andréa Filato

Revisão: Silvana Gouveia e Hebe Ester Lucas

Capa: Alexandre Mieda (sobre projeto original)

Projeto gráfico e diagramação: Figurativa Arte e Projeto Editorial

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

Deitel, H.M., 1945- .
C++ : como programar / H.M. Deitel, P.J. Deitel ;
tradução Edson Furmarkiewicz ; revisão técnica Fábio Lucchini.
— São Paulo: Pearson Prentice Hall, 2006.

Título original: C++ how to program
Bibliografia.
ISBN 978-85-4301-373-2

1. C++ (Linguagem de programação para
computadores) I. Deitel, Paul J. II. Título.

05-9603

CDD-005.133

Índices para catálogo sistemático:

1. C++ : Linguagem de programação : Computadores :
Processamento de dados 005.133

8^a reimpressão – Abril 2015
Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil Ltda.,
uma empresa do grupo Pearson Education
Rua Nelson Francisco, 26
CEP 02712-100 – São Paulo – SP – Brasil
Fone: 11 2178-8686 – Fax: 11 2178-8688
e-mail: vendas@pearson.com

Marcas Comerciais

Borland e C++ Builder são marcas comerciais notórias ou marcas comerciais registradas da Borland.

Cygwin é uma marca comercial e uma obra com direitos autorais protegidos pela Red Hat, Inc. nos Estados Unidos e em outros países.

Dive Into é uma marca comercial registrada da Deitel & Associates, Inc.

GNU é uma marca comercial notória da Free Software Foundation.

Java e todas as marcas baseadas em Java são marcas comerciais notórias ou marcas comerciais registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países. A Pearson Education é uma empresa independente da Sun Microsystems, Inc.

Linux é uma marca comercial registrada de Linus Torvalds.

Microsoft, Microsoft® Internet Explorer e o logotipo Windows são marcas comerciais registradas ou marcas comerciais notórias da Microsoft Corporation nos Estados Unidos e/ou em outros países.

Janela de navegador Netscape © 2004 Netscape Communications Corporation. Utilizada com permissão. A Netscape Communications não autorizou, patrocinou, endossou ou aprovou esta publicação e não é responsável pelo seu conteúdo.

Object Management Group, OMG, Unified Modeling Language e UML são marcas comerciais registradas do Object Management Group, Inc.

Dedicató

Para:

Stephen Clamage

Presidente do comitê J16, 'Programming Language C++' que é responsável pelo padrão C++; engenheiro sênior de pessoal, Sun Microsystems, Inc., Divisão de Software.

Don Kostuch

Consultor independente

e Mike Miller

Ex-vice-presidente Executivo do Comitê de Design das Working Groups de Design C e C++, Presidente da J16, por sua orientação, amizade e incansável devoção em insistir que nós fizéssemos 'a coisa certa' e por nos ajudar a fazer isso.

É um privilégio trabalhar com profissionais C++ tão competentes.

Harvey M. Deitel e Paul J. Deitel

Sumário

Prefácio	xxi
Antes de você começar	xxxix
1 Introdução aos computadores, à Internet e à World Wide Web.....	1
1.1 O que é um computador?.....	2
1.3 Organização do computador	3
1.4 Primeiros sistemas operacionais.....	4
1.5 Computação pessoal distribuída e computação cliente/servidor	4
1.6 A Internet e a World Wide Web	4
1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível	5
1.8 História do C e do C++	6
1.9 C++ Standard Library	6
1.10 História do Java.....	7
1.11 Fortran, COBOL, Ada e Pascal	7
1.12 Basic, Visual Basic, Visual C++, C# e .NET	8
1.13 Tendência-chave do software: tecnologia de objeto	8
1.14 Ambiente de desenvolvimento C++ típico	9
1.15 Notas sobre o C++ e este livro	11
1.16 Test-drive de um aplicativo C++.....	11
1.17 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML (obrigatório).....	16
1.18 Síntese	20
1.19 Recursos na Web	20
2 Introdução à programação em C+.....	26
2.1 Introdução	27
2.2 Primeiro programa C++: imprimindo uma linha de texto	27
2.3 Modificando nosso primeiro programa C++	30
2.4 Outro programa C++: adicionando inteiros	31
2.5 Conceitos de memória	34
2.6 Aritmética.....	35
2.7 Tomada de decisão: operadores de igualdade e operadores relacionais.....	37
2.8 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)	41
2.9 Síntese	48
3 Introdução a classes e objetos	55
3.1 Introdução	56
3.2 Classes, objetos, funções-membro e membros de dados	56
3.3 Visão geral dos exemplos do capítulo	57
3.4 Definindo uma classe com uma função-membro	57
3.5 Definindo uma função-membro com um parâmetro	60

3.6 Membros de dados, funções e funções 62
3.7 Inicializando objetos com construtores 67
3.8 Colocando uma classe em um arquivo separado para reusabilidade 70
3.9 Separando a interface da implementação 73
3.10 Validando dados com funções 78
3.11 Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional) 82
3.12 Síntese 87
4 Instruções de controle: parte 1 93
4.1 Introdução 94
4.2 Algoritmos 94
4.3 Pseudocódigo 94
4.4 Estruturas de controle 95
4.5 Instrução de seleção 98
4.6 A instrução de seleção <code>if/else</code> 99
4.7 A instrução de repetição 103
4.8 Formulando algoritmos: repetição controlada por contador 104
4.9 Formulando algoritmos: repetição controlada por sentinelas 109
4.10 Formulando algoritmos: instruções de controle aninhadas 116
4.11 Operadores de atribuição 120
4.12 Operadores de incremento e decremento 121
4.13 Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional) 123
4.14 Síntese 126
5 Instruções de controle: parte 2 140
5.1 Introdução 141
5.2 Princípios básicos de repetição controlada por contador 141
5.3 A instrução de repetição 143
5.4 Exemplos com a estrutura 146
5.5 Instrução de repetição <code>do/while</code> 150
5.6 A estrutura de seleção múltipla 151
5.7 Instruções <code>break</code> e <code>continue</code> 159
5.8 Operadores lógicos 161
5.9 Confundindo operadores de igualdade com operadores de atribuição 164
5.10 Resumo de programação estruturada 165
5.11 Estudo de caso de engenharia de software: identificando estados e atividades dos objetos no sistema ATM (opcional) 169
5.12 Síntese 173
6 Funções e uma introdução à recursão 181
6.1 Introdução 182
6.2 Componentes de um programa em C++ 183
6.3 Funções da biblioteca de matemática 184
6.4 Definições de funções com múltiplos parâmetros 185
6.5 Protótipos de funções e coerção de argumentos 189
6.6 Arquivos de cabeçalho da biblioteca padrão C++ 190
6.7 Estudo de caso: geração de números aleatórios 190
6.8 Estudo de caso: jogo de azar e introdução a 196

6.9	Classes de armazenamento	200
6.10	Regras de escopo	202
6.11	Pilha de chamadas de função e registros de ativação	204
6.12	Funções com listas de parâmetro vazias	206
6.13	Funções inline	208
6.14	Referências e parâmetros de referência	209
6.15	Argumentos-padrão	213
6.16	Operador de solução de escopo.unário.....	215
6.17	Sobrecarga de funções	216
6.18	Templates de funções	218
6.19	Recursão	221
6.20	Exemplos que utilizam recursão:.série.de.Fibonacci.....	226 222
6.22	Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)	228
6.23	Síntese	233
7	Arrays e vetores	250
7.1	Introdução	251
7.2	Arrays	251
7.3	Declarando arrays	252
7.4	Exemplos que utilizam arrays	253
7.5	Passando arrays para funções	267
7.6	Estudo de caso: <code>classmateBook</code> utilizando um array para armazenar notas	271
7.7	Pesquisando arrays com pesquisa linear	276
7.8	Classificando arrays por inserção	276
7.9	Arrays multidimensionais	279
7.10	Estudo de caso: <code>classmateBook</code> utilizando um array bidimensional	282
7.11	Introdução ao temporizador da C++ Standard Library	286
7.12	Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional)	292
7.13	Síntese	297
8	Ponteiros e strings baseadas em ponteiro	311
8.1	Introdução	312
8.2	Declarações de variável ponteiro e inicialização	312
8.3	Operadores de ponteiro	313
8.4	Passando argumentos para funções por referência com ponteiros	315
8.5	Utilizando <code>const</code> com ponteiros	319
8.6	Classificação por seleção utilizando passagem por referência	324
8.7	Operadores <code><</code> <code>></code> <code>eof</code>	327
8.8	Expressões e aritmética de ponteiro	330
8.90	Relações entre ponteiros e arrays.....	3352
8.11	Estudo de caso: simulação de embaralhamento e distribuição de cartas	336
8.12	Ponteiros de função	341
8.13	Introdução ao processamento de string baseada em ponteiro.....	345
8.13.1	Fundamentos de caracteres e strings baseadas em ponteiro.....	346
8.13.2	Funções de manipulação de string da biblioteca de tratamento. <code>de.string</code>	347
8.14	Síntese	354

9 Classes: um exame mais profundo, parte 1	375
9.1 Introdução	376
9.2 Estudo de caso da classe	376
9.3 Escopo de classe e acesso a membros de classe	382
9.4 Separando a interface da implementação	382
9.5 Funções de acesso e funções utilitárias	384
9.6 Estudo de caso da classe construtores com argumentos-padrão	386
9.7 Destrutores	391
9.8 Quando construtores e destrutores são chamados	392
9.9 Estudo de caso da classe uma armadilha sutil — retornar uma referência a um membro de dado	395
9.10 Atribuição padrão de membro a membro	399
9.12 Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)	399
9.13 Síntese	405
10 Classes: um exame mais profundo, parte 2	411
10.1 Introdução	412
10.2 Objetos const (constante) e funções-membro	412
10.3 Composição: objetos como membros de classes	421
10.4 Funções friend e classes friend	426
10.5 Utilizando o ponteiro	429
10.6 Gerenciamento de memória dinâmico com os operadores	430
10.7 Membros de classe static	435
10.8 Abstração de dados e ocultamento de informações	440
10.8.2 Exemplo: tipo de dados abstrato array	441
10.8.3 Exemplo: tipo de dados abstrato queue	442
10.9 Classes contêineres e iteradores	442
10.10 Classes proxy	442
10.11 Síntese	445
11 Sobrecarga de operadores; objetos string.e.array.....	450
11.1 Introdução	451
11.2 Fundamentos de sobrecarga de operadores	451
11.3 Restrições à sobrecarga de operadores	452
11.4 Funções operadoras como membros de classes globais	454
11.5 Sobrecarregando operadores de inserção e extração de fluxo	455
11.6 Sobrecarregando operadores unários	457
11.7 Sobrecarregando operadores binários	458
11.8 Estudo de caso: classarray	458
11.9 Convertendo entre tipos	469
11.9.1 Estudo de caso: classstring	469
11.9.2 Sobrecarregando	479
11.9.3 Estudo de caso: uma classe	480
11.9.4 Classe string da biblioteca-padrão	484
11.9.5 Construtores explicit	487
11.10 Síntese	490

12 Programação orientada a objetos: herança	501
12.1 Introdução	502
12.2 Classes básicas e derivadas	503
12.3 Membros <code>protected</code>	505
12.4 Relacionamento entre classes básicas e derivadas	505
12.4.1 Criando e utilizando uma classe <code>CommissionEmployee</code>	505
12.4.2 Criando uma classe <code>BasePlusCommissionEmployee</code> para utilizar herança	510
12.4.3 Criando uma hierarquia de herança <code>CommissionEmployee</code> – <code>BasePlusCommissionEmployee</code>	515
12.4.4 Hierarquia de heranças entre <code>CommissionEmployee</code> – <code>BasePlusCommissionEmployee</code> utilizando dados <code>protected</code>	519
12.4.5 Hierarquia de heranças entre <code>CommissionEmployee</code> – <code>BasePlusCommissionEmployee</code> utilizando dados <code>private</code>	525
12.5 Construtores e destrutores em classes derivadas	532
12.6 Herança <code>public</code> , <code>protected</code> e <code>private</code>	539
12.7 Engenharia de software com herança.....	540
12.8 Síntese	541
13 Programação orientada a objetos: polimorfismo.....	545
13.1 Introdução	546
13.2 Exemplos de polimorfismo	547
13.3 Relacionamentos entre objetos em uma hierarquia de herança.....	548
13.3.1 Invocando funções de classe básica a partir de objetos de classe derivada	548
13.3.2 Apontando ponteiros de classe derivada para objetos da classe básica	554
13.3.3 Chamadas de função-membro de classe derivada via ponteiros de classe básica	555
13.3.4 Funções virtuais.....	557
13.3.5 Relacionamentos de herança e polimorfismo entre objetos de classe básica	562
13.4 Campos de tipo e instruções	563
13.5 Classes abstratas e funções puras	563
13.6 Estudo de caso: sistema de folha de pagamento utilizando polimorfismo	564
13.6.1 Criando a classe básica <code>Employee</code>	565
13.6.2 Criando a classe derivada <code>CommissionEmployee</code>	566
13.6.3 Criando a classe derivada <code>ConcreteEmployee</code>	570
13.6.4 Criando a classe derivada <code>ConcreteCommissionEmployee</code>	570
13.6.5 Criando a classe derivada concreta <code>BasePlusCommissionEmployee</code>	574
13.6.6 Demonstrando o processamento polimórfico	574
13.7 Polimorfismo, funções <code>virtual</code> e vinculação dinâmica ‘sob o capô’ (opcional)	579
13.8 Estudo de caso: sistema de folha de pagamento utilizando polimorfismo e informações de tipo de tempo de execução com <code>downcasting</code> , <code>casttypeid</code> e <code>type_info</code>	582
13.9 Destrutores virtuais	585
13.10 Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional)	585
13.11 Síntese	591
14 Templates	595
14.1 Introdução	596
14.2 Templates de funções	596
14.3 Sobrecarregando templates de função	599
14.4 Templates de classe	599
14.5 Parâmetros sem tipo e tipos-padrão para templates de classes	605

14.6 Notas sobre templates e herança	606
14.7 Notas sobre templates e friends	606
14.8 Notas sobre templates e membros	606
14.9 Síntese	607
15 Entrada/saída de fluxo	611
15.1 Introdução	612
15.2 Fluxos	613
15.2.1 Fluxos clássicos versus fluxos-padrão	613
15.2.2 Arquivos de cabeçalho da biblioteca	613
15.2.3 Classes de entrada/saída de fluxo e objetos	614
15.3 Saída de fluxo de variáveis	615
15.3.2 Saída de caractere utilizando a função <code>membro</code>	616
15.4 Entrada de fluxo	616
15.4.1 Funções-membro <code>getline</code>	617
15.4.2 Funções-membro <code>peek</code> , <code>putback</code> e <code>ignore</code> de <code>istream</code>	619
15.4.3 E/S fortemente tipada (<code>safe</code>)	619
15.5 E/S não formatada utilizando <code>read</code> e <code>write</code>	620
15.6 Introdução aos manipuladores de fluxos	620
15.6.1 Base de fluxo integrado (<code>oct</code> , <code>hex</code> e <code>setbase</code>)	621
15.6.2 Precisão de ponto flutuante (<code>precision</code> , <code>setprecision</code>)	621
15.6.3 Largura de campo (<code>setw</code> , <code>setw</code>)	622
15.6.4 Manipuladores de fluxo de saída definidos pelo usuário	624
15.7 Estados de formato de fluxo e manipuladores de fluxo	624
15.7.1 Zeros finais e pontos de fração (<code>decimals(n)</code>)	626
15.7.2 Alinhamento (<code>left</code> , <code>right</code> e <code>internal</code>)	626
15.7.3 Preenchimento (<code>fill</code> , <code>setfill</code>)	628
15.7.4 Base de fluxo integrado (<code>oct</code> , <code>hex</code> , <code>showbase</code>)	630
15.7.5 Números de ponto flutuante; notação científica (<code>scientific</code> , <code>fixed</code>)	631
15.7.6 Controle de letras maiúsculas/minúsculas (<code>supticks</code>)	631
15.7.7 Especificando o formato booleano (<code>boolalpha</code>)	632
15.7.8 Configurando e redefinindo o estado de formato via a função <code>membro</code>	633
15.8 Estados de erro de fluxo	634
15.9 Associando um fluxo de saída a um fluxo de entrada	636
15.10 Síntese	636
16 Tratamento de exceções	644
16.1 Introdução	645
16.2 Visão geral do tratamento de exceções	645
16.3 Exemplo: tratando uma tentativa de divisão por zero	646
16.4 Quando utilizar o tratamento de exceções	650
16.5 Revisando o tratamento de exceção	653
16.7 Processando exceções inesperadas	653
16.8 Desempilhamento de pilha	654
16.9 Construtores, destrutores e tratamento de exceções	654
16.10 Exceções e herança	654
16.11 Processando falhas	656
16.12 Classe <code>auto_ptr</code> e alocação de memória dinâmica	659
16.13 Hierarquia de exceções da biblioteca-padrão	661

16.14 Outras técnicas de tratamento de erro	662
16.15 Síntese	663
17 Processamento de arquivo	668
17.1 Introdução	669
17.2 Hierarquia de dados	669
17.3 Arquivos e fluxos	671
17.4 Criando um arquivo seqüencial	671
17.5 Lendo dados de um arquivo seqüencial	674
17.6 Atualizando arquivos seqüenciais	680
17.7 Arquivos de acesso aleatório	680
17.8 Gravando dados aleatoriamente em um arquivo de acesso aleatório	681
17.10 Lendo um arquivo de acesso aleatório seqüencialmente	687
17.11 Estudo de caso: um programa de processamento de transação	689
17.12 Entrada/saída de objetos	695
17.13 Síntese	696
18 Classe string e processamento de fluxo de string.....	703
18.1 Introdução	704
18.2 Atribuição e concatenação de strings	705
18.3 Comparando strings	707
18.4 Substrings	709
18.5 Trocando strings	709
18.6 Características de strings	710
18.7 Localizando strings e caracteres em strings	712
18.8 Substituindo caracteres em uma string	714
18.9 Inserindo caracteres em strings	716
18.10 Conversão para strings * baseadas em ponteiro no estilo C	717
18.11 Iteradores	718
18.12 Processamento de fluxo de string	719
18.13 Síntese	722
19 Programação Web	727
19.1 Introdução	728
19.2 Tipos de solicitação HTTP	728
19.3 Arquitetura de múltiplas camadas	729
19.4 Acessando servidores Web	730
19.5 Apache HTTP Server	731
19.6 Solicitando documentos XHTML	731
19.7 Introdução à CGI	731
19.8 Transações HTTP simples	732
19.9 Scripts CGI simples	733
19.10 Envmando entradas para um script CGI	737
19.11 Utilizando formulários XHTML para enviar entrada	741
19.12 Outros cabeçalhos	745
19.13 Estudo de caso: uma página Web interativa	748
19.14 Cookies	752
19.15 Arquivos do lado do servidor	755
19.16 Estudo de caso: carrinho de compras	761

19.17 Síntese	773
19.18 Recursos na Internet e na Web	775
20 Pesquisa e classificação	780
20.1 Introdução	781
20.2 Algoritmos de pesquisa.....	782
20.2.1 Eficiência da pesquisa linear.....	782
20.2.2 Pesquisa binária.....	783
20.3 Algoritmos de classificação	787
20.3.1 Eficiência da classificação por seleção	787
20.3.2 Eficiência da classificação por inserção	788
20.4 Síntese	788
21 Estruturas de dados	798
21.1 Introdução	799
21.2 Classes auto-referenciais.....	799
21.3 Alocação de memória e estruturas de dados dinâmicas	800
21.4 Listas vinculadas.....	800
21.5 Pilhas	813
21.6 Filas	817
21.7 Árvores	820
21.8 Síntese	828
22 Bits, caracteres, strings e structures	846
22.1 Introdução	847
22.2 Definições de estruturas	847
22.4 Utilizando estruturas com funções	849
22.5 <code>typedef</code>	849
22.6 Exemplo: simulação de embaralhamento e distribuição de cartas de alto desempenho	850
22.7 Operadores de bits	852
22.8 Campos de bit.....	859
22.9 Biblioteca de tratamento de caractere	863
22.10 Funções de conversão de string baseada em ponteiro	868
22.11 Funções de pesquisa da biblioteca de tratamento de strings baseadas em ponteiro	873
22.12 Funções de memória da biblioteca de tratamento de strings baseadas em ponteiro	874
22.13 Síntese	881
23 Standard Template Library (STL)	890
23.1 Introdução à Standard Template Library (STL)	891
23.1.1 Introdução aos contêineres.....	893
23.1.2 Introdução aos iteradores	896
23.1.3 Introdução aos algoritmos	899
23.2 Contêineres de seqüênci.....	901
23.2.1 Contêiner de seqüênci.....	902
23.2.2 Contêiner de seqüênci.....	908
23.2.3 Contêiner de seqüênci.....	911
23.3 Contêineres associativos	913
23.3.1 Contêiner associativo multiset	913

23.3.2 Contêiner associativo	915
23.3.3 Contêiner associativo multimap	915
23.3.4 Contêiner associativo map	918
23.4 Adaptadores de contêiner	919
23.4.1 Adaptador stack	919
23.4.2 Adaptador queue	921
23.4.3 Adaptador priority_queue	922
23.5 Algoritmos	924
23.5.1 fill , fill_n , generate e generate_n	924
23.5.2 equal, mismatch e lexicographical_compare	926
23.5.3 remove, remove_if, remove_copy e remove_copy_if	927
23.5.5 Algoritmos de itátore replace, copy e replace_copy	930..... 932
23.5.6 Algoritmos de pesquisa e classificação básica	934
23.5.7 swap, iter_swap e swap_ranges	937
23.5.8 copy_backward, merge, unique e reverse	938
23.5.9 inplace_merge, unique_copy e reverse_copy	940
23.5.10 Operações set	940
23.5.11 lower_bound, upper_bound e equal_range	943
23.5.12 Heapsort	945
23.5.13 min e max	948
23.5.14 Algoritmos de STL não discutidos neste capítulo	948
23.6 Classe bitset	950
23.7 Objetos de função	953
23.8 Síntese	955
23.9 Recursos sobre C++ na Internet e na Web	955
24 Outros tópicos	963
24.1 Introdução	964
24.2 Operador const_cast	964
24.3 namespace	965
24.4 Palavras-chave de operador	968
24.5 Membros de classe stable	970
24.6 Ponteiros para membros de classe ()	972
24.7 Herança múltipla	973
24.8 Herança múltipla e classes básicas	976
24.9 Síntese	981
24.10 Observações finais	982
A Tabela de precedência e associatividade de operadores	986
Precedência de operadores	986
B Conjunto de caracteres ASCII	988
C Tipos fundamentais	989
D Sistemas de numeração	991
D.1 Introdução	992
D.2 Abreviando números binários como números octais e números hexadecimais	993
D.3 Convertendo números octais e hexadecimais em números binários	995

D.4	Convertendo de octal, binário ou hexadecimal para decimal.....	995
D.5	Convertendo de decimal para octal, binário ou hexadecimal.....	996
D.6	Números binários negativos: notação de complemento de dois	997
E	Tópicos sobre o código C legado	1001
E.1	Introdução	1002
E.2	Redirecionando entrada/saída em sistemas UNIX/Linux/Mac OS X e Windows	1002
E.3	Listas de argumentos de comprimento variável	1003
E.4	Utilizando argumentos de linha de comando	1005
E.5	Notas sobre a compilação de programas de arquivo de múltiplas fontes.....	1005
E.6	Terminação de programa com exit	1007
E.7	Qualificador de tipo long	1009
E.8	Quotas para constantes de inteiro e de ponto flutuante.....	1009
E.9	Tratamento de sinal	1009
E.10	Alocação dinâmica de memória com realloc	1011
E.11	O desvio incondicional	1012
E.12	Uniões	1012
E.13	Especificações de linkagem	1016
E.14	Síntese	1016
F	Pré-processador	1021
F.1	Introdução	1022
F.2	A diretiva de pré-processador #include	1022
F.3	Diretiva de pré-processador #define: constantes simbólicas	1022
F.4	Diretiva de pré-processador #define: macros	1023
F.5	Compilação condicional	1024
F.6	As diretivas de pré-processador e	1025
F.7	Os operadores # e ###error#pragma	1025
F.8	Constantes simbólicas predefinidas	1026
F.9	Assertivas	1026
F.10	Síntese	1026
G	Código para o estudo de caso do ATM	1030
G.1	Implementação do estudo de caso ATM	1030
G.2	ClassATM.....	1031
G.3	ClassScreen.....	1037
G.4	ClassKeypad.....	1037
G.5	ClassCashDispenser.....	1038
G.6	ClassDepositSlot	1039
G.7	ClassAccount	1040
G.8	ClassBankDatabase.....	1042
G.9	ClassTransaction	1045
G.10	ClassBalanceInquiry	1048
G.12	ClassWithdraw	1050
G.13	Programa de teste ATMCaseStudy.cpp	1055
G.14	Síntese	1056
H	UML 2: tipos de diagramas adicionais	1057
H.1	Introdução	1057

H.2 Tipos de diagramas adicionais	1057
I Recursos sobre C++ na Internet e na Web.....	1059
I.1 Recursos	1059
I.2 Tutoriais.....	1060
I.3 FAQs.....	1061
I.4 Visual C++	1061
I.5 Newsgroups.....	1061
I.6 Compiladores e ferramentas de desenvolvimento.....	1061
I.7 Standard Template Library	1062
J.1 Introdução à XHTML.....	1063
J.2 Editando XHTML	1064
J.3 Primeiro exemplo de XHTML	1064
J.4 Cabeçalhos	1066
J.5 Vinculando	1068
J.6 Imagens	1070
J.7 Caracteres especiais e mais quebras de linha.....	1074
J.8 Listas não ordenadas.....	1075
J.9 Listas aninhadas e listas ordenadas.....	1077
J.10 Tabelas de XHTML básicas.....	1077
J.11 Tabelas e formatação de XHTML intermediárias.....	1081
J.12 Formulários XHTML básicos	1083
J.13 Formulários XHTML mais complexos	1086
J.14 Recursos na Internet e na Web	1092
K Caracteres especiais de XHTML.....	1096
L Utilizando o depurador do Visual Studio .NET	1097
L.1 Introdução	1098
L.2 Pontos de interrupção e o comando Continue	1098
L.3 As janelas Local e Watch	1103
L.4 Controlando a execução utilizando os comandos Step Into , Step Over , Step Out e Continue	1105
L.5 A janela Autos	1108
L.6 Síntese	1109
M Utilizando o depurador do GNU C++	1111
M.1 Introdução	1112
M.2 Pontos de interrupção e os comandos step , continue e eprint	1112
M.3 Os comandos print e set	1118
M.4 Controlando a execução utilizando os comandos next , finish e next	1119.....1121
M.6 Síntese	1124
Bibliografia	1027
Índice remissivo.....	1131

Prefácio

"O principal mérito da língua é a clareza ..." Galeno

Bem-vindo ao **C++ e como programar**, Quinta Edição. É uma linguagem de programação de primeira classe para desenvolver aplicativos de computador de capacidade industrial e de alto desempenho. Acreditamos que este livro e o material de suíto aquilo de que os instrutores e alunos precisam para uma experiência educacional em C++ informativa, interessante, desejada divertida. Neste prefácio, fornecemos uma visão geral dos **Novos recursos** da Quinta Edição.

"Passeio pelo livro" deste prefácio apresenta aos instrutores, alunos e profissionais como programar o conteúdo do livro, como os recursos para destaque dos exemplos de código, 'limpeza de código' e destaque de código. Apresentamos informações sobre compiladores livres/gratuitos que podem ser encontrados na Web. Discutimos também o abrangente conjunto de materiais que ajudam os instrutores a maximizar a experiência de aprendizagem de seus alunos (disponível no site em inglês).

Recursos do C++ Como programar, Quinta Edição

Na Deitel & Associates, escrevemos livros-texto de ciência da computação de nível universitário e livros profissionais. Para criar **Como programar**, Quinta Edição, colocamos a edição anterior de **Como programar sob o microscópio**. A nova edição tem muitos recursos atraentes:

- Revisões importantes do **Conteúdo**: capítulos foram significativamente reescritos e atualizados. Adaptamos o texto para maior clareza e precisão. Também adequamos nosso uso da terminologia do C++ ao documento-padrão do ANSI/ISO que define a linguagem.
- Capítulos menores: capítulos maiores foram divididos em capítulos menores, mais gerenciáveis (por exemplo, o Capítulo 1 da Quarta Edição foi dividido nos capítulos 1–2; o Capítulo 2 da Quarta Edição transformou-se nos capítulos 4–5).
- Abordagem que introduz classes e objetos a **Média**: uma parte um método didático que introduz classes e objetos antecipadamente no currículo. Os conceitos básicos e a terminologia da tecnologia de objeto são apresentados ao longo do Capítulo 1. Na edição anterior, os alunos começavam a desenvolver classes e objetos personalizados reutilizáveis somente no Capítulo 6, mas, nesta edição, eles fazem isso já no Capítulo 3, que é inteiramente novo. Os capítulos 4–7 foram cuidadosamente reescritos a partir de uma perspectiva de 'introdução a classes e objetos'. Esta nova edição é orientada a objeto, onde aparece desde o início e por todo o texto. Deslocar a discussão sobre objetos e classes para capítulos anteriores faz com que os 'pensem sobre objetos' imediatamente e dominem esses conceitos mais completamente. A programação orientada a objeto é de modo algum trivial, mas é divertido escrever programas orientados a objetos, e os alunos podem ver resultados imediatamente.
- Estudos de caso integrados: adicionamos diversos estudos de caso distribuídos por múltiplas seções e capítulos que, com freqüência, baseiam-se em uma classe introduzida anteriormente no livro para demonstrar novos conceitos de programação nos capítulos posteriores. Esses estudos de caso incluem o **desenvolvimento de software** das seções de **classes** e **atividades** nas várias seções dos capítulos 9–13, e os capítulos 12–13 e o estudo de caso opcional ATM com OOD/UML nos capítulos 1–7, 9, 13 e Apêndice G.
- Estudo de caso integrado: adicionamos um novo estudo de caso para reforçar nossa apresentação de classes anteriores. Ele utiliza classes e objetos nos capítulos 3–7 para construir visualmente uma classe que representa o livro de notas de um instrutor e realiza vários cálculos com base em um conjunto de notas de alunos, como cálculo de médias, determinação das notas máximas e mínimas e impressão de um gráfico de barras.
- Unified Modeling Language™ 2.0 (UML 2.0) — Introduzido: Unified Modeling Language (UML) tornou-se a linguagem de modelagem gráfica preferida dos projetistas de sistemas orientados a objetos. Todos os diagramas da UML neste livro obedecem à nova especificação da UML 2.0. Utilizamos diagramas de classes UML para representar visualmente classes e seus relacionamentos de herança, e usamos diagramas de atividades UML para demonstrar o fluxo de controle em cada instrução de controle do C++. Fazemos uso especialmente intenso da UML no estudo de caso opcional ATM com OOD/UML.
- Estudo de caso opcional ATM com **OOD/UML**: o estudo de caso opcional de simulador de elevador da edição anterior por um novo estudo de caso opcional de **automata eletrônico** (ATM) com OOD/UML nas seções "Estudo de caso de engenharia de software" dos capítulos 1–7, 9 e 13. O novo estudo de caso é mais simples, menor, mais 'real' e mais apropriado aos cursos de programação de primeiro e segundo anos. As nove seções de estudo de caso apresentam uma cuidadosa introdução passo a passo ao projeto orientado a objetos utilizando a UML. Introduzimos um subconjunto simplificado e conciso da UML 2.0 e então guiamos o leitor no seu primeiro projeto concebido para projetistas e programadores iniciantes.

em orientação a objetos. Nossa objetivo nesse estudo de caso é ajudar alunos a desenvolver um projeto orientado a objetos complementar os conceitos de programação orientada a objetos que eles começam a aprender no Capítulo 1 e a implementar no Capítulo 3. O estudo de caso foi revisado por uma eminente equipe de profissionais acadêmicos e empresariais especializados em OOD/UML. O estudo de caso não é um exercício; é, mais exatamente, uma experiência de aprendizagem desenvolvida de ponta, que conclui com uma revisão detalhada da implementação de código C++ de 877 linhas. Fazemos um passeio detalhado pelas nove seções desse estudo de caso mais adiante.

- Processo de compilação e linkagem para programas de arquitetura multiplataforma. O Capítulo 10 fornece um diagrama e uma discussão detalhados do processo de compilação e linkagem que produz um aplicativo executável.
- Explicação das pilhas de chamadas. No Capítulo 6 fornecemos uma discussão detalhada (com ilustrações) sobre as pilhas de chamadas de função e sobre os registros de ativação para explicar como o C++ é capaz de monitorar a função que está atualmente em execução, como as variáveis automáticas de funções são mantidas na memória e como uma função sabe quando retornar depois de completar a execução.
- Introdução aos strings e vector. As bibliotecas padrão do C++ strings e vector são utilizadas para tornar os exemplos anteriores mais orientados a objetos. As classes strings e vector são utilizadas para tornar os exemplos anteriores mais orientados a objetos.
- Classes. Utilizamos a classe string vez de strings baseadas em ponteiro no estilo C para a maioria das manipulações de string por todo o livro. Continuamos a incluir discussões sobre strings 11 e 22 para que os alunos pratiquem manipulações de ponteiro, para ilustrar a alocação dinâmica de memória com nossa própria classe para preparar alunos para atividades e projetos no mercado de trabalho nos quais eles trabalharão com strings* em código C e C++ legado.
- Template de classe. Utilizamos o template de classe vez de manipulações de array baseadas em ponteiro no estilo C por todo o livro. Entretanto, começamos discutindo arrays baseados em ponteiro no estilo C no Capítulo 7 para preparar os alunos para trabalhar com código C e C++ legado no mercado de trabalho e utilizá-lo como uma base para construir sua própria classe personalizada no Capítulo 11, "Sobrecarga de operadores".
- Tratamento adaptado de herança e polimorfismo. Os Capítulos 12–13 foram cuidadosamente adaptados, tornando o tratamento de herança e polimorfismo mais claro e mais acessível aos alunos iniciantes. Substituímos uma hierarquia hierárquica entre Circle/Cylinder utilizada nas edições anteriores para introduzir herança e polimorfismo. A nova hierarquia é mais natural.
- Discussão e ilustração de como o polimorfismo funciona. O Capítulo 13 tem um diagrama e uma explicação detalhada de como o C++ pode implementar polimorfismo dinâmico internamente. Isso fornece aos alunos um entendimento sólido de como essas capacidades realmente funcionam. Mais importante, ajuda os alunos a entenderem o overhead do polimorfismo — em termos de consumo de memória e tempo de processador adicional. Isso ajuda os alunos a determinar quando usar polimorfismo e quando evitá-lo.
- Programação Web. No Capítulo 19, "Programação Web", tem tudo aquilo de que os leitores precisam para começar a desenvolver seus próprios aplicativos baseados na Web que executarão na Internet! Os alunos aprenderão a construir os chamados aplicativos em camadas, em que a funcionalidade fornecida em cada camada pode ser distribuída para computadores separados pela rede ou executados no mesmo computador. Utilizando o popular servidor HTTP Apache (que está disponível gratuitamente download a partir de www.apache.org) apresentamos o protocolo CGI (Common Gateway Interface) e discutimos como a CGI permite que um servidor Web se comunique com a camada superior (por exemplo, um navegador Web que executa no computador do usuário) e com scripts CGI (isto é, nossos programas C++) que executam em um sistema remoto. Os exemplos do capítulo concluem com o estudo de caso de comércio eletrônico (vitrine on-line que permite aos usuários adicionar livros a um carrinho de compras eletrônico).
- Standard Template Library (STL). Esse talvez seja um dos tópicos mais importantes do livro em função da avaliação que faz do reuso de software. A STL define poderosos componentes reutilizáveis, baseados em um template, que implementam estruturas de dados e algoritmos usualmente utilizados para processar essas estruturas de dados. O Capítulo 23 introduz e discute seus três componentes-chave — os contêineres, os iteradores e os algoritmos. Mostramos que o uso de componentes STL fornece um enorme poder expressivo e pode reduzir muitas linhas de código a uma única instrução.
- XHTML. O World Wide Web Consortium (W3C) declarou a Hypertext Markup Language (HTML) como uma tecnologia de legado que não sofrerá mais nenhum desenvolvimento. A HTML está sendo substituída pela Extensible Hypertext Markup Language (XHTML) — uma tecnologia baseada em XML que está se tornando rapidamente o padrão para descrever conteúdo. Utilizamos XHTML no Capítulo 19, "Programação Web"; o Apêndice J e o Apêndice K, introduzem a XHTML.
- Conformidade-padrão ANSI/ISO C++. Somos nossa apresentação contra o mais recente documento-padrão ANSI/ISO C++ para maior completude e exatidão. Se você precisar de detalhes técnicos adicionais sobre C++, leia o documento-padrão C++. Uma cópia PDF eletrônica do documento-padrão C++, número INCITS/ISO/IEC 14882-2003, está disponível (em inglês) em webstore.ansi.org/ansidocstore/default.asp.

- Novos apêndices sobre depuradores. Adicionamos dois novos apêndices sobre utilização de depuradores — o Apêndice L, “Utilizando o depurador Visual Studio .NET”, e o Apêndice M, “Utilizando o depurador GNU C++”. Ambos estão no final do livro.
- Novo design interativo. Desenvolvemos os estilos internos do novo projeto. As novas fontes são mais agradáveis à leitura e o novo pacote de arte é mais apropriado para as ilustrações mais detalhadas. Agora colocamos a ocorrência de cada termo-chave no texto em destaque em negrito para facilitar a consulta. Enfatizamos os componentes na tela com a fonte Helvetica negrito (por exemplo, `cout < "Hello, world!"`) e enfatizamos o texto do programa com a fonte Arial (por exemplo, `x = 5`).
- Variação de tons e fontes na sintaxe. Apresentamos o código C++ em sintaxe com algumas variações de tons e fontes. Isso melhora significativamente a legibilidade de código — um objetivo especialmente importante, já que este livro contém 17 mil linhas de código. Nossas convenções de sintaxe incluem:

comentários aparecem em fonte Courier

palavras-chave aparecem em azul negrito

constantes e valores literais aparecem em cinza

erros aparecem em itálico

todos os demais códigos aparecem em preto, normal

- Destaque de código. Esse uso do destaque de código torna fácil para os leitores localizar os novos recursos de cada programa apresentado e ajuda os alunos a revisar o material rapidamente na preparação para exames ou para aulas de laboratório.
- ‘Limpeza de código’. É o nosso termo para utilizar extensos comentários e identificadores significativos, aplicar convenções de recuo uniforme, alinhar chaves verticalmente, utilizar um corredor lâmina com uma chave direita e utilizar o espaçamento vertical para destacar unidades de programa significativas como instruções de controle e funções. Esse resultado em programas fáceis de ler e autodocumentados. Fizemos a ‘limpeza de código’ de todos os programas de código tanto no texto como no material auxiliar do livro. Trabalhamos muito para tornar nosso código exemplar.
- Teste de código em múltiplas plataformas. Exibimos os exemplos de código em várias plataformas C++ populares. Em sua maioria, todos os exemplos do livro são portados facilmente para todos os compiladores populares compatíveis com o padrão ANSI/ISO. Publicaremos qualquer problema em www.deitel.com/books/cpphtp5/index.html
- Erros e avisos mostrados para múltiplas plataformas. Exibimos programas que contêm erros intencionais para ilustrar um conceito-chave, mostramos as mensagens de erro que resultam em várias plataformas populares.

- Graduação e revisão de revisão (data da publicação). O livro é examinado por uma equipe de 30 eminentes revisores. Ao ler este livro, se tiver dúvidas, envie uma mensagem de correio eletrônico para deitel@deitel.com e respostas imediatas.

Visite nosso site Web:

www.deitel.com/newsletter/subscribe.html para obter atualizações deste livro e as últimas informações sobre C++ (em inglês). Utilizamos o site Web e o boletim para manter nossos leitores e clientes corporativos informados das últimas notícias sobre serviços da Deitel. Verifique o seguinte site Web regularmente para obter erratas, atualizações relacionadas ao software C++, dicas gratuitas e outros recursos:

www.deitel.com/books/cpphtp5/index.html

A abordagem de ensino

Este livro contém uma rica coleção de exemplos, exercícios e projetos extraídos de vários campos para fornecer ao aluno uma variedade de resolver interessantes problemas do mundo real. O livro concentra-se nos princípios da boa engenharia de software, principalmente a clareza da programação. Evitamos terminologia obscura e especificações de sintaxe em favor de ensinar por exemplo. Somos educadores que ministram cursos de programação de linguagens em salas de aulas de empresas em todo o mundo. O dr. Harvey Deitel tem 20 anos de experiência no ensino universitário, incluindo a posição de chefe do Departamento de Ciência da Computação no Boston College e 15 anos de experiência no ensino em empresas. Paul Deitel tem 12 anos de experiência no ensino em empresas. Os Deitels ministram cursos de C++ em todos os níveis para o governo, indústria, militares e clientes academicos da Deitel & Associates.

Aprendendo C++ com a abordagem Live-Code (“código ativo”)

C++ Como programar Quinta Edição é repleto de programas em C++ — cada novo conceito é apresentado no contexto de um programa funcional completo que é imediatamente seguido por uma ou mais amostras de execuções que exibem as entradas e saídas. Esse estilo exemplifica a maneira como ensinamos e escrevemos sobre programação. Chamamos esse método (e o nome) de **abordagem Live-Code**. Utilizamos as linguagens de programação para ensinar linguagens de programação. Os exemplos no livro são quase como digitá-los e executá-los em um computador. Fornecemos todos os códigos-fonte para os exemplos citados que acompanham o livro em www.deitel.com — tornando fácil para que os alunos executem cada exemplo à medida que estudam.

Acesso à World Wide Web

Todos os exemplos de código fonte de *Learn to Program, Quinta Edição* (e de nossas outras publicações) estão disponíveis (em inglês) na Internet como downloads a partir de

www.deitel.com

O registro é rápido e fácil e os downloads são gratuitos. Sugerimos fazer download de todos os exemplos (ou copiá-los do CD que panha este livro) e, em seguida, executar cada programa à medida que você lê o texto correspondente. Fazer alterações nos e-mail imediatamente ver os efeitos dessas alterações é uma excelente maneira de aprimorar sua experiência de aprendizagem em C

Objetivos

Cada capítulo inicia com uma declaração de objetivos. Isso permite que os alunos saibam o que esperar e oferece uma oportunidade da leitura do capítulo, de verificar se eles alcançaram esses objetivos. Isso dá confiança ao aluno e é uma fonte de reforço positivo.

Citações

Os objetivos da aprendizagem são seguidos por citações. Algumas são humorísticas, *lightning*, filosóficas e outras oferecem insights interessantes. Esperamos que você se divirta relacionando as citações com o material do capítulo. Muitas citações merecem uma leitura após o estudo do capítulo.

Sumário

O Sumário, no início dos capítulos, oferece uma visão geral dos capítulos, de modo que os alunos possam antecipar o que virá e belecer um processo de aprendizagem adequado e eficiente para eles.

Aproximadamente 17.000 linhas de código apresentadas em 260 programas de exemplo com entradas e saídas de programa

Nossos programas baseados em Live-Code variam de algumas poucas linhas de código a exemplos mais substanciais. Cada programa é seguido por uma janela contendo o diálogo de entrada/saída produzido quando o programa é executado, de forma que os alunos podem confirmar que os programas são executados como o esperado. Relacionar saídas às instruções que produzem o programa é uma maneira de aprender e reforçar os conceitos. Nossos programas demonstram os diversos recursos de C++. O código é numerado em linhas e apresenta sintaxe diferenciada — com palavras-chave C++, comentários e outros textos do programa, cada um com um propósito diferente. Isso facilita a leitura do código — os alunos gostarão dos estilos de sintaxe especialmente ao ler programas maiores.

735 ilustrações/figuras

Incluímos uma quantidade enorme de gráficos, tabelas, ilustrações, programas e saídas de programa. Modelamos o fluxo de controle com diagramas de atividades UML. Os diagramas de classes UML modelam os membros de dados, construtores e funções-membro de classes. Utilizamos tipos adicionais de diagramas UML por todo o nosso “Estudo de caso opcional de encadernação de software ATM com OOD/UML”.

571 dicas de programação

Incluímos dicas de programação para ajudar os alunos a focalizar aspectos importantes do desenvolvimento de programa. Desenvolvemos essas dicas na forma de

Boas práticas de programação, **Erros comuns de programação**, **Dicas de desempenho**, **Dicas de portabilidade** e **Dicas de desempenho**. Observações de engenharia de software. Essas dicas e práticas representam o melhor de seis décadas combinadas de programação e experiência pedagógica. Uma de nossas alunas, especializada em matemática, disse-nos que essa abordagem é como o destaque de axiomas, teoremas, lemas e corolários em livros de matemática — fornece uma base sólida para criar um bom software.

**Boas práticas de programação**

Boas práticas de programação são dicas para escrever programas claros. Essas técnicas ajudam os alunos a criar programas mais legíveis, autodocumentados e mais fáceis de manter.

**Erros comuns de programação**

Alunos que começam a aprender a programação (ou uma linguagem de programação) tendem a cometer certos erros com freqüência. Focalizar esses erros comuns de programação aumenta a probabilidade de os alunos cometerem os mesmos equívocos e ajuda a reduzir as longas filas que se formam na frente da sala dos instrutores fora do horário de aula!

**Dicas de desempenho**

Em nossa experiência, ensinar os alunos a escrever programas claros e comprehensíveis é de longe o objetivo mais importante para um primeiro curso de programação. Mas os alunos querem escrever programas que executam mais rápido, utilizem menos memória,

exijam menos pressionamentos de tecla ou ousem mais sobre outros aspectos. Os alunos realmente se preocupam com desempenho. Eles querem saber o que podem fazer para 'turbinar' seus programas. Portanto, destacamos as oportunidades de melhorar o desempenho do programa — tornando a sua execução mais rápida ou minimizando a quantidade de memória que eles ocupam.



Dicas de portabilidade

O desenvolvimento de softwares é uma atividade complexa e cara. Freqüentemente, organizações que desenvolvem software devem produzir versões personalizadas para uma variedade de computadores e sistemas operacionais. Por isso, hoje há uma forte ênfase em portabilidade, isto é, na produção de softwares que executem em uma variedade sistemas operacionais com pouca ou nenhuma alteração. Alguns programadores assumem que, se eles implementam um aplicativo em padrão C++, o aplicativo será portável. Esse simplesmente não é o caso. Alcançar portabilidade exige um projeto cuidadoso e cauteloso. Há muitas armadilhas. Incluímos as Dicas de portabilidade para ajudar os alunos a escrever código portável e fornecemos sugestões sobre como o C++ pode alcançar seu alto grau de portabilidade.



Observações de engenharia de software

O paradigma da programação orientada a objetos necessita de uma reconsideração completa sobre a maneira como construímos sistemas de software. O C++ é uma linguagem eficaz para alcançar a boa engenharia de software. Observações de engenharia de software destacam questões arquitetônicas e de projeto que afetam a construção de sistemas de software, especialmente sistemas de larga escala. Muito do que o aluno aprende aqui será útil em cursos de nível superior e na indústria quando ele começar a trabalhar com sistemas grandes e complexos do mundo real.



Dicas de prevenção de erros

Quando pela primeira vez projetamos esse 'tipo de dica', achamos que as utilizariamos estritamente para dizer às pessoas como testar e depurar programas C++. Na realidade, boa parte das dicas descreve os aspectos do C++ que reduzem a probabilidade de 'bugs', simplificando assim os processos de teste e depuração.

Seções de síntese

Cada capítulo termina com recursos pedagógicos adicionais. Nova nesta edição, cada capítulo termina com uma breve seção c que recapitula os tópicos apresentados. As sínteses também ajudam o aluno na transição para o capítulo seguinte.

Resumo (1.126 itens de resumo)

Apresentamos um resumo completo, em estilo lista de itens, no final de cada capítulo. Em média, há 40 itens de resumo por capítulo. Isso focaliza a revisão do aluno e reforça conceitos-chave.

Terminologia (1.682 termos)

Incluímos uma lista em ordem alfabética dos termos importantes definidos em cada capítulo — mais uma vez, para mais reforço médio, há 82 termos por capítulo. Cada termo também aparece no índice, e a ocorrência que define cada termo é destacada com um número de página para permitir localizar as definições dos termos-chave rapidamente.

609 exercícios e respostas de revisão (a contagem inclui partes separadas)

Extensos exercícios de revisão e respostas são incluídos em cada capítulo. Isso oferece ao aluno uma oportunidade de adquirir com o material e preparar-se para os exercícios regulares. Encorajamos os alunos a fazer todos os exercícios de revisão e as suas respostas.

849 exercícios

Todo capítulo conclui com um conjunto substancial de exercícios que inclui revisar terminologia e conceitos importantes; escrever instruções C++ individuais; escrever pequenas partes de funções e classes C++; escrever funções, classes e programas C++, e escrever projetos de especialização da graduação. O grande número de exercícios permite que os instrutores personalizem:

de acordo com as necessidades específicas de suas classes e variem as atividades de curso a cada semestre. Os instrutores podem usar esses exercícios para preparar deveres de casa, pequenos questionários e exames importantes. As soluções para a maioria dos exercícios estão incluídas no manual de soluções, disponível em inglês para os professores na [site do livro](#).

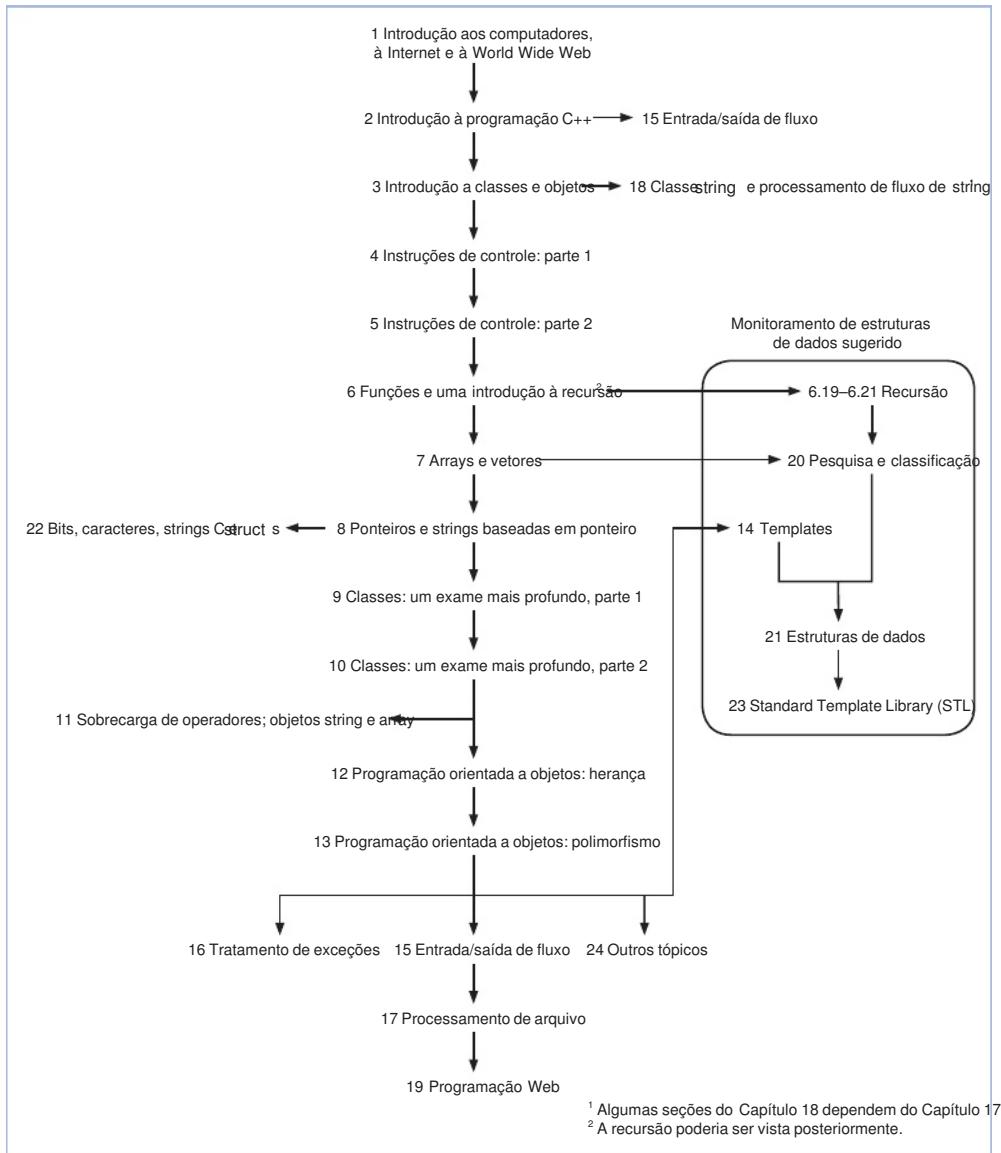
Aproximadamente 6.000 entradas de índice

Incluímos um extenso índice. Isso ajuda os alunos a encontrar termos ou conceitos por palavra-chave. O índice é um excelente recurso para as pessoas que estão lendo o livro pela primeira vez e é especialmente útil para programadores profissionais que utilizam o livro como uma referência.

Passeio pelo livro

Nesta seção, fazemos um passeio pelas muitas capacidades do C++ que você está aprendendo.

Figura 1 ilustra as dependências entre os capítulos. Recomendamos o estudo desses tópicos na ordem indicada pelas setas, em seqüências sejam possíveis. Este livro é amplamente utilizado em todos os níveis dos cursos de programação C++. Pesquise ‘C++’ e ‘Deitel’ na Web para localizar conteúdos curriculares de disciplinas que utilizaram as recentes edições deste livro.



¹ Algumas seções do Capítulo 18 dependem do Capítulo 17.
² A recursão poderia ser vista posteriormente.

Figura 1 Fluxograma ilustrando as dependências entre os capítulos

O Capítulo 1 Introdução aos computadores, à Internet e à Web discute que são computadores, como eles funcionam e como são programados. Esse capítulo fornece uma breve história do desenvolvimento das linguagens de programação desde os tempos da máquina, passando pelas linguagens assembly, até as linguagens de alto nível. A srcrem da linguagem de programação discutida. Nossas publicações da série [downloads.html](#) outras plataformas estão disponíveis ([http://tinyurl.com/Downloads](#)) em

O capítulo inclui uma introdução para um ambiente de programação C++ típico. Guiamos os leitores por um ‘tes

drive’ de um aplicativo C++ nas plataformas Windows e Linux. Esse capítulo também introduz os conceitos básicos e a termin

da tecnologia de objetos e da UML.

O Capítulo 2 Introdução à programação orientada a objetos fornece uma rápida introdução à programação de aplicativos na linguagem de programação C++. O capítulo apresenta aos não-programadores os conceitos e construtos básicos de programação. Os programas do capítulo ilustram como exibir dados na tela e como obter dados do usuário no teclado. O capítulo termina com tratamentos de

da tomada de decisão e operações aritméticas.

O Capítulo 3 Introdução a classes e objetos fornece uma introdução amigável a classes e objetos. Cuidadosamente desenvolvido, o Capítulo 3 apresenta aos alunos uma abordagem prática e fácil para a orientação a objeto desde o início. Ele foi desen

sobretudo baseado em experiências de exemplo simples empresas reais desenvolverem seus próprios sistemas de software para organizar programas orientados a objetos em C++. Primeiro, motivamos a noção de classes com um exemplo simples. Em seguida, apresentamos uma sequência cuidadosamente elaborada de sete programas funcionais completos para demonstrar a criação de suas próprias classes. Esses exemplos de estudo de caso integrado no desenvolvimento de uma classe de livro de notas que os instrutores podem utilizar para manter um quadro de notas de testes dos alunos. Esse estudo de caso é aprimorado nos capítulos seguintes, culminando com a versão apresentada no Capítulo 7, “Arrays e estruturas”. O estudo de caso de classe descreve como definir uma classe e como utilizá-la para criar um objeto. O estudo de caso discute como declarar e definir um membro para implementar os comportamentos da classe, como declarar membros de dados para implementar os atributos da classe e como chamar funções-membro de um objeto para fazê-las realizar suas tarefas. O estudo de caso aprofunda a discussão sobre classes e criamos os objetos para armazenar o nome do curso que o aluno representa. O Capítulo 3 explica as diferenças entre membros de dados de uma classe e variáveis locais de uma função e como utilizar um construtor para assegurar que os membros de dados de uma classe sejam inicializados quando o objeto é criado. Mostramos como promover a reusabilidade de software separando uma classe do código de cliente (por exemplo, a utilização da classe). Também introduzimos outro princípio fundamental de engenharia de software — a separação da interface da implementação. O capítulo inclui um diagrama detalhado e uma discussão que explicam a compilação e o processo de vinculação que produz um aplicativo executável.

O Capítulo 4 Instruções de controle: parte 1 idealiza o processo de desenvolvimento de programa envolvido na criação de classes úteis. Esse capítulo discute como escolher uma declaração do problema e desenvolver um programa C++ funcional a parti

tochada da discussão de passo a passo. Estamos sempre voltados para o controle de ponto a ponto da programação orientada a objeto simplificando a discussão de passo a passo. Estamos sempre voltados para o controle de ponto a ponto da programação orientada a objeto simplificando a discussão de passo a passo. Utilizando classes de Book do Capítulo 3 e introduzimos os operadores de atribuição, incremento e decremento do C++. O capítulo inclui duas versões aprimoradas da classe Book, ambas baseadas na versão final do Capítulo 3. Cada uma dessas versões inclui uma função-membro que utiliza instruções de controle para calcular a média de um conjunto de notas dos alunos. Na primeira versão, o método utiliza a repetição controlada por contador para um usuário inserir 10 notas, e então determina a nota média. Na segunda versão, a função-membro utiliza repetição controlada por sentinelas para inserir um número arbitrário de notas do usuário e, então, calcula a média das notas que foram inseridas. O capítulo utiliza diagramas de atividades de UML simples para mostrar o fluxo de controle para cada uma das instruções de controle.

O Capítulo 5 Instruções de controle: parte 2 continua a discussão sobre as instruções de controle C++ com exemplos da instrução de repetição, a instrução de seleção, a instrução break e a instrução continue. Criamos versões aprimoradas da classe Book que utilizam uma instrução para contar o número de notas. A, B, C, D e F inseridas pelo usuário. Essa versão utiliza a repetição controlada por sentinelas para inserir as notas. Enquanto as notas do usuário, uma função-membro modifica membros de dados que monitoram a contagem de notas em cada categoria baseada em letra. Outra função-membro da classe então utiliza esses membros de dados para exibir um relatório de resumo das notas inseridas. O capítulo inclui uma discussão de operadores lógicos.

O Capítulo 6 Funções e uma introdução à representação de dados fornece um exame mais profundo dos objetos e suas funções-membro. Discutimos as funções da biblioteca-padrão do C++ e examinamos minuciosamente como os alunos podem criar suas próprias funções. As técnicas apresentadas no Capítulo 6 são essenciais à produção de programas adequadamente organizados, especialmente de grande porte que os programadores de sistema e os programadores de aplicativo podem eventualmente desenvolver em um mundo real. A estratégia “dividir para conquistar” é apresentada como um meio eficaz de resolver problemas complexos divididos em componentes de interação mais simples. O primeiro exemplo de uma função com múltiplos parâmetros continua com um exemplo de uma função com múltiplos parâmetros. Os alunos apreciarão o tratamento e a simulação de números aleatórios do jogo de dados. A discussão do jogo de dados, que faz uso elegante de instruções de controle. O capítulo discute os chamados ‘aprimoramentos’ em relação ao C, incluindo funções com parâmetros de referência, argumentos-padrão, o operador unário de solução de escopo, sobrecarga de funções e templates de função. Apresentamos também as capacidades de chamada por valor e por referência. A tabela de arquivos de cabeçalho introduz muitos dos arquivos de cabeçalho que o leitor utilizará por todo o livro. Fornecemos uma discussão detalhada (com ilustrações) sobre a pilha de chamadas de função e sobre os registros de ativação para explicar cor

capaz de monitorar quais funções estão executando atualmente, como variáveis automáticas de funções são mantidas na memória uma função sabe onde retornar depois de completar a execução. O capítulo então oferece uma introdução sólida à recursão e irá apresentar uma tabela que resume os exemplos de recursão e os exercícios distribuídos por todo o restante do livro. Alguns textos deixam a referência para um capítulo posterior no livro; achamos que este tópico é mais bem abordado gradualmente por todo o texto. A extensa coleção de exercícios no final do capítulo inclui vários problemas clássicos de recursão, incluindo as Torres de Hanói.

O Capítulo 7, *Arrays e vetores*, explica como processar listas e tabelas de valores. Discutimos a estrutura de dados em arrays de itens de dados do mesmo tipo e demonstramos como os arrays facilitam as tarefas realizadas por objetos. As primeiras partes do capítulo utilizam arrays baseados em ponteiro no estilo C, que, como você verá no Capítulo 8, são, na realidade, ponteiros para o conteúdo de array na memória. Em seguida, apresentamos arrays como objetos completos com todos os recursos, na última seção do capítulo que introduzimos o template de classe `array` da biblioteca-padrão — uma robusta estrutura de dados de array. O capítulo apresenta inúmeros exemplos de arrays unidimensionais e bidimensionais. Os exemplos no capítulo investigam várias manipulações de array comuns, como cálculos de médias, classificação de dados e passagem de arrays para funções. O capítulo finaliza com exercícios de caso de estudo, em que utilizamos arrays para armazenar as notas dos alunos até o final da execução de um programa. Versões

anteriores desse capítulo proveram a mesma funcionalidade, mas não manipulavam os valores das notas individuais, apenas os conjuntos totais de notas para cada aluno. Neste capítulo, podemos manipular diretamente os valores das notas individuais de notas na memória, eliminando assim a necessidade de inserir repetidamente o mesmo conjunto de notas. A primeira versão armazena as notas em um array unidimensional e pode produzir um relatório contendo as notas médias e as notas mínimas e máximas e um gráfico de barras que representa a distribuição das notas. A segunda versão (isto é, a versão final no estudo de caso) usa um array bidimensional para armazenar as notas de vários alunos de múltiplos exames em um semestre. Essa versão pode calcular a média semestral de cada aluno, bem como as notas mínimas e máximas de todas as notas recebidas no semestre. A classe também gera um gráfico de barras que exibe a distribuição total das notas para o semestre. Outro recurso-chave deste capítulo é a discussão de técnicas de classificação e pesquisa elementares. Os exercícios no final do capítulo incluem uma variedade de problemas interessantes e desafiantes, como técnicas aprimoradas de classificação, projeto de um sistema simples de reservas de passagens aéreas, uma introdução a jogos de gráficos de tartaruga (tornados famosos na linguagem de programação LOGO) e os problemas Passeio do Cavalo e Oito Rainhas que introduzem a noção de programação heurística amplamente empregada no campo da inteligência artificial. Os exercícios terminam com muitos problemas de recursão, que incluem classificação por seleção, palíndromos, pesquisa linear, Oito Rainhas, impressão de um array, impressão de uma string de trás para a frente e localização do valor mínimo em um array.

O Capítulo 8, *Ponteiros e strings*, apresenta um dos recursos mais poderosos da linguagem C++ — os ponteiros. O capítulo fornece explicações detalhadas de operadores de ponteiro, chamada por referência, expressões de ponteiro, aritmética de ponteiro, relacionamento entre ponteiros e arrays, arrays de ponteiros e ponteiros para funções. Demonstramos como construir com ponteiros para impor o princípio de menor privilégio a fim de construir um software mais robusto. Além disso, introduzir

programação de strings para manipulação de dados de texto. O capítulo introduz as classes de manipulação de strings no estilo C básico e discutimos algumas das funções de tratamento de strings no estilo C mais populares, como `strlen` (inserir uma linha de texto), `strcpy` (copiar uma string), `strncat` (concatenar duas strings), `strcmp` (comparar duas strings), `strncpy` (tokenizar uma string em suas partes), `strspn` (remover o comprimento de uma string). Utilizamos os objetos `string` (introduzidos no Capítulo 3) em vez de strings em ponteiro do estilo C, onde quer que possível. Entretanto, incluímos as strings do Capítulo 8 para ajudar o leitor a dominar ponteiros e a se preparar para o mundo profissional em que ele verá muito código legado C que foi implementado nas últimas três décadas. Portanto, o leitor irá se familiarizar com os métodos mais predominantes de criar e manipular strings em C++. Muitas pessoas acham que o tópico ponteiros é, de longe, a parte mais difícil de um curso introdutório de programação. Em C e 'C++ bruto', arrays e strings são ponteiros para o conteúdo de array e na memória (até mesmo nomes de função são ponteiros). Estudar esse capítulo cuidadosamente deve recompensá-lo com um entendimento de ponteiros. O capítulo é repleto de exercícios desafiadores. Os exercícios incluem uma simulação da clássica corrida entre a lebre e a tartaruga, algoritmos de embaralhar e distribuir cartas, classificação rápida recursiva e modos de percorrer recorrendo a um labirinto. Uma seção especial intitulada "Construindo seu próprio computador" também está incluída e explica a programação em linguagem de máquina, prosseguindo com o projeto e a implementação de um simulador de computador que leva o aluno a executar programas de linguagem de máquina. Esse recurso único do texto será especialmente útil ao leitor que quer entender como os computadores realmente funcionam. Nossos alunos gostam desse projeto e frequentemente implementam aprimoramentos subversivos, muitos dos quais são sugeridos nos exercícios. Uma segunda seção especial inclui exercícios desafiadores de manipulação de strings relacionados com análise e processamento de textos, impressão de datas em vários formatos, proteção de cheque, escrita por e-mail de um cheque, código Morse e conversões de medidas métricas para medidas inglesas.

O Capítulo 9, *Classes*, é um exame mais profundo da nossa discussão sobre programação orientada a objetos. Este capítulo utiliza um rico estudo de caso para ilustrar como acessar membros de classe, separar a interface da implementação, utilizar funções de acesso e funções utilitárias, inicializar objetos com construtores, destruir objetos com destrutores, por cópia-padrão de membro a membro e reusabilidade de software. Os alunos aprendem a ordem em que construtores e destrutores são chamados durante o tempo de vida de um objeto. Uma modificação comum nos testes de classes que podem ocorrer quando uma função-membro retorna uma referência é a criação de um problema de dados encapsulado da classe. Os exercícios do capítulo desafiam o aluno a desenvolver classes para horas, datas, retângulos e a jogar o jogo da velha (Tic-Tac-Toe). Em geral, os alunos gostam de programação de jogos. Leitores com inclinação para a matemática apreciarão os exercícios sobre

criar a classe `complex` (para números complexos), a classe `ratio` (para números racionais) e a classe `integer` (para inteiros arbitrariamente grandes).

O Capítulo 10 Classes: um exame mais profundo do que o estudo de classes e apresenta conceitos adicionais de programação orientada a objetos. O capítulo discute como declarar e utilizar objetos constantes, funções-membro constantes, ci — o processo de construir classes que têm objetos de outras classes como membros, tem de ser feito especiais de acesso aos membros das classes, o ponteiro `this`, que permite que um objeto saiba seu próprio endereço, alocação dinâmica de memória, membros de classe para manipular dados de escopo de classe, exemplos populares de tipos de dados abstratos (arrays, strings e filas), classes contêineres e iteradores. Em nossa discussão sobre obj mencionamos a palavra-chave `new` que é utilizada de maneira útil para permitir a modificação da implementação 'não visível' nos objetos. Discutimos a alocação dinâmica de memória utilizando o `operator new`. Se o `operator new` falha, o programa termina por padrão por geração uma exceção no padrão C++. Motivamos a discussão sobre os membros das classes baseado em videogame. Enfatizamos a importância de ocultar detalhes de implementação de clientes de uma classe; então, c as classes proxy, que fornecem um meio de ocultar dos clientes de uma classe a implementação (incluindo os dados

~~Capítulo 11 Classes e exercícios de classe~~ Os exercícios de capítulo incluem o desenvolvimento de uma classe de conta-poupança e uma classe para

O Capítulo 11 Sobre carga de operadores; objetos strings. Apresenta um dos tópicos mais populares em nossos cursos de C++. Os alunos realmente se divertem com este material. Eles o consideram uma combinação perfeita com a discussão detal como personalizar classes valiosas nos capítulos 9 e 10. A sobre carga de operadores permite ao programador dizer ao compilador como personalizar operadores existentes com objetos de novos tipos. O C++ já sabe utilizar esses operadores com objetos de tipos predito como inteiros, números de ponto flutuante e caracteres. Mas suponha que a classe que significa a classe o sinal de adição quando utilizados entre objetos programadores utilizam o sinal de ativos para se referirem à concatenação. No Capítulo 11 o programador aprenderá a 'sobre carregar' o sinal de adição, para que, quando esses dois objetos em uma expressão, o compilador gere uma chamada de função para uma 'função de operador' que irá concatenar as duas strings. O capítulo discute os princípios básicos da sobre carga de operadores, as restrições na sobre carga de operador sobre carga com funções-membro e funções não-membro, a sobre carga de operadores unário e binário e a conversão entre tipos. O Capítulo 11 apresenta a coleção de estudos de caso substanciais incluindo, uma classe de array, uma classe classe de data, uma classe de inteiros arbitrariamente grandes e uma classe de números complexos (estas duas últimas aparecem no código-fonte completo nos exercícios). Alunos com aptidão para a matemática ganham exercícios classe Este material é diferente da maioria das linguagens e cursos de programação. A sobre carga de operadores é um tópico complicado. Utilizar a sobre carga de operadores ajuda a adicionar, de modo racional, 'acabamento' extra a suas classes. As discussões Array e da classe string são particularmente valiosas para alunos que já utilizaram a classe `C++ Standard Library`

~~Capítulo 12 Programação orientada a objetos: herança~~ O que é herança? Como é que a herança pode ser usada? O que é sobre carga de operadores? — como na matemática — em vez de com chamadas de função, como o aluno fez nos exercícios do Capítulo 10.

O Capítulo 12 Programação orientada a objetos: herança. Apresenta as capacidades mais fundamentais das linguagens de programação orientada a objetos — a herança: uma forma de reusabilidade de software em que novas classes são desenhadas rapidamente e facilmente absorvendo as capacidades de classes existentes e adicionando novas capacidades apropriadas. No conto de cabide hierarquia employee, esse capítulo substancialmente revisado apresenta uma sequência de cinco exemplos para demonstrar classes , dados , dados protected e boa engenharia de software com herança. Começamos demonstrando uma classe com membros de dados e funções-membro para manipular esses dados. Em seguida, implementamos uma segunda classe com capacidades adicionais, duplicando intencional e tediosamente grande parte do código do primeiro exemplo. O terceiro e quarto iniciam nossa discussão de herança e reuso de software — utilizamos a classe do primeiro exemplo como uma classe básica e rápida e simples, fazemos com que seus dados e funcionalidades sejam herdados por uma nova classe derivada. Esse exemplo demonstra que uma classe derivada não pode acessar diretamente os membros da classe básica. Isso motiva nosso quarto exemplo, no qual introduzimos classes herdados da classe básica e demonstramos que a classe derivada pode, de fato, acessar os dados herdados da classe básica. O último exemplo na sequência demonstra uma adequada engenharia de software definindo os dados da classe herdados utilizando funções-membro da classe básica (que foram herdadas pela classe derivada) para manipular os dados da classe básica na classe derivada. O capítulo discute as noções de classes básicas e derivadas, protegendo herança public, herança protected, herança private , classes básicas diretas e indiretas, construtores e destrutores em classes básicas e classes derivadas e engenharia de software com herança. O capítulo também

~~Capítulo 13 Programação orientada a objetos: polimorfismo~~ Outra capacidade fundamental da programação orientada a objetos: comportamento polimórfico. O Capítulo 13, completamente revisado, fundamenta os conceitos de herança e dos no Capítulo 12 e focaliza os relacionamentos entre classes em uma hierarquia de classes e as poderosas capacidades de que esses relacionamentos permitem. Quando muitas classes são relacionadas a uma classe básica comum por herança, cada classe derivada pode ser tratada como um objeto de classe básica. Isso permite que os programas sejam escritos de maneira geral, independentemente dos tipos dos objetos específicos da classe derivada. Novos tipos de objetos podem ser tratados pelo programa, tornando os sistemas mais extensíveis. O polimorfismo permite aos programadores lidar a complexa lógica da lógica 'em linha reta' mais simples. Um gerenciador de tela de um videogame, por exemplo, pode enviar uma mensagem

cada objeto em uma lista vinculada de objetos para estes serem desenhados. Cada objeto sabe como se desenhar. Um novo objeto pode ser adicionado ao programa sem modificar esse programa (contanto que esse novo objeto também saiba desse próprio). O capítulo discute os mecanismos para alcançar o comportamento polimórfico de funções entre classes abstratas (a partir das quais os objetos não podem ser instanciados) e classes concretas (a partir das quais os objetos podem ser instanciados). As classes abstratas são úteis para fornecer uma interface herdável para classes por toda a hierarquia. Demos classes abstratas e o comportamento polimórfico herdado no Capítulo 12. Introduzimos uma classe básica Employee abstrata, a partir da qual classes HourlyEmployee e SalariedEmployee herdam diretamente e a classe BasePlusCommissionEmployee herda indiretamente. No passado, nossos clientes profissionais insistiam para que fornecêssemos uma explicação mais profunda que mostrasse precisamente como o polimorfismo é implementado em C++ e, portanto, precisamente tempos de execução e 'custos' de memória se incorrem ao programar com essa poderosa capacidade. Respondemos desenvolvendo ilustração e uma explicação *prestabilizada* (que o compilador C++ cria automaticamente para suportar o polimorfismo. Para concluir, introduzimos as informações de tipo de tempo de execução e coerção dinâmica, que permite que um programa determine um tipo de objeto em tempo de execução e, em seguida, atue nesse objeto de

~~calcular sobre o tipo de tempo de execução de diferentes tipos de dados. Para todos os outros tipos de tempo de execução, o tempo de execução é sempre 100% igual ao tempo de execução do tipo de tempo de execução.~~

O Capítulo 14 Templates discute um dos recursos de reutilização de software mais poderosos do C++, a saber, templates. Os templates de função e de classe permitem aos programadores especificar, com um único segmento de código, um intervalo de funções sobrecarregadas relacionadas (chamadas especializações de template de função) ou um intervalo inteiro de classes (chamadas especializações de template de classe). Essa técnica é chamada programação genérica. Os templates de função foram introduzidos no Capítulo 6. Este capítulo apresenta discussões adicionais e exemplos sobre template de função. Poderíamos escrever um único template de classe para uma classe de pilha, então fazer o C++ gerar especializações separadas de template de classe, classe de 'pilhas' de uma classe de 'pilhas' de outra classe de 'pilhas' e assim por diante. O capítulo discute como utilizar parâmetros de tipo, parâmetros sem tipo e tipos-padrão para templates de classe. Também discutimos os relacionamentos entre templates e outros recursos C++, como sobrecarga, herança e membros. Desafiam o aluno a escrever uma variedade de templates de função e templates de classe e empregar esses templates em programas completos. Aprimorando significativamente o tratamento de templates em nossa discussão sobre contêineres, iteradores e algoritmos da Standard Template Library (STL) no Capítulo 23.

O Capítulo 15 Entrada/saída de fluxo contém um tratamento abrangente das capacidades de entrada/saída do C++ padrão. Este capítulo discute uma série de capacidades suficientes para realizar a maioria das operações de E/S comuns e fornece uma visão das capacidades restantes. Muitos dos recursos de E/S são orientados a objeto. Esse estilo de E/S utiliza outros recursos do C++ referências, sobrecarga de funções e de operadores. As várias capacidades de E/S do C++, incluindo saída com o operador de

~~definições personalizadas de saída de E/S. O capítulo discute os tipos de formatação de E/S que podem ser definidos para manipulação de fluxo e o operador de extração de E/S. A extensibilidade é um dos recursos mais valiosos do C++. O C++ fornece vários manipuladores de fluxo que realizam tarefas de formatação. Esse capítulo discute manipuladores de fluxo que fornecem capacidades como exibir inteiros em várias bases, controlar precisão de ponto flutuante, configurar larguras de campo, exibir ponto de fração decimal e zeros finais, justificar saída, configurar e desconfigurar estado de formato e configurar o caractere de preenchimento em c. Também apresentamos um exemplo que cria manipuladores de fluxo de saída definidos pelo usuário.~~

O Capítulo 16 Tratamento de exceções discute como o tratamento de exceções permite aos programadores escrever programas que são robustos, tolerantes a falha e apropriados para ambientes críticos à missão e ambientes críticos ao negócio. O capítulo mostra quando o tratamento de exceções é apropriado; introduz as capacidades básicas de tratamento de exceções com blocos `try` e `catch`; indica como e quando relançar uma exceção; explica como escrever uma especificação de exceção e processar exceções inesperadas; e discute os importantes laços entre exceções e construtores, destrutores e herança. Os exercícios neste capítulo mostram ao aluno a diversidade e o poder das capacidades de tratamento de exceções do C++. Discutimos como relançar uma exceção e ilustramos como ela pode falhar quando a memória esgotar. Muitos compiladores C++ mais antigos retornam 0 por padrão quando `new` falha. Mostramos o novo estilo de `nothrow` para garantir uma exceção (alocação ruim). Ilustramos como utilizar a função `set_new_handler` para especificar uma função personalizada a ser chamada para lidar com situações de esgotamento de memória. Discutimos como utilizar o template `new` para excluir dinamicamente memória alocada implicitamente, evitando, assim, vazamentos de memória. Para concluir o capítulo, apresentamos a hierarquia de exceção da Standard Library.

O Capítulo 17 Processamento de arquivos discute técnicas para criar e processar tanto arquivos seqüenciais como de acesso aleatório. O capítulo começa com uma introdução à hierarquia de dados de bits a bytes, campos, registros e arquivos. Em seguida, apresentamos a visão de arquivos e fluxos do C++. Discutimos arquivos seqüenciais e construímos programas que mostram como abrir e fechar arquivos, como armazenar em e ler dados seqüencialmente de um arquivo. Então discutimos arquivos de acesso aleatório e como ler dados seqüencialmente de um arquivo de acesso aleatório. O estudo de caso combina as técnicas de acesso tanto seqüencial como aleatoriamente em um programa completo de processamento de transações. Os alunos em nossos cursos corporativos mencionaram que, depois de estudar o material sobre processamento de arquivo, eles foram capazes de produzir programas substancialmente melhores de processamento de arquivo que se tornaram imediatamente úteis em suas empresas. Os exercícios exigem que o leitor implemente uma variedade de programas que criam e processam tanto arquivos seqüenciais como arquivos de acesso aleatório.

O Capítulo 18 Classes¹ e processamento de fluxo-de-dados² as capacidades do C++ de inserir dados de strings na memória e gerar saída de dados para strings na memória; essas capacidades são frequentemente referidas como formata ou processamento de fluxo de string. A classe componente necessário da Standard Library. Preservamos o tratamento de strings baseadas em ponteiro no estilo C no Capítulo 8 e posteriores por várias razões. Primeiro, ele reforça o entendimento sobre ponteiros. Segundo, durante a próxima década, os programadores em C++ precisarão ser capazes de ler e modificar as quantidades de código C legado que acumularam no último quarto do século — esse código processa strings como ponteiros, assim o faz uma grande parte do código C++ que foi escrito na indústria nos últimos anos. No Capítulo 18 discutimos atribuição, concatenação e comparação de strings. Mostramos como determinar várias características de uma string e a capacidade de uma se ela está ou não vazia. Discutimos como usar as várias funções 'find' que permitem localizar uma substring em uma string, para a frente ou para trás) e mostramos como localizar a primeira ou a última ocorrência de um caractere selecionado de caracteres e como localizar a primeira ou a última ocorrência de um caractere que não está entre os caracteres selecionada. Mostramos como substituir, apagar e inserir caracteres em uma string e converter um objeto em uma string³ no estilo C.

O Capítulo 19 Aplicativos Web⁴ é dedicado ao estudo de Web applicative. Vou explicar como construir sistemas Web aplicativos de em que a funcionalidade fornecida em cada camada pode ser distribuída para computadores separados pela Internet ou existir no mesmo computador. Em particular, construímos um aplicativo de livraria on-line de três camadas. As informações da livraria armazenadas na camada inferior do aplicativo, também chamada de camada de dados. Em aplicativos de capacidade industrializada de dados é, em geral, um banco de dados como Oracle, MySQL ou Microsoft SQL Server. Para simplificar, utilizamos arquivos de texto e empregamos as técnicas de processamento de arquivo do Capítulo 17 para acessar e modificar esses arquivos. O usuário insere solicitações e recebe respostas na camada superior do aplicativo, também chamada camada de interface com o camada cliente, que geralmente é um computador com um navegador Web popular como Microsoft Internet Explorer, o Mac Safari™, o Mozilla Firefox, o Opera® ou Netscape®. Os navegadores Web sabem comunicar-se com sites Web por toda a Internet. A camada intermediária, também chamada de camada da lógica do negócio, contém um servidor Web e um programa específico do aplicativo (por exemplo, nosso aplicativo de livraria). O servidor Web comunica-se com o programa C++ (e vice-versa) via protocolo CGI (Common Gateway Interface). Esse programa é referido como um script CGI. Utilizamos o servidor HTTP Apache popular, que está disponível gratuitamente para download a partir do site www.apachefriends.org. A instalação do Apache para muitas plataformas populares, incluindo sistemas Linux e Windows, estão disponíveis nesse site e em www.prenhall.com/deitel. O servidor Web sabe como conversar com a camada cliente por meio da Internet utilizando um protocolo chamado HTTP (Hypertext Transfer Protocol). Discutimos os dois métodos HTTP mais populares para enviar dados para um site Web — GET e POST. Então discutimos o papel crucial do servidor Web em programação Web e fornecemos um exemplo simples que

solicita um servidor Web extensões HyperText Markup Language (HTML). O Capítulo 20⁵ aborda conexões simples que obtêm ficheiros do servidor e o renderiza em um navegador. Outros exemplos demonstram como processar entrada do usuário baseada em formulários via técnicas de processamento de string introduzidas no Capítulo 18. Em nossos exemplos baseados em formulário utilizamos campos de senha, caixas de seleção e campos de texto. Apresentamos um exemplo de portal interativo de uma empresa de viagens que exibe os preços de passagens aéreas para várias cidades. Os membros do clube de viagem podem efetuar logon e visualizar as passagens aéreas. Discutimos também vários métodos de armazenar dados específicos do cliente, que incluem campos cookie, informações armazenadas em uma página Web, mas não renderizadas pelo navegador Web) e cookies — pequenos arquivos que o navegador armazena na máquina do cliente. Os exemplos do capítulo terminam com um estudo de caso de uma livraria que permite aos usuários adicionar livros a um carrinho de compras. Esse estudo de caso contém vários scripts CGI que interagem para formar um aplicativo completo. A livraria on-line é protegida por senha, desse modo os usuários devem primeiro efetuar logon para ter acesso. Os recursos na Web do capítulo incluem informações sobre a especificação CGI, bibliotecas C++ CGI e sites Web relacionados ao servidor Apache HTTP.

O Capítulo 20 Pesquisa e classificação⁶ discute duas das mais importantes classes de algoritmos em ciência da computação. Consideramos uma variedade de algoritmos específicos a cada uma e os comparamos com relação ao seu consumo de memória de processador (introduzindo a notação O, que indica o grau de dificuldade que um algoritmo pode ter para resolver um problema). Pesquisar dados envolve determinar se um valor (chamado chave de pesquisa) está presente nos dados e, se estiver, encontrar o valor. Nos exemplos e exercícios deste capítulo, discutimos vários algoritmos de pesquisa, incluindo: pesquisa binária e recursivas de pesquisa linear e binária. Pelos exemplos e exercícios, o Capítulo 20 discute a classificação por intercalação de strings.

O Capítulo 21 Estruturas de dados⁷ discute as técnicas utilizadas para criar e manipular estruturas de dados dinâmicas. O capítulo inicia com discussões de classes auto-referenciais e a alocação dinâmica de memória, então prossegue com uma discussão sobre como criar e manter várias estruturas de dados dinâmicas, incluindo listas vinculadas, filas, pilhas e árvores. Para cada tipo de estrutura de dados, apresentamos programas funcionais completos e mostramos saídas de exemplo. O capítulo também ajuda o aluno a entender ponteiros. O capítulo inclui muitos exemplos que utilizam indireção e dupla indireção — um conceito particularmente difícil. Um

¹ XHTML é uma linguagem de marcação para identificar os elementos de um documento XHTML (página 46) de modo que um navegador possa renderizar (isto é, exibir) essa página na tela do computador. XHTML é uma nova tecnologia projetada pelo World Wide Web Consortium para substituir a Markup Language (HTML) como o principal meio de especificar conteúdo Web. Nos apêndices J e K introduzimos a XHTML.

blema ao trabalhar com ponteiros é que os alunos têm dificuldade em visualizar as estruturas de dados e como seus nós estão v entre si. Incluímos ilustrações que mostram os links e a seqüência em que são criados. O exemplo de árvore binária é uma ex base para o estudo de ponteiros e estruturas de dados dinâmicas. Esse exemplo cria uma árvore binária, impõe eliminação c e introduz maneiras de percorrer recursivamente uma árvore pré-ordem, na ordem e pós-ordem. Os alunos sentem-se verdade realizados quando estudam e implementam esse exemplo. Eles particularmente apreciam ver que o modo de percorrer na ordem valores de nó na ordem classificada. Incluímos uma coleção substancial de exercícios. Um destaque dos exercícios é a seção "Construindo seu próprio compilador". Os exercícios orientam o aluno pelo desenvolvimento de um programa de conversão de ' para pós-fixo' e um programa de avaliação de expressão pós-fixo. Então, modificamos o algoritmo de avaliação pós-fixo para ger de linguagem de máquina. O compilador coloca esse código em um arquivo (utilizando as técnicas do Capítulo 17). Os alunos executam a linguagem de máquina produzida por compiladores nos simuladores de software criados nos exercícios do Capítulo 35 exercícios incluem pesquisar recursivamente uma lista de endereços, imprimir recursivamente uma lista de trás para a frente nó de árvore binária, percorrer uma árvore binária na ordem, imprimir árvores, escrever parte de um compilador de otimização, e um interpretador, inserir em/excluir de qualquer lugar em uma lista de endereços vinculada, implementar listas e filas sem ponte

~~da dupla manipulação de ponteiros e de listas encadeadas para a criação de árvores binárias. No Capítulo 21, o leitor já terá aprendido a manipular contêineres, iteradores e algoritmos STL no Capítulo 23. Os contêineres STL são estruturas de dados pré-empacotadas organizadas que a maior parte dos programadores irá considerar suficiente para a vasta maioria dos aplicativos que precisarão imp~~

A STL é um grande salto para alcançar a visão de reutilização.

O Capítulo 22 Bits, caracteres, strings C++ apresenta uma variedade de recursos importantes. Este capítulo começa comparando estruturas C++ com classes e, então, definindo e utilizando estruturas no estilo C. Mostramos como declarar estr inicializá-las e passá-las para funções. O capítulo apresenta uma simulação de embaralhamento e distribuição de cartas de alto penho. É uma excelente oportunidade para o instrutor enfatizar a qualidade de algoritmos. As poderosas capacidades de mani de bits do C++ permitem aos programadores escrever programas que exercitam capacidades de hardware de baixo nível. Isso programas a processar strings de bit, configurar bits individuais e armazenar informações de maneira mais compacta. Essas cap muitas vezes encontradas somente em linguagens assembly de baixo nível, são estimadas por programadores que escrevem sc sistema, como sistemas operacionais e software de rede. Como você se lembra, introduzimos a manipulação de string C no Capítulo 8 e apresentamos funções mais populares de manipulação de string. No Capítulo 22, continuamos nossa apres de caracteres e strings no estilo do C. Mostramos as várias capacidades de manipulação do caractere da biblioteca — como a capacidade de testar um caractere para determinar se ele é um dígito, um caractere alfabetico, um caractere alfanum dígito hexadecimal, uma letra minúscula ou maiúscula. Apresentamos as demais funções de manipulação de string das várias l casas relacionadas com strings; como sempre, cada função é apresentada no contexto de um programa C++ funcional e comple

~~pete de um bom número de exemplos e exercícios para aprimorar a capacidade de uso das estruturas de dados e algoritmos principais de strings no estilo C para o benefício de programadores C++ que irão, provavelmente, trabalhar com código C legado.~~

O Capítulo 23 Standard Template Library (STL) em todo este livro, discute a importância do reuso de software. Reconhecendo que muitas estruturas de dados e algoritmos são comumente utilizadas por programadores C++, o comitê do pac adicionou a Standard Template Library (STL) à C++ Standard Library. A STL define poderosos componentes reutilizáveis, baseados um template que implementa muitas estruturas de dados e algoritmos comuns utilizados para processar essas estruturas de dad oferece uma prova de conceito para a programação genérica com templates — introduzida no Capítulo 14 e demonstrada em de Capítulo 21. Esse capítulo introduz a STL e discute seus três componentes-chave — contêineres (estruturas de dados popularizadas em template), iteradores e algoritmos. Os contêineres STL são estruturas de dados capazes de armazenar objetos de quaisquer tipos de dados. Veremos que há três categorias de contêineres — contêineres de primeira classe, contêineres adaptadores e semiclasse. Os iteradores STL, que têm propriedades semelhantes às dos ponteiros, são utilizados por programas para manipular os elementos de um contêiner STL. De fato, os arrays-padrão podem ser manipulados como contêineres STL, utilizando ponteiros-padrão como iteradores. Veremos que a manipulação de contêineres com iteradores é conveniente e fornece um poder expressivo quando combinada com os algoritmos STL — em alguns casos, reduzindo muitas linhas de código a uma única instrução. Os algoritmos STL são funções que realizam manipulações de dados comuns como pesquisa, classificação e comparação de elementos (ou contêineres inteiros). Há, aproximadamente, 70 algoritmos implementados na STL. A maioria desses algoritmos utiliza iteradores para acessar elementos de contêiner. Veremos que cada contêiner de primeira classe suporta tipos específicos de iterador, alguns dos quais são mais poderosos que outros. O tipo de iterador suportado por um contêiner determina se o contêiner pode ou não ser utilizado com um algoritmo específico. Os iteradores encadeados, que são iteradores que permitem a manipulação direta de elementos dentro de um contêiner, são suportados por todos os contêineres que suportam os requisitos mínimos do algoritmo. Uma vez que os iteradores de um contêiner suportam os requisitos mínimos do algoritmo, então o algoritmo pode processar os elementos desse contêiner. Isso também significa que os programadores criem algoritmos que podem processar os elementos de múltiplos tipos diferentes de contêiner. O Capítulo 23 discute como implementar estruturas de dados com ponteiros, classes e memória dinâmica. O código baseado em ponteiro é mais seguro e mais leve que o código baseado em ponteiros. A implementação de estruturas de dados com ponteiros é mais difícil de implementar e mais suscetível a erros de memória. Implementar estruturas de dados adicionais como deques (filas com dupla terminação), filas de prioridade, conjuntos etc. exige considerável trabalho adicional. Além disso, se muitos programadores em um projeto grande implementarem contêineres com algoritmos semelhantes para diferentes tarefas, o código torna-se difícil de modificar, manter e depurar. Uma vantagem da STL é que os algoritmos são portáteis, ou seja, podem ser usados em diferentes ambientes sem precisar de alterações significativas.

programadores podem reutilizar os contêineres, iteradores e algoritmos STL para implementar representações e manipulações comuns. Esse reuso resulta em uma considerável economia de tempo de desenvolvimento e de recursos. Esse é um capítulo acessível que deve convencê-lo do valor da STL e encorajar mais investigações.

O Capítulo 24 *Outros tópicos* é uma coleção de diversos tópicos sobre o C++. Este capítulo discute um operador de coerção adicional `const_cast`. Esse operador, juntamente com `(Capítulo 5) dynamic_cast` (`Capítulo 13) interpret_cast` (`Capítulo 17`), fornecem um mecanismo mais robusto para conversão entre tipos do que os operadores de coerção C++ `src`inais herdados do C (que agora são considerados obsoletos). Discutimos namespaces, um recurso particularmente crucial para desenvolvedores que criam sistemas substanciais, especialmente para aqueles que criam sistemas de bibliotecas de classes. Os namespaces evitam colisões de nome, o que pode colocar obstáculos aos esforços de criação de softwares grandes. O capítulo discute as palavras-chave operadoras, que são úteis para programadores cujos teclados não suportam certos caracteres utilizados em símbolos de operadores como `!, &, ^, ~ e |`. Esses operadores também podem ser utilizados por programadores que não gostam de símbolos de operadores obsoletos. Discutimos a palavra-chave `new`, que permite que um membro de uma classe seja criado. Anteriormente, isso era realizado 'fazendo coerção de classe', o que é considerado uma prática perigosa. Discutimos também os operadores de 'ponteiro para memória'.

* O Apêndice A *Tabela de precedência das operações de coerção* fornece uma lista das precedências relativas dos símbolos de operadores do C++, em que cada operador aparece em uma linha com seu símbolo de operador, nome e associatividade.

O Apêndice B *Conjunto de caracteres ASCII* apresenta o conjunto de caracteres ASCII, que todos os programas neste livro utilizam.

O Apêndice C *Tipos fundamentais* lista todos os tipos fundamentais definidos no C++.

O Apêndice D *Sistemas de numeração* aborda os sistemas numéricos binário, octal, decimal e hexadecimal. Considera como converter números entre bases e explica as representações binárias do complemento de um e do complemento de dois.

O Apêndice E *Tópicos sobre código C legado* apresenta tópicos adicionais incluindo vários tópicos avançados não comumente discutidos em cursos introdutórios. Mostramos como redirecionar a entrada de um programa para vir de um arquivo, redirecionar a saída de programa para ser colocada em um arquivo, redirecionar a saída de um programa para ser redirecionada para a entrada de outro programa, acrescentar a saída de um programa a um arquivo existente. Desenvolvemos funções que utilizam listas de argumentos de cor variável e mostramos como passar argumentos de linha de comando para a função principal de um programa. Discutimos como compilar programas cujos componentes se distribuem por múltiplos arquivos; como registrar funções externas ao término de um programa e como terminar a execução de um programa. Discutimos também os qualificadores de tipo `const` e `volatile`, a especificação do tipo de uma constante numérica utilizando os sufíxos de inteiro e de ponto flutuante, a utilização da biblioteca de tratamento de sinal para interceptar eventos inesperados, a criação e utilização de arrays dinâmicos com `malloc`, a utilização de `new` e `delete` como uma técnica de economia de espaço e o uso de especificações de linkagem quando programas

Em *Precessando o trabalho com código C legado*, o leitor verá o que se pode fazer para integrar o código principal de um projeto para integrar o código C legado. Isso é um determinado momento da carreira.

O Apêndice F *Pré-processador* fornece discussões detalhadas das diretivas de pré-processador. O apêndice inclui informações mais completas sobre a diretiva `#include` que faz com que uma cópia de um arquivo especificado seja incluída no lugar da diretiva antes de o arquivo ser compilado e sobre as diretivas de constantes e macros simbólicas. Explica a compilação condicional para permitir ao programador controlar a execução de diretivas de pré-processador e a compilação de código de programas. São discutidos operadores que converte seu operando em uma string que compreende dois tokens. São apresentadas as várias constantes simbólicas predefinidas de pré-processador (`_DATE`, `_STDC`, `_TIME` e `_TIMESTAMP`). Por fim, é discutida a macro `__FILE__` do arquivo de cabeçalho, o que é valioso em testes, verificação e validação de um programa.

O Apêndice G *Código para o estudo de caso* fornece a implementação de nosso estudo de caso sobre projeto orientado a objetos com a UML. Esse apêndice é discutido na visão geral do estudo de caso (apresentado a seguir).

O Apêndice H *UML 2: tipos de diagramas adicionais* fornece uma visão geral dos tipos de diagrama UML 2 que não são encontrados no estudo de caso com OOD/UML.

O Apêndice I *Recursos sobre C++ na Internet* fornece uma listagem de valiosos recursos sobre C++, como demonstrações, informações sobre compiladores populares, livros eletrônicos, conferências, classificados, periódicos, revistas, ajudas, tutoriais, FAQs (perguntas feitas com frequência), newsgroups, cursos baseados na Web, notícias de produtos e ferramentas de desenvolvimento C++.

O Apêndice J *Introdução à XHTML* fornece uma introdução à XHTML — uma linguagem de marcação para descrever os elementos de uma página Web de modo que um navegador, como o Microsoft Internet Explorer ou o Netscape, possa exibir essa página. O leitor deve conhecer o conteúdo desse apêndice antes de estudar o Capítulo 19, *Programação Web*. Esse apêndice não aborda nenhuma programação em C++. Alguns tópicos-chave tratados nesse apêndice incluem incorporação de texto e imagens em documentos XHTML, vinculação a outros documentos XHTML, incorporação de caracteres especiais (como símbolo de direitos autorais de marca comercial) em um documento XHTML, separação de partes de um documento XHTML com linhas horizontais, apresentação de informações em listas e tabelas e coleta de informações de usuários que navegam em um site.

O Apêndice K *Caracteres especiais de XHTML* fornece muitos caracteres XHTML especiais comumente utilizados, chamados de referências de entidade de caractere.

O Apêndice H Utilizando o depurador do Visual Studio .NET demonstra recursos-chave do Visual Studio .NET Debugger, que permite que um programador monitore a execução de aplicativos para localizar e remover erros de lógica. Esse apêndice a instruções passo a passo para os alunos aprenderem a utilizar o depurador de uma maneira prática.

O Apêndice I Utilizando o depurador do GNU C++ demonstra recursos-chave do GNU C++ Debugger, que permite que um programador monitore a execução de aplicativos para localizar e remover erros de lógica. Esse apêndice apresenta instruções passo a passo para os alunos aprenderem a utilizar o depurador de uma maneira prática.

A Bibliografia lista mais de cem livros e artigos para encorajar o aluno a ler mais sobre C++ e POO.

O Índice abrangente permite que o leitor localize por palavra-chave qualquer termo ou conceito em todo o texto.

Projeto orientado a objetos de um ATM com a UML:

um passeio pelo estudo de caso opcional de engenharia de software

Nesta seção faremos um passeio pelo estudo de caso opcional deste livro do projeto orientado a objetos com UML. Este passeio o conteúdo das nove seções de “Estudo de caso de engenharia de software” (nos capítulos 1–7, 9 e 13). Depois de completar esse passeio, o leitor estará inteiramente familiarizado com um projeto e uma implementação orientados a objetos cuidadosamente realizadas para um significativo aplicativo C++.

O projeto apresentado no estudo de caso ATM foi desenvolvido na Deitel & Associates, Inc. e minuciosamente examinado por uma eminentemente equipes de revisão acadêmica e de profissionais da indústria. Elaboramos esse projeto para atender aos requisitos de s de cursos introdutórios. Sistemas ATM reais utilizados por bancos e seus clientes em todo o mundo são baseados em projetos sofisticados que levam em consideração muitas outras questões além das abordadas aqui. Nossa principal objetivo por todo o projeto foi criar um projeto simples que fosse claro para os iniciantes em OOD e UML e, ao mesmo tempo, demonstrar conceitos-chave de OOD e técnicas de modelagem relacionadas à UML. Trabalhamos muito para manter o projeto e o código relativamente pequeno que funcionasse bem em um curso introdutório.

A Seção 1.1 Estudo de caso de engenharia de software: introdução à tecnologia de projeto com UML

de caso do projeto orientado a objetos com a UML. A seção introduz os conceitos básicos e a terminologia da tecnologia de projeto, incluindo classes, objetos, encapsulamento, herança e polimorfismo. Discutimos a história da UML. Essa é a única seção obrigatória do estudo de caso.

A Seção 2.1 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)

cute um documento de requisitos para um sistema que iremos projetar e implementar — o software para um caixa eletrônico simples (automated teller machine ATM). Investigamos de maneira geral a estrutura e o comportamento dos sistemas orientados a objetos. Discutimos como a UML facilitará o processo de projeto nas seções subsequentes de “Estudo de caso de engenharia de software”, fornecendo vários tipos de diagramas adicionais para modelar nosso sistema. Incluímos uma lista de URLs e referências bibliográficas sobre projetos orientados a objetos com a UML. Discutimos a interação entre o sistema ATM especificado pelo documento de requisitos e seu usuário. Especificamente, investigamos os cenários que podem ocorrer entre o usuário e o próprio sistema — estes são casos de uso. Modelamos essas interações utilizando a UML.

A Seção 3.1 Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional) — começa a projetar o sistema ATM. Identificamos suas classes, ou ‘blocos de construção’, extraíndo os substantivos simples e compostos do documento de requisitos. Organizamos essas classes em um diagrama de classes UML que descreve a classe da nossa simulação. O diagrama de classes também descreve relações entre as classes, como adjetivos e adjetivos compostos.

A Seção 4.1 Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional) — focaliza os atributos das classes discutidas na Seção 3.1. Atributos (valores que um objeto pode ter). Como veremos nas seções a seguir, alterações nos atributos do objeto freqüentemente afetam o comportamento do objeto. Para definir os atributos para as classes no nosso estudo de caso, extraímos os adjetivos que descrevem os substantivos simples e os substantivos compostos (que definiram nossas classes) do documento de requisitos, então colocamos os atributos no diagrama de classes que aparece na Seção 3.11.

A Seção 5.1 Estudo de caso de engenharia de software: identificando estados e atividades de objetos no sistema ATM (opcional) — discute como um objeto, em qualquer dado momento, ocupa uma condição específica chamada de estado. Isso ocorre quando esse objeto recebe uma mensagem para alterar seu estado. Afinal de contas, é o que identifica o conjunto de possíveis estados que um objeto pode ocupar e modela as transições de estado desse objeto. Utilizamos a UML para modelar esses estados — o trabalho que ele realiza no seu tempo de vida. A UML torna esse flu-

rograma que trata de especificar a lógica desse sistema. Mais uma vez, ATM executa dírias tarefas de acordo com o que o usuário deseja. O usuário é autenticado.

A Seção 6.2 Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional) — identifica as operações, ou serviços, de nossas classes. Extraímos do documento de requisitos os verbos e frases com ve que especificam as operações para cada classe. Modificamos então o diagrama de classes da Seção 3.11 a fim de incluir cada operação e sua classe associada. Nesse ponto no estudo de caso, teremos coletado todas as possíveis informações do documento de requisitos. Entretanto, à medida que os capítulos seguintes introduzirem tópicos como herança, modificaremos nossas classes e diagramas.

A Seção 7.1 Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional) — um ‘esboço’ do modelo para nosso sistema ATM. Nessa seção vemos como ele funciona. Investigamos o comportamento da si-

discutindo colaborações— mensagens que objetos enviam uns para os outros para se comunicar. As operações de classe que descobrimos na Seção 6.22 se tornam as colaborações entre os objetos no nosso sistema. Determinamos as colaborações e então as reunimos em um diagrama de comunicação—diagrama da UML para modelar colaborações. Esse diagrama revela quais objetos colaboram e quando. Apresentamos um diagrama de comunicação das colaborações entre objetos para realizar uma consulta de saldo no ATM. Apresentamos então um diagrama de seqüências UML para modelar interações em um sistema. Esse diagrama enfatiza a ordem cronológica das mensagens. Um diagrama de seqüências modela como objetos no sistema interagem para executar transações de retirada e de depósito.

A Seção 9.12—Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)—faz uma pausa no projeto do comportamento de nosso sistema. Começamos o processo de implementação para enfatizar o material discutido no Capítulo 9. Utilizando o diagrama de classes da UML da Seção 3.11, e os atributos e as operações discutidas nas seções 4.13 e 6.22, mostramos como implementar uma classe em C++ a partir de um projeto. Não implementamos todas as classes — porque ainda não completamos o processo de projeto. Trabalhando a partir de nossos diagramas da UML, criamos o código para a classe

A Seção 13.10—Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional)—nossa discussão sobre programação orientada a objetos. Examinamos a herança — classes que compartilham características comuns podem herdar atributos e operações de uma classe ‘básica’. Nessa seção, investigamos como nosso sistema ATM pode se beneficiar da utilização de herança. Documentamos nossas descobertas em um diagrama de classes que modela relacionamentos de herança — a UML denomina esses relacionamentos heranças. Modificamos o diagrama de classes da Seção 3.11 utilizando a herança para agrupar classes com características semelhantes. Esta seção conclui o projeto da parte de modelagem da nossa simulação. Implementamos esse modelo completamente em 877 linhas de código C++ no Apêndice G.

O Apêndice G—Código para o estudo de caso do ATM—envolve em sua maior parte projetar o modelo (isto é, os dados e a lógica) do sistema ATM. Nesse apêndice, implementamos esse modelo em C++. Utilizando todos os diagramas UML que criamos, apresentamos as classes C++ necessárias para implementar o modelo. Aplicamos os conceitos do projeto orientado a objetos com a UML e programação orientada a objetos em C++ que você aprendeu nos capítulos. Ao final do apêndice, os alunos terão completado o projeto e a implementação de um sistema prático e devem estar confiantes para abordar sistemas maiores, como aqueles que engenheiros de softwares construem.

O Apêndice H—UML 2: tipos de diagramas adicionais fornece uma visão geral dos tipos de diagrama UML 2 que não são encontrados no estudo de caso com OOD/UML.

Material complementar

A Sala Virtual deste livro (sv.pearson.com.br) oferece: para os professores — apresentações em PowerPoint em português, manual de solução e código-fonte dos exemplos do livro em inglês; e para os alunos — código-fonte dos exemplos do livro e exercícios de múltipla escolha em inglês.

O boletim de correio eletrônico gratuito Deitel® Buzz Online

Nosso boletim de correio eletrônico gratuito Deitel® Buzz Online é enviado para aproximadamente 38.000 assinantes registrados e inclui comentários sobre tendências e desenvolvimentos da indústria, links para artigos gratuitos e recursos de nossos livros publicados e das próximas publicações, agendas de lançamento de produtos, erratas, desafios, relatos, informações sobre nossos cursos de treinamento corporativos e muito mais. Também é o canal para notificar nossos leitores rapidamente sobre questões relacionadas a este livro. Todas as informações estão disponíveis em inglês. Para assiná-lo, visite

www.deitel.com/newsletter/subscribe.html

Agradecimentos à equipe da edição srcinal

Um dos grandes prazeres de escrever um livro é o reconhecimento dos esforços de muitas pessoas cujos nomes não podem aparecer na capa, mas cujo trabalho duro, cooperação, amizade e compreensão foram cruciais à produção do livro. Muitas pessoas na Deitel & Associates, Inc. dedicaram longas horas para trabalhar conosco neste projeto.

- Andrew B. Goldberg é graduado pelo Amherst College, onde obteve o grau de bacharelado em ciência da computação Andrew atualizou os capítulos 1–13 com base na nova apresentação antecipada de classes e outras revisões de conteúdo. Ele co-projetou e co-escreveu o novo estudo de caso opcional ATM com OOD/UML que aparece nos capítulos 1–7, 9 e 13. Além disso, foi colaborador do Capítulo 19 e co-autor dos apêndices G e H.
- Jeff Jeff é graduado em ciência da computação pelo Harvard College. Jeff contribuiu para os capítulos 18, 20 e 22, apêndice 14–24.
- Su Zhang é bacharel e doutora em ciência da computação pela McGill University e contribuiu para os capítulos 14–24.
- Cheryl Yaeger graduou-se em três anos pela Boston University em ciência da computação e contribuiu para os capítulos 4, 6, 8, 9 e 13.
- Barbara Deitel, diretora financeira na Deitel & Associates, Inc., pesquisou as citações no começo de cada capítulo e fez o copi desque deste livro.

- Abbey Deitel, presidente da Deitel &Associates, Inc., é graduada em administração industrial pela Carnegie Mellon University e colaborou com o Prefácio e o Capítulo 1. Fez copidesque de vários capítulos do livro, gerenciou o processo de revisão e sugeriu o tema e os nomes de bugs para a capa do livro.
- Christi Kelsey graduou-se pela Purdue University, com bacharelado em gerenciamento e especialização em sistemas de informações. Christi contribuiu para o Prefácio e o Capítulo 1. Editou o Índice, paginou o manuscrito e coordenou muitos aspectos do nosso relacionamento de publicação com a Prentice Hall.

Tivemos a felicidade de trabalhar neste projeto com uma equipe talentosa e dedicada de profissionais de publicação da Prentice Hall. Apreciamos especialmente os esforços extraordinários de nossa editora de ciência da computação, Kate Hargett, e de sua mentora em publicações — Marcia Horton, editora-chefe da divisão de engenharia e de ciência da computação da Prentice Hall. Jennifer Cappello fez um trabalho extraordinário de recrutamento da equipe de revisão e gerenciamento do processo de revisão da Prentice Hall. Vince O'Brien, Tom Manshreck e John Lovell fizeram um trabalho formidável como gerentes de produção deste projeto. Os talentos de Paul Belfanti, Carole Anson, Xiaohong Zhu e Geoffrey Cassar ficam evidentes no projeto interno do livro e na apresentação final.

Sobre o autor para mais informações sobre o autor, consulte o site de complementação da quinta edição e de nossos revisores da quinta edição.

Revisores acadêmicos

Richard Albright, Goldey Beacom College
 Karen Arlien, Bismarck State College
 David Branigan, DeVry University, Illinois
 Jimmy Chen, Salt Lake Community College
 Martin Dulberg, North Carolina State University
 Ric Heishman, Northern Virginia Community College
 Richard Holladay, San Diego Mesa College
 William Honig, Loyola University
 Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire
 Brian Larson, Modesto Junior College
 Robert Myers, Florida State University
 Gavin Osborne, Saskatchewan Institute of Applied Science and Technology
 Wolfgang Pelz, The University of Akron
 Donna Reese, Mississippi State University

Revisores da indústria

Curtis Green, Boeing Integrated Defense Systems
 Mahesh Hariharan, Microsoft
 James Huddleston, Consultor independente
 Ed James-Beckham, Borland Software Corporation
 Don Kostuch, Consultor independente
 Meng Lee, Hewlett-Packard
 Kriang Lerdsuwanakij, Siemens Limited
 William Mike Miller, Edison Design Group, Inc.
 Mark Schimmel, Borland International
 Vicki Scott, Metrowerks
 James Snell, Boeing Integrated Defense Systems
 Raymond Stephenson, Microsoft

Revisores do Estudo de caso de engenharia de software com OOD/UML

Sinan Si Alhir, Consultor independente
 Karen Arlien, Bismarck State College
 David Branigan, DeVry University, Illinois

Martine Dulberg, North Carolina State University College
 Richard Holladay, San Diego Mesa College
 Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire
 Brian Larson, Modesto Junior College
 Gavin Osborne, Saskatchewan Institute of Applied Science and Technology
 Praveen Sadhu, Infodat International, Inc.
 Cameron Skinner, Embarcadero Technologies, Inc. / OMG
 Steve Tockey, Construx Software

Revisores pós-publicação de C++ 4/e

Butch Anton, Wi-Tech Consulting

Karen Arlien, Bismarck State College

Jimmy Chen, Salt Lake Community College

Martin Dulberg, North Carolina State University

William Honig, Loyola University

Don Kostuch, Consultor independente

Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire

Brian Larson, Modesto Junior College

Kriang Lerdswanakij, Siemens Limited

Robert Myers, Florida State University

Gavin Osborne, Saskatchewan Institute of Applied Science and Technology

Wolfgang Pelz, The University of Akron

David Papurt, Consultor Independente

Dawn Reese, Mississippi State University

Catherine Wyman, DeVry University, Phoenix

Salih Yurtas, Texas A&M University

Pressionados por um rígido prazo final de entrega, eles examinaram minuciosamente cada aspecto do texto e fizeram inúmeras sugestões para melhorar a exatidão e a integridade da apresentação.

Bem, aí está! Bem-vindo ao fascinante mundo do C++ e da programação orientada a objetos. Esperamos que você goste do exame da programação de computador contemporânea. Boa sorte! A medida que você ler este livro, apreciaríamos sinceramente comentários, críticas, correções e sugestões para melhorar o texto. Envie qualquer correspondência para:

clientes@pearsoned.com

ou em inglês para

deitel@deitel.com

Responderemos prontamente e publicaremos correções e esclarecimentos em:

www.deitel.com/books/cpphtp5/index.html

Esperamos que você se divirta aprendendo com a gente, Quinta Edição tanto quanto nós nos divertimos ao escrevê-lo!

Dr. Harvey M. Deitel

Paul J. Deitel

Sobre os autores

O dr. Harvey M. Deitel é presidente e executivo-chefe de estratégias da Deitel & Associates Inc., tem 43 anos de experiência no campo de computação, incluindo extensa experiência acadêmica e corporativa. O dr. Deitel obteve os graus de B.S e M.S. do Massachussetts Institute of Technology e um Ph.D. da Boston University. Ele foi um dos pioneiros nos projetos dos sistemas operacionais de nível virtual na IBM e no MIT que desenvolveram as técnicas amplamente implementadas hoje em dia em sistemas como UNIX, Linux, Windows XP. Ele tem 20 anos de experiência no ensino universitário, incluindo livre-docência e a chefia do Departamento de Ciência da Computação no Boston College antes de fundar a Deitel & Associates, Inc., com seu filho, Paul J. Deitel. O dr. Deitel proferiu centenas de seminários profissionais para importantes corporações, instituições acadêmicas, organizações governamentais e órgãos internacionais. Ele e Paul são os co-autores de dezenas de livros e pacotes de multimídia e estão escrevendo muito mais. Com traduções para japonês, alemão, russo, espanhol, chinês tradicional, chinês simplificado, francês, polonês, italiano, português, grego, urdu e textos da Deitel ganharam reconhecimento internacional.

Paul J. Deitel é executivo-chefe de tecnologia da Deitel & Associates, Inc., é graduado pela Sloan School of Management do MIT onde estudou Tecnologia da Informação. Por meio da Deitel & Associates, ministrou cursos sobre C++, Java, C, Internet e World Wide Web para clientes de grandes empresas, incluindo IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, National Severe Storm Laboratory, PalmSource, White Sands Missile Range, Rogue Wave Software, Boeing, Cambridge Technology Partners, TJX, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems e várias outras organizações. Paul é um dos professores corporativos mais experientes do mundo em Java e C++, tendo conduzido cerca de cem cursos de treinamento profissional em Java e C++. Deu palestras sobre C++ e Java para o Boston Chapter da Association for Computing Machinery. Ele e seu pai, dr. Harvey M. Deitel, são autores de livros-texto sobre a ciência da computação líderes de vendas no mundo todo.

Sobre a Deitel & Associates, Inc.

A Deitel & Associates, Inc., é uma organização internacionalmente reconhecida de treinamento corporativo e criação de software especializada em linguagens de programação de computadores, tecnologia de software para a World Wide Web e a Internet e tecnologia de objetos. A empresa oferece cursos direcionados a instrutores sobre as principais linguagens de programação e suas aplicações, como Java, Advanced Java, C, C++, linguagens de programação .NET, XML, Perl, Python; tecnologia de objetos; e programação orientada a objetos.

para a Internet e a World Wide Web. Os fundadores da Deitel & Associates, Inc. são o dr. Harvey M. Deitel e Paul J. Deitel. Os c
da empresa incluem muitas das maiores empresas de computadores do mundo, agências governamentais, setores do serviço
organizações comerciais. Ao longo dos seus 29 anos de parceria editorial com a Prentice Hall, a Deitel & Associates, Inc. publica
textos de programação de ponta, livros profissionais, cursos interativos, ^{www.deitel.com/training} e ^{www.deitel.com/courses} e ^{www.deitel.com/training} ^{www.deitel.com/training}
Courses ^{www.deitel.com/training} ^{www.deitel.com/training} ^{www.deitel.com/training} ^{www.deitel.com/training} ^{www.deitel.com/training} ^{www.deitel.com/training} ^{www.deitel.com/training}
systems CMSs) para CMSs populares como o WebCT, o Blackboard e o CourseCompass da Pearson. A Deitel & Associates e os
os podem ser contatados por e-mail (em inglês) em:

deitel@deitel.com

Para saber mais sobre a Deitel & Associates, suas publicações no ^{www.deitel.com/books} Séries Complete Training mundial,
visite o nosso site:

www.deitel.com

e assine o [bulletin Buzz Online](http://www.deitel.com/newsletter/index.html) gratuito via correio eletrônico em:

www.deitel.com/newsletter/subscribe.html

Aqueles que desejam comprar livros da Deitel, Cyber Classrooms, Complete Training Courses e cursos de treinamento bas
na Web podem fazer isso via:

www.deitel.com/books/index.html

Antes de começar

Siga as instruções nesta seção para assegurar que os exemplos do livro sejam adequadamente copiados no computador antes de começar a utilizar este livro.

Convenções de fontes e nomes

Utilizamos fontes para separar componentes na tela (como nomes de menu e itens de menu) e código ou comandos C++. Nesta seção, usaremos **negritas** para exemplificar e **itálicas** para enfatizar código e comandos C++ com **underline** serifas (por exemplo, `Hello;`).

Recursos no CD que acompanha C++ Como programar, Quinta Edição

O CD que acompanha este livro inclui:

- Centenas de exemplos de código ativo (Live-Code) em C++.
- Os links para compiladores C++ gratuitos e ambientes de desenvolvimento integrado (IDEs).
- Centenas de recursos na Web, incluindo referências gerais, tutoriais, FAQs, newsgroups e informações sobre a STL.

Se você tiver alguma pergunta, envie correio eletrônico para support@deitel.com

Copiando e organizando arquivos

Todos os exemplos do C++ Como programar, Quinta Edição incluídos no CD que acompanha este livro. Siga os passos na seção a seguir, "Copiando os exemplos do livro a partir do CD", para copiar os diretórios da unidade de disco. Sugermos que você trabalhe a partir da sua unidade de disco em vez da sua unidade de CD por duas razões: o CD é de leitura, portanto pode salvar seus aplicativos no CD do livro; além disso, os arquivos podem ser acessados mais rapidamente de uma unidade de disco do que de um CD. Os exemplos no livro também estão disponíveis para download no site do livro em www.deitel.com/books/cpphtp5/index.html e no nosso site em: www.prenhall.com/deitel.

Pressupomos para o propósito desta seção "Antes de começar" que você esteja utilizando um computador que execute o Windows. As capturas de tela na seção a seguir podem diferir ligeiramente do que você vê no seu computador, dependendo do Windows 2000 ou Windows XP. Se você estiver executando um sistema operacional diferente e tiver perguntas sobre como copiar os arquivos de exemplo no computador, consulte seu instrutor.

Copiando os exemplos do livro a partir do CD

1. Inserindo o CD no computador que acompanha o C++ Como programar, Quinta Edição, a unidade de CD do computador. A janela exibida na Figura 1 deve aparecer. Se a página **Aparecerá esta janela** não aparecer, prossiga para o passo 2.
2. Abrindo o diretório de CD utilizando o Meu Computador. Se a página mostrada na Figura 1 não aparecer, dê um clique duplo no ícone **Meu Computador** na área de trabalho. **Não** dê um clique duplo na unidade de CD-ROM (Figura 2) para carregar o CD (Figura 1).
3. Abrindo o diretório de CD-ROM. Se a página na Figura 1 aparecer, **Não** clique nele para ver o conteúdo do CD (Figura 1) para acessar o conteúdo do CD.
4. Copiando o diretório `examples`. Dê um clique com o botão direito do mouse no ícone **examples** (Figura 3) e então selecione

Copiar. Em seguida, **Coloque o CD no meu computador** (Figura 4). Selecione a opção **Sim** na caixa de diálogo.

Copiar. Em seguida, **Coloque o CD no meu computador** (Figura 4). Selecione a opção **Sim** na caixa de diálogo. [Acesse a essa unidade por todo o livro. Você pode escolher salvar os arquivos em uma unidade diferente com base na configuração do seu computador, na configuração do laboratório da sua faculdade ou em suas preferências pessoais. Se você trabalha em um laboratório de informática, consulte seu instrutor para informações adicionais a fim de confirmar onde os exemplos devem ser salvos.]

Os arquivos de exemplo que você copiou para seu computador do CD são de leitura. Em seguida, você removerá a propriedade de leitura para que possa modificar e executar os exemplos.

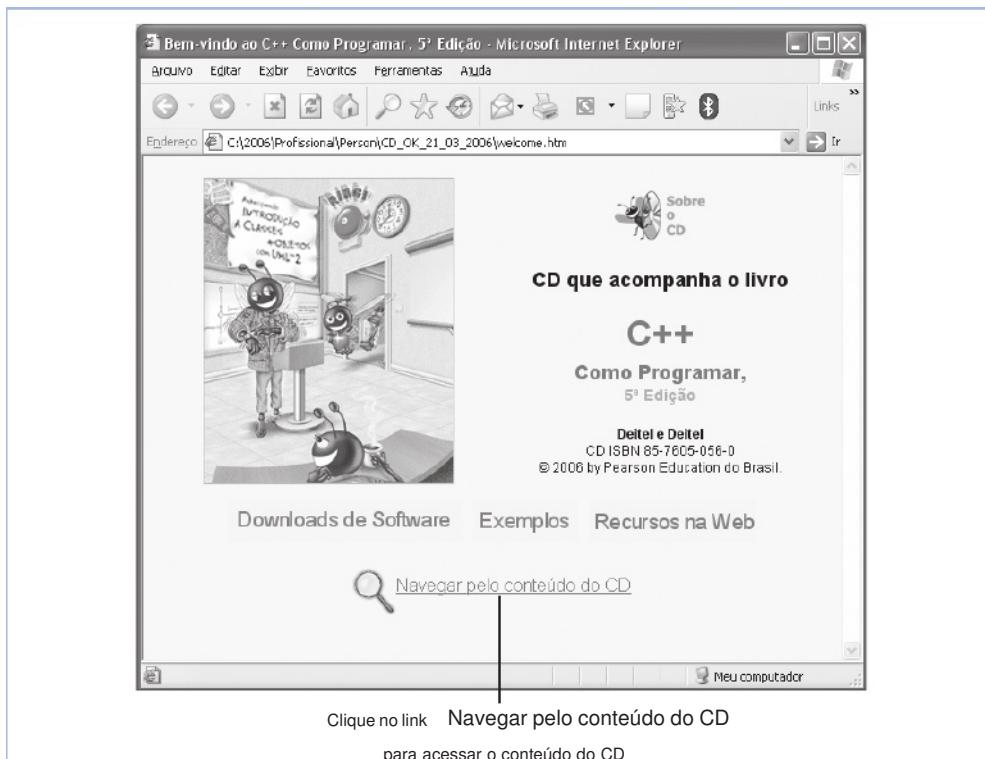


Figura 1 Página de boas-vindas do CD C++ Como programar

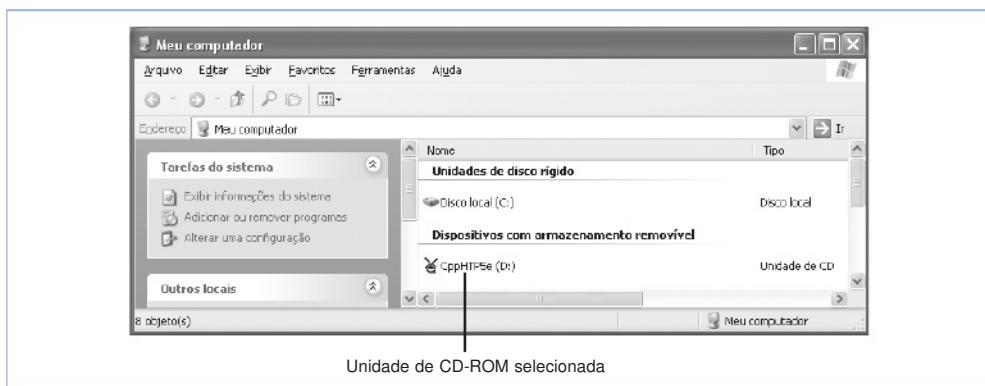


Figura 2 Localizando a unidade de CD-ROM.

Alterando a propriedade de leitura dos arquivos

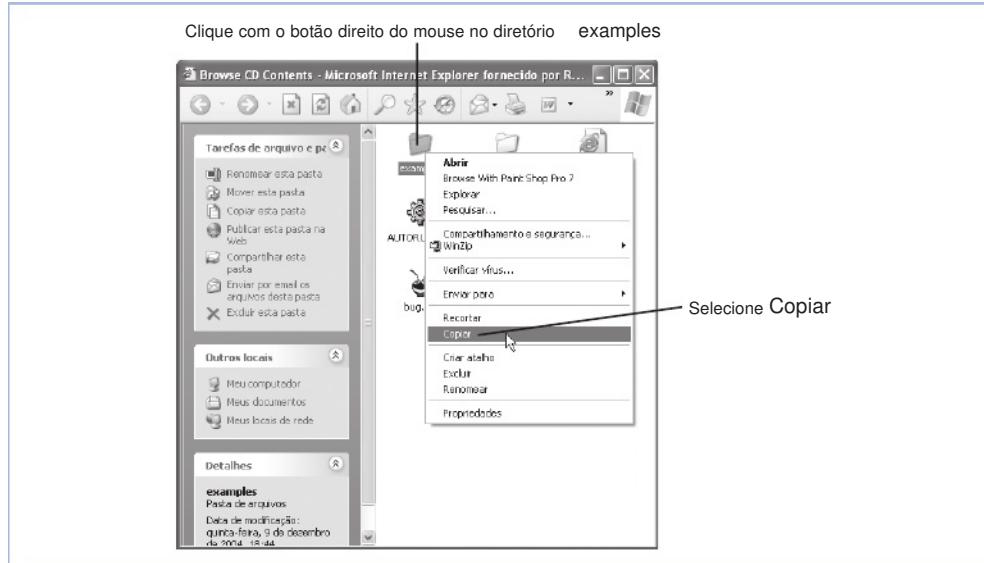


Figura 3 Copiando o diretório examples

Alterando a propriedade de leitura dos arquivos

1. Abrindo a caixa de diálogo Propriedades. Dê um clique com o botão direito do mouse no diretório examples e selecione Propriedades no menu. A caixa de diálogo Propriedades de examples aparece (Figura 4).

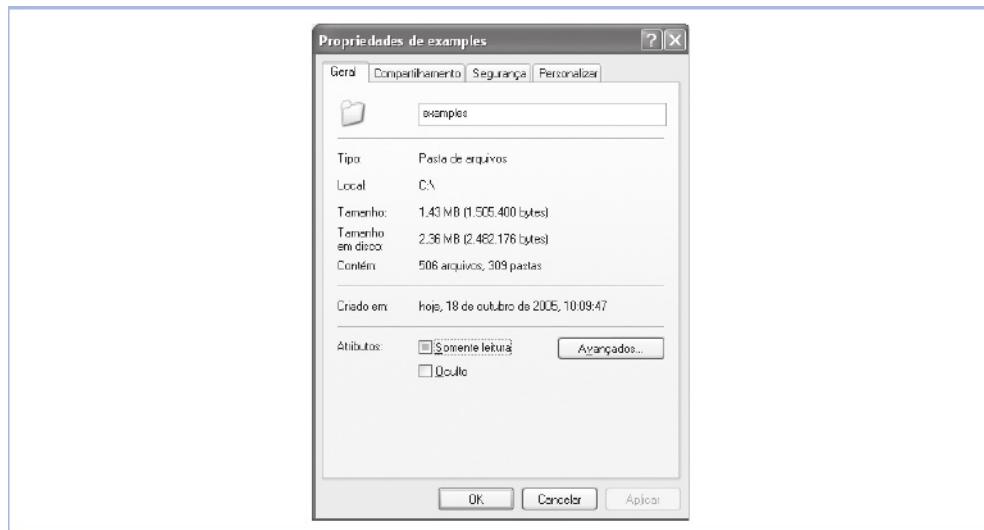


Figura 4 Caixa de diálogo Propriedades de examples

2. Alterando a propriedade para somente leitura. Na seção Atributos dessa caixa de diálogo, clique na caixa de seleção Somente leitura para remover a marca de seleção (Figura 5). Clique em Aplicar para aplicar as alterações.

3. Alterando a propriedade para todos os arquivos para somente leitura. Clique em Aplicar para aplicar as alterações de atributo (Figura 6). Nessa janela, clique no botão de opção Aplicar as alterações a esta pasta, subpastas e arquivos para remover a propriedade de somente leitura para todos os arquivos e diretórios no diretório examples.

Agora você está pronto para começar seus estudos sobre o programa. Esperamos que você goste deste livro! Você pode nos contatar facilmente em www.mecanismo.com. Responderemos prontamente.

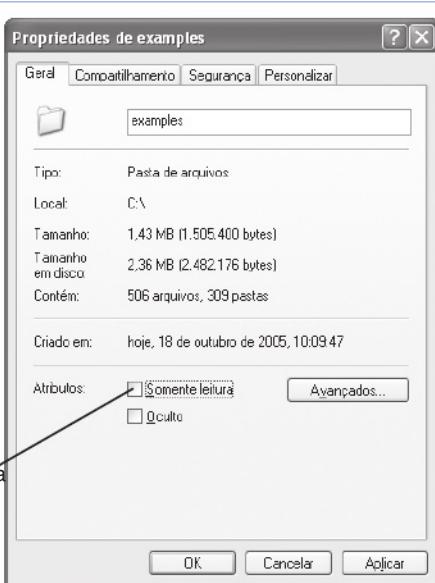


Figura 5 Desmarcando a caixa de seleção Somente leitura .

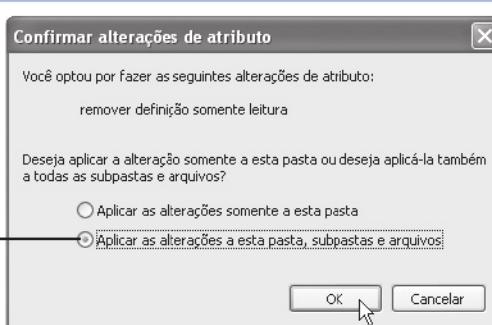


Figura 6 Removendo o atributo Somente leitura para todos os arquivos no diretório examples



O principal mérito da língua é a clareza.
Galeno

Nossa vida é desperdiçada em detalhes... Simplifique,
simplifique.
Henry David Thoreau

Ele tinha um talento maravilhoso para empacotar bem os pensamentos e torná-los portáteis.
Thomas B. Macaulay

O homem ainda é o computador mais extraordinário.
John F. Kennedy

Introdução aos computadores, à Internet e à World Wide Web

OBJETIVOS

Neste capítulo, você aprenderá:

Conceitos básicos de hardware e software.

Conceitos básicos de tecnologia de objeto, como classes, objetos, atributos, comportamentos, encapsulamento e herança.

Os diferentes tipos de linguagens de programação.

As linguagens de programação que são mais amplamente utilizadas.

Um típico ambiente de desenvolvimento de programa em C++.

A história da linguagem orientada a objetos usada como sistema de modelagem padrão da indústria, a UML.
A história da Internet e da World Wide Web.

A fazer um test-drive de aplicativos C++ em dois ambientes C++ populares — o GNU C++, que executa em Linux e o Visual C++ .NET da Microsoft, que executa no Windows® XP.

P á m S	<ul style="list-style-type: none"> 1.1 Introdução 1.2 O que é um computador? 1.3 Organização do computador 1.4 Primeiros sistemas operacionais 1.5 Computação pessoal distribuída e computação cliente/servidor 1.6 A Internet e a World Wide Web 1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível 1.8 História do C e do C++ 1.9 C++ Standard Library 1.10 História do Java 1.11 Fortran, COBOL, Ada e Pascal 1.12 Basic, Visual Basic, Visual C++, C# e .NET 1.13 Tendência-chave do software: tecnologia de objeto 1.14 Ambiente de desenvolvimento C++ típico 1.15 Notas sobre o C++ e este livro 1.16 Test-drive de um aplicativo C++ 1.17 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML (obrigatório) 1.18 Síntese 1.19 Recursos na Web
------------------	--

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

1.1 Introdução

Bem-vindo ao C++! Trabalhamos muito para criar o que esperamos que você julgue ser uma experiência de aprendizagem informativa, divertida e amigável. Experiências podem variar quando programada para diferentes públicos, portanto, esta apresentação é especialmente orientada para pessoas que desejam aprender a programar. Como programar, Quinta edição, uma ferramenta de aprendizagem eficaz para cada uma dessas audiências.

A essência do livro enfatiza como alcançar clareza de programa por meio de comprovadas técnicas de programação orientada a objetos — este é um livro de ‘classes e objetos desde o início’ — e os não-programadores aprenderão a programação de maneira intuitiva. A apresentação é clara, direta e abundantemente ilustrada. Ensinamos os recursos do C++ no contexto de programas funcionais completos e mostramos as saídas produzidas quando esses programas são executados em um computador — com a bordagem live-code (código ativo). Os programas de exemplo são incluídos no CD que acompanha este livro ou você pode fazer download desses programas em www.prenhall.com/deitel.

Os primeiros capítulos introduzem os princípios básicos sobre computadores, programação de computadores e a linguagem de programação de computadores em C++, fornecendo uma base sólida para o tratamento mais profundo do C++ nos capítulos posteriores. Programadores experientes tendem a ler os primeiros capítulos rapidamente e, então, acham o tratamento de C++ no restante tanto rigoroso como desafiador.

A maioria das pessoas está familiarizada de alguma maneira com as coisas empolgantes que os computadores fazem. Utilizando um livro-texto, você aprenderá a comandar computadores para fazer essas coisas. Os computadores (freqüentemente referidos como ‘máquinas’ ou ‘computadores’) são controlados por instruções que você escreve para instruir o computador. C++ é uma das mais populares linguagens de desenvolvimento de software atualmente. Este texto fornece uma introdução à programação de C++ padronizada nos Estados Unidos pelo American National Standards Institute (ANSI) em todo o mundo pelos esforços da International Organization for Standardization (ISO).

O uso de computadores está aumentando em quase todos os campos de trabalho. Os custos de computação têm diminuído drasticamente por causa do rápido desenvolvimento em tecnologia de hardware e software. Os computadores que ocupavam grandes quartos e custavam milhões de dólares há algumas décadas agora podem ser gravados em chips de silício menores que uma unha, de apenas alguns poucos dólares. (Aqueles computadores grandes eram chamados de ‘mainframes’ e eram utilizados hoje em negócios, órgãos do governo e na indústria.) Felizmente, o silício é um dos materiais mais abundantes na terra — é um ingrediente comum. A tecnologia do chip de silício tornou a computação tão econômica que um bilhão de computadores de uso geral estão em utilização em todo o mundo, auxiliando pessoas no comércio, indústria, governo e na vida pessoal.

Ao longo dos anos, muitos programadores aprenderam a metodologia de programação estruturada. Agora, com a introdução da programação estruturada e uma estimulante e mais recente metodologia, a programação orientada a objetos. Por que e

ambas? Atualmente, a orientação a objeto é a metodologia de programação chave utilizada pelos programadores. Você criará e com muitos objetos de software neste texto. Entretanto, descobrirá que sua estrutura interna é muitas vezes construída com ténica de programação estruturada. Além disso, a lógica de manipular objetos é ocasionalmente expressa com programação estruturada.

Você está embarcando em um caminho desafiante e recompensador. À medida que avança, se você tiver qualquer dúvida, e-mail para

deitel@deitel.com

Responderemos prontamente. Para manter-se atualizado com o desenvolvimento do C++ na Deitel & Associates, assine nossa gratuita de correio eletrônico, Buzz Online:

www.deitel.com/newsletter/subscribe.html

Esperamos que você goste de aprender com

1.2 O que é um computador?

Um computador é um dispositivo capaz de realizar computações e tomar decisões lógicas milhões e (até bilhões) de vezes mais rapidamente que o homem. Por exemplo, muitos computadores pessoais de hoje podem realizar um bilhão de adições por segundo. Uma pessoa operando uma calculadora de mesa pode gastar a vida toda para fazer cálculos e ainda não concluir a mesma quantidade de cálculos que um poderoso computador pessoal pode realizar em um segundo! (Questões a ponderar: Como você saberia se somou os números corretamente? Como você saberia se o computador somou os números corretamente?) Os rápidos de hoje podem realizar trilhões de adições por segundo!

Os computadores processam o controle de conjuntos de instruções chamados de linguagem de computador. Esses programas orientam o computador por meio de conjuntos ordenados de ações especificadas por pessoas chamadas de programadores.

Um computador consiste em vários dispositivos referidos como hardware (por exemplo, o teclado, a tela, o mouse, o disco rígido, a memória, os DVDs e as unidades de processamento). Os programas que executam em um computador são referidos como software. Os custos de hardware têm caído significativamente nos últimos anos, a ponto de os computadores pessoais terem se tornado muito populares. Neste livro, você aprenderá métodos comprovados que estão reduzindo custos de desenvolvimento de software, programação orientada a objetos e (em nosso Estudo de caso opcional de engenharia de software nos capítulos 2–7, 9 e 13) orientado a objetos.

1.3 Organização do computador

Independentemente das diferenças na aparência física, praticamente todos os computadores podem ser considerados como unidades lógicas de seções:

1. **Unidade de entrada**: Esta é a seção 'receptora' do computador. Ele obtém informações (dados e programas de computador) de dispositivos de entrada e coloca essas informações à disposição das outras unidades para processamento. A maioria das informações é inserida em computadores por meio de dispositivos de entrada, como teclados e mouse. As informações também podem ser inseridas de muitas outras maneiras, incluindo falar com seu computador, digitalizar imagens e fazer seu computador receber informações de uma rede, como a Internet.
2. **Unidade de saída**: Esta é a seção de 'envio' do computador. Ela pega as informações que o computador processou e as coloca em vários dispositivos de saída para tornar as informações disponíveis à utilização fora do computador. A maioria das informações enviadas para a saída de computadores é exibida em telas, impressas em papel ou utilizadas para controlar dispositivos. Os computadores também podem gerar saída de suas informações para redes, como a Internet.
3. **Unidade de memória**: Esta é a seção de armazenamento de relativamente baixa capacidade e rápido acesso do computador. Ela armazena programas de computador enquanto estão sendo executados. Retém informações que foram inseridas pela unidade de entrada, para se tornarem imediatamente disponíveis para processamento quando necessário. A unidade de memória também retém informações processadas até que elas possam ser colocadas em dispositivos de saída pela unidade de saída. As informações na unidade de memória são, em geral, perdidas quando o computador é desligado. A unidade de memória costuma ser chamada de memória principal [Historicamente, essa unidade era chamada de 'memória básica', mas o uso desse termo está desaparecendo gradualmente hoje.]
4. **Unidade de aritmética e lógica (arithmetic and logic unit ALU)**: Esta é a seção de 'produção' do computador. Ela é responsável pela realização de cálculos, como adição, subtração, multiplicação e divisão. Contém os mecanismos de decisão que permitem ao computador, por exemplo, comparar dois itens da unidade de memória para determinar se são iguais ou não.
5. **Unidade central de processamento (central processing unit CPU)**: Esta é a seção 'administrativa' do computador. Ela coordena e supervisiona a operação das outras seções. A CPU diz à unidade de entrada quando as informações devem ser transferidas para a unidade de memória, informa à ALU quando as informações da unidade de memória devem ser utilizadas para realizar cálculos e instrui a unidade de saída sobre quando enviar as informações da unidade de memória para certos dispositivos de saída.

saída. Muitos computadores de hoje têm múltiplas CPUs e, portanto, podem realizar muitas operações simultaneamente. Computadores são chamados de **múltiplos usuários**.

6. **Unidade de armazenamento secundário** Esta é a seção de armazenamento de alta capacidade e longo prazo do computador. Programas ou dados que não são utilizados ativamente pelas outras unidades, em geral, são colocados em dispositivos de armazenamento secundário, como as unidades de disco, até que sejam novamente necessários, possivelmente horas, dias ou mesmo anos mais tarde. As informações no armazenamento secundário exigem muito mais tempo para serem acessadas que as informações na memória principal, mas o custo por unidade de armazenamento secundário é muito menor que a memória principal. Outros dispositivos de armazenamento secundários incluem CDs e DVDs, que podem armazenar centenas de milhares de caracteres e bilhões de caracteres, respectivamente.

1.4 Primeiros sistemas operacionais

Os primeiros computadores podiam realizar apenas **uma tarefa de vez**. Isso costuma ser chamado de **processamento em lotes**.

Quando um usuário executava um único programa para uma tarefa, processamento de dados em lote era feito, com frequência, tinham de esperar horas ou até mesmo dias antes de as impressões retornarem para suas mesas.

Os sistemas de software **sistemas operacionais** foram desenvolvidos para tornar a utilização de computadores mais conveniente. Os primeiros sistemas operacionais tornaram a transição entre trabalhos mais suave e mais rápida e, portanto, aumentaram a quantidade de trabalho que os computadores poderiam processar.

Quando os computadores se tornaram mais poderosos, ficou evidente que o processamento em lotes de um único usuário e cliente, porque uma grande quantidade de tempo era gasta esperando dispositivos de entrada/saída lentos completarem suas tarefas. Muitos trabalhos ou tarefas **compartilhavam** recursos do computador para alcançar melhor utilização. Isso é alcançado pela **multiprogramação**. A multiprogramação envolve a operação simultânea de muitos trabalhos que competem para compartilhar os recursos do computador. Com os primeiros sistemas operacionais baseados em multiprogramação, os usuários ainda submetiam trabalhos em unidades de cartões perfurados e esperavam horas ou dias para resultados.

Na década de 1960, vários grupos na indústria e nas universidades foram os pioneiros dos sistemas operacionais de **tempo compartilhado**. O compartilhamento de tempo é um caso especial da multiprogramação em que usuários acessam o computador meio de terminais, em geral dispositivos com teclados e telas. Dúzias ou até mesmo centenas de usuários compartilham o computador uma vez. Na verdade, o computador não executa todos eles simultaneamente. Em vez disso, executa uma pequena parte do trabalho de um usuário e, em seguida, atende o próximo usuário, talvez fornecendo serviço para cada usuário várias vezes por segundo. Os programas dos usuários parecem estar executando simultaneamente. Uma vantagem do compartilhamento de tempo é que as solicitações de usuários recebem respostas quase imediatas.

1.5 Computação pessoal distribuída e computação cliente/servidor

Em 1977, a Apple Computer popularizou a **computação pessoal**. Os computadores tornaram-se tão econômicos que as pessoas podiam comprá-los para utilização pessoal ou profissional. Em 1981, a IBM, o maior fornecedor de computadores do mundo, lançou o Personal Computer. Essa computação pessoal rapidamente legitimada em negócios, indústria e órgãos do governo, como muitos da IBM, era intensamente utilizada.

Esses computadores eram unidades ‘independentes’ — as pessoas transportavam discos de um lado para o outro para o compartilhamento de informações (frequentemente chamado de ‘rede peão’). Embora computadores pessoais antigos não fossem suficientemente poderosos para compartilhamento de tempo de vários usuários, essas máquinas podiam ser interconectadas em redes de comunicação, às vezes por linhas telefônicas e, às vezes (em area networks – LAN), dentro de uma organização. Isso levou ao fenômeno da **computação distribuída**, em que a computação de uma organização, em vez de ser realizada somente em alguma central de processamento, é distribuída por redes para os sites em que o trabalho da organização é realizado. Os computadores pessoais são bastante poderosos para tratar os requisitos de computação de usuários individuais, bem como as tarefas básicas de comunicação de passar eletronicamente informações entre computadores.

Os computadores pessoais de hoje são tão poderosos quanto as máquinas de milhões de dólares de apenas algumas décadas atrás. As máquinas desktop mais poderosas, **as chamadas**, fornecem enormes capacidades a usuários individuais. As informações são facilmente compartilhadas através de redes de computadores **em que os computadores** chamados de **computação cliente/servidor**.

Computação cliente/servidor tornou-se largamente utilizado para escrever software para sistemas operacionais, redes de computadores e aplicativos cliente/servidor distribuídos. Sistemas operacionais populares de hoje, como UNIX, Linux, Mac OS X e sistemas baseados no Windows da Microsoft, fornecem os tipos de capacidades discutidas nesta seção.

1.6 A Internet e a World Wide Web

A **Internet** — uma rede global de computadores — foi iniciada há quase quatro décadas com o financiamento fornecido pelo Departamento de Defesa dos Estados Unidos. Originalmente projetada para conectar os principais sistemas de computador de cerca de duas dezenas de universidades e organizações de pesquisa, a Internet hoje é acessível por computadores em todo o mundo.

Com o surgimento da **World Wide Web** — que permite aos usuários de computador localizar e visualizar documentos baseados em multimídia sobre quase qualquer assunto pela Internet — a Internet explodiu, tornando-se um dos principais mecanismos de comunicação do mundo.

A Internet e a World Wide Web estão certamente entre as criações mais importantes e profundas da raça humana. A grande maioria dos aplicativos de computador era executada em computadores que não se comunicavam entre si. Os aplicativos de hoje eram escritos para se comunicar entre os computadores do mundo. A Internet funde computação e tecnologias de comunicação. Faz nosso trabalho mais fácil. Ela torna informações acessíveis mundialmente de maneira instantânea e conveniente. Permite aos interessados empresas de pequeno porte local obter exposição mundial. E está mudando a maneira como os negócios são feitos. As pessoas podem procurar os melhores preços em praticamente todos os produtos ou serviços. Comunidades de interesse especial podem permanecer em contato entre si. Os pesquisadores podem tornar-se instantaneamente clientes dos últimos avanços científicos. Depois de Capítulo 19, "Programação Web", você será capaz de desenvolver aplicativos de computador baseados na Internet.

1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível

Os programadores escrevem instruções em várias linguagens de programação, algumas diretamente compreensíveis por computadores e outras requerendo passos intermediários. Muitas linguagens de computador estão atualmente em uso. Essas linguagens podem ser divididas em três tipos gerais:

1. Linguagens de máquina
2. Linguagens assembly
3. Linguagens de alto nível

Qualquer computador pode entender diretamente **instruções de sua própria linguagem de máquina**. A linguagem de máquina é a 'linguagem natural' de um computador e como tal é definida pelo seu hardware. A linguagem de máquina é, muitas vezes, referida como **objeto**. Este termo é anterior à 'programação orientada a objetos'. Essas duas utilizações de 'objeto' estão relacionadas.] As linguagens de máquina consistem geralmente em strings de números (em última instância reduzidas a 1s e 0s) que instruem os computadores a realizar suas operações mais elementares uma de cada vez. As linguagens de máquina são de máquina (isto é, uma linguagem particular de máquina pode ser utilizada apenas em um tipo de computador). Essas linguagens são muito complexas para os seres humanos, como ilustrado pela próxima seção de um antigo programa de linguagem de máquina que ganha horas extras ao salário-base e armazena o resultado no salário bruto:

```
+1300042774  
+1400593419
```

⁺¹²⁰⁰²⁷⁴⁰²⁷ A programação de linguagem de máquina era simplesmente muito lenta, tediosa e propensa a erro para a maioria dos programadores. Em vez de utilizar as strings de números que os computadores poderiam entender diretamente, os programadores começaram a usar abreviações em inglês para representar operações elementares. Essas abreviações formaram a base das tradutores chamados **assembler**, que foram desenvolvidos para converter os primeiros programas de linguagem assembly em linguagem de máquina a velocidades de computador. A seção a seguir de um programa de linguagem assembly também soma os ganhos extras ao salário-base e armazena o resultado no salário bruto:

```
load    basepay  
add     overpay  
store   grosspay
```

Embora tal código seja mais claro para humanos, ele é incompreensível para computadores até ser traduzido em linguagem de máquina.

O uso de computadores aumentou rapidamente com o advento de linguagens assembly, mas os programadores ainda tinham muitas instruções para realizar até as tarefas mais simples. Para acelerar o processo de programação foram desenvolvidas linguagens de alto nível em que instruções únicas poderiam ser escritas para realizar tarefas substanciais. Os programas tradutores chamados compiladores convertem os programas de linguagem de alto nível em linguagem de máquina. Linguagens de alto nível permitem aos programadores escrever instruções que se parecem com o inglês cotidiano e contêm notações matemáticas comumente usadas. Um programa de folha de pagamentos escrito em uma linguagem de alto nível poderia conter uma instrução assim:

```
grossPay = basePay + overtimePay;
```

Do ponto de vista do programador, obviamente, as linguagens de alto nível são preferíveis à linguagem de máquina e linguagem assembly. O C, o C++, as linguagens .NET da Microsoft (por exemplo, Visual Basic .NET, Visual C++ .NET e C#) e o Java estão entre as linguagens de programação de alto nível mais amplamente utilizadas.

O processo de compilação de um programa de linguagem de alto nível em linguagem de máquina pode consumir uma quantidade considerável de tempo de processamento. Programas desenvolvidos para executar programas de linguagem de alto nível diretamente, embora muito mais lentamente. Os interpretadores são populares em ambientes de desenvolvimento devido ao fato de que novos recursos são adicionados, e os erros, corrigidos. Depois que um programa está completamente desenvolvido, é compilado para produzir uma versão executável.

1.8 História do C e do C++

O C++ desenvolveu-se a partir do C, que se desenvolveu a partir de duas linguagens de programação anteriores, o BCPL e o B. Foi desenvolvido em 1967 por Martin Richards como uma linguagem para escrever software de sistemas operacionais e compilado para sistemas operacionais. Ken Thompson modelou muitos recursos em sua linguagem B com base nas suas contrapartes em C, utilizando o B para criar versões anteriores do sistema operacional UNIX na Bell Laboratories em 1970.

A linguagem C foi desenvolvida da linguagem B por Dennis Ritchie na Bell Laboratories. O C utiliza muitos conceitos importados do BCPL e B. No início, o C tornou-se amplamente conhecido como a linguagem de desenvolvimento do sistema operacional UNIX, a maioria de sistemas operacionais é escrita em C e/ou C++. O C está agora disponível para a maioria dos computadores e é independente de hardware. Com design cuidadoso, é possível escrever programas portáveis para diferentes tipos de computadores.

A utilização disseminada de C com vários tipos de computadores (às vezes chamados de máquinas) levou a muitas variações. Isso era um problema sério para desenvolvedores de programa, que precisavam escrever programas portáveis que executassem em várias plataformas. Uma versão-padrão de C era necessária. O American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) padronizaram o C em todo o mundo; o documento-padrão em cor ANSI ISO 9899: 1990 foi publicado em 1990 e é referido como



Dica de portabilidade 1.1

Como o C é uma linguagem padronizada, amplamente disponível e independente de hardware, os aplicativos escritos em C podem ser freqüentemente executados com pouca ou nenhuma modificação em um amplo espectro de sistemas de computador.

O C++, uma extensão do C, foi desenvolvido por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. O C++ adiciona vários recursos que aprimoram a linguagem C, mas, sobretudo, proporciona capacidades de programação orientada a objetos.

Uma revolução na comunidade de softwares está em progresso. Construir software de maneira rápida, correta e econômica permanece um objetivo difícil de alcançar e isso em uma época em que a demanda por novos e mais poderosos programas de computador cresce vertiginosamente. A programação orientada a objetos é essencialmente de software reutilizável que modelam itens do mundo real. Os desenvolvedores de software estão descobrindo que utilizar um projeto e uma abordagem de implementação modulares e orientados a objetos pode torná-los muito mais produtivos do que seriam com o uso de técnicas anteriores de programação. Programas orientados a objetos são mais fáceis de entender, corrigir e modificar.

1.9 C++ Standard Library

Os programas C++ consistem em partes chamadas

grâmas. Portanto, a maioria dos programas usa a programação orientada a objetos. A programação orientada a objetos depende a propriedades de classe; a se é aprender a utilizar as classes e funções na C++ Standard Library. Por todo o livro, discutimos muitas dessas classes e funções. P. J. Plauger, Standard C Library (Upper Saddle River, NJ: Prentice Hall PTR, 1992), é uma leitura obrigatória para programadores que precisam de um entendimento profundo sobre as funções de biblioteca ANSI C incluídas no C++, como implementá-las e utilizá-las para escrever código portável. Em geral, as bibliotecas de classes padrão são disponibilizadas por fornecedores de coleções. Muitas bibliotecas de classes de uso especial são disponibilizadas por fornecedores de softwares independentes.



Observação de engenharia de software 1.1

Utilize uma abordagem de 'blocos de construção' para criar programas. Evite reinventar a roda. Utilize partes existentes sempre que possível. Chamada de **reutilização de software**, essa prática é fundamental para a programação orientada a objetos.



Observação de engenharia de software 1.2

Ao programar em C++, você normalmente utilizará os seguintes blocos de construção: classes e funções da C++ Standard Library, classes e funções que você e seus colegas criam, e classes e funções de várias bibliotecas independentes populares.

Incluímos muitas observações de engenharia de software por todo o livro para explicar conceitos que afetam e melhoram

o seu trabalho de programação. Deixamos claras dicas para testar e — ou remover erros de programação comuns (técnicas para examinar e evitar o desempenho de sistemas de software); ajudá-lo a escrever programas que executam mais rapidamente (técnica de utilização de técnicas para ajudá-lo a escrever programas que podem ser executados, com pouca ou nenhuma modificação, em uma variedade de computadores); essas dicas também incluem observações gerais sobre como o C++ alcança um alto grau de portabilidade) e erros (técnicas para remover bugs de programas e, sobretudo, técnicas para escrever programas livres de bugs em primeiro lugar). dessas técnicas são apenas diretrizes. Sem dúvida, você desenvolverá seu próprio estilo de programação preferido.

A vantagem de criar suas próprias funções e classes é que você saberá exatamente como elas funcionam. Você será capaz de lidar com o código C++. A desvantagem é o demorado e complexo esforço para projetar, desenvolver e manter novas funções e classes corretas e operarem eficientemente.



Dica de desempenho 1.1

Utilizar funções e classes da C++ Standard Library em vez de escrever suas próprias versões pode melhorar o desempenho do programa, porque elas são escritas cuidadosamente para executar com eficiência. Essa técnica também diminui o tempo de desenvolvimento de programas.



Dica de portabilidade 1.2

Utilizar funções e classes da C++ Standard Library em vez de escrever suas próprias funções e classes melhora a portabilidade do programa, porque elas são incluídas em cada implementação C++.

1.10 História do Java

Os microprocessadores têm um impacto profundo em dispositivos inteligentes eletrônicos voltados para o consumo popular. Nascendo isso, a Sun Microsystems financiou, em 1991, um projeto de pesquisa corporativa interna com o codinome Green. O resultado no desenvolvimento de uma linguagem baseada em C++ que seu criador, James Gosling, chamou de Oak em nome de uma árvore de carvalho vista por sua janela na Sun. Descobriu-se mais tarde que já havia uma linguagem de computador chamada Green. Quando um grupo da Sun visitou uma cafeteria (lidação de streaming de um tipo de café importado) foi sugerido; e o nome pegou.

O projeto Green passou por algumas dificuldades. O mercado de dispositivos eletrônicos inteligentes voltados para o consumo não se desenvolvia no início da década de 1990 tão rapidamente quanto a Sun antecipara. O projeto corria o risco de ser cancelado, mas a World Wide Web explodiu em popularidade em 1993, e a Sun percebeu o imediato potencial de utilizar Java para conteúdos dinâmicos (por exemplo, interatividade, animações e assim por diante) a páginas Web. Isso deu nova vida ao projeto.

Em 1995, a Sun anunciou formalmente o Java. O Java gerou interesse imediato na comunidade de negócios por causa do sucesso da World Wide Web. O Java é agora utilizado para desenvolver aplicativos corporativos de grande porte, melhorar a qualidade de servidores Web (os computadores que fornecem o conteúdo que vemos em nossos navegadores Web), fornecer APIs para dispositivos voltados para o consumo popular (como telefones celulares, pagers e PDAs) e para muitos outros propósitos atuais do C++, como o Visual Studio da Microsoft C++Builder™ da Embarcadero.

1.11 Fortran, COBOL, Ada e Pascal

Centenas de linguagens de alto nível foram desenvolvidas, mas somente algumas, como o FORTRAN (FORmula TRANslator) foram amplamente aceitas. O FORTRAN foi uma linguagem desenvolvida pela IBM Corporation em meados da década de 1950 para ser utilizada em aplicativos científicos de engenharia que exigem complexos cálculos matemáticos. O FORTRAN ainda é amplamente utilizado, especialmente em aplicativos de engenharia.

O COBOL (Common Business Oriented Language) foi desenvolvido no final da década de 1950 por fabricantes de computadores e usuários de computadores no governo norte-americano e na indústria. O COBOL é utilizado para aplicativos comerciais que manipulam precisamente grandes quantidades de dados. Grande parte do software utilizado em grandes empresas ainda é programada em COBOL.

Durante a década de 1960, muitos grandes esforços no desenvolvimento de software encontraram diversas dificuldades. E os prazos de entrega de software atrasavam, os custos excediam enormemente os orçamentos e os produtos finais não eram de qualidade. As pessoas começaram a perceber que o desenvolvimento de software era uma atividade muito mais complexa do que havia sido. Uma pesquisa feita na década de 1960 resultou na evolução de uma abordagem disciplinada para escrever programas mais claros, mais fáceis de testar e depurar e mais fáceis de modificar do que os grandes programas produzidos com a técnica anterior.

Um dos resultados mais tangíveis dessa pesquisa foi o desenvolvimento da linguagem de programação Pascal. A linguagem Pascal foi criada por Niklaus Wirth em 1971. Parte de seu nome veio de Blaise Pascal, matemático e filósofo francês do século XVII. A linguagem foi criada para ser usada em ambientes de laboratório de programação. O Pascal não possui muitos recursos necessários em aplicativos comerciais, industriais e governamentais, assim não foi amplamente aceito nesses ambientes.

A linguagem de programação Ada foi desenvolvida sob o patrocínio do Departamento de Defesa (DOD) dos Estados Unidos durante a década de 1970 e o início da década de 1980. Centenas de linguagens separadas foram utilizadas a fim de produzir temas de software para controle e comando maciços do DOD. O DOD queria que uma única linguagem atendesse a maioria das necessidades. A linguagem foi batizada como Ada em homenagem a Lady Ada Lovelace, filha do poeta Lord Byron. Lady Lovelace é considerada a primeira pessoa a escrever um programa de computador do mundo no início do século XIX (para o dispositivo de computação conhecido como Máquina Analítica, projetado por Charles Babbage). Uma importante capacidade da linguagem Ada é a capacidade de lidar com tipos de dados complexos.

chamada **multitarefa**, permite que os programadores especifiquem quantas atividades devem ocorrer paralelamente. O Java, por meio de uma técnica chamada multithreading, também permite que os programadores escrevam programas com atividades paralelas que não fazem parte do padrão C++, o multithreading é disponibilizado por várias bibliotecas de classes suplementares.

1.12 Basic, Visual Basic, Visual C++, C# e .NET

A linguagem de programação **BASIC** (Beginner's All-Purpose Symbolic Instruction Code) foi desenvolvida em meados da década de 1960, no Dartmouth College, como um meio de escrever programas simples. O principal propósito do BASIC foi familiarizar iniciantes com as técnicas de programação. A linguagem Visual Basic da Microsoft, introduzida no início de 1990 para simplificar o desenvolvimento de aplicativos Microsoft Windows, tornou-se uma das linguagens de programação mais populares no mundo.

As últimas ferramentas de desenvolvimento da Microsoft fazem parte de sua estratégia de escopo corporativo para integrar a linguagem de programação **C#** (C-sharp) e implementá-la em todos os sistemas operacionais Microsoft, que fornece aos desenvolvedores as capacidades necessárias para criar e executar aplicativos de computador que possam executar em computadores de qualquer tipo e na Web em aplicativos de computador. Essa estratégia é implementada por Microsoft, que fornece aos desenvolvedores as capacidades necessárias para criar e executar aplicativos de computador que possam executar em computadores de qualquer tipo e na Web em aplicativos de computador.

Visual C++ .NET baseada no C# (uma nova linguagem baseada no C++ e no Java que foi desenvolvida especificamente para a plataforma .NET). Os desenvolvedores que utilizam .NET podem escrever componentes de software na linguagem com as quais estão mais familiarizados e então formar aplicativos combinando esses componentes com componentes escritos em qualquer linguagem .NET.

1.13 Tendência-chave do software: tecnologia de objeto

Um dos autores, Harvey Deitel, lembra-se da grande frustração que foi sentida, na década de 1960, por organizações de desenvolvimento de softwares, especialmente as que trabalhavam em projetos de larga escala. Durante seus anos universitários, ele teve o privilégio de trabalhar nas férias para um importante fornecedor de computador em equipes de desenvolvimento de sistemas operacionais de tempo real e compartilhamento de tempo. Essa foi uma grande experiência para um aluno de faculdade. Mas, no verão de 1967, a reunião se impôs quando a empresa se “descomprometeu” a produzir comercialmente um sistema particular em que centenas das pessoas trabalhado por muitos anos. Era difícil fazer esse software funcionar. Software é um ‘material complexo’.

Os aprimoramentos na tecnologia de software emergiram com os benefícios da programação estruturada (e as disciplinas de **análise e design de sistemas estruturados**) sendo realizada na década de 1970. Mas foi somente quando a tecnologia de programação orientada a objetos se tornou amplamente utilizada na década de 1990 que os desenvolvedores de software sentiram, por fim, que as ferramentas necessárias para dar passos importantes no processo de desenvolvimento de software.

De fato, a tecnologia de objetos data de meados de 1960. A linguagem de programação C++, desenvolvida na AT&T por Bjarne Stroustrup, é considerada a linguagem de programação orientada a objetos mais antiga. Ela surgiu em 1970, durante a década de 1970, linguagem originalmente projetada para sistemas operacionais AT&T, que era usada na Europa e lançada em 1967. O C++ absorveu os recursos do C e acrescentou capacidades do Simula para criar e manipular estruturas de dados. Nem o C nem o C++ foram originalmente projetados para larga utilização fora dos laboratórios de pesquisa da AT&T. Mas o suporte a grandes sistemas de software se desenvolveu rapidamente para cada linguagem.

O que são objetos e por que eles são especiais? Na realidade, a tecnologia de objetos é um esquema de empacotamento que a criar unidades significativas de software. Estas podem ser grandes e fortemente focalizadas em áreas particulares de aplicativos, como objetos data, objetos tempo, objetos cheque, objetos pagamento, objetos fatura, objetos áudio, objetos vídeo, objetos arquivo, registro e assim por diante. De fato, quase todo substantivo pode ser razoavelmente representado como um objeto.

Vivemos em um mundo de objetos. Dê uma olhada à sua volta. Há carros, aviões, pessoas, animais, edifícios, sinais de trânsito, elevadores e muito mais. Antes de as linguagens orientadas a objetos aparecerem, as linguagens de programação (como FORTRAN, COBOL, Pascal, Basic e C) eram o foco das ações (verbos) em vez do foco de coisas ou objetos (substantivos). Os programadores viviam em um mundo de objetos programando utilizando principalmente verbos. Isso tornava a escrita de programas inconveniente. Agora, com a disponibilidade de linguagens orientadas a objetos populares, como C++ e Java, os programadores continuam a viver em um mundo orientado a objetos e podem programar de maneira orientada a objetos. Esse é um processo mais natural do que a programação procedural e resultou em aprimoramentos de produtividade significativos.

Um problema-chave com a programação procedural é que as unidades de programa não espelham facilmente as entidades do mundo real de modo eficaz, então essas unidades não são particularmente reutilizáveis. Não é incomum programadores recomeçarem o projeto a partir do zero a cada novo projeto e ter de escrever software semelhante ‘a partir do zero’. Isso desperdiça tempo e dinheiro, já que as ‘reinventam a roda’ repetidamente. Com a tecnologia de objeto, as unidades de software reutilizáveis, chamadas de **componentes**, tendem a ser muito mais reutilizáveis em projetos futuros. Utilizar bibliotecas de componentes reutilizáveis, como a **(Microsoft Foundation Classes, .NET Framework Class Library da Microsoft)** aquelas produzidas pela Rogue Wave e por muitas outras organizações de desenvolvimento de software, pode reduzir significativamente a quantidade de esforço requerido para certos tipos de sistemas (comparado ao esforço que seria necessário para reinventar essas capacidades em novos projetos).



Observação de engenharia de software 1.3

Extensas bibliotecas de classes de componentes de software reutilizáveis estão disponíveis pela Internet e World Wide Web. Muitas dessas bibliotecas estão disponíveis sem nenhuma taxa.

Algumas organizações afirmam que o benefício-chave que a programação orientada a objetos fornece não é a reutilização. Em vez disso, essas organizações argumentam que a POO tende a produzir software mais compreensível, mais organizado e de manter, modificar e depurar. Isso pode ser significativo, porque foi estimado que até 80 por cento dos custos de software são associados aos esforços senciniais para o desenvolvimento do software, mas com a contínua evolução e manutenção desse software todo o seu tempo de vida.

Quaisquer que sejam os benefícios percebidos da orientação a objeto, está claro que ela será a principal metodologia de programação pelas próximas décadas.

1.14 Ambiente de desenvolvimento C++ típico

Vamos considerar os passos na criação e execução de um aplicativo C++ utilizando um ambiente de desenvolvimento C++ (ilustrado na Figura 1.1). Em geral, os sistemas C++ consistem em três partes: um ambiente de desenvolvimento de programa, a linguagem Standard Library. Os programas C++, em geral, passam por seis fases:

Nota: Essas fases são: editor, pré-processamento, compilação, linkagem, carregamento e execução.

- e . A discussão a seguir explica um típico ambiente de desenvolvimento C++.

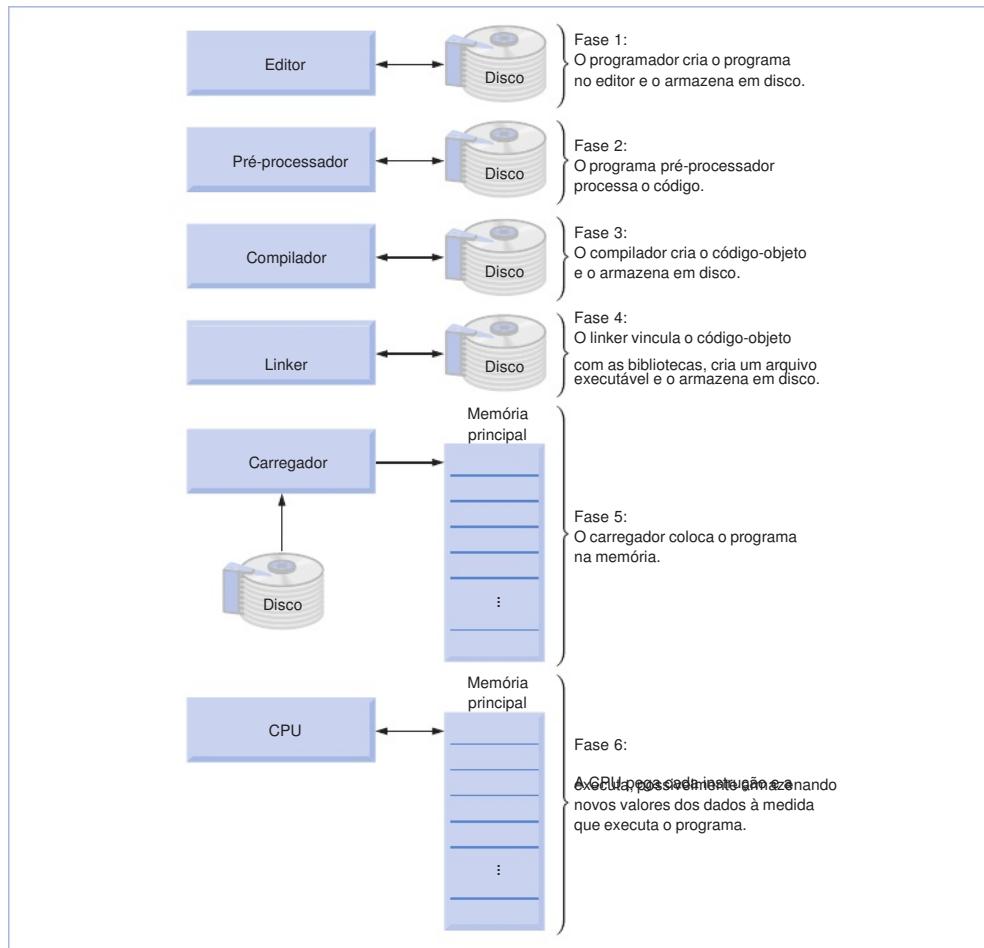


Figura 1.1 Típico ambiente C++.

em www.deitel.com/books/downloads.htm fornecemos as publicações da série Dive Into™ para ajudá-lo a começar a utilizar as várias ferramentas de desenvolvimento C++ populares [C in Boutin e o Microsoft C++ 6; o Microsoft Visual C++.NET, o GNU C++ em Linux e o GNU C++ no ambiente Cygwin]. Nós fornecemos outras publicações da série Dive Into™ disponíveis à medida que instrutores as solicitarem.]

Fase 1: Criando um programa

A Fase 1 consiste em editar um arquivo comum (em geral, conhecido simplesmente como digitação) um programa C++ (em geral referido como) utilizando o editor, faz quaisquer correções necessárias e salva o programa em um dispositivo de armazenamento secundário, tal como sua unidade de disco. Os nomes de arquivo de código-fonte C++ costumam terminar com as extensões .cc ou .c (observe que está em letra maiúscula), que indicam que um arquivo contém código-fonte C++. Consulte a documentação do seu ambiente C++ para obter informações adicionais sobre extensões de nome do arquivo C++.

Dois editores amplamente utilizados em sistemas Linux são os de software C++ para Microsoft Windows como o Borland CodeWarrior (www.borland.com) e o Metrowerks CodeWarrior (www.metrowerks.com) e o Microsoft Visual C++ (www.microsoft.com/visualstudio).

cópia de bloco de notas (bloco de notas) é o Microsoft WordPad, parte de seu pacote de programação. Você também pode utilizar o bloco de notas para escrever código-fonte em arquivos simples.

Fases 2 e 3: Pré-processamento e compilação de um programa C++

Na Fase 2, o programador dá o comando `programa`. Em um sistema C++, `programa` executa automaticamente antes de a fase de conversão do compilador iniciar (então chamamos a fase 2 de pré-processamento, e a fase 3 de compilação). O pré-processador C++ obedece a `comandos chamados de`, que indicam que certas manipulações devem ser realizadas no programa antes da compilação. Essas manipulações normalmente incluem outros arquivos de texto compilados e realizam várias substituições de texto. As diretivas de pré-processador mais comuns são discutidas nos primeiros capítulos. Uma discussão detalhada de todos os recursos de pré-processador aparece no Apêndice F, "Pré-processador". Na fase 3, o `cc` converte o programa C++ em código de linguagem de máquina (também referido como código-objeto).

Fase 4: Vinculando

A Fase 4 é chamada de `linkagem`. Em geral, os programas C++ contêm referências a funções e dados definidos em outra parte, como nas bibliotecas-padrão ou nas bibliotecas privadas de grupos de programadores que trabalham em um projeto particular. Normalmente, o código-objeto produzido pelo compilador C++ contém 'lacunas' devido a essas partes ausentes do código. O linkador une o código das funções ausentes para produzir uma (sem partes ausentes). Se o programa compilar e vincular corretamente, uma imagem executável é produzida.

Fase 5: Carregamento

A Fase 5 é chamada de `carregamento`. Antes que um programa possa ser executado, ele deve primeiro ser colocado na memória. Isso é feito pelo `carregador` que pega a imagem executável do disco e a transfere para a memória. Componentes adicionais de biblioteca compartilhadas que suportam o programa também são carregados.

Fase 6: Execução

Por fim, o computador, sob o controle da `memória`, executa o programa uma instrução por vez.

Problemas que podem ocorrer em tempo de execução

Os programas nem sempre funcionam na primeira tentativa. Cada uma das fases anteriores pode falhar por causa de vários erros discutidos por todo o livro. Por exemplo, um programa executável poderia tentar realizar uma operação de divisão por zero (uma operação ilegal para a aritmética de número inteiro em C++). Isso faria o programa C++ imprimir uma mensagem de erro. Se ocorresse, você teria de retornar à fase de edição, fazer as correções necessárias e passar novamente pelas demais fases para as correções resolverem o(s) problema(s).

A maioria dos programas em C++ realiza entrada e/ou saída de dados. Certas `funções C++` obtêm sua entrada de `entrada padrão` (pronuncia-se 'ci-in'), que normalmente é o `pedaço de redirecionado para outro dispositivo`. A saída dos dados é freqüentemente para `saída padrão` (pronuncia-se 'ci-out'), que, normalmente, é a tela do computador, mas `sair` pode ser redirecionado para outro dispositivo. Quando dizemos que um programa imprime um resultado, normalmente é significado que o resultado é exibido na tela. Os resultados de saída também podem ser direcionados para dispositivos de saída como discos e impressoras. Erros de tempo de execução fazem com que os programas sejam imediatamente encerrados sem terem realizado seus trabalhos com sucesso. Erros de tempo de execução não permitem que os programas executem até sua conclusão, produ-



Erro comum de programação 1.1

Os erros, como divisão por zero, ocorrem enquanto um programa executa, então são chamados de `erros de runtime`. Erros de tempo de execução fazem com que os programas sejam imediatamente encerrados sem terem realizado seus trabalhos com sucesso. Erros de tempo de execução não permitem que os programas executem até sua conclusão, produ-

Note zindo freqüentemente resultados incorretos em alguns sistemas, a divisão por zero não é um erro fatal. Veja a documentação do seu sistema.]

1.15 Notas sobre o C++ e este livro

Programadores em C++ experientes às vezes ficam orgulhosos por serem capazes de fazer algum uso exótico, desvirtuado ou da linguagem. Essa é uma prática de programação pobre. Ela torna os programas mais difíceis de ler, mais propensos a se estranhamente, mais difíceis de testar e depurar e mais difíceis de adaptar em caso de alteração de requisitos. Este livro é projeto para programadores iniciantes, então salientamos a clareza do programa. Segue nossa primeira ‘boa prática de programação’.



Boa prática de programação 1.1

Escreva seus programas C++ de uma maneira simples e direta. Isso é às vezes chamado KIS ('keep it simple', 'mantenha a coisa simples'). Não ‘estenda’ a linguagem tentando usos bizarros.

Você já ouviu que o C e o C++ são linguagens portáveis e que programas escritos em C e C++ podem ser executados em computadores diferentes. Portabilidade é um objetivo vago. O documento-padrão C ANSI contém uma longa lista de questões de portabilidade; e livros completos foram escritos para discutir portabilidade.



Dica de portabilidade 1.3

Embora seja possível escrever programas portáveis, há muitos problemas entre diferentes compiladores C e C++ e diferentes computadores que podem dificultar o alcance da portabilidade. Escrever programas em C e C++ não garante a portabilidade. Com freqüência, o programador precisará lidar diretamente com variações de compilador e computador. Como um grupo, esses problemas são às vezes chamados de variações de forma.

Auditamos nossa apresentação com o documento ANSI/ISO C++ padrão para alcançar completude e exatidão. Entretanto, é uma linguagem rica e há alguns recursos que não abrangemos. Se precisar de detalhes técnicos adicionais sobre o C++, recomendação é que você leia o documento-padrão C++, que pode ser encomendado no site Web da ANSI em webstore.ansi.org/ansidocstore/default.aspx

O título do documento é ‘Information Technology – Programming Languages – C++’ e seu número de documento é INCITS/ISO/14882-2003.

Incluímos uma extensa bibliografia de livros e artigos sobre C++ e programação orientada a objetos. Além disso, incluímos um apêndice de recursos sobre C++ na Internet e em sites Web que se relacionam com C++ e a programação orientada a objetos. Vários sites Web na Seção 1.19, incluindo links para compiladores C++ gratuitos, sites de recursos e alguns jogos C++ divertidos, tutoriais de programação de jogos.



Boa prática de programação 1.2

Leia os manuais da versão de C++ que você está utilizando. Consulte com freqüência esses manuais para certificar-se de que está ciente da rica coleção de recursos C++ e de que os está utilizando corretamente.



Boa prática de programação 1.3

Seu computador e seu compilador são bons professores. Se depois de ler o manual da linguagem C++ você ainda não estiver seguro da maneira como um recurso C++ funciona, experimente utilizar um pequeno ‘programa de teste’ e ver o que acontece. Configure suas opções de compilador para ‘avisos máximos’. Estude cada mensagem gerada pelo compilador e corrija os programas para eliminar as mensagens.

1.16 Test-drive de um aplicativo C++

Nesta seção, você irá executar e interagir com seu primeiro aplicativo. Você irá inserir números aleatórios e o aplicativo irá informar se sua suposição era correta ou não. Se sua suposição estiver correta, o aplicativo irá incrementar o contador de acertos. Se sua suposição estiver errada, o aplicativo irá decrementar o contador de erros. O aplicativo irá terminar quando o contador de erros atingir 5. Se não estiver correta, o aplicativo indica se sua suposição é um número maior ou menor que o correto. Não há limite ao número de suposições que você pode fazer para este test-drive, modificamos esse aplicativo a partir do exercício que você será solicitado a criar no Capítulo 6, “Funções e uma introdução à recursão”. Em geral, esse aplicativo seleciona números testes para você adivinhar toda vez que executa-lo, porque ele escolhe os números aleatoriamente. Nossa aplicativo modificado faz as mesmas suposições ‘corretas’ toda vez que você executar o programa. Isso permite utilizar as mesmas suposições e ver os resultados que mostramos quando orientamos você pela interação com seu primeiro aplicativo C++.]

Demonstraremos como executar um aplicativo C++ de duas maneiras – no modo Windows XP e utilizando um shell no Linux (semelhante ao modo Windows). O aplicativo executa de maneira semelhante em ambas as

plataformas. Muitos ambientes de desenvolvimento que estão disponíveis podem compilar, construir e executar aplicativos C++ C++Builder da Borland, Metrowerks, GNU C++, Visual C++ .Net da Microsoft etc. Enquanto não fazemos test-drive de cada um de ambientes, fornecemos informações na Seção 1.19 relacionadas com compiladores C++ disponíveis para download na Internet. seu instrutor para obter informações sobre seu ambiente de desenvolvimento específico. Além disso, fornecemos várias publicações da série Deitel™ para ajudá-lo na introdução com os vários compiladores C++. Esses estão gratuitamente disponíveis para download em www.deitel.com/books/cpptp5/index.html.

Nos passos seguintes, você executará o aplicativo e inserirá vários números para adivinhar o número correto. Os elementos cisionalidade que você vê nesse aplicativo são típicos daqueles que você aprenderá a programar neste livro. Por todo o livro, utilizaremos fontes para distinguir entre os recursos que você vê na tela por exemplo, elementos que não são diretamente relacionados com a tela. Nossa convenção serve para enfatizar elementos de tela como títulos e menus (por exemplo, o menu em uma fonte *bold sans serif*) ou nomes de arquivo, texto exibido por um aplicativo e valores que você deve inserir em um aplicativo (por exemplo, `GuessNumber500`, em uma fonte *Gothic*). Como você já notou, a **destaque da definição** de cada termo aparece em azul e negrito. Para as figuras nesta seção, destacamos a entrada de usuário necessária a cada passo e indicamos

Figura 1.2 Abrindo uma janela Prompt do MS-DOS alterando o diretório.

Executando um aplicativo C++ a partir do Prompt do MS-DOS do Windows XP

1. Verificando sua configuração. **Leia a seção** de começo deste livro-texto para se certificar de que você copiou corretamente os exemplos do livro para a unidade de disco.
2. Localizando o aplicativo concluído. **Abra uma janela** de comando. Para os leitores que utilizam o Windows 95, 98 ou 2000, **selecione** Programas > Acessórios > Prompt do MS-DOS. Para os usuários do Windows XP, selecione Iniciar > Todos os programas > Acessórios > Prompt de comando. Para mudar para o diretório do aplicativo completado, **digite** `cd \examples\ch01\GuessNumber\Windows` depois pressione **Enter** (Figura 1.2). O comando utilizado para alterar diretórios.
3. Executando o aplicativo `GuessNumber`. Agora que você está no diretório que contém o aplicativo, **digite** o comando `GuessNumber` (Figura 1.3) e pressione **Enter**. [Note que é o nome real do aplicativo; entretanto, o Windows assume a extensão padrão.]
4. Inserindo sua primeira suposição. O aplicativo **exibe** "Please type your first guess.", então exibe um ponto de interrogação como um prompt na próxima linha (Figura 1.3).
5. Inserindo outra suposição. O aplicativo **exibe** "High. Try again.", o que significa que o valor que você inseriu é maior do que o escolhido pelo aplicativo como a suposição correta. Portanto, você deve inserir um número menor na próxima inserção. No prompt `Please type your first guess.` (Figura 1.5), o aplicativo **exibe** novamente "High. Try again.", porque o valor inserido ainda é maior do que o número da suposição correta.



Figura 1.2 Abrindo uma janela Prompt do MS-DOS alterando o diretório.

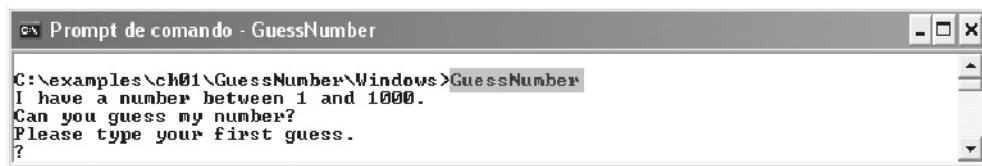


Figura 1.3 Executando o aplicativo `GuessNumber`

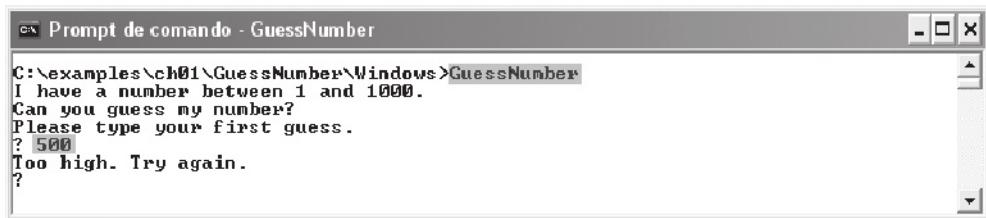


Figura 1.4 Inserindo sua primeira suposição.

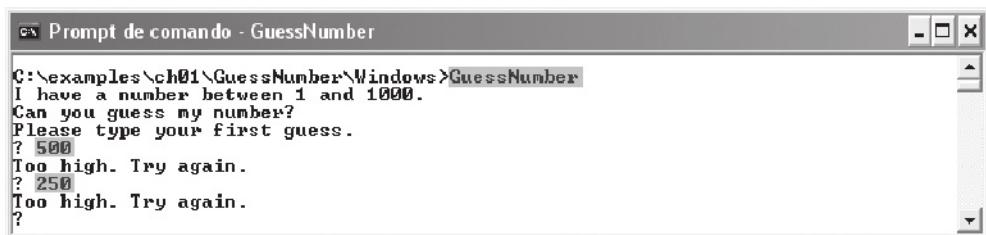


Figura 1.5 Inserindo uma segunda suposição e recebendo feedback.

6. Inserindo suposições adicionais. Continue o jogo inserindo valores até adivinhar o número correto. Uma vez que adivinhar a resposta, o aplicativo exibirá "Excellent! You guessed the number!" (Figura 1.6).
7. Jogando novamente ou saindo do aplicativo. Depois de adivinhar o número correto, o aplicativo pergunta se você gostaria de jogar novamente. Se responder com 'n', o aplicativo encerra. Se responder com 'y', o aplicativo volta ao ponto de interrogação ('?') para que você possa fazer sua primeira suposição no novo jogo. Inserir o caracte fecha o aplicativo e o retorna para o diretório do aplicativo (Figura 1.8). Toda vez que você executar esse aplicativo desde o ícone do desktop, escolherá os mesmos números para você adivinhar.
8. Feche a janela Prompt do MS-DOS.

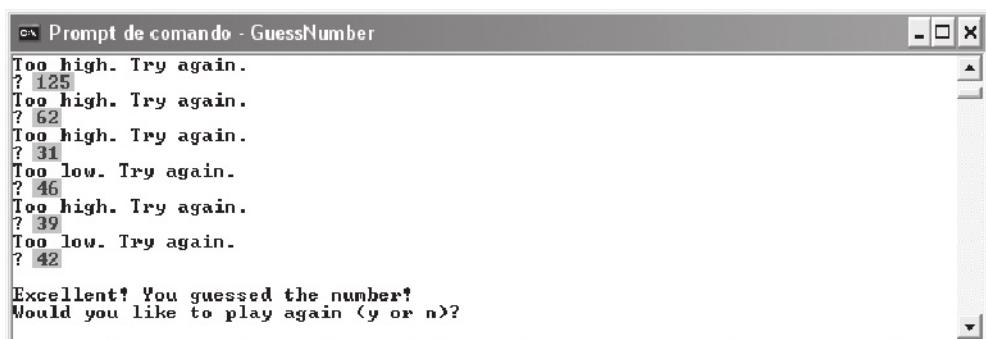


Figura 1.6 Inserindo suposições adicionais e adivinhando o número correto.

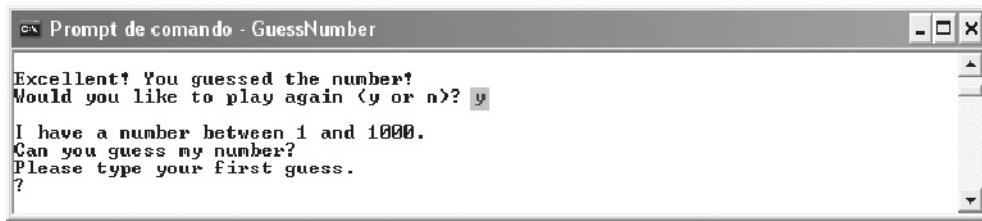


Figura 1.7 Jogando novamente.



Figura 1.8 Saindo do jogo.

Executando um aplicativo C++ utilizando GNU C++ com Linux

Para esse test-drive, assumimos que você sabe copiar os exemplos para seu diretório inicial. Consulte seu instrutor se tiver qualquer dúvida em relação a como copiar os arquivos para o sistema Linux. Além disso, para as figuras nesta seção, utilizamos uma fonte em negrito para indicar a entrada de usuário necessária a cada passo. O prompt no shell é parcialmente visível, mas o sistema utiliza o caractere tilde (~) para representar o diretório inicial e cada prompt termina com o caractere de quebra de linha.

1. Localizando o aplicativo concluído. A partir de um shell do Linux, mude para o diretório do aplicativo completo (Figura 1.9) digitando
cd Examples/ch01/GuessNumber/GNU_Linux
2. Compilando o aplicativo. O comando é utilizado para alterar diretórios.
guessNumber Para executar um aplicativo no compilador GNU C++, ele deve ser primeiro compilado digitando
g++ GuessNumber.cpp -o GuessNumber
3. Executando o aplicativo. Para executar o arquivo executável digitado, GuessNumber no próximo prompt, então pressione Enter (Figura 1.11).
guessNumber
4. Inserindo sua primeira suposição. O aplicativo exibe "Please type your first guess.", então exibe um ponto de interrogação (como um prompt na próxima linha (Figura 1.11)). [Figura 1.12] Esse é o mesmo aplicativo que modificamos e no qual fizemos o test-drive para Windows, mas as saídas poderiam variar, com base no comando usado.]

```
~$ cd examples/ch01/GuessNumber/GNU_Linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

Figura 1.9 Alterando o diretório do aplicativo depois de efetuar logon com sua conta Linux.

```
~/examples/ch01/GuessNumber/GNU_Linux$ g++ GuessNumber.cpp -o GuessNumber
~/examples/ch01/GuessNumber/GNU_Linux$
```

Figura 1.10 Compilando o aplicativo GuessNumber com o comando g++

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Figura 1.11 Executando o aplicativo GuessNumber

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Figura 1.12 Inserindo uma suposição inicial.

5. Inserindo outra suposição no prompt, insira “500”. O aplicativo exibe “Too high. Try again.”, o que significa que o valor que você inseriu é maior que o número que o aplicativo escolheu como a suposição correta (Figura 1.12). No próximo prompt, insira “500” novamente para continuar o jogo.
6. Inserindo suposições adicionais e continue o jogo (Figura 1.14) inserindo valores até você adivinhar o número correto. Quando adivinhar a resposta, o aplicativo exibirá “You guessed the number!” (Figura 1.14).
7. Jogando novamente ou fechando o aplicativo. Depois de adivinhar o número correto, o aplicativo pergunta se você gostaria de ir para outro jogo. No próximo prompt, insira “y” ou “n”, inserir o caractere com que o aplicativo escolha um novo número e exiba a mensagem “Please type your first guess.” seguida por um prompt de ponto de interrogação (Figura 1.15) para que você possa fazer sua primeira suposição fechar o jogo. Inserir o caractere apelativo e o retorna para o diretório do aplicativo no shell (Figura 1.16). Toda vez que você executar esse aplicativo de início (isto é, se ele escolherá os mesmos números para você adivinhar.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
Too low. Try again.
?
```

Figura 1.13 Inserindo uma segunda suposição e recebendo feedback.

```

Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384

Excellent! You guessed the number.
Would you like to play again (y or n)?

```

Figura 1.14 Inserindo suposições adicionais e adivinhando o número correto.

```

Excellent! You guessed the number.
Would you like to play again (y or n)? y

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.

?

```

Figura 1.15 Jogando novamente.

```

Excellent! You guessed the number.
Would you like to play again (y or n)? n

~/examples/ch01/GuessNumber/GNU_Linux$
```

Figura 1.16 Saindo do jogo.

1.17 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML (obrigatório)

Agora vamos começar nossa primeira introdução à orientação a objetos, uma maneira natural de pensar o mundo e escrever programas. Introduzido passo a passo a orientação a objetos! Nossa objetivo aqui é ajudar você a desenvolver uma maneira orientada de pensar e apresentar o **Unified Modeling Language™ (UML™)**, uma linguagem gráfica que permite às pessoas que projetam sistemas de software orientados a objetos utilizar uma notação-padrão da indústria para representá-los.

Nesta seção obrigatória, introduzimos a terminologia e os conceitos básicos da orientação a objetos. As seções opcionais nos capítulos 2–7, 9 e 13 apresentam projeto e implementação orientados a objetos do software para um sistema de caixa automática (*machine- ATM*) simples. As seções “Estudo de caso de engenharia de software” no final dos capítulos 2–7

- analisam um típico documento de requisitos que descreve um sistema de software (o ATM) a ser construído;
- determinam os objetos necessários para implementar esse sistema;

- determinam os atributos que os objetos terão;
- determinam os comportamentos que esses objetos exibirão;
- especificam como os objetos interagem para atender os requisitos de sistema.

As seções “Estudo de caso de engenharia de software” no final dos capítulos 9 e 13 modificam e aprimoram o design apresentado nos capítulos 2–7. O Apêndice G contém uma implementação funcional e completa em C++ do sistema ATM orientado a objetos.

Embora nosso estudo de caso seja uma versão reduzida de um problema de nível industrial, abrangemos, apesar disso, as práticas comuns da indústria. Você experimentará uma introdução sólida ao projeto orientado a objetos com a UML. Além disso, aperfeiçoará suas habilidades de leitura de código fazendo um passeio pela completa implementação em C++ do ATM cuidadosamente escrita e bem-dокументada.

Conceitos básicos da tecnologia de objeto

Iniciamos nossa introdução à orientação a objetos com uma terminologia-chave. Onde quer que você olhe no mundo real, você

Teléfonos, casas, animais, plantas, corpos, ruínas, edifícios, computadores e assim por diante. Os humanos pensam em termos de objetos ao nosso redor todos os dias.

As vezes dividimos objetos em duas categorias: animados e inanimados. Os objetos animados são, em certo sentido, objetos — eles se movem e fazem coisas. Por outro lado, os objetos inanimados não se movem por conta própria. Objetos de ambos, porém, têm algumas coisas em comum: ~~Todos~~^{Por exemplo}, tamanho, forma, cor e peso. Eles todos exibem comportamentos (por exemplo, uma bola rola, rebate, infla e murcha; o bebê chora, dorme, engatinha, anda e pisca; um carro acelera, é desviado; uma toalha absorve água). Estudaremos os tipos de atributos e comportamentos dos objetos de software.

Os humanos aprendem sobre os objetos existentes estudando seus atributos e observando seus comportamentos. Diferentemente, podem ter atributos semelhantes e podem exibir comportamentos semelhantes. É possível fazer comparações, por exemplo, entre adultos, e entre humanos e chimpanzés.

O projeto orientado a objetos ([object-oriented design OOD](#)) modela software em termos semelhantes àqueles que as pessoas utilizam para descrever objetos do mundo real. Ele tira proveito de relacionamentos de classe, em que os objetos de certa classe, uma classe de veículos, têm as mesmas características — carros, caminhões e patins têm muito em comum. O OOD tira proveito de relacionamentos ~~de classe~~, em que novas classes de objetos são derivadas absorvendo-se características de classes existentes e adicionando-se características únicas dessas mesmas classes. Um objeto da classe ‘conversível’ certamente tem as características mais gerais ‘automóvel’, porém, mais especificamente, seu teto se abre e se fecha.

O projeto orientado a objetos fornece uma maneira natural e intuitiva de visualizar o processo de planejamento do software.

~~As OOD divide os objetos para seu uso individual, os objetos são tratados como entidades separadas entre si, os objetos juntos mandam um soldado permanecer em atenção), os objetos também se comunicam via mensagens. Um objeto conta bancária pode enviar uma mensagem para reduzir seu saldo em certa quantia porque o cliente retirou essa quantia em dinheiro.~~

O OOD encapsula (isto é, empacota) ~~atributos~~ (comportamentos) em objetos — os atributos e as operações de um objeto estão intimamente ligados. Os objetos têm a propriedade de ~~de~~ isso significa que os objetos podem saber como se comunicar com outros ~~por meio de~~ definidas, mas normalmente eles não têm permissão para saber como os outros objetos são implementados — os detalhes de implementação são ocultados dentro dos próprios objetos. Na verdade, para dirigir um carro, por exemplo, sem conhecer os detalhes de como motores, transmissões, freios e sistemas de escapamento funcionam internamente — contanto que saibamos utilizar o acelerador, o freio, a direção e assim por diante. O ocultamento de informações, é crucial à boa engenharia de software.

Linguagens como [C++](#) ([classes a objeto](#)) A programação nessa linguagem é chamada de [orientada a objetos](#) ([object-oriented programming OOP](#)) e permite aos programadores de computador implementar um projeto orientado a objetos como um sistema de software funcional. Linguagens como [C](#) ([classes portadoras](#)) a programação tende a ser [orientada para a ação](#). No C, a unidade de programação é o bloco, a unidade de programação básica da qual os objetos são [por fim criados](#) (um termo da POO para ‘criados’). As classes C++ contêm funções que implementam operações e dados que implementam atributos.

Programadores de C concentram-se em escrever funções. Os programadores agrupam as ações que realizam alguma tarefa em funções e agrupam funções para formar programas. Os dados são certamente importantes em C, mas a visualização e a organização dos dados existem principalmente em suporte das ações que os ~~funcionários~~ realizam. A classificação de sistema ajudam o programador de C a determinar o conjunto de funções que trabalharão juntas para implementar o sistema.

Classes, membros de dados e funções-membro

Os programadores C++ concentram-se em [criar seus próprios classes](#). Cada classe contém dados, bem como o conjunto de funções que manipula esses dados ([métodos](#) ou funções que utilizam a classe). Os componentes de dados de uma classe são [chamados de](#), uma classe conta bancária poderia incluir o número e o saldo de uma conta. Os componentes de função de uma classe são chamados de (em geral, chamados de) outras linguagens de programação orientada a objetos como o Java). Por exemplo, uma classe

conta bancária poderia incluir funções-membro para fazer um depósito (aumentando o saldo), fazer uma retirada (diminuindo o saldo) e consultar o saldo atual. O programador utiliza tipos predefinidos (e outros tipos definidos pelo usuário) como os ‘blocos de construção’ para construir novos tipos definidos pelo usuário ([classes](#)) na especificação de sistema ajudam o programador em C++ a determinar o conjunto de classes a partir da qual os objetos que são criados trabalham juntos para implementar o sistema.

As classes estão para os objetos como as plantas arquitetônicas estão para as casas — uma classe é um ‘plano’ para o objeto da classe. Assim como podemos construir muitas casas a partir de uma planta, podemos instanciar (criar) muitos objetos de uma classe. Você não pode fazer refeições na cozinha de uma planta; isso só é possível em uma cozinha real. Você não pode dormir no quarto de uma planta arquitetônica; você só pode dormir no quarto de uma casa.

As classes podem ter relacionamentos com outras classes. Por exemplo, em um projeto orientado a objetos de um banco, a ‘caixa de banco’ precisa se relacionar com outras classes, como a classe ‘cliente’, a classe ‘gaveta de dinheiro’, a classe ‘cofre’ e os relacionamentos são chamados de [aninhados de classes](#).

Empacotar software como classes possibilita que os sistemas de classes sejam reutilizáveis. Assim como corretores de imóveis costumam dizer que os três fatores mais importantes para a compra de um imóvel são localização, localização e localização, os sistemas de classes devem ser orientados ao fator de ‘reutilização, reutilização e reutilização’.



Observação de engenharia de software 1.4

A reutilização de classes existentes ao construir novas classes e programas economiza tempo, dinheiro e esforço. A reutilização também ajuda os programadores a construir sistemas mais confiáveis e eficientes, porque classes e componentes existentes costumam passar por extensos testes, depurações e ajustes de desempenho.

De fato, com a tecnologia de objetos, você pode construir grande parte do novo software necessário combinando classes existentes como fabricantes de automóveis combinam partes intercambiáveis. Cada nova classe que você cria terá o potencial de tornar um valioso ativo de software que você e outros programadores podem reutilizar para acelerar e aprimorar a qualidade e os esforços futuros no desenvolvimento de software.

Introdução à análise e projeto orientados a objetos (Object-oriented analysis and design – OOAD)

Logo você estará escrevendo programas em C++. Como criar o código para seus programas? Talvez, como muitos programadores, você simplesmente ligará seu computador e começará a digitar. Essa abordagem pode funcionar para pequenos programas apresentados nos primeiros capítulos do livro), mas se você fosse contratado para criar um sistema de software para cont

rofessores de programação automática, sistema de bádminton portátil ou o sistema de eletrônico de vôlei das Escolas Olímpicas. Para sistemas tão complexos, você não pode simplesmente sentar e começar a escrever programas.

Para criar as melhores soluções, você deve seguir um [processo detalhado](#) para projeto (isto é, determinar [o que o sistema deve fazer](#) e [desenvolver](#) que atenda esses requisitos (isto é, [designar](#) o que deve fazê-lo).

Idealmente, você passaria por esse processo e revisaria cuidadosamente o design (ou teria seu design revisado por outros programadores) antes de escrever qualquer código. Se esse processo envolve analisar e projetar o sistema de um ponto de vista de objetos, ele é chamado de [análise e projeto orientados a objetos](#) (object-oriented analysis and design – OOAD). Programadores experientes sabem que análise e design podem poupar muitas horas ajudando a evitar uma abordagem de desenvolvimento de sistemas mal planejada que tem de ser abandonada no meio de sua implementação, possivelmente desperdiçando tempo, esforço considerável.

O OOAD é o termo genérico para o processo de análise de um problema e desenvolvimento de uma abordagem para resolver problemas como os discutidos nesses primeiros poucos capítulos não exigem um processo exaustivo de OOAD. Pode ser suficiente, antes de começarmos a escrever código, [usar um método informal baseado em texto de expressar a lógica do programa](#). Na realidade, o pseudocódigo não é uma linguagem de programação, mas pode ser utilizado como um esboço orientado ao escrever o código. Introduzimos o pseudocódigo no Capítulo 4.

Uma vez que os problemas e os grupos de pessoas que os resolvem aumentam em tamanho, os métodos OOAD rapidamente se tornam inadequados do que o pseudocódigo. Idealmente, um grupo deve estabelecer um acordo comum sobre um processo rigoroso definido para resolver seu problema e sobre uma maneira uniforme de comunicar os resultados desse processo para os outros. Existem muitos processos OOAD diferentes, uma única linguagem gráfica para todos os processos OOAD, chamada de UML, está sendo amplamente utilizada. Essa linguagem, conhecida como Unified Modeling Language (UML), foi desenvolvida em meados da década de 1990 sob a direção inicial de três metodologistas de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

História da UML

Na década de 1980, um número crescente de organizações começou a utilizar POO para construir seus aplicativos e desenvolver necessidade de um processo OOAD padrão. Muitos metodologistas — inclusive Booch, Rumbaugh e Jacobson — produziram e veram individualmente processos separados para satisfazer essa necessidade. Cada processo tinha sua própria notação ou ‘linguagem’ (na forma de diagramas gráficos), para transportar os resultados de análise e design.

Por volta do início de 1990, diferentes organizações e até divisões dentro de uma mesma organização estavam utilizando diferentes processos e notações. Ao mesmo tempo, essas organizações também queriam utilizar ferramentas de software que suportassem seus processos particulares. Os fornecedores de software acharam difícil fornecer ferramentas para tantos processos. Claras notação-padrão e processos-padrão eram necessários.

Em 1994, James Rumbaugh associou-se a Grady Booch na Rational Software Corporation (agora uma divisão da IBM) e começaram a trabalhar para unificar seus populares processos. Eles logo se associaram a Ivar Jacobson. Em 1996, o grupo das primeiras versões da UML para a comunidade de engenharia de software e solicitaram feedback. Por volta da mesma época, a **Object Management Group™ (OMG)** solicitou sugestões para uma linguagem de modelagem comum. O OMG é uma organização sem fins lucrativos que promove a padronização de tecnologias orientadas a objetos publicando diretrizes e especificações, como a UML. Várias corporações — entre elas HP, IBM, Microsoft, Oracle e Rational Software — já reconhecido a necessidade de uma linguagem de modelagem comum. Em resposta à solicitação de propostas do OMG, essas corporações formaram a **UML Partners** — o consórcio que desenvolveu a UML versão 1.1 e a sugeriu para o OMG. O OMG aceitou a proposta e, em 1997, assumiu a responsabilidade pela manutenção e revisão constantes da UML. Em março de 2003, o OMG lançou a UML versão 2.0.

Nesta versão 2 da UML, a versão 1.1 é mantida, mas não é mais utilizada para novas implementações da linguagem. A melhoria da versão 2 é que a UML é agora a única linguagem de modelagem padrão da indústria, utilizando a UML versão 2, então apresentamos a terminologia e a notação da UML versão 2 por todo este livro.

O que é a UML?

A Unified Modeling Language é agora o esquema de representação gráfica mais amplamente utilizado para modelar sistemas e objetos. Ela de fato unificou os vários esquemas de notação populares. Aquelas que projetam sistemas usam a linguagem (de diagramas) para modelar seus sistemas, como fazemos por todo este livro.

Um recurso atraente da UML é sua flexibilidade ([UML é capaz de ser aprimorada com novos recursos](#)) e é independente de qualquer processo OOAD particular. Os modeladores de UML são livres para utilizar vários processos ao projetar; mas agora todos os desenvolvedores podem expressar seus projetos com um conjunto-padrão de notações gráficas.

A UML é uma linguagem gráfica complexa rica em recursos. Em nossas seções “Estudo de caso de engenharia de software” desenvolvimento do software de um caixa automático (ATM), apresentamos um subconjunto conciso e simples desses recursos. Utilizamos esse subconjunto para guiá-lo pela primeira experiência de design com a UML, destinada a programadores iniciar tecnologia orientada a objetos em um curso de programação do primeiro ou segundo semestre.

Esse estudo de caso foi cuidadosamente desenvolvido sob a orientação de revisores acadêmicos e profissionais. Espera-se que você se divirta trabalhando nele. Se tiver qualquer dúvida, entre em contato conosco e responderemos prontamente.

[Recursos da UML na Internet e Web](#)

Para obter informações adicionais sobre a UML, consulte os seguintes sites Web. Para sites sobre UML adicionais, consulte os na Internet e Web listados no fim da Seção 2.8.

[www.uml.org](#)

Esta página de recursos da UML do Object Management Group (OMG) fornece documentos de especificação da UML e outras tecnologias orientadas a objetos.

[www.ibm.com/software/rational/uml](#)

Esta é a página de recurso da UML da IBM Rational — a sucessora da Rational Software Corporation (a empresa que criou a UML).

[Leituras recomendadas](#)

Muitos livros sobre UML já foram publicados. Os seguintes livros recomendados fornecem informações sobre o projeto orientado a objetos com a UML.

- Arlow, J., and I. Neustadt. [UML distilled, second edition: practical object-oriented analysis and design](#). Londres: Pearson Education Ltd., 2002.
- Fowler, M. [UML distilled, third edition: a brief guide to the standard object modeling language](#). Boston: Addison-Wesley, 2004.
- Rumbaugh, J., I. Jacobson, and G. Booch. [The unified modeling language user guide](#). Reading, MA: Addison-Wesley, 1999.

[www.amazon.comwww.](#)

Parágrafo Adicional: Rational sabe que a Rational Software Corporation também é uma das empresas que criaram a UML e que os sites de livros sobre a UML.

[Seção 1.17 Exercícios de revisão](#)

1.1 Liste três exemplos de objetos do mundo real que não mencionamos. Para cada objeto, liste vários atributos e comportamentos.

1.2 Pseudocódigo é _____.

- outro termo para OOAD
- uma linguagem de programação utilizada para exibir diagramas de UML

- c) uma maneira informal de expressar a lógica do programa
 - d) um esquema de representação gráfica para modelar sistemas orientados a objetos
- 1.3 A UML é utilizada principalmente para .
- a) testar sistemas orientados a objetos
 - b) projetar sistemas orientados a objetos
 - c) implementar sistemas orientados a objetos
 - d) a e b são alternativas corretas

Respostas aos exercícios de revisão da Seção 1.17

- 1.1 [Nota: As respostas podem variar.] a) Os atributos de uma televisão incluem o tamanho da tela, o número de cores que ela pode exibir, o canal atual e volume atual. Uma televisão liga e desliga, muda de canais, exibe vídeo e reproduz sons. b) Os atributos de uma cafeteira incluem o volume máximo de água que ela pode conter, o tempo necessário para fazer um bule de café e a temperatura da chapa sob o bule. Uma cafeteira liga e desliga, faz e esquenta café. c) Os atributos de uma tartaruga incluem a idade, tamanho do casco e peso. Uma tartaruga caminha, protege-se dentro de seu casco, sai de seu casco e alimenta-se de vegetais.
- 1.2 c.
- 1.3 b.

1.18 Síntese

Este capítulo introduziu conceitos básicos de hardware e software e explorou o papel do C++ no desenvolvimento de aplicativos cliente/servidor distribuídos. Você estudou a história da Internet e da World Wide Web. Discutimos os diferentes tipos de linguagens de programação, sua história e as linguagens de programação que são mais amplamente utilizadas. Discutimos também a C++ Standard Library, que contém classes e funções reutilizáveis que ajudam os programadores em C++ a criar programas C++ portáveis.

Apresentamos conceitos básicos da tecnologia de objeto, incluindo classes, objetos, atributos, comportamentos, encapsular herança. Você também aprendeu a história e o propósito da UML — a linguagem gráfica padrão da indústria para modelar sistemas.

Você aprendeu os passos típicos para criar e executar um aplicativo C++. Por fim, você fez o ‘test-drive’ de um aplicativo C++ exemplo semelhante aos tipos de aplicativo que você aprenderá a programar neste livro.

No próximo capítulo, você criará seus primeiros aplicativos C++. Você verá vários exemplos que demonstram como exibir mensagens de programas na tela e obter informações do usuário no teclado para processamento. Analisamos e explicamos cada exemplo para ajudá-lo a facilitar a introdução na programação C++.

1.19 Recursos na Web

Esta seção fornece muitos recursos da Web que serão úteis para você à medida que aprende a linguagem C++. Os sites incluem C++, ferramentas de desenvolvimento C++ para alunos e profissionais e alguns links para jogos divertidos construídos com C++. Esta seção também lista seus próprios sites Web onde você pode localizar downloads e recursos associados com este livro. Você encontra recursos na Web adicionais no Apêndice I.

Sites Web da Deitel & Associates

www.deitel.com/books/cppHTP5/index.html
O site do *How to Program*, Fifth Edition da Deitel & Associates. Aqui o leitor encontrará links para os exemplos do livro (também incluídos no CD que acompanha o livro) e outros recursos, *domínio dos signatários* que ajudam a começar a aprender os vários ambientes de desenvolvimento integrado (development environments) C++.

www.deitel.com

Consulte o site da Deitel & Associates para obter atualizações, correções e recursos adicionais de todas as publicações da Deitel.

www.deitel.com/newsletter/subscribe.html
Visite esse site para assinar o boletim de *domínio eletrônico* de seguir o programa de publicações da Deitel & Associates.

www.deitel.com/DeitelHall para as publicações da Deitel. Aqui você encontrará informações detalhadas sobre nossos produtos, capítulos de exemplo e Companion Web Sites. Esse site contém recursos específicos do livro e do capítulo para alunos e instrutores.

Compiladores e ferramentas de desenvolvimento

www.thefreecountry.com/developercy/ccompilers.shtml
Esse site lista compiladores C e C++ gratuitos para uma variedade de sistemas operacionais.

msdn.microsoft.com/visualc

O site Microsoft Visual C++ fornece informações de produto, resumos, materiais suplementares e informações de compra do compilador Visual C++.

www.borland.com/bcppbuilder

Esse é um link para o site da Borland. Uma versão de linha de comando gratuita está disponível para download.

www.compilers.net

O Compilers.net foi projetado para ajudar usuários a localizar compiladores.

developer.intel.com/software/products/compilers/cwin/index.htm

Um download de avaliação do Intel C++ está disponível nesse site.

www.kai.com/C_plus_plus

Esse site oferece uma versão de avaliação de 30 dias da Kai C++.

www.symbian.com/developer/development/cppdev.html

O Symbian fornece um C++ Developer's Pack e links para vários recursos, incluindo código e ferramentas de desenvolvimento para programar C++ que implementam aplicativos móveis para o sistema operacional Symbian, que é popular em dispositivos como telefones celulares.

Recursos

www.hal9k.com/cug

O site C/C++ Users Group (CUG) contém recursos, periódicos, shareware e freeware sobre C++.

www.devx.com

O DevX é um recurso abrangente para programadores que fornece as últimas notícias, ferramentas e técnicas para várias linguagens de programação. O Zone oferece dicas, fóruns de discussão, ajuda técnica e newsletters on-line.

www.acm.org/crossroads/xrds3-2/ovp32.html

O site Association for Computing Machinery (ACM) oferece uma abrangente listagem de recursos C++, incluindo textos recomendados, periódicos e revistas, padrões publicados, newsletters, FAQs e newsgroups.

www.acu.informika.ru/resources/public/terse/cpp.htm

O site Association of C & C++ Users (ACCU) contém links para tutoriais de C++, artigos, informações de desenvolvedor, discussões e sinopses de livros.

www.cuj.com

O C/C++ User's Journal é uma revista on-line que contém artigos, tutoriais e downloads. O site apresenta notícias sobre C++, fóruns e links para informações sobre ferramentas de desenvolvimento.

www.research.att.com/~bs/homepage.html

Esse é o site de Bjarne Stroustrup, projetista da linguagem de programação C++. Esse site fornece uma lista de recursos do C++, FAQs e informações C++ úteis.

Jogos

www.codearchive.com/list.php?go=0708

Esse site tem vários jogos em C++ disponíveis para download.

www.mathtools.net/C_C_Games/

Esse site inclui links para numerosos jogos construídos com C++. O código-fonte para a maioria dos jogos está disponível para download.

www.gametutorials.com/Tutorials/GT/GT_Pg1.htm

Esse site tem tutoriais sobre programação de jogo em C++. Cada tutorial inclui uma descrição do jogo e uma lista dos métodos e funções utilizados no tutorial.

www.forum.nokia.com/main/0,6566,050_20,00.html

Visite esse site da Nokia para aprender a utilizar C++ para programar jogos de alguns dispositivos sem fio da Nokia.

Resumo

- Os vários dispositivos que abrangem um sistema de computador são referidos como hardware.
- Os programas que executam em um computador são referidos como software.
- Um computador é capaz de realizar computações e tomar decisões lógicas a velocidades de milhões e (até bilhões) de vezes mais rápidas que o homem.
- Os computadores processam dados sob o controle de conjuntos de instruções chamados programas de computador, que orientam o computador por meio de conjuntos ordenados de ações especificadas por programadores de computador.
- A unidade de entrada é a seção 'receptora' do computador. Ela obtém informações de dispositivos de entrada e as coloca à disposição das unidades para processamento.

- A unidade de saída é a seção de 'envio' do computador. Ela recebe informações (dados e programas de computador) de vários dispositivos de saída e coloca essas informações à disposição de outras unidades de modo que as informações possam ser processadas.
- A unidade de memória é a seção de armazenamento de relativamente baixa capacidade e acesso rápido do computador. Ela retém informações que foram inseridas pela unidade de entrada, disponibilizando-as imediatamente para processamento quando necessário, e retém informações já foram processadas até que essas informações possam ser colocadas em dispositivos de saída pela unidade de saída.
- A unidade lógica e aritmética (ALU) é a seção de 'fabricação' do computador. É responsável por realizar cálculos e tomar decisões.
- A unidade central de processamento (CPU) é a seção 'administrativa' do computador. Ela coordena e supervisiona a operação das outras unidades.
- A unidade de armazenamento secundário é a seção de armazenamento de alta capacidade e longo prazo do computador. Normalmente, os programas ou dados que não estão sendo utilizados por outras unidades são colocados em dispositivos de armazenamento secundários (por exemplo, discos) até que sejam necessários novamente, possivelmente horas, dias, meses ou mesmo anos mais tarde.
- Os sistemas operacionais foram desenvolvidos para ajudar a tornar o uso de computadores mais conveniente.
- A multiprogramação envolve o compartilhamento dos recursos de um computador entre os trabalhos que disputam sua atenção, de modo que esses pareçam executar simultaneamente.
- Com a computação distribuída, a computação de uma organização é distribuída em redes para os sites em que o trabalho da organização é realizado.
- Qualquer computador pode entender diretamente apenas sua própria linguagem de máquina, que geralmente consiste em strings de númerais instruindo os computadores a realizar suas operações mais elementares.
- Abreviações em inglês formam a base das linguagens assembly. Os programas tradutores chamados assemblers convertem programas em linguagem assembly em linguagem de máquina.
- Compiladores traduzem programas de linguagem de alto nível em programas de linguagem de máquina. Linguagens de alto nível (como C) contêm palavras inglesas e notações matemáticas convencionais.
- Os programas interpretadores executam diretamente os programas de linguagem de alto nível, eliminando a necessidade de compilá-los em linguagem de máquina.
- O C++ evoluiu a partir do C, que evoluiu de duas linguagens de programação, BCPL e B.
- O C++ é uma extensão do C desenvolvida por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. O C++ aprimora a linguagem C e fornece capacidades para programação orientada a objetos.
- Objetos são componentes de software reutilizáveis que modelam itens do mundo real. Utilizar uma abordagem de projeto e implementação modular e orientadas a objetos pode tornar grupos de desenvolvimento de software mais produtivos do que com as técnicas de programação anteriores.
- Os programas C++ consistem em partes chamadas classes e funções. Você pode programar cada parte de que possa precisar para fornecer seu programa C++. Mas a maioria dos programadores em C++ tira proveito das ricas coleções de classes e funções existentes na C++ Standard Library.
- O Java é utilizado para criar conteúdo dinâmico e interativo para páginas Web, desenvolver aplicativos corporativos, aprimorar funcionalidades de servidor Web, fornecer aplicativos para dispositivos de consumo popular e muito mais.
- A linguagem FORTRAN (FORmula TRANslator) foi desenvolvida pela IBM Corporation em meados da década de 1950 para aplicativos científicos e de engenharia que requerem complexos cálculos matemáticos.
- A linguagem COBOL (COmmon Business Oriented Language) foi desenvolvida no final da década de 1950 por um grupo de fabricantes de computador e usuários de computadores do governo e da indústria. O COBOL é utilizado principalmente para aplicativos comerciais que exigem manipulação de dados precisa e eficiente.
- A linguagem Ada foi desenvolvida sob o patrocínio do Departamento da Defesa dos Estados Unidos durante a década de 1970 e início de 1980. A Ada fornece multitarefa, que permite aos programadores especificar que muitas atividades devem ocorrer paralelamente.
- A linguagem BASIC (Beginner's All-Purpose Symbolic Instruction Code) de programação foi desenvolvida em meados de 1960 no Dartmouth College como uma linguagem para escrever programas simples. O principal propósito da BASIC foi familiarizar os iniciantes com as técnicas de programação.
- A linguagem Visual Basic da Microsoft foi introduzida no início da década de 1990 para simplificar o processo de desenvolvimento de aplicativos para Microsoft Windows.
- A Microsoft tem uma estratégia de escopo corporativo para integrar a Internet e a Web em aplicativos de computador. Essa estratégia é implementada na plataforma .NET da Microsoft.
- As três principais linguagens de programação da plataforma .NET são Visual Basic .NET (baseada na linguagem BASIC original), Visual C# .NET (baseada em C++) e C# (uma nova linguagem baseada em C++ e Java que foi desenvolvida especificamente para a plataforma .NET).
- Os desenvolvedores em .NET podem escrever componentes de software em sua linguagem preferida e, assim, formar aplicativos combinando esses componentes com componentes escritos em qualquer linguagem .NET.

- Em geral, os sistemas C++ consistem em três partes: um ambiente de desenvolvimento de programa, a linguagem e a C++ Standard Lib
- Os programas C++, em geral, passam por seis fases: edição, pré-processamento, compilação, linkagem, carregamento e execução.
- Nomes de arquivo de código-fonte C++ costumam ter ~~par, como extensões~~
- Um programa pré-processador executa automaticamente antes de a fase de conversão do compilador iniciar. O pré-processador C++ opera comandos chamados diretivas de pré-processador, que indicam que certas manipulações devem ser realizadas no programa antes da pilhação.
- Em geral, o código-objeto produzido pelo compilador C++ contém ‘lacunas’ por causa das referências a funções e dados definidos em outra parte. Um linker vincula o código-objeto com o código das funções ausentes a fim de produzir uma imagem executável (sem partes ausentes).
- O carregador pega a imagem executável a partir de disco e a transfere para a memória para execução.
- A maioria dos programas em C++ realiza entrada e/ou saída de dados. Os dados são freqüentemente lidos a partir de um padrão que, em geral, é o terminal redirecionado a partir de outro dispositivo. A saída dos dados é, em geral, enviada para o terminal ou para o dispositivo de saída.
- A Unified Modeling Language (UML) é uma linguagem gráfica que permite que as pessoas construam sistemas para representar seus processos orientados a objetos.
- O projeto orientado a objetos (object-oriented design OOD) modela componentes de software em termos de objetos do mundo real. Ela tira proveito dos relacionamentos de classe, em que os objetos de certa classe têm as mesmas características. Ele também tira proveito de relacionamentos de herança, em que as novas classes de objetos criadas são derivadas absorvendo-se características de classes existentes, adicionando-se características únicas dessas mesmas classes. O OOD encapsula dados (atributos) e funções (comportamento) em objetos — as funções de um objeto são intimamente conectadas.
- Os objetos têm a propriedade de ocultamento de informações — normalmente os objetos não têm permissão de saber como outros objetos implementados.
- A programação orientada a objetos (object-oriented programming OOP) permite aos programadores implementar projetos orientados a objetos como sistemas funcionais.
- Os programadores em C++ criam seus próprios tipos definidos pelo usuário chamados classes. Toda classe contém dados (conhecidos como membros de dados) e o conjunto de funções (conhecido como funções-membro) que manipula esses dados e fornece serviços para clientes.
- As classes podem ter relacionamentos com outras classes. Esses relacionamentos são chamados associações.
- Empacotar software como classes possibilita que os sistemas de software futuros reutilizem as classes. Grupos de classes relacionadas freqüentemente empacotados como componentes reutilizáveis.
- Uma instância de uma classe é chamada objeto.
- Com a tecnologia de objeto, os programadores podem construir grande parte do software que será necessária combinando partes intercar e padronizadas chamadas classes.
- O processo de analisar e projetar um sistema a partir de um ponto de vista orientado a objetos é chamado análise e projeto orientados a objetos (object-oriented analysis and design OOAD).

Terminologia

ação	C padrão ANSI/ISSO	computação cliente/servidor
.NET Framework Class Library da Microsoft	C++	computação distribuída
abordagem de código ativo	C++ padrão ANSI/ISSO	computação pessoal
Ada	Standard Library	computador
American National Standards Institute (ANSI)	carregador	conteúdo dinâmico
análise	desse	dados
análise e design de sistemas estruturais	design	decisão
análise e projeto orientados a objetos (object-oriented analysis and design – OOA&D)	designer	dependente de máquina
assembly language	designer	designer
associação	código-fonte	diretivas de pré-processador
atributo de um objeto	código-objeto	dispositivo de entrada
BASIC (Beginner's All-Purpose Symbolic Instruction Code)	compartilhamento de tempo	dispositivo de saída
Booch, Grady	compilador	documento de requisitos
C	componente	editor
	comportamento de um objeto	encapsular

entrada/saída (E/S)	linker	projeto orientado a objetos (object-oriented design – OOD)
erros de tempo de execução	membro de dados	pseudocódigo
estação de trabalho extensível	memória	Rational Software Corporation
fase de carga	memória básica	redes locais (local area networks – LANs)
fase de compilação	memória principal	reutilização de software
fase de edição	método	
fase de execução	MFC (Microsoft Foundation Class)	Sombaugh, James
fase de link	multiprocessador	servidor de arquivos
fase de pré-processamento	multiprogramação	sistema operacional
FORTRAN (FORMula TRANslator)	multitarefa	software
função	multithreading	supercomputador
funcão-membro	Object Management Group (OMG)	Tarefa
hardware	objeto	throughput
herança	ocultamento de informações	tipo definido pelo usuário
imagem executável	operação	tradução
independente de máquina	plataforma	unidade aritmética e lógica (ALU)
instanciar	plataforma .NET	unidade central de processamento (central processing unit – CPU)
interface	plataforma de hardware	unidade de armazenamento secundária
International Organization for Standardization	portátil	unidade de entrada
(ISO)	processamento em lote	unidade de memória
Internet	programa de computador	unidade de saída
interpretador	programa tradutor	unidade lógica
Jacobson, Ivar	programação estruturada	Object Modeling Language (UML)
Java	programação orientada a objetos (Object oriented programming – OOP)	Visual Basic .NET
linguagem assembly	programação procedural	Visual C++ .NET
linguagem de alto nível	programador de computador	World Wide Web
linguagem de máquina		

Exercícios de revisão

- 1.1 Preencha as lacunas em cada uma das seguintes sentenças:
- A empresa que popularizou a computação pessoal foi _____.
 - O computador que tornou a computação pessoal legítima nos negócios e na indústria foi _____.
 - Os computadores processam dados sob o controle de conjuntos de instruções chamados _____.
 - As seis principais unidades lógicas do computador são _____, _____, _____, _____, _____ e _____.
 - As três classes de linguagens discutidas no capítulo são _____, _____ e _____.
 - Os programas que traduzem programas de linguagem de alto nível em linguagem de máquina são chamados _____.
 - C é amplamente conhecido como a linguagem de desenvolvimento do sistema operacional _____.
 - A linguagem _____ foi desenvolvida por Wirth para ensinar programação estruturada.
 - O Departamento da Defesa dos Estados Unidos desenvolveu a linguagem Ada com uma capacidade chamada _____, que permite aos programadores especificar que muitas atividades podem ocorrer em paralelo.
- 1.2 Preencha as lacunas em cada uma das seguintes frases sobre o ambiente C++.
- Os programas C++ normalmente são digitados em um computador _____ utilizando um programa _____.
 - Em um sistema C++, um programa _____ executa antes de a fase de conversão do compilador iniciar.
 - O programa _____ combina a saída do compilador com várias funções de biblioteca para produzir uma imagem executável.
 - O programa _____ transfere a imagem executável de um programa C++ do disco para a memória.
- 1.3 Preencha as lacunas de cada uma das sentenças a seguir (com base na Seção 1.17):
- Os objetos têm a propriedade de _____ — embora os objetos possam saber comunicar-se entre si por meio de interfaces bem definidas, normalmente não têm permissão de saber como outros objetos são implementados.
 - Os programadores em C++ se concentram na criação de _____, que contêm membros de dados e as funções-membro que manipulam esses membros de dados e fornecem serviços para clientes.
 - As classes podem ter relacionamentos com outras classes. Esses relacionamentos são chamados _____.
 - O processo de analisar e projetar um sistema de um ponto de vista orientado a objetos é chamado _____.

- e) O OOD também tira proveito de ~~retrabalhamentos~~ que novas classes de objetos são derivadas absorvendo-se características de classes existentes e, em seguida, adicionando-se características únicas dessas mesmas classes.
- f) _____ é uma linguagem gráfica que permite que as pessoas que projetam sistemas de software utilizem uma notação-padrão da indústria para representá-las.
- g) O tamanho, forma, cor e peso de um objeto ~~são~~ considerados do objeto.

Respostas dos exercícios de revisão

- 1.1 a) Apple. b) IBM Personal Computer. c) programas. d) unidade de entrada, unidade de saída, unidade de memória, unidade de aritmética e lógica, unidade central de processamento, unidade de armazenamento secundária. e) linguagens de máquina, linguagens assembly, linguagens de alto nível. f) compiladores. g) UNIX h) Pascal. i) multitarefa.
- 1.2 a) editor. b) pré-processador. c) linker. d) carregador.
- 1.3 ~~desenvolvimento de informações. Usando Modelagem UML para atribuir orientação a objetos (~~
- Exercícios**
- 1.4 Categorize cada um dos itens seguintes como hardware ou software:
- CPU
 - Compilador C++
 - ALU
 - Pré-processador C++
 - unidade de entrada
 - um programa de editor
- 1.5 Por que você poderia querer escrever um programa em uma linguagem independente de máquina em vez de uma linguagem dependente de máquina? Por que uma linguagem dependente de máquina talvez fosse mais apropriada para escrever certos tipos de programas?
- 1.6 Preencha as lacunas em cada uma das seguintes afirmações:
- Qual unidade lógica do computador recebe informações de fora do computador ~~para~~ utilização pelo computador?
 - O processo de instrução do computador para resolver problemas ~~específicos~~ é chamado
 - Que tipo de linguagem de computador utiliza abreviações semelhantes ao inglês ~~para~~ instruções de linguagem de máquina?
 - ~~Quais unidades lógicas do computador realizam operações que já foram processadas pelo computador para vários dispositivos de memória?~~
 - Qual unidade lógica do computador retém informações?
 - Qual unidade lógica do computador realiza cálculos?
 - Qual unidade lógica do computador toma decisões lógicas?
 - O nível de linguagem de computador mais conveniente para o programador ~~escrever~~ programas rápida e facilmente é
 - A única linguagem que um computador entende ~~diretamente~~ é chamada desse computador.
 - Qual unidade lógica do computador coordena as atividades de ~~todas~~ as outras unidades lógicas?
- 1.7 Por que hoje se concentra tanta atenção na programação orientada a objetos em geral e no C++ em particular?
- 1.8 Por que você talvez prefira experimentar um erro fatal em vez de um erro não fatal? Por que você poderia preferir que seu programa sofresse um erro fatal em vez de um erro não fatal?
- 1.9 Dê uma resposta breve a cada uma das seguintes perguntas:
- Por que esse texto discute a programação estruturada além da programação orientada a objetos?
 - Quais são os passos típicos (mencionados no texto) de um processo de projeto orientado a objetos?
 - Que tipos de mensagens as pessoas enviam entre si?
 - Os objetos enviam entre si mensagens por meio de interfaces bem-definidas. Que interfaces um rádio de carro (objeto) apresenta para seu usuário (um objeto pessoa)?
- 1.10 Segundo este texto, o que é o que se aplica a quase todos os tipos de objetos, tanto animados, quanto inanimados, mas só a alguns deles? Discuta, nomeando, por exemplo, um despertador), abstração, modelagem, mensagens, encapsulamento, interface, ocultamento de informações, membros de dados e funções-membro.

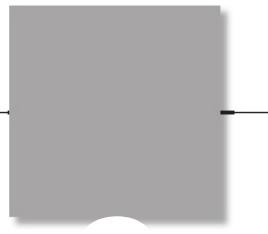


Que há num nome simples?
que chamamos rosa,
que com outro nome
seria tão perfumado.

William Shakespeare
Quando preciso tomar uma
decisão, sempre pergunto: “O
que seria mais divertido?”
Peggy Walker

“Tome mais chá”, a Lebre de
Março disse para Alice, muito
sinceramente. “Eu ainda não
bebi nada”, Alice respondeu
em um tom ofendido: “então
não posso tomar mais.” “Você
quis dizer que não podes tomar
menos”, disse Leirão: “é muito
mais fácil tomar mais do que
não tomar nada.”
Lewis Carroll

Pensamentos elevados devem
ter uma linguagem elevada.
Aristófanes



Introdução à programação em C++

OBJETIVOS

Neste capítulo, você aprenderá:

- A escrever programas de computador simples em C++.
- A escrever instruções de entrada e saída simples.
- A utilizar tipos fundamentais.
- Os conceitos básicos de memória de computador.
- A utilizar operadores aritméticos.
- A precedência dos operadores aritméticos.
- A escrever instruções de tomada de decisão simples.

- O
r
a
m
u
s
- 2.1 Introdução
 - 2.2 Primeiro programa C++: imprimindo uma linha de texto
 - 2.3 Modificando nosso primeiro programa C++
 - 2.4 Outro programa C++: adicionando inteiros
 - 2.5 Conceitos de memória
 - 2.6 Aritmética
 - 2.7 Tomada de decisão: operadores de igualdade e operadores relacionais
 - 2.8 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)
 - 2.9 Síntese

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

2.1 Introdução

Introduzimos agora a programação C++, que facilita uma abordagem disciplinada ao design de programa. A maioria dos programas que você estudará neste livro processa informações e exibe os resultados. Neste capítulo, apresentamos cinco exemplos que demonstram como seus programas exibem mensagens e obtêm informações do usuário para processamento. Os três primeiros exemplos simplesmente exibem mensagens na tela. O quarto é um programa que obtém dois números de um usuário, calcula sua soma e exibe o resultado. A discussão que o acompanha mostra como realizar vários cálculos aritméticos e salvar seus resultados para uso posterior. O quinto exemplo demonstra os fundamentos de tomada de decisão mostrando como comparar dois números e, então, exibir mensagens com base nos resultados da comparação. Analisamos cada programa uma linha por vez para ajudar a facilitar a introdução à programação C++. Para ajudar a aplicar as habilidades aprendidas aqui, fornecemos vários problemas de programação nos exercícios do capítulo.

2.2 Primeiro programa C++: imprimindo uma linha de texto

O C++ utiliza notações que podem parecer estranhas aos não-programadores. Agora consideraremos um programa simples que imprime uma linha de texto (Figura 2.1). Esse programa ilustra vários recursos importantes da linguagem C++. Consideraremos cada linha individualmente.

As linhas 1 e 2

// Figura 2.1: fig02_01.cpp

Programa de impressão de texto. Começam corindo que o restante de cada linha é comentários. Os programadores inserem comentários para documentar programas e também para ajudar as pessoas a ler e a entender programas. Os comentários não fazem com que o computador execute qualquer ação quando o programa está em execução — eles são ignorados pelo compilador C++ e não fazem com que qualquer objeto de linguagem de máquina seja gerado. Um comentário de texto descreve o propósito do programa. Um comentário que inicia com // é chamado de comentário de uma única linha porque termina no final da linha. Programadores em C++ também podem utilizar o estilo C em que um comentário — possivelmente contendo muitas linhas — inicia com /* e termina com */.

```

1 // Figura 2.1: fig02_01.cpp
2 // Programa de impressão de texto.
3 #include <iostream> // permite que o programa gere saída de dados na tela
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     std::cout << "Welcome to C++!\n"; // exibe a mensagem
9
10    return 0; // indica que o programa terminou com sucesso
11
12 } // fim da função main

```

Welcome to C++!

Figura 2.1 Programa de impressão de texto.



Boa prática de programação 2.1

Cada programa deve iniciar com um comentário que descreve o propósito do programa, autor, data e hora. (Não mostramos o auto data e hora nos programas deste livro porque essas informações seriam redundantes.)

A linha 3

`#include <iostream> // permite que o programa gere saída de dados na tela`

é uma diretiva de pré-processador que é uma mensagem para o pré-processador C++ (introduzido na Seção 1.14). As linhas que iniciam com # são processadas pelo pré-processador antes de o programa ser compilado. Essa linha instrui o pré-processador a incluir o programa o conteúdo do cabeçalho de fluxo de entrada/saída. Esse arquivo deve ser incluído em qualquer programa que realize a saída de dados na tela ou a entrada de dados a partir do teclado utilizando entrada/saída de fluxo no estilo C. O programa na Figura 2.1 gera saída de dados na tela, como veremos em breve. Discutimos os arquivos de cabeçalho em mais detalhes no Capítulo 6 e explicamos o conteúdo de Capítulo 15.



Erro comum de programação 2.1

Esquecer de incluir o arquivo `cabecalho.h` em um programa que realiza entrada de dados a partir do teclado ou saída de dados na tela faz com que o compilador emita uma mensagem de erro, porque o compilador não pode reconhecer referências a componentes de fluxo (por exemplo,

A linha 4 é simplesmente uma linha em branco. Os programadores utilizam linhas em branco, caracteres de espaço em branco de tabulação (isto é, ‘tabs’) para facilitar a leitura de programas. Juntos, esses ~~caracteres são~~ conhecidos como **espacos em branco**. Os caracteres de espaço em branco são normalmente ignorados pelo compilador. Neste e nos vários capítulos seguintes, discutiremos convocações de utilização de caracteres de espaço em branco para aprimorar a legibilidade do programa.



Boa prática de programação 2.2

Utilize linhas em branco e caracteres de espaço em branco para aprimorar a legibilidade do programa.

A linha 5

// a função main inicia a execução do programa

“A função main inicia a execução do programa é outro comentário de uma única linha que indica que a execução de programa inicia na próxima linha.

A linha 6

é uma **parte** de cada programa C++. Os parênteses são parte de um bloco de construção de programa chamado **função**. Os programas C++, em geral, consistem em uma ou mais funções e classes (como você aprenderá no Capítulo 3). Todo programa deve ter exatamente **uma**. A Figura 2.1 contém somente uma função. Os programas C++ começam executando na função `main`, mesmo se não for a primeira função no programa. A palavra-chave `main` indica que ‘retorna’ um valor do tipo inteiro (número inteiro). **Um** é uma palavra em código que é reservada pelo C++ para uma utilização específica. A lista completa de palavras-chave C++ pode ser encontrada na Figura 4.3. Explicaremos o que significa uma função ‘`return`’ quando demonstrarmos como criar suas próprias funções na Seção 3.5 e quando estudarmos as funções em maior profundo no Capítulo 6. Por enquanto, simplesmente inclua a estrutura `main` em cada um de seus programas.

A chave esquerda (linha 7) deve corresponder a uma direita (linha 12) deve terminar o corpo de cada função. A linha 8

```
std::cout << "Welcome to C++!\n" // exibe a mensagem
```

instrui o computador a realizar uma ação — a saber, imprimir cadeias de caracteres contida entre as aspas duplas. Uma string às vezes é chamada de caractere-mensagem ou string literal. Nós nos referimos a caracteres entre aspas duplas genericamente como strings. Os caracteres de espaço em branco em strings não são ignorados pelo compilador.

A linha 8 inteira, incluindo o operador `<`, a string `come to C++\n'e ponto-e-virgula()`, é chamada de instrução. Toda instrução em C++ deve terminar com um ponto-e-vírgula (também conhecido como diretivas de pré-processador). As linhas 9 e 10 terminam com um ponto-e-vírgula. A saída e a entrada em C++ são realizadas com

fluxo de caracteres. Portanto, quando a instrução precedente é executada, `Welcome to C#` é exibida na saída padrão (`Console.out-`). Normalmente, o console é conectado à tela. Discutiremos os recursos detalhadamente no Capítulo 15, "Entrada/saída de fluxo".

Note que colocamos `nos` antes de `de`. Isso é necessário quando utilizamos nomes trazidos no programa pela diretiva de pré-processador `#include <iostream>`. A notação `:cout` especifica que estamos utilizando um nome `ques pertence` ao 'namespace'. Os nomes (`o fluxo de entrada padrão`) — introduzidos no Capítulo 1 — também pertencem ao 'namespace'. Os namespaces são um recurso avançado do C++ que discutimos em profundidade no Capítulo 24, "Outros tópicos". Por enquanto, você deve simplesmente lembrar de cada nome que deve ocorrer em um programa. Isso pode ser incômodo — na Figura 2.13, introduzimos pedidamente os nomes de cada uso de um nome no namespace.

O operador é referido como o **operador de inserção de fluxo**. Quando esse programa executa, o valor à direita do operador, o **operando direito**, é inserido no fluxo de saída. Note que o operador aponta na direção de onde entram os dados. Em geral, os caracteres do operando direito são impressos exatamente como aparecem entre as aspas duplas. Na tela, impressos, os caracteres na tela. A barra invertida (****) é o **adactere de escape**. Ela indica que um caractere "especial" deve ser enviado para a saída. Quando uma barra invertida é encontrada em uma string de caracteres, o próximo caractere é combinado com a barra invertida para formar a **seqüência de escape**. A seqüência de escape indica a **caráter de nova linha**. Isso faz com que o (isto é, o indicador da posição atual na tela) se mova para o começo da próxima linha na tela. Algumas outras seqüências de escape comuns são listadas na Figura 2.2.



Erro comum de programação 2.2

Omitir o ponto-e-vírgula no fim de uma instrução C++ é um erro de sintaxe. (Novamente, as diretivas de pré-processador não terminam em um ponto-e-vírgula.) Uma linguagem de programação específica as regras da criação de um programa adequado nessa linguagem. Um erro ocorre quando o compilador encontra o código que viola as regras da linguagem C++ (isto é, sua sintaxe). O compilador normalmente emite uma mensagem de erro para ajudar o programador a localizar e corrigir o código incorreto. Erros de sintaxe também são chamados de **erros de compilação** porque o compilador os detecta durante a fase de compilação. Não será possível executar seu programa até você corrigir todos os erros de sintaxe nele. Como você verá, alguns erros de compilação não são erros de sintaxe.

A linha 10

`return 0;` // indica que o programa terminou com sucesso
é uma das várias maneiras que utilizaremos para indicar que o programa terminou com sucesso. Quando a instrução é utilizada no final de um bloco, como mostrado aqui, indica que o programa terminou com sucesso. No Capítulo 6, discutimos as funções em detalhes, e as razões de incluir essa instrução se tornarão claras. Por enquanto, simplesmente inclua essa instrução em cada programa; caso contrário, o compilador pode produzir um aviso em alguns sistemas.



Boa prática de programação 2.3

Muitos programadores tornam o último caractere impresso por uma função um **caractere de quebra de linha** (que é a função de saída de uma função). Isso faz com que o cursor de tela posicionado no começo de uma nova linha. Convenções dessa natureza encorajam a reusabilidade de software — um objetivo-chave no desenvolvimento de software.



Boa prática de programação 2.4

Recue o corpo inteiro de cada função um nível dentro das chaves que delimitam o corpo da função. Isso faz a estrutura funcional de um programa destacar-se e ajuda a tornar o programa mais fácil de ler.



Boa prática de programação 2.5

Defina uma convenção para o tamanho de recuo preferido, então aplique-a uniformemente. A tecla Tab pode ser utilizada para criar recuos, mas as paradas de tabulação podem variar. Recomendamos utilizar paradas de tabulação de $\frac{1}{4}$ de polegada (0,63 cm) (preferivelmente) três espaços para formar um nível de recuo.

Seqüênciadeescape	Descrição
\n	Nova linha. Posiciona o cursor de tela para o início da próxima linha.
\t	Tabulação horizontal. Move o cursor de tela para a próxima parada de tabulação.
\r	Retorno de carro. Posiciona o cursor da tela no início da linha atual; não avança para a próxima linha.
\a	Alerta. Aciona o aviso sonoro do sistema.
\	Barras invertidas. Utilizadas para imprimir um caractere de barra invertida.
\'	Aspas simples. Utilizadas para imprimir um único caractere de aspas simples.
\"	Aspas duplas. Utilizadas para imprimir um caractere de aspas duplas.

Figura 2.2 Seqüências de escape.

2.3 Modificando nosso primeiro programa C++

Esta seção continua nossa introdução à programação C++ com dois exemplos, mostrando como modificar o programa na Figura 2 para imprimir texto em uma linha utilizando múltiplas instruções e imprimir texto em várias linhas utilizando uma única instrução.

Imprimindo uma única linha de texto com múltiplas instruções

Welcome to C++ pode ser impresso de várias maneiras. Por exemplo, a Figura 2.3 realiza a inserção de fluxo em múltiplas instruções (linhas 8–9), mas ainda produz a mesma saída que o [Programa da Figura 2.1](#). [Nota do tradutor: Figura 2.1 é a figura anterior, utilizamos um fundo cinza na tabela de código para destacar os recursos-chave que cada programa introduz.] Cada inserção de fluxo retoma a impressão a partir do ponto onde a anterior parou. A primeira inserção de fluxo (linha 8) imprime tudo o que havia no espaço; e a segunda inserção de fluxo (linha 9) começa a imprimir a mesma linha logo depois do espaço. Em geral, o C++ permite ao programador expressar instruções de várias maneiras.

Imprimindo múltiplas linhas de texto com uma única instrução

Uma única instrução pode imprimir múltiplas linhas utilizando caracteres de nova linha, como na linha 8 da Figura 2.4. Toda vez que uma sequência de estrelinha (\n) for encontrada no fluxo de saída, o cursor de tela é posicionado no começo da linha seguinte. Para obter uma linha em branco em sua saída, coloque dois caracteres de nova linha um após o outro, como na linha 8.

```

1 // Figura 2.3: fig02_03.cpp
2 // Imprimindo uma linha de texto com múltiplas instruções.
3 #include <iostream> // permite que o programa gere saída de dados na tela
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     std::cout << "Welcome "
9     std::cout << "to C++!\n";
10
11    return 0; // indica que o programa terminou com sucesso
12
13 } // fim da função main

```

Welcome to C++!

Figura 2.3 Imprimindo uma linha de texto com múltiplas instruções.

```

1 // Figura 2.4: fig02_04.cpp
2 // Imprimindo múltiplas linhas de texto com uma única instrução.
3 #include <iostream> // permite que o programa gere saída de dados na tela
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     std::cout << "Welcome\n to\n C++!\n";
9
10    return 0; // indica que o programa terminou com sucesso
11
12 } // fim da função main

```

Welcome
to

C++!

Figura 2.4 Imprimindo múltiplas linhas de texto com uma única instrução.

2.4 Outro programa C++: adicionando inteiros

Nosso próximo programa utiliza o objeto de **fluxo de entrada** de extração de fluxo, para obter dois inteiros digitados por um usuário no teclado, calcula a soma desses valores e gera a saída. A figura 2.5 mostra o programa e os exemplos de entradas e saídas. Observe que destacamos a entrada do usuário em negrito.

Os comentários nas linhas 1 e 2

```
// Figura 2.5: fig02_05.cpp
```

// Programa de adição que exibe a soma de dois números.

declaram o nome do arquivo e o propósito do programa. A diretiva de pré-processador C++

```
#include <iostream> // permite ao programa realizar entrada e saída
```

na linha 3 inclui o conteúdo do arquivo de cabeçalho do programa.

O programa inicia a execução com (linha 5). A chave esquerda (linha 7) marca o começo do bloco do direita correspondente (linha 25) marca o fim de

As linhas 9–11

```
int number1;// primeiro inteiro a adicionar
int number2;// segundo inteiro a adicionar
int sum; // soma de number1 e number2
```

São declarações Os identificadores `number1`, `number2` e `sum` são os nomes de variáveis. Uma variável é uma posição na memória do computador onde um valor pode ser armazenado para utilização por um programa. Essas declarações especificam que as variáveis `number1`, `number2` e `sum` são dados de tipo int que significa que essas variáveis armazenarão valores inteiros como 7, -11, 0 e 31.914. Todas as variáveis devem ser declaradas com um nome e um tipo de dados antes que possam ser utilizadas no programa. Diversas variáveis do mesmo tipo podem ser declaradas em uma declaração ou em múltiplas declarações. Poderia ser declarado todas as três variáveis em uma declaração assim:

```
int number1, number2, sum;
```

```
1 // Figura 2.5: fig02_05.cpp
2 // Programa de adição que exibe a soma de dois números.
3 #include <iostream> // permite ao programa realizar entrada e saída

4 // a função main inicia a execução do programa
5 int main()
6 {
7     // declarações de variável
8     int number1;// primeiro inteiro a adicionar
9     int number2;// segundo inteiro a adicionar
10    int sum; // soma de number1 e number2

11
12
13    std::cout << "Enter first integer: " ; // solicita dados ao usuário
14    std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
15
16    std::cout << "Enter second integer: " ; // solicita dados ao usuário
17    std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
18
19    sum = number1 + number2; adiciona os números; armazena o resultado em sum
20
21    std::cout << "Sum is " << sum << std::endl; // exibe sum; termina a linha
22
23    return 0; // indica que o programa terminou com sucesso
24
25 } // fim da função main
```

Enter first integer: 45

Enter second integer: 72

Sum is 117

Figura 2.5 O programa de adição que exibe a soma de dois inteiros inseridos a partir do teclado.

Isso torna o programa menos legível e impede que forneçamos comentários que descrevem o propósito de cada variável. Se mais nome é declarado em uma declaração (como mostrado aqui), os nomes ~~são separados por vírgulas~~ (~~separada por vírgulas~~)



Boa prática de programação 2.6

Coloque um espaço depois de ~~cada vírgula~~ os programas mais legíveis.



Boa prática de programação 2.7

Alguns programadores preferem declarar cada variável em uma linha separada. Esse formato permite inserção fácil de um comentário descritivo ao lado de cada declaração.

Em breve, discutiremos o tipo de dados para especificar números reais e octópolos e para especificar dados de caractere. Os números reais são números com pontos de fração decimal, como 19.1e0.0 ou 1111.0000. As variáveis só podem ser nomeadas se forem minúsculas, uma única maiúscula, um único dígito ou uma palavra especial (por exemplo, `const`, `double` e `char`) — são freqüentemente chamados de tipos primitivos ou tipos predefinidas. Os nomes dos tipos fundamentais são palavras-chave e, portanto, devem aparecer em letras minúsculas. O Apêndice C contém a lista completa dos tipos fundamentais.

Um nome variável (ou identificador) é qualquer string válida que não seja uma palavra-chave. Um identificador é uma série de caracteres consistindo em letras, dígitos e sublinhados, com um dígito ~~distintos~~ fazendo a ~~distinção entre~~ entre letras maiúsculas e minúsculas — letras minúsculas e maiúsculas são ~~diferentes~~ identificadores diferentes.



Dica de portabilidade 2.1

O C++ permite identificadores de qualquer comprimento, mas sua implementação do C++ pode impor algumas restrições sobre o comprimento de identificadores. Utilize identificadores de 31 caracteres ou menos para assegurar portabilidade.



Boa prática de programação 2.8

Escolher identificadores significativos ajuda a ~~fazer um~~ programar uma pessoa pode entender o programa simplesmente lendo-o em vez de ter de referir-se a manuais ou comentários.



Boa prática de programação 2.9

Evite usar abreviações em identificadores. Isso aumenta a legibilidade do programa.



Boa prática de programação 2.10

Evite identificadores que iniciem com sublinhados e sublinhados duplos, porque os compiladores C++ podem utilizar nomes assim para seus próprios propósitos internamente. Isso impedirá que os nomes que você escolhe sejam confundidos com nomes escolhidos por compiladores.



Dica de prevenção de erro 2.1

Linguagens como C++ são ‘alvos inconstantes’. À medida que elas evoluem, mais palavras-chave poderiam ser adicionadas à linguagem. Evite utilizar palavras ‘sobrecarregadas de significado’ como ‘object’ para identificadores. Ainda que ‘object’ não seja atualmente uma palavra-chave em C++, ela poderia vir a tornar-se uma; portanto, a compilação futura com novos compiladores poderia quebrar o código existente.

As declarações de variáveis podem ser colocadas quase em qualquer lugar em um programa, mas devem aparecer antes de variáveis correspondentes serem utilizadas no programa. Por exemplo, no programa da Figura 2.5, a declaração na linha 9 `int number1; // primeiro inteiro a adicionar` poderia ter sido colocada imediatamente antes da linha 14

```
std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
a declaração na linha 10
int number2; // segundo inteiro a adicionar
poderia ter sido colocada imediatamente antes da linha 17
```

```
std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
```

e a declaração na linha 11

```
int sum; // soma de number1 e number2
poderia ter sido colocada imediatamente antes da linha 19
sum = number1 + number2; adiciona os números; armazena o resultado em sum
```



Boa prática de programação 2.11

Sempre coloque uma linha em branco entre uma declaração e instruções executáveis adjacentes. Isso faz com que as declarações destaqueem e contribui para a clareza do programa.



Boa prática de programação 2.12

~~Se preferir adicionar uma declaração e uma instrução adjacentes, pode separá-las por uma linha em branco.~~ Se preferir adicionar uma declaração e uma instrução adjacentes, pode separá-las por uma linha em branco.

A linha 13

```
std::cout << "Enter first integer: " ; // solicita dados ao usuário
imprime a string first integer: (também conhecida como um literal string) na tela. Essa mensagem é chamada
prompt porque direciona o usuário para uma ação específica. Gostamos de pronunciar esse construtor precedente como 'a string de caractere'. A linha 14
std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
utiliza objeto fluxo de entrada (do namespace) operador de extração de fluxo, para obter um valor do teclado.
Utilizar o operador de extração de fluxo aceita a entrada de caractere a partir do fluxo de entrada padrão, que geralmente
é o teclado. Gostamos de pronunciar a instrução anterior como 'ler um valor número simplesmente
fornece' number1'
```



Dica de prevenção de erro 2.2

Os programas devem validar a correção de todos os valores de entrada para impedir que informações errôneas afetem os cálculos de um programa.

Quando o computador executa a instrução anterior, ele espera que o usuário insira um número intável (não intérpreteável) para o computador. O computador converte a representação de caractere do número em um inteiro e atribui (copia) esse (ou valor) para a variável number1. Qualquer referência subsequente a esse programa utilizará esse mesmo valor.

Os objetos de fluxo std::cout facilitam a interação entre o usuário e o computador. Como essa interação assemelha-se a um diálogo, ela é freqüentemente [chamada de conversação ou computação interativa](#).

A linha 16

```
std::cout << "Enter second integer: " ; // solicita dados ao usuário
imprime Enter second integer: na tela, pedindo para o usuário executar uma ação. A linha 17
std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
obtém um valor para a variável number2 a partir do usuário.
```

A instrução de atribuição na linha 19

```
sum = number1 + number2; adiciona os números; armazena o resultado em sum
calcula a soma das variáveis number1 e number2 e atribui o resultado à variável sum usando o operador de atribuição. A instrução
é lida como 'obter o valor de number1 + number2'. A maioria dos cálculos é realizada em instruções de atribuição. O operador
e o operando são chamados de operadores binários porque cada um tem dois operandos. No caso desse operador
só number1 e number2. No caso do operador de atribuição, os dois operandos são o valor da expressão number1 + number2 e number2.
```



Boa prática de programação 2.13

Coloque espaços de ambos os lados de um operador binário. Isso destaca o operador e torna o programa mais legível.

A linha 21

```
std::cout << "Sum is " << sum << std::endl; // exibe sum; termina a linha
exibe a string de caractere seguida pelo valor numérico da variável sum. std::endl — chamada manipulador
de fluxo. O nome endl é uma abreviação de 'end line' e pertence ao manipulador de fluxo. Ele gera saída
de um caractere de nova linha e, depois, 'esvazia o buffer de saída'. Isso simplesmente significa que, em alguns sistemas em
```

saídas acumulam na máquina até haver conteúdo suficiente para ‘saber’ a posição das saídas. As saídas acumuladas sejam exibidas nesse momento. Isso pode ser importante quando as saídas exigem que o usuário execute uma ação inserir dados.

Observe que a instrução precedente gera saída de múltiplos valores de diferentes tipos. O operador de inserção de fluxo ‘sabe’ saída de cada tipo de dados. Utilizar múltiplos operadores de inserção é desnecessário ter múltiplas instruções para gerar saída de múltiplas partes dos dados.

Os cálculos também podem ser feitos em instruções de saída. Poderíamos ter combinado as instruções nas linhas 19 e 21 nessa mesma linha.

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

eliminando, assim, a necessidade de uma variável

Um recurso poderoso do C++ é que os usuários podem criar seus próprios tipos de dados chamados classes (introduzimos capacidade no Capítulo 3 e a exploramos profundamente nos capítulos 9 e 10). Os usuários então podem ‘ensinar’ o C++ a reutilizar e saída de valores desses novos tipos de dados utilizando suas próprias regras de operadores – um tópico que exploramos no Capítulo 11).

2.5 Conceitos de memória

Os nomes de variáveis `number1` e `sum` correspondem na realidade à memória do computador. Cada variável tem um nome, um tipo, um tamanho e um valor.

No programa de adição da Figura 2.5, quando a instrução

```
std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
```

na linha 14 é executada, os caracteres digitados pelo usuário são convertidos em um inteiro que é colocado em uma posição da memória que o nome `number1` foi atribuído pelo compilador C++. Suponha que o usuário inseriu o valor 45. O computador coloca o valor 45 na posição `number1`, como mostrado na Figura 2.6.

Sempre que um valor é colocado em uma posição da memória, o valor sobrescreve o valor anterior nessa posição; portanto, é estruturado para colocar um novo valor em uma posição da memória.

Retornando ao nosso programa de adição, quando a instrução

```
std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
```

na linha 17 é executada, suponha que o usuário inseriu o valor 72. Esse valor é colocado na posição de memória aparece como

mostrado na Figura 2.7. Observe que essas posições não são necessariamente adjacentes na memória, mas a instrução

`sum = number1 + number2;` adiciona os números; armazena o resultado em `sum` que realiza a adição também substitui qualquer valor que já estava na memória. A soma calculada de `number1` é colocada na posição `sum` (sem considerar qual valor já pode estar lá – o valor é perdido). Depois de calculada, a memória aparece como na Figura 2.8. Observe que os valores permanecem exatamente como eram antes de serem utilizados no cálculo. Esses valores foram utilizados, mas não destruídos, enquanto o computador realizou o cálculo. Portanto, quando um valor é lido de uma posição da memória, o processo é do tipo



Figura 2.6 Posição da memória mostrando o nome e o valor de uma variável

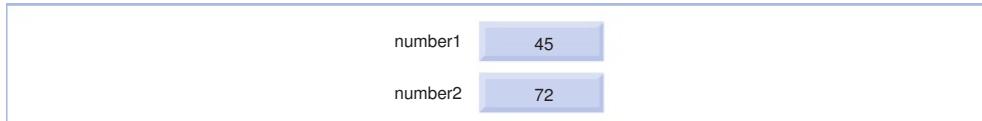


Figura 2.7 Posições da memória depois de armazenar os valores de number1 e number2

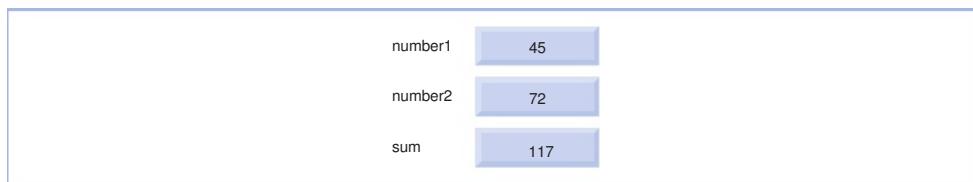


Figura 2.8 Posições da memória depois de calcular e armazenar a soma de number1 e number2.

2.6 Aritmética

A maioria dos programas realiza cálculos aritméticos. A Figura 2.9 resume os operadores aritméticos do C++. Observe o uso de vários símbolos especiais não utilizados em álgebra. Além disso, indica multiplicação, exponenciação, porcentagem (%) e o operador módulo que será discutido em breve. Os operadores aritméticos na Figura 2.9 são todos operadores binários, isto é, operadores que aceitam dois operandos. Por exemplo, a expressão `number1 + number2` contém o operador binário `+` dos operandos `number1` e `number2`.

A **divisão de inteiro** (isto é, aquela em que tanto o numerador como o denominador são inteiros) produz um quociente do tipo inteiro; por exemplo, a expressão `a / b` é avaliada como `a` e a expressão `7 / 5` produz `1`. Observe que qualquer parte fracionária na divisão de inteiro é descartada (isto é, é `0`) — nenhum arredondamento ocorre.

O C++ fornece o operador `módulo (%)` que fornece o resto da divisão de inteiros. O operador módulo pode ser utilizado somente com operandos inteiros. A expressão `a % b` produz o resto depois de dividir `a` por `b`. Portanto, `7 % 4` produz `3` e `17 % 5` produz `2`. Nos capítulos posteriores, discutimos muitas aplicações interessantes do operador módulo, tais como determinar se um número é um múltiplo de outro (um caso especial disso é determinar se um número é ímpar ou par).



Erro comum de programação 2.3

Tentar utilizar o operador módulo (%) com operandos não inteiros é um erro de compilação.

Expressões aritméticas em linha reta

As expressões aritméticas em C++ devem ser inseridas no computador. Portanto, as expressões `a / b`, `a * b`, `a + b` e `a - b` devem ser escritas como

$$\frac{a}{b}$$

em geral não é aceitável para compiladores, embora existam alguns pacotes de software de uso especial que suportem notação mais natural para expressões matemáticas complexas.

Parênteses para agrupar subexpressões

Os parênteses são utilizados em expressões C++ da mesma maneira que em expressões algébricas. Por exemplo, para multiplicar a quantidade `a * b + c`, escrevemos `(b + c) * a`.

Regras de precedência de operadores

O C++ aplica os operadores em expressões aritméticas em uma seqüência precisa determinada pelas regras de precedência de operador que em geral são as mesmas que aquelas seguidas em álgebra:

OperaçãoC++	Operador aritméticoC++	Expressão algébrica	ExpressãoC++
Adição	+	$f + 7$	<code>f + 7</code>
Subtração	-	$p - c$	<code>p - c</code>
Multiplicação	*	$b m o u b \cdot m$	<code>b * m</code>
Divisão	/	x / y ou $\frac{x}{y}$ ou $x \div y$	<code>x / y</code>
Módulo	%	$r \text{ mod } s$	<code>r % s</code>

Figura 2.9 Operadores aritméticos.

1. Os operadores em expressões contidas dentro de pares de parênteses são avaliados primeiro. Portanto, os ser utilizados para forçar a ordem de avaliação a ocorrer em qualquer sequência. Diz-se que o programador os parênteses estão no nível 'mais alto de precedência'. Em casos de ~~parênteses aninhados~~, como
 $((a + b) + c)$
os operadores no par mais interno de parênteses são aplicados primeiro.
2. Operações de multiplicação, divisão e módulo são aplicadas em seguida. Se uma expressão contém várias operações de multiplicação, divisão e módulo, os operadores são aplicados da esquerda para a direita. Diz-se que a multiplicação, a divisão e módulo estão no mesmo nível de precedência.
3. Operações de adição e de subtração são aplicadas por último. Se uma expressão contém várias operações de adição e subtração, os operadores são aplicados da esquerda para a direita. Adição e subtração também têm o mesmo nível de precedência.

O conjunto de regras de precedência de operadores define a ordem em que o C++ aplica operadores. Quando dizemos que certos operadores são aplicados da esquerda para a direita, estamos nos referindo a operadores.

Os operadores de adição associam-se da esquerda para a direita, contando o primeiro operador associado a essa soma para determinar o valor da expressão inteira. Veremos que alguns operadores se associam da direita para a esquerda. A 2.10 resume essas regras de precedência de operador. Essa tabela será expandida à medida que os operadores adicionais do C introduzidos. Uma tabela completa de precedência está incluída no Apêndice A.

Exemplo de expressões algébricas e expressões em C++

Agora considere várias expressões à luz das regras de precedência de operadores. Cada exemplo lista uma expressão algébrica e uma expressão equivalente em C++. O seguinte é um exemplo de uma média aritmética de cinco termos:

$$\text{Álgebra: } \frac{a+b+c+d+e}{5}$$

$$\text{C++: } m = (a + b + c + d + e) / 5;$$

Os parênteses são exigidos porque a divisão tem precedência mais alta que a adição. A quantidade inteira deve ser dividida por cinco parênteses são omitidos erroneamente, obtendo-se, que é incorretamente avaliado como

$$\frac{a+b+c+d+e}{5}$$

O seguinte é um exemplo da equação de uma linha reta:

$$\text{Álgebra: } y = mx + b$$

$$\text{C++: } y = m * x + b;$$

Nenhum parêntese é requerido. A multiplicação é aplicada primeiro porque a multiplicação tem uma precedência mais alta que a adição.

O exemplo a seguir contém operadores de multiplicação, divisão, adição, subtração e atribuição:

$$\text{Álgebra: } z = p \% q + w / x - y$$

$$\text{C++: } z = p * r \% q + w / x - y;$$



Operador(es)	Operação(ões)	Ordem de avaliação (precedência)
()	Parênteses	Avaliados primeiro. Se os parênteses estão aninhados, a expressão no par mais interno é avaliada primeiro. Se há vários pares de parênteses 'no mesmo nível' (isto é, não aninhados), eles são avaliados da esquerda para a direita.
* / %	Multiplicação Divisão Módulo	Avaliado em segundo lugar. Se houver vários, eles são avaliados da esquerda para a direita.
+-	Adição Subtração	Avaliado por último. Se houver vários, eles são avaliados da esquerda para a direita.

Figura 2.10 Precedência de operadores aritméticos.

Os números dentro de círculos sob a instrução indicam a ordem em que o C++ aplica os operadores. A multiplicação, o módulo divisão são avaliados primeiro na ordem da esquerda para a direita (isto é, eles se associam da esquerda para a direita) porque precedência mais alta que a adição e a subtração. A adição e a subtração são aplicadas a seguir. Elas também são aplicadas da esquerda para a direita. Então, o operador de atribuição é aplicado.

Avaliação de um polinômio de segundo grau

Para desenvolver um melhor entendimento das regras de precedência de operadores, considere a avaliação de um polinômio de segundo grau ($y = ax^2 + bx + c$):

$y = a * x * x + b * x + c;$

6 1 2 4 3 5

Os números dentro de círculos sob a instrução indicam a ordem em que o Java aplica os operadores. Não há nenhum operador de multiplicação ou exponenciação em C++, então representamos brevemente discutiremos a função de biblioteca `Math.pow()` que realiza exponenciação. Por causa de algumas questões sutis relacionadas com a ordem de avaliação dos operadores de exponenciação detalhadas no Capítulo 6.



Erro comum de programação 2.4

Algumas linguagens de programação utilizam os operadores `**` para representar a exponenciação. O C++ não suporta esses operadores de exponenciação; utilizá-los para exponenciação resulta em erros.

Suponha que as variáveis no polinômio precedente de segundo grau sejam inicializadas da seguinte maneira:
 $a = 2$, $b = 3$, $x = 5$. A Figura 2.11 ilustra a ordem em que os operadores são aplicados.

Como em álgebra, é aceitável colocar parênteses desnecessários em uma expressão para tornar a expressão mais clara. Eles são chamados de **parênteses redundantes**. Por exemplo, a instrução de atribuição precedente poderia estar entre parênteses como a seguir:

$y = (a * x * x) + (b * x) + c;$



Boa prática de programação 2.14

Utilizar parênteses redundantes em expressões aritméticas complexas pode tornar as expressões mais claras.

2.7 Tomada de decisão: operadores de igualdade e operadores relacionais

Esta seção introduz uma versão simplificada do C++ que permite que um programa tome uma decisão com base na verdade ou falsidade de alguma condição. Se a condição for satisfeita, isto é, se a condição for verdadeira, a instrução no corpo da instrução

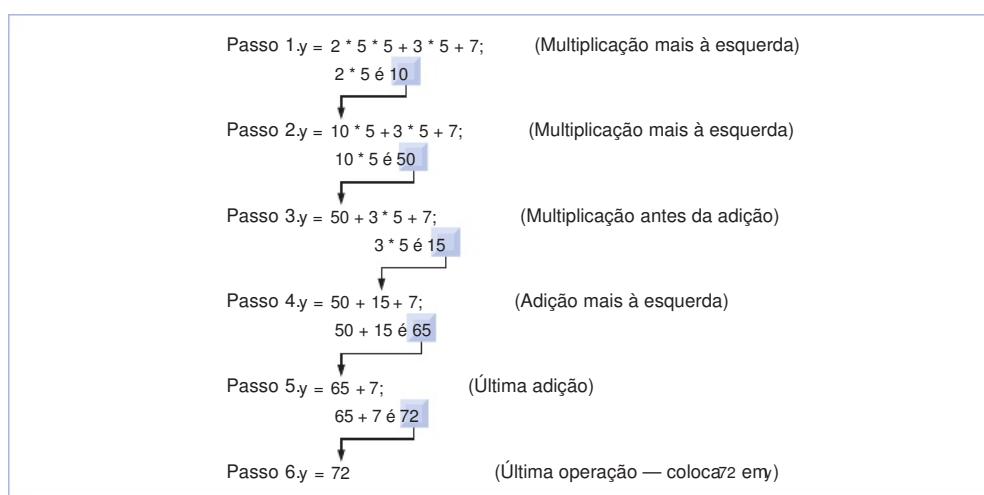


Figura 2.11 Ordem em que um polinômio de segundo grau é avaliado.

será executada. Se a condição não for satisfeita, isto é, se a condição for falsa, a instrução no corpo não será executada. Verem exemplo brevemente.

As condições em instruções ser formadas utilizando operadores de igualdade e operadores relacionais resumidos na Figura 2.12. Todos os operadores relacionais têm o mesmo nível de precedência e se associam da esquerda para a direita. Os operadores de igualdade têm o mesmo nível de precedência, que é mais baixo que aquele dos operadores relacionais, e eles se associam da esquerda para a direita.



Erro comum de programação 2.5

Ocorrerá um erro de sintaxe se algum dos operadores receber com espaços entre seu par de símbolos.



Erro comum de programação 2.6

— 1 —

é em geral considerado desejável que os algoritmos executados no lado de software possam ser executados de forma eficiente, um de lógica que tem um efeito em tempo de execução. Você entenderá a razão disso quando aprender sobre operadores lógicos no Capítulo 5. Um de lógica faz com que um programa falhe e termine prematuramente. Isto permite a um programa continuar executando, mas geralmente produz resultados incorretos.



Erro comum de programação 2.7

Confundir o operador de igualdade com o operador de atribuição em erros de lógica. O operador de igualdade deve ser lido como 'é igual a', e o operador de atribuição deve ser lido como 'obtém' ou 'obtém o valor de' ou 'recebe o valor de'. Algumas pessoas preferem ler o operador de igualdade como 'duplo igual'. Como discutimos na Secção 5.9, confundir esses operadores pode não causar necessariamente um erro de sintaxe fácil de reconhecer, mas pode causar erros de lógica extremamente sutis.

O seguinte exemplo utiliza seis instruções para dois números inseridos pelo usuário. Se a condição em qualquer dessas instruções for satisfeita, a instrução de saída associada é exibida. A Figura 2.13 mostra o programa e os diálogos de entrada/saída de três execuções de exemplo.

analogos de entradas

```
using std::cout; // o programa utiliza cout
```

```
using std::cin; // o programa utiliza cin
```

```
using std::endl; // o programa utiliza endl
```

~~que eliminam a necessidade de repetir `coprefine`~~ que eliminam a necessidade de repetir `coprefine`. Uma vez que inserimos essas declarações, podemos escrever `cout`, `cin` em vez de `de:cout`, `de:cin` e `endl` em vez de `de:endl`, respectivamente, no restante do código. De qualquer modo, em diante no livro, cada exemplo contém uma ou mais declarações usando.]

Operador algébrico de igualdade ou relacional padrão	Operador de igualdade ou relacional em C++	Exemplo de condição em C++	Significado da condição em C++
Operadores relacionais			
>	>	y > x	x é maior que
<	<	y < x	x é menor que
≥	≥=	y ≥= x	x é maior que ou igual a
≤	≤=	y ≤= x	x é menor que ou igual a
Operadores de igualdade			
=	==	y == x	x é igual a
≠	!=	y != x	x não é igual a

Figura 2.12 Operadores de igualdade e operadores relacionais.

```

1 // Figura 2.13: fig02_13.cpp
2 // Comparando inteiros utilizando instruções if, operadores relacionais
3 // e operadores de igualdade.
4 #include <iostream> // permite ao programa realizar entrada e saída
5
6 using std::cout; // o programa utiliza cout
7 using std::cin; // o programa utiliza cin
8 using std::endl; // o programa utiliza endl
9
10 // a função main inicia a execução do programa
11 int main()
12 {
13     int number1;// primeiro inteiro a comparar
14     int number2;// segundo inteiro a comparar
15
16     cout << "Enter two integers to compare: " ; // solicita dados ao usuário
17     cin >> number1 >> number2; // lê dois inteiros fornecidos pelo usuário
18
19     if ( number1 == number2 )
20         cout << number1 << " == " << number2 << endl;
21
22     if ( number1 != number2 )
23         cout << number1 << " != " << number2 << endl;
24
25     if ( number1 < number2 )
26         cout << number1 << " < " << number2 << endl;
27
28     if ( number1 > number2 )
29         cout << number1 << " > " << number2 << endl;
30
31     if ( number1 <= number2 )
32         cout << number1 << " <= " << number2 << endl;
33
34     if ( number1 >= number2 )
35         cout << number1 << " >= " << number2 << endl;
36
37     return 0; // indica que o programa terminou com sucesso
38
39 } // fim da função main

```

Enter two integers to compare: 3 7

3 != 7
3 < 7
3 <= 7

Enter two integers to compare: 22 12

22 != 12
22 > 12
22 >= 12

Enter two integers to compare: 7 7

7 == 7
7 <= 7
7 >= 7

Figura 2.13 Operadores de igualdade e operadores relacionais.



Boa prática de programação 2.15

Coloque declarações imediatamente depois que elas referenciam.

As linhas 13–14

```
int number1; // primeiro inteiro a comparar  
int number2; // segundo inteiro a comparar
```

declararam as variáveis utilizadas no programa. Lembre-se de que as variáveis podem ser declaradas em uma ou em múltiplas cotações.

O programa utiliza operações de extração de fluxo em cascata (linha 17) para inserir dois inteiros. Lembre-se de que podemos crescer (em vez de `<< cin`) por causa da linha 7. Primeiro um valor é lido para a variável `number1` e o valor é lido para a variável `number2`.

A instrução nas linhas 19–20

```
if (number1 == number2) " << number2 << endl;
```

compara os valores de variáveis `number1` e `number2` para testá-los quanto à igualdade. Se os valores forem iguais, a instrução na linha 20 exibirá uma linha de texto para indicar que os números são iguais. ~~Se as condições das instruções que iniciam nas linhas 22, 25, 28, 31 e 34, a instrução correspondente do corpo exibirá uma linha apropriada de texto.~~

Note que cada instrução ~~Figura 2.13 tem uma única instrução no seu corpo e que cada instrução do corpo é recuada. No Capítulo 4, mostramos como especificar corpos de múltiplas instruções (incluindo as instruções de corpo em um par de chaves), criando o que é chamado de bloco composto.~~



Boa prática de programação 2.16

Recue a(s) instrução(ões) no corpo de uma instrução para manter a legibilidade.



Boa prática de programação 2.17

Para legibilidade não deve haver mais de uma instrução por linha em um programa.



Erro comum de programação 2.8

Colocar um ponto-e-vírgula imediatamente depois do parêntese direito depois da condição em uma instrução erro de lógica (embora não seja um erro de sintaxe). O ponto-e-vírgula faz com que se passe para a instrução não realiza nenhuma ação, independentemente de sua condição ser ou não verdadeira. Pior ainda, a instrução scinal do corpo da instrução se tornaria uma instrução em seqüência ~~ao invés de precedência associativa~~, fazendo com que o programa produza resultados incorretos freqüentemente.

Observe o uso do espaço em branco na Figura 2.13. Recorde que os caracteres de espaço em branco como tabulações, novas linhas e espaços normalmente são ignorados pelo compilador. Então, as instruções podem ser divididas em diversas linhas e espaçadas de acordo com as preferências do programador. É um erro de sintaxe dividir identificadores longos (como o número `000`) em várias linhas.



Erro comum de programação 2.9

É um erro de sintaxe dividir um identificador inserindo caracteres de espaço em branco (por exemplo, escrevendo `ma in`).



Boa prática de programação 2.18

Uma instrução longa pode se estender por várias linhas. Se uma única instrução precisar ter sua linha quebrada, escolha pontos de quebra significativos, como depois de uma vírgula em uma lista separada por vírgulas ou depois de um operador em uma expressão extensa. Se uma instrução se estende por duas ou mais linhas, recue todas as linhas subsequentes e alinhe à esquerda o grupo.

A Figura 2.14 mostra a precedência e a associatividade dos operadores introduzidos neste capítulo. Os operadores são mostados de cima para baixo em ordem decrescente de precedência. Note que todos esses operadores, com a exceção do operador de atribuição `=`, se associam da esquerda para a direita. A adição associa da esquerda para a direita, é associativa, expressão como `x = y = 0` seja avaliada como se tivesse sido escrita `(x + y) = 0`. O operador de atribuição da direita para a esquerda, para que uma expressão como `x = y = 0` seja avaliada como se tivesse sido escrita que, como logo veremos, primeiramente atribui o resultado dessa atribuição.—

Operadores	Associatividade	Tipo
()	daesquerdaparaadireita	parênteses
*	daesquerdaparaadireita	multiplicativo
-	daesquerdaparaadireita	aditivo
<<	daesquerdaparaadireita	inserção/extracção de fluxo
< <= > >=	daesquerdaparaadireita	relacional
==	daesquerdaparaadireita	igualdade
=	dadireitaparaaesquerda	atribuição

Figura 2.14 Precedência e associatividade dos operadores discutidos até agora.



Boa prática de programação 2.19

Consulte o gráfico de precedência e associatividade de operadores ao escrever expressões contendo muitos operadores. Confirme os operadores na expressão são realizados na ordem em que você espera. Se não estiver certo quanto à ordem de avaliação em expressão complexa, divida a expressão em instruções menores ou utilize parênteses para forçar a ordem de avaliação, exatamente como faria em uma expressão algébrica. Certifique-se de observar que alguns operadores podem ter precedência de atribuição (para a esquerda em vez de da esquerda para a direita).

2.8 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)

Agora, iniciaremos nosso estudo de caso opcional orientado a objetos de projeto e implementação. As seções “Estudo de caso de engenharia de software” no final deste e dos vários próximos capítulos facilitarão seu entendimento da orientação a objeto. Desenvolveremos um software para um sistema de caixa eletrônico simples (macATM), fornecendo uma experiência de projeto e implementação completa, cuidadosamente elaborada passo a passo. Nos capítulos 3–7, 9 e 13, realizaremos os vários passos de um projeto orientado a objetos utilizando o UML e, ao mesmo tempo, relacionaremos esses passos com conceitos orientados a objetos discutidos nos capítulos O Aprendiz e Implementa o ATM utilizando as técnicas de programação orientada a objetos (object-oriented programming) em C++. Apresentamos a solução completa do estudo de caso. Isso não é um exercício; em vez disso, é uma experiência de aprendizagem completa que conclui com uma revisão detalhada do código C. Implementaremos nosso projeto. Ele possibilitará que você se familiarize com os tipos de problemas substanciais encontrados na indústria e suas potenciais soluções.

Iniciamos nosso processo de projeto apresentando um [documento de requisitos](#) que especifica o propósito geral do sistema ATM e o que este deve fazer. Por todo o estudo de caso, nós nos referimos ao documento de requisitos para determinar precisamente a funcionalidade que o sistema deve incluir.

Documento de requisitos

Um banco local pretende instalar um novo [caixa eletrônico](#) (macATM) para permitir que os usuários (isto é, os clientes do banco) realizem transações financeiras básicas (Figura 2.15). Cada usuário pode ter somente uma conta no banco. Os usuários do ATM devem ser capazes de visualizar seus saldos bancários, sacar dinheiro (isto é, retirar dinheiro de uma conta) e depositar dinheiro (isto é, colocar dinheiro em uma conta).

A interface com o usuário do caixa eletrônico contém os seguintes componentes de hardware:

- uma tela que exibe as mensagens para o usuário;
- um teclado que recebe a entrada numérica do usuário;
- um dispensador de cédulas que disponibiliza o dinheiro para o usuário e
- uma abertura para depósito que recebe os envelopes com o depósito do usuário.

O dispensador de cédulas é carregado diariamente com \$1000. [Notas de \$1000 não são levadas desse estudo de caso, certos elementos do ATM descritos aqui não simulam exatamente aqueles de um ATM real. Por exemplo, um ATM real geral contém um dispositivo que lê o número da conta de um usuário a partir de um cartão ATM, enquanto esse ATM solicita que o usuário digite o número de uma conta no teclado. Normalmente, um ATM real também imprime um recibo no fim de uma sessão. Toda a saída desse ATM aparece na tela.]

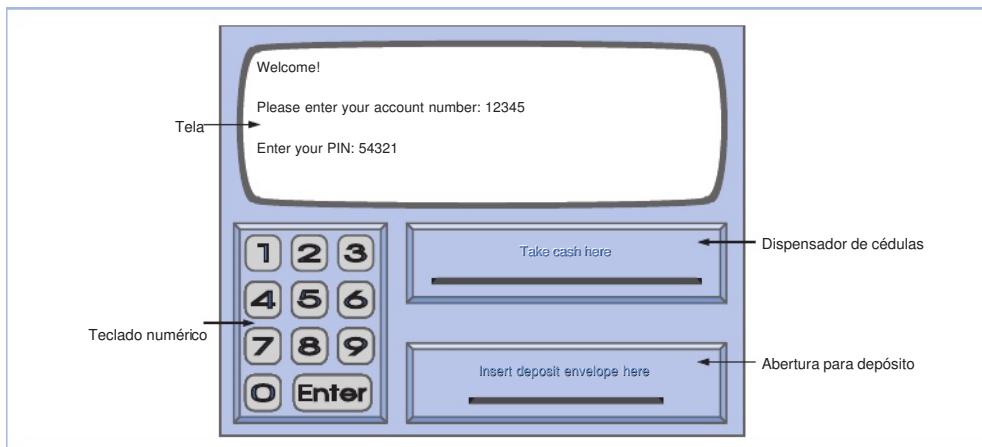


Figura 2.15 A interface com o usuário do caixa eletrônico.

O banco quer que você desenvolva um software para realizar as transações financeiras iniciadas pelos clientes do banco por meio de um ATM. O banco integrará o software com o hardware do ATM em um momento posterior. O software deve encapsular a funcionalidade dos dispositivos de hardware (por exemplo, o dispensador de cédulas, a abertura para depósito) dentro dos componentes de software, mas ele próprio não precisa se preocupar com a maneira como esses dispositivos realizam suas tarefas. O hardware do ATM não foi desenvolvido; assim, em vez de escrever seu software para ser executado no ATM, você deve desenvolver uma primeira versão do software para executar em um computador pessoal. Essa versão deve utilizar o monitor do computador para simular a tela do ATM e o teclado do computador para simular o teclado do ATM.

Uma sessão do ATM consiste na autenticação de um usuário (isto é, provar a identidade do usuário) com base em um número de conta e em um pin.

Portanto, o usuário usa o software de identificação para iniciar a sessão no ATM, para interagir com o banco de dados de informações da conta do banco. Um banco de dados é uma coleção organizada de dados armazenados em um computador.] Para cada conta bancária, o banco [não desempenha nenhum papel] armazena um número de conta, um PIN e um saldo que indica a quantidade de dinheiro disponível na conta. Portanto, se o banco planeja construir somente um ATM, portanto não precisamos nos preocupar com múltiplos ATMs acessando esse banco de dados ao mesmo tempo. Além disso, supomos que o banco não faz nenhuma alteração nas informações no banco de dados enquanto um usuário está acessando o ATM. Além disso, qualquer sistema de negócios como um ATM depara-se com questões de segurança razoavelmente complexas que estão bem além do escopo de um curso de ciência da computação no primeiro ou segundo semestre. Entretanto, fazemos a suposição simplificadora de que o banco confia no ATM para acesso e manipulação das informações no banco de dados sem medidas de segurança significativas.]

Ao acessar inicialmente o ATM, o usuário deve experimentar a seguinte seqüência de eventos (mostrada na Figura 2.15):

1. A tela exibe uma mensagem de boas-vindas e solicita que o usuário insira o número da conta.
2. O usuário insere um número de conta de cinco dígitos utilizando o teclado numérico.
3. A tela solicita que o usuário insira o PIN associado com o número da conta especificada.
4. O usuário insere um PIN de cinco dígitos utilizando o teclado numérico.
5. Se o usuário inserir um número de conta válido e o PIN correto para essa conta, a tela exibe o menu principal (Figura 2.16).

~~Se o usuário inserir um número de conta inválido ou inserir um PIN incorreto, a tela exibe uma mensagem apropriada e então o ATM~~

Depois que o ATM autentica o usuário, o menu principal (Figura 2.16) exibe uma opção numerada para cada um dos três tipos de transações: consulta de saldos (opção 1), retirada (opção 2) e depósito (opção 3). O menu principal também exibe uma opção que permite ao usuário sair do sistema (opção 4). O usuário então opta por realizar uma transação (inserindo 1, 2 ou 3) ou sair do sistema (inserindo 4). Se o usuário inserir uma opção inválida, a tela exibe uma mensagem de erro e, então, reexibe o menu principal.

Se o usuário inserir 1 para fazer uma consulta de saldos, a tela exibe o saldo da conta do usuário. Para fazer isso, o ATM deve percorrer o banco de dados do banco.

As seguintes ações ocorrem quando o usuário insere 2 para fazer uma retirada:

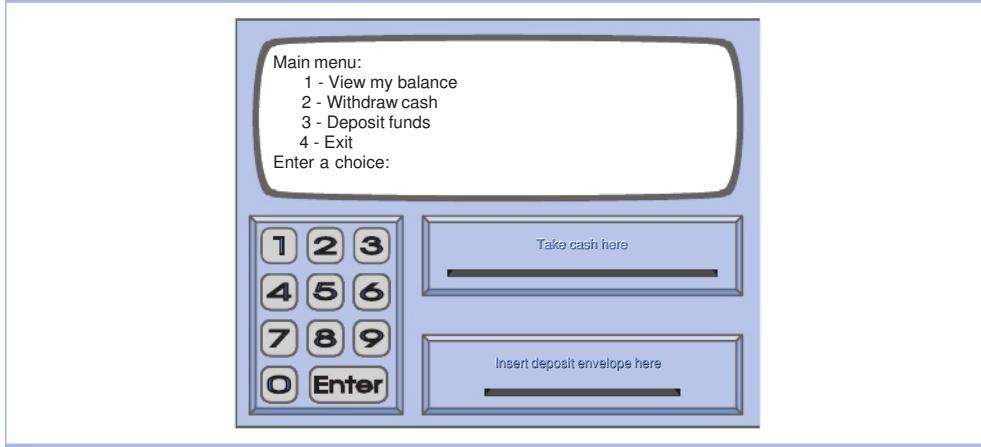


Figura 2.16 Menu principal do ATM.

6. A tela exibe um menu (mostrado na Figura 2.17) que contém quantias-padrão de saque: \$ 20 (opção 1), \$ 40 (opção 2), (opção 3), \$ 100 (opção 4) e \$ 200 (opção 5). O menu também contém uma opção para permitir que o usuário cancele a transação (opção 6).
7. O usuário insere uma seleção de menu (1-6) utilizando o teclado.
8. Se a quantia de saque escolhida for maior que o saldo da conta do usuário, a tela exibe uma mensagem declarando isso e dando a opção de cancelar a transação. O ATM prossegue para o passo 9 se o usuário optar por cancelar a transação.
9. Se o dispensador de cédulas contiver dinheiro suficiente para atender à solicitação, o ATM prossegue para o passo 10. Caso contrário, a tela exibe uma mensagem indicando o problema e solicitando que o usuário escolha uma quantia de saque menor. O ATM retorna para o passo 1.

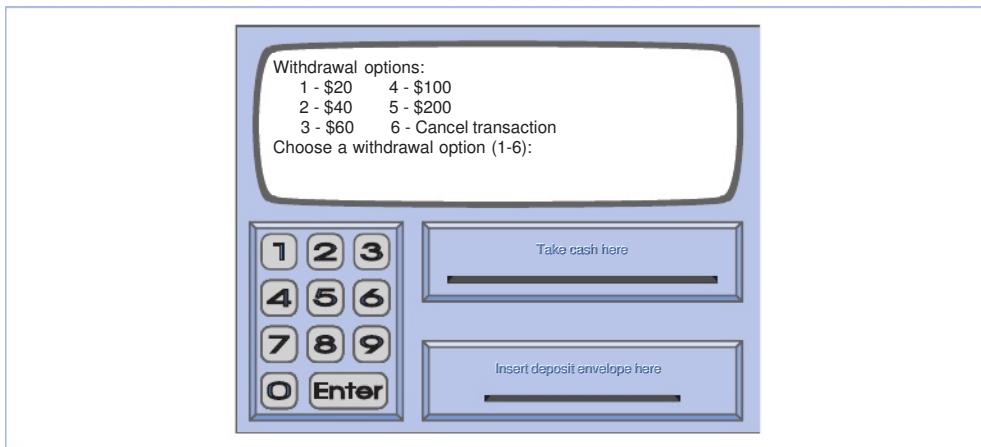


Figura 2.17 Menu de saque do ATM.

10. O ATM debita (isto é, subtrai) a quantia de saque do saldo da conta do usuário no banco de dados do banco.
 11. O dispensador de cédulas entrega a quantia de dinheiro desejada para o usuário.
 12. A tela exibe uma mensagem lembrando o usuário de pegar o dinheiro.
- As seguintes ações ocorrem quando o usuário insere 3 (enquanto o menu principal é exibido) para efetuar um depósito:
13. A tela solicita que o usuário insira uma quantia de depósito ou que digite 0 (zero) para cancelar a transação.
 14. O usuário insere uma quantia de depósito ou 0 utilizando [Nota: O decimal monetário confunde um ponto de fração decimal nem um sinal de cifrão, portanto o usuário não pode digitar uma quantia monetária real (por exemplo, \$ 1,25). Em vez disso, o usuário deve inserir uma quantia de depósito como um número de centavos (por exemplo, 125). O ATM divide esse número por 100 para obter um número que represente uma quantia monetária (por exemplo, $125 \div 100 = 1,25$).]
 15. Se o usuário especificar uma quantia de depósito, o ATM processará e aguardará por cancelar a transação (inserindo 0), o ATM exibe o menu principal (Figura 2.16) e espera pela entrada do usuário.
 16. A tela exibe uma mensagem solicitando que o usuário insira um envelope de depósito na abertura de depósito.
 17. Se a abertura de depósito receber um envelope de depósito dentro de dois minutos, o ATM credita (isto é, adiciona) a quantia de depósito ao saldo de conta do usuário no banco [Nota: Esse detalhamento permanece imediatamente disponível para saque. Primeiro o banco deve verificar fisicamente a quantia de dinheiro no envelope de depósito e quaisquer cheques envelope devem ser transferidos do emissor do cheque para a conta do depositário. Quando qualquer um desses eventos ocorrer, o banco atualiza apropriadamente o saldo do usuário armazenado no seu banco de dados. Isso ocorre independentemente do sistema ATM.] Se a abertura para depósito não receber o envelope de um depósito dentro desse período, a tela exibe a mensagem informando que o sistema cancelou a transação devido à inatividade. O ATM exibe então o menu principal e espera a entrada do usuário.

Depois que o sistema realiza com sucesso uma transação, ele deve reexibir o menu principal (Figura 2.16) para que o usuário realize outras transações. Se o usuário escolher sair do sistema (opção 4), a tela exibe uma mensagem de agradecimento e então a mensagem de boas-vindas para o próximo usuário.

Analizando o sistema ATM

A instrução anterior é um exemplo simplificado de um documento de requisitos. Em geral, esse documento é o resultado de um processo de coleta de requisitos que poderia incluir entrevistas com potenciais usuários do sistema e especialistas em campos relacionados ao sistema. Por exemplo, um analista de sistemas contratado para preparar um documento de requisitos para software de operações bancárias (por exemplo, o sistema ATM descrito aqui) poderia entrevistar especialistas financeiros para obter um melhor entendimento sobre o que o software deve fazer. O analista utilizaria as informações obtidas para auxiliar o projeto de sistemas de orientar os projetistas de sistemas.

O processo de coleta de requisitos é uma tarefa-chave da primeira etapa do ciclo de vida do software. O processo especifica as etapas pelas quais o software evolui desde o momento em que é inicialmente concebido até o momento em que é usado. Essas etapas em geral incluem: análise, projeto, implementação, teste e depuração, implantação, manutenção e retirada. Há vários modelos de ciclo de vida de software, cada um com suas próprias preferências e especificações para o momento e a frequência com que os engenheiros de software devem realizar [Nota: A maioria dessas etapas] cada etapa uma vez em sucessão, enquanto outros iterativos podem repetir uma ou mais etapas várias vezes por todo um ciclo de vida do produto.

A etapa da análise do ciclo de vida de software se concentra na definição do problema a ser solucionado. Ao projetar qualquer sistema, deve-se certamente corretamente o problema. [Nota: Um aspecto particularmente importante é determinar o problema correto.] Os analistas de sistemas coletam os requisitos que indicam o problema específico a ser solucionado. Nossa documentação de requisitos descreve nosso sistema ATM em detalhes suficientes para que você não precise passar por uma extensa etapa de análise — isso é feito para você.

Para capturar o que um sistema proposto deve fazer, os desenvolvedores costumam empregar uma técnica conhecida como [Nota: Esse processo identifica os usos do sistema, cada um dos quais representa uma capacidade diferente que o sistema fornece para seus clientes. Por exemplo, ATMs em geral têm vários casos de uso, como 'Visualizar saldo de conta', 'Sacar dinheiro', 'Depositar fundos', 'Transferir fundos entre contas' e 'Comprar selos de postagem']. O sistema ATM simplificado que criamos nesse estudo de caso permite ao leitor contrapor a descrição do sistema (Figura 2.18) as descrições dos casos de uso do ATM no documento de requisitos; as listas de passos necessários para realizar cada tipo de transação (isto é, consulta de saldo, e depósito) na verdade descreveram os três casos de uso do nosso ATM — 'Visualizar saldo em conta', 'Sacar dinheiro' e 'Depositar fundos'.

Diagramas de casos de uso

Agora introduzimos o primeiro de vários diagramas UML em nosso estudo de caso de ATM. Criamos um diagrama para modelar as interações entre os clientes de um sistema (neste estudo de caso, clientes do banco) e o sistema. O objetivo é mostrar os tipos de interações entre usuários e um sistema sem fornecer os detalhes — estes são fornecidos em outros diagramas UML.

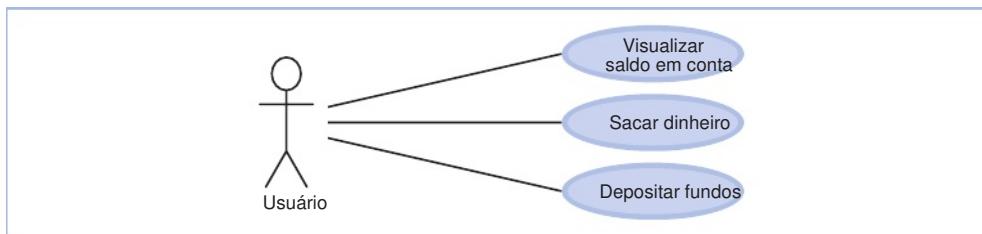


Figura 2.18 Diagrama de casos de uso para o sistema ATM da perspectiva do usuário.

apresentamos por todo o estudo de caso). Os diagramas de caso de uso costumam ser acompanhados por texto informal que explica os casos de uso em mais detalhes — como o texto que aparece no documento de requisitos. Os diagramas de casos de uso são criados durante a etapa de análise do ciclo de vida do software. Em sistemas maiores, diagramas de casos de uso são ferramentas muito úteis mas indispensáveis, que ajudam projetistas de sistema a permanecer concentrados em atender às necessidades dos usuários.

A Figura 2.18 mostra o diagrama dos casos de uso para nosso sistema ATM. O ator 'usuário' representa um papel que uma entidade externa — como uma pessoa ou outro sistema — reproduz ao interagir com o sistema. Para nosso ATM é o Usuário que pode visualizar um saldo em conta, sacar dinheiro e depositar fundos na ATM. O Usuário não é uma pessoa real, mas sim um papel que uma pessoa real — ao reproduzir a parte de um Usuário — pode reproduzir ao interagir com o sistema. Observe que um diagrama de casos de uso pode incluir múltiplos atores. Por exemplo, o diagrama de casos de uso para o sistema de um banco real poderia também incluir um ator chamado Administrador que recarrega o dispensador de cédulas todos os dias.

Identificamos o ator em nosso sistema examinando o documento de requisitos, que declara 'usuários de ATM devem ser capazes de ver o saldo em suas contas, sacar dinheiro e depositar fundos'. Portanto, o ator em cada um dos três casos de uso é o Usuário que interage com o ATM. Uma entidade externa — uma pessoa real — desempenha o papel do Usuário para realizar transações finais. A Figura 2.18 mostra um ator, cujo nome, Usuário, aparece abaixo do ator no diagrama. A UML modela cada caso de uso como um óval conectado a um ator com uma linha sólida.

Os engenheiros de software (mais precisamente, os projetistas de sistemas) devem analisar o documento de requisitos ou um documento de casos de uso e projetar o sistema antes que os programadores o implementem em uma linguagem de programação particular. Na etapa de análise, os projetistas de sistemas se concentram no entendimento do documento de requisitos para produzir uma especificação de projeto de alto nível que descreve o que o sistema deve fazer. A saída da etapa de projeto deve especificar claramente como o sistema deve ser construído a fim de satisfazer esses requisitos. Nas várias seções "Estudo de caso sobre o software" a seguir, realizamos os passos do processo de um projeto simples orientado a objetos (OOD) no sistema ATM para produzir uma especificação de projeto que contém uma coleção de diagramas da UML e texto de suporte. Lembre-se de que a UML é principalmente uma linguagem de especificação de projeto que pode ser utilizada com qualquer processo OOD. Há muitos desses processos, dos quais o mais conhecido é o Rational Unified Process (RUP), desenvolvido pela Rational Software Corporation (agora uma divisão da IBM). O RUP é um processo rico concebido para aplicativos com 'poder industrial'. Para esse estudo de caso, apresentamos nosso próprio processo de projeto simplificado.

Projetando o sistema ATM

Agora, começamos a etapa de projeto do nosso sistema ATM. Tudo junto de componentes que interage para resolver um problema. Por exemplo, para que o sistema ATM realize as tarefas projetadas, nosso sistema ATM tem uma interface com o mundo exterior (Figura 2.15), contém um software que executa as transações financeiras e interage com um banco de dados das informações da conta bancária. A estrutura do sistema descreve os objetos do sistema e seus inter-relacionamentos. O comportamento do sistema descreve como o sistema muda à medida que seus objetos interagem entre si. Cada sistema apresenta tanto uma estrutura quanto um comportamento — os projetistas devem especificar ambos. Há vários tipos distintos de estruturas e comportamentos de sistemas. Por exemplo, as interações entre objetos no sistema diferem daquelas entre o usuário e o sistema, contudo ambos constituem uma estrutura de comportamento de sistema.

A UML 2 especifica 13 tipos de diagramas para documentar os modelos dos sistemas. Cada um modela uma característica da estrutura ou comportamento de um sistema. Listamos aqui somente os seis tipos de diagramas utilizados em nosso estudo — um desses (diagrama de classes) modela a estrutura do sistema — os cinco restantes modelam o comportamento do sistema. Vamos dar uma visão geral dos tipos de diagramas da UML remanescentes no Apêndice H, "UML 2: Tipos de diagrama adicionais".

1. **Diagramas de casos de uso**, como o da Figura 2.18, modelam as interações entre um sistema e suas entidades externas (atores) em termos de casos de uso (capacidades do sistema, como 'Visualizar saldo em conta', 'Sacar dinheiro' e 'Depositar fundos').
2. **Diagramas de classes**, que você estudará na Seção 3.11, modelam as classes, ou os 'blocos de construção' utilizados em um sistema. Todo substantivo ou 'coisa' descrito no documento de requisitos é candidato a ser uma classe no sistema (por exemplo, 'conta', 'usuário', 'transação').

'conta', 'teclado numérico'). Os diagramas de classes nos ajudam a especificar os relacionamentos estruturais entre partes do sistema. Por exemplo, o diagrama de classes do sistema ATM especificará que o ATM é fisicamente composto de uma tela, dispensador de cédulas e uma abertura para depósito.

3. **Diagramas de estados de máquinas** que você estudará na Seção 3.11, modelam como um **objeto** muda de estado. O que um objeto é indicado pelos valores de todos os atributos do objeto em um determinado momento. Quando um objeto muda de estado, esse objeto pode comportar-se diferentemente no sistema. Por exemplo, depois de validar o PIN de um usuário, o ATM passa do estado 'usuário não autenticado' para o estado 'usuário autenticado' quando o ATM permite ao usuário realizar transações financeiras (por exemplo, visualizar o saldo em conta, sacar dinheiro, depositar fundos).
4. **Diagramas de atividades** que você também estudará na Seção 5.11, modelam o fluxo de trabalho do objeto (sequência de eventos) durante a execução do programa. Um diagrama de atividades modela as ações que o objeto realiza e especifica a ordem em que ele realiza essas ações. Por exemplo, um diagrama de atividades mostra que o ATM deve obter o saldo em conta do usuário (a partir do banco de dados com as informações da conta bancária) antes de a tela poder exibir o saldo para o usuário.
5. **Diagramas de comunicação** (também chamados de **diagramas de colaboração**) nas versões anteriores da UML modelam as interações entre os objetos em um sistema, quando interações ocorrem. Você aprenderá na Seção 7.12 que esses diagramas mostram quais objetos devem interagir para realizar uma transação no ATM. Por exemplo, o ATM deve se comunicar com o banco de dados de informações da conta bancária para obter o saldo em uma conta.
6. **Diagramas de seqüência** também modelam as interações entre os objetos em um sistema, mas diferentemente dos diagramas de comunicação, eles enfatizam as interações que ocorrem. Você aprenderá na Seção 7.12 que esses diagramas ajudam a mostrar a ordem em que as interações ocorrem ao executar uma transação financeira. Por exemplo, a tela solicita que o usuário insira uma quantia de saque antes de o dinheiro ser entregue.

Na Seção 3.11, continuamos a projetar nosso sistema ATM identificando as classes no documento de requisitos. Realizamos extrair substantivos simples chave e substantivos compostos chave do documento de requisitos. Utilizando essas classes, desenhamos nosso primeiro rascunho do diagrama de classes que modela a estrutura do nosso sistema ATM.

Recursos na Internet e na Web

Os URLs a seguir fornecem as informações sobre o projeto orientado a objetos com a UML.

www-306.ibm.com/software/rational/uml/

Lista as perguntas feitas com freqüência sobre a UML, fornecidas pela IBM Rational.

www.softdocwiz.com/Dictionary.htm

Hospeda o Unified Modeling Language Dictionary, que lista e define todos os termos utilizados na UML.

www-306.ibm.com/software/rational/offering/design.html

Fornece informações sobre o software da IBM Rational disponíveis para projetar sistemas. Fornece downloads gratuitos de versões para avaliação de 30 dias de vários produtos, como o IBM Rational Rose.

www.embarcadero.com/products/describe/index.html

Fornece uma licença gratuita para avaliação de 15 dias para a ferramenta de modelagem UML da Embarcadero Technologies.

www.borland.com/together/index.html

Fornece uma licença de 30 dias gratuita para download de uma versão beta da Borland Together, uma ferramenta de desenvolvimento de software que suporta a UML.

www.ilogix.com/rhapsody/rhapsody.cfm

Fornece uma licença de 30 dias gratuita para download de uma versão de avaliação gratuita de Rhapsody, um ambiente de desenvolvimento orientado ao modelo baseado em UML 2.

argouml.tigris.org

Contém informações e downloads do ArgoUML, uma ferramenta de UML gratuita e de código-fonte aberto.

www.objectsbydesign.com/books/booklist.html

Lista livros sobre a UML e projeto orientado a objetos.

www.objectsbydesign.com/tools/umltutorial_byCompany.html

Lista as ferramentas de software que utilizam a UML, como IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody e Gentleware Poseidon for UML.

www.ootips.org/ood-principles.html

Fornece respostas à pergunta: 'O que faz um bom projeto orientado a objetos?'

www.cetus-links.org/oo_uml.html

Introduz a UML e fornece links para inúmeros recursos da UML.

www.agilemodeling.com/essays/umlDiagrams.htm

Fornecer descrições e tutoriais detalhados sobre cada um dos 13 tipos de diagramas da UML.

Leituras recomendadas

Os livros a seguir fornecem informações sobre o projeto orientado a objetos com a UML.

Booch, Object-oriented analysis and design with applications Boston: Addison-Wesley, 2004.

Eriksson, H., UML 2 tool New York: John Wiley, 2003.

Kruchten, The rational unified process: an introduction Boston: Addison-Wesley, 2004.

Larman, Applying UML and patterns: an introduction to object-oriented analysis and design Saddle River, NJ: Prentice Hall, 2002.

Rouques, UML in practice: the art of modeling software systems demonstrated through well-known examples and solutions New York: Addison-Wesley, 2004.

Rosenberg, D. and K. Sjøtun, Applying use case driven object modeling with UML: an annotated reading MA: Addison-Wesley, 2000.

Rumbaugh, J., I. Jacobson and G. Booch, Complete UML training manual Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson and G. Booch, UML modeling language reference Reading MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson and G. Booch, Software development: a unified approach Reading MA: Addison-Wesley, 1999.

Exercícios de revisão do estudo de caso de engenharia de software

- 2.1 Suponha que permitimos aos usuários do nosso sistema ATM transferir dinheiro entre duas contas bancárias. Modifique o diagrama de caso de uso da Figura 2.18 para refletir essa alteração.
- 2.2 _____ modelam as interações entre objetos em um sistema com ênfase em quando essas interações ocorrem.
 - a) Diagramas de classes
 - b) Diagramas de seqüências
 - c) Diagramas de comunicação
 - d) Diagramas de atividades
- 2.3 Qual das opções a seguir lista as etapas de um ciclo de vida de software típico em uma ordem seqüencial?
 - a) projeto, análise, implementação, teste
 - b) projeto, análise, teste, implementação
 - c) análise, projeto, teste, implementação
 - d) análise, projeto, implementação, teste

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 2.1 A Figura 2.19 contém um diagrama de casos de uso para uma versão modificada do nosso sistema ATM que também permite aos usuários transferir dinheiro entre contas.
- 2.2 b.
- 2.3 d.

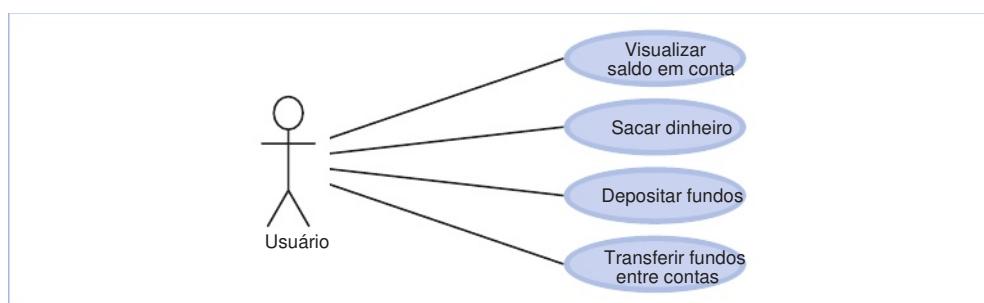


Figura 2.19 Diagrama de casos de uso para uma versão modificada do nosso sistema ATM que também permite aos usuários transferir dinheiro entre contas.

2.9 Síntese

Você aprendeu muitos recursos importantes do C++ neste capítulo, incluindo como exibir dados na tela, inserir dados a partir do teclado para construir programas interativos simples. Exploramos como as variáveis são armazenadas na memória e recuperadas a partir dela. Você também aprendeu a utilizar os operadores aritméticos para fazer cálculos. Discutimos a ordem em que o C++ executa os operadores (isto é, as regras de precedência de operadores), bem como a associatividade dos operadores. Você também aprendeu que o C++ permite que um programa tome decisões. Por fim, introduzimos os operadores de igualdade e relacionais, que você utiliza para formar condições em instruções.

Os aplicativos não orientados a objeto apresentados aqui introduziram os conceitos básicos de programação. Como você verá no Capítulo 3, os aplicativos C++ em geral contêm apenas algumas linhas de código que normalmente criam os objetos que fazem o trabalho do aplicativo e, então, os objetos ‘assumem a partir daí’. No Capítulo 3, você aprenderá a implementar suas próprias classes e a utilizar objetos dessas classes nos aplicativos.

Resumo

- Comentários de uma única linha/`/*...*/` Os programadores inserem comentários para documentar programas e aprimorar sua legibilidade.
- Os comentários não fazem com que o computador realize qualquer ação quando o programa está em execução — eles são ignorados pelo compilador C++ e não fazem com que qualquer código-objeto de linguagem de máquina seja gerado.
- Uma diretiva de pré-processador `#include <iostream>` instrui o pré-processador do C++ a incluir o conteúdo do arquivo de cabeçalho de fluxo de entrada/saída no programa. Esse arquivo contém as informações necessárias para compilar programas que operadores `<>`
- Os programadores utilizam espaço em branco (isto é, linhas em branco, caracteres de espaço em branco e caracteres de tabulação) para tornar os programas mais fáceis de ler. Os caracteres de espaço em branco são ignorados pelo compilador.
- Os programas C++ começam executando a função `main`, que não aparece primeiro no programa.
- A palavra-chave `return` indica que `main` retorna um valor inteiro.
- Uma chave esquerda `{` inicia o corpo de cada função. Uma chave direita `}` encerra o corpo de cada função.
- Uma string entre aspas duplas é, às vezes, referida como uma string de caracteres, mensagem ou literal string. Os caracteres de espaço em branco em strings não são ignorados pelo compilador.
- Cada instrução deve terminar com um ponto-e-vírgula (também conhecido como terminador de instrução).
- A saída e a entrada em C++ são feitas com fluxos de caracteres.
- O objeto de fluxo `cout` — normalmente conectado à tela — é utilizado para gerar saída de dados. A saída de múltiplos itens de dados pode ser gerada concatenando operadores de inserção de fluxo (`<<`).
- O objeto de fluxo `cin` — normalmente conectado ao teclado — é utilizado para inserir dados. Múltiplos itens de dados podem ser inseridos concatenando-se os operadores de extração de fluxo (`>>`).
- Os objetos de fluxo `estd::cout` facilitam a interação entre o usuário e o computador. Como essa interação se assemelha a um diálogo, ela é frequentemente chamada de computação conversacional ou computação interativa.
- A notação `estd::cout` especifica que estamos utilizando um nome, que pertence ao ‘namespace’
- Quando uma barra invertida (isto é, um caractere de escape) é encontrada em uma string de caracteres, o próximo caractere é combinado com a barra invertida para formar uma seqüência de escape.
- A seqüência de `\n` especifica nova linha. Ela faz com que o cursor (isto é, o indicador da posição de tela atual) se move para o começo da próxima linha na tela.
- Uma mensagem que direciona o usuário para executar uma ação específica é conhecida como um prompt.
- A palavra-chave `return` é um de vários meios de sair de uma função.
- Todas as variáveis em um programa C++ devem ser declaradas antes de ser utilizadas.
- Um nome de variável em C++ é qualquer identificador válido que não seja uma palavra-chave. Um identificador é uma série de caracteres consistindo em letras, dígitos e sublinhados; identificadores não podem iniciar com um dígito. Identificadores C++ podem ter qualquer comprimento; entretanto, alguns sistemas e/ou implementações C++ podem impor algumas restrições ao comprimento de identificadores.
- O C++ diferencia letras maiúsculas de minúsculas.
- A maioria dos cálculos é realizada em instruções de atribuição.

- Uma variável é uma posição na memória do computador onde um valor pode ser armazenado para utilização por um programa.
 - As variáveis ~~contém~~ armazenam valores inteiros, isto é, números inteiros como 7, -11, 0, 31.914.
 - Cada variável armazenada na memória do computador tem um nome, um valor, um tipo e um tamanho.
 - Sempre que um novo valor é colocado em uma posição da memória, o processo é destrutivo; isto é, o novo valor substitui o valor anterior na posição. O valor anterior é perdido.
 - Quando um valor é lido a partir da memória, o processo é não destrutivo; isto é, uma cópia do valor é lida, deixando o valor original intacto na posição da memória.
 - O manipulador ~~cout~~ gera a saída de um caractere de nova linha e então ‘esvazia o buffer de saída’.
 - O C++ avalia expressões aritméticas em uma seqüência precisa determinada pelas regras de precedência e de associatividade de operadores.
 - Parênteses podem ser utilizados para forçar que a ordem de avaliação ocorra em qualquer seqüência desejada pelo programador.
 - A divisão de inteiros (isto é, tanto o numerador como o denominador são inteiros) produz um quociente de inteiro. Qualquer parte fracionária na divisão de inteiros é truncada — não ocorre nenhum arredondamento.
 - O operador módulo ~~mod~~ conhece o resto de divisão de inteiros. O operador módulo pode ser utilizado somente com operandos inteiros.
 - A instrução ~~if~~ permite que um programa tome uma decisão quando certa condição é satisfeita. O formato de uma instrução

```
if ( condição )  
    instrução;
```

Se a condição for verdadeira, a instrução é executada. Se a condição não for satisfeita, isto é, a condição for falsa, a instrução de corpo não será executada.

- As condições em ~~instruções~~ são comumente formadas com o uso de operadores de igualdade e operadores relacionais. O resultado do uso desses operadores é sempre o valor verdadeiro ou falso.
 - A declaração

```
using std::cout;
```

é uma declaração que elimina a necessidade de repetir o prefixo `cout` que incluímos essa vez, podímos escrever `cout << nome` em vez de `cout << nome` no resto de um programa.

Terminologia

/* */ comentário	erro de compilação	main função
< 'é menor que'	erro de lógica	manipulador de fluxo
=<'é menor que ou igual a'	erro de lógica não fatal	memória
>'é maior que'	erro de sintaxe	memória, posição
>=>'é maior que ou igual a'	erro em tempo de compilação	mensagem
arquivo de cabeçalho de fluxo de entrada	objeto de fluxo de entrada padrão (objeto de fluxo de saída padrão (
<iostream>	espaço em branco	operações de inserção de fluxo de concatenação
associatividade da esquerda para a direita	fechar uma função	operações de inserção de fluxo de encadeamento
associatividade de operadores	fluxo	operações de inserção de fluxo em cascata
associatividade dos operadores	função	operador
bloco	gravação destrutiva	operador aritmético
caractere de escape (identificador	operador binário
caractere de novar linha (if , instrução	operador de atribuição (
cin , objeto	igualdade, operadores de	operador de extração de fluxo (
comentário(!= 'é não igual a'	operador de inserção de fluxo (
condição	== 'é igual a'	operador de multiplicação (
corpo de uma função	instrução	operador módulo (
cout , objeto	instrução composta	operadores relacionais
cursor	int , tipo de dados	operando
decisão	inteiror()	parênteses
declaração	leitura não destrutiva	parênteses aninhados
diretiva de pré-processador	lista separada por vírgulas	parênteses redundantes
distinção de letras maiúsculas e minúsculas	literais	ponto-e-vírgula
divisão de inteiros	literal string	terminador de instrução

programa autodocumentado	<code>return</code> , instrução	terminador de instrução (
<code>prompt</code>	seqüência de escape	tipo de dados
realizar uma ação	<code>string</code>	<code>using</code> , declaração
regras de precedência de operador	<code>string</code> de caractere	variável

Exercícios de revisão

- 2.1 Preencha as lacunas em cada uma das seguintes sentenças:
- Todo programa C++ inicia a execução na função
 - A _____ inicia o corpo de cada função e a _____ termina o corpo de cada função.
 - Toda instrução em C++ termina com um(a) _____.
 - A seqüência de `\n` representa o caractere _____, que faz com que o cursor se posicione no começo da próxima linha na tela.
 - A instrução _____ é utilizada para tomar decisões.
- 2.2 Determine se cada uma das seguintes afirmações é verdadeira ou falsa. Se a resposta for falsa, explique por quê. Suponha que as instruções ; é utilizada.
- Os comentários fazem com que o computador imprima teletype depois que o programa é executado.
 - A seqüência de `\escape` quando tem sua saída gerada com o operador de inserção de fluxo, faz com que o cursor se posicione no começo da próxima linha na tela.
 - Todas as variáveis devem ser declaradas antes de ser utilizadas.
 - Todas as variáveis devem ser atribuídas a um tipo quando são declaradas.
 - O C++ considera as variáveis `num1` e `num2` idênticas.
 - As declarações podem aparecer em quase qualquer lugar no corpo de uma função C++.
 - O operador `mod` pode ser utilizado apenas com operandos inteiros.
 - Os operadores aritméticos têm o mesmo nível de precedência.
 - Um programa C++ que imprime três linhas de saída deve conter três instruções de inserção de fluxo.
- 2.3 Escreva uma única instrução C++ para realizar cada uma das seguintes tarefas (suponha que as declarações zadas):
- Declare as variáveis `num1` e `num2` como sendo `String`.
 - Solicite ao usuário inserir um inteiro. Termine sua mensagem de solicitação com dois espaços (deixe o espaço entre a pergunta e o sinal de igual).
 - Leia um inteiro do usuário no teclado e armazene o valor inserido em uma variável do tipo inteiro.
 - Se a variável `num1` não for igual a `7`, imprima "The variable number is not equal to 7".
 - Imprima a mensagem "This is a C++ program" em uma linha.
 - Imprima a mensagem "This is a C++ program" em duas linhas. Termine a primeira linha com `\n`.
 - Imprima a mensagem "This is a C++ program" com cada palavra em uma linha separada.
 - Imprima a mensagem "This is a C++ program" com cada palavra separada da palavra seguinte por uma tabulação.
- 2.4 Escreva uma instrução (ou comentário) para realizar cada uma das seguintes tarefas (suponha que as declarações zadas):
- Determine se um programa calcula o produto de três inteiros.
 - Declare as variáveis `result` como tipo `int` (em instruções separadas).
 - Peça para o usuário inserir três inteiros.
 - Leia três inteiros a partir do teclado e armazene-os nas variáveis `a`, `b` e `c`.
 - Compute o produto dos três inteiros contidos nas variáveis `a`, `b` e `c`. O resultado é armazenado na variável `result`.
 - Imprima "The product is: " seguido pelo valor da variável `result`.
 - Retorne um valor para indicar que o programa terminou com sucesso.
- 2.5 Utilizando as instruções que você escreveu no Exercício 2.4, escreva um programa completo que calcula e exibe o produto de três inteiros. Onde apropriado, adicionar comentários. [Este código ficará para escrever as instruções necessárias.]
- 2.6 Identifique e corrija os erros em cada uma das seguintes instruções (suponha que a saída é correta).
- `if (c < 7);`
cout << "c is less than 7\n" ;
 - `if (c => 7)`
cout << "c is equal to or greater than 7\n" ;

Respostas dos exercícios de revisão

- 2.1 a) main b) chave esquerda\chave direita (c) ponto-e-vírgula. d) nova linha. e)
- 2.2 a) Falsa. Os comentários não fazem com que qualquer ação seja realizada quando o programa é executado. Eles são utilizados para documentar programas e melhorar sua legibilidade.
- b) Verdadeira.
- c) Verdadeira.
- d) Verdadeira.
- e) Falsa. O C++ diferencia letras maiúsculas de minúsculas, então essas variáveis são únicas.
- f) Verdadeira.
- g) Verdadeira.
- h) Falsa. Os operadores têm a mesma precedência, e os operadores precedência menor.
- i) Falsa. Uma única instrução múltiplas seqüências\deve ser\imprimir várias linhas.
- 2.3 a) `int c, thisIsAVariable, q76354, number;`
b) `std::cout << "Enter an integer: " ;`
c) `std::cin >> age;`
d) `if (number != 7)`
 `std::cout << "The variable number is not equal to 7\n" ;`
e) `std::cout << "This is a C++ program\n";`
f) `std::cout << "This is a C++\nprogram\n";`
g) `std::cout << "This\nis\na\nC++\nprogram\n";`
h) `std::cout << "This\nis\na\nC++\nprogram\n" ;`
- 2.4 a) // Calcula o produto de três inteiros
b) `int x;`
`int y;`
`int z;`
`int result;`
c) `cout << "Enter three integers: " ;`
d) `cin >> x >> y >> z;`
e) `result = x * y * z;`
f) `cout << "The product is " << result << endl;`

2.5 (Ver programa abaixo)

```

1 // Calcula o produto de três inteiros
2 #include <iostream> // permite ao programa realizar entrada e saída
3
4 using std::cout; // o programa utiliza cout
5 using std::cin; // o programa utiliza cin
6 using std::endl; // o programa utiliza endl
7
8 // a função main inicia a execução do programa
9 int main()
10 {
11     int x; // primeiro inteiro a multiplicar
12     int y; // segundo inteiro a multiplicar
13     int z; // terceiro inteiro a multiplicar
14     int result; // o produto dos três inteiros
15
16     cout << "Enter three integers: " ; // solicita dados ao usuário
17     cin >> x >> y >> z; // lê três inteiros de usuário
18     result = x * y * z; // multiplica os três inteiros; resultado de armazenamento
19     cout << "The product is " << result << endl; // imprime resultado; termina a linha
20
21     return 0; // indica que o programa executou com sucesso
22 } // fim da função main

```

- 2.6 a) Erro: Ponto-e-vírgula depois do parêntese direito da condição na instrução
 Correção: Remova o ponto-e-vírgula depois do ~~parêntese direito~~. Esse erro é que a instrução de saída será executada se a condição na ~~instrução de saída~~ [] O ponto-e-vírgula depois do parêntese direito é uma instrução nula (ou vazia) — uma instrução que não faz nada. Aprenderemos mais sobre a instrução nula no próximo capítulo.
- b) Erro: O operador relacional
 Correção: Mude para = e você também pode querer mudar ‘igual a ou maior que’ para ‘maior que ou igual a’.

Exercícios

- 2.7 Discuta o significado de cada um dos seguintes objetos:
 a) std::cin
 b) std::cout
- 2.8 Preencha as lacunas em cada uma das seguintes sentenças:
 a) _____ são utilizados para documentar um programa e aprimorar sua legibilidade.
 b) O objeto utilizado para imprimir as informações na tela é _____.
 c) Uma instrução C++ que toma uma decisão é _____.
 d) A maioria de cálculos é geralmente feita por instruções _____.
 e) O objeto _____ insere valores a partir do teclado.
- 2.9 Escreva uma única instrução C++ ou linha que realize cada uma das seguintes tarefas:
 a) Imprima a mensagem com dois números.
 b) Atribua o produto de variável para a variável.
 c) Declare que um programa realiza um cálculo de folha de pagamento (isto é, utilize texto que ajuda a documentar um programa).
 d) Insira três valores de inteiro a partir do teclado nas variáveis de inteiro
- 2.10 Determine quais das seguintes sentenças são falsas. Seja explique suas respostas.
 a) Operadores C++ são avaliados da esquerda para a direita.
 b) Todos os seguintes nomes de variáveis são válidos: `one_seo_valido281345, j7, her_sales, his_account_total, a, b, c, z, z2`.
 c) A instrução `<< "a = 5;"` é um exemplo típico de uma instrução de atribuição.
 d) Uma expressão aritmética em C++ válida sem parênteses é avaliada da esquerda para a direita.
 e) Todos os seguintes nomes de variáveis são inválidos: `se_só_máximo221`.
- 2.11 Preencha as lacunas em cada uma das seguintes sentenças:
 a) Que operações aritméticas estão no mesmo nível de precedência que a multiplicação?
 b) Quando os parênteses são aninhados, qual conjunto de parênteses é avaliado primeiro em uma expressão aritmética?
 c) Uma posição na memória do computador que pode conter valores diferentes em várias vezes por toda a execução de um programa chamada de _____.
- 2.12 O que é impresso, se algo for, quando cada uma das seguintes instruções C++ é executada? Se nada for impresso, então responda “não”. Assuma `a = 2` e `y = 3`.
 a) cout << x;
 b) cout << x + x;
 c) cout << "x=";
 d) cout << "x = " << x;
 e) cout << x + y << " = " << y + x;
 f) z = x + y;
 g) cin >> x >> y;
 h) // cout << "x + y = " << x + y;
 i) cout << "\n";
- 2.13 Qual das instruções C++ a seguir contém variáveis cujos valores são substituídos?
 a) `pira i> p>k e Z> d >> e >> f;`
 b) cout << “variables whose values are replaced” ;
 c) cout << “a = 5” ;
- 2.14 Dada a equação algébrica $x^7 + ax^6 + bx^5 + cx^4 + dx^3 + ex^2 + fx + g = 0$, qual das seguintes, se houver alguma, são instruções C++ corretas para essa equação?
 a) $y = a * x * x * x + 7;$
 b) $y = a * x * x * (x + 7);$
 c) $y = (a * x) * x * (x + 7);$
 d) $y = (a * x) * x * x + 7;$

e) $y = a * (x * x * x) + 7;$
 f) $y = a * x * (x * x + 7);$

- 2.15 Declare a ordem de avaliação dos operadores em cada uma das seguintes instruções. O resultado final é realizado.
- $x = 7 + 3 * 6 / 2 - 1;$
 - $x = 2 \% 2 + 2 * 2 - 2 / 2;$
 - $x = (3 * 9 * (3 + (9 * 3 / (3))));$
- 2.16 Escreva um programa que solicita ao usuário inserir dois números, obtém os dois números do usuário e imprime a soma, produto, diferença e quociente dos dois números.
- 2.17 Escreva um programa que imprime os números 1 a 4 na mesma linha com cada par de números adjacentes separados por um espaço. Isto é, de várias maneiras:
- Utilizando uma instrução com um operador de inserção de fluxo.
 - Utilizando uma instrução com quatro operadores de inserção de fluxo.
 - Utilizando quatro instruções.
- 2.18 Escreva um programa que pede para o usuário inserir dois inteiros, obtém os números do usuário, e então imprime o maior número seguido pelas palavras "maior". Se os números forem iguais, imprime "os dois são iguais".
- 2.19 Escreva um programa que insere três inteiros a partir do teclado e imprime a soma, a média, o produto, o menor e o maior desses números. O diálogo de tela deve se parecer com o seguinte:

```
Entre com três valores inteiros: 13 27 14
Soma: 54
Média: 18
Produto: 4914
O menor: 13
O maior: 27
```

- 2.20 Escreva um programa que lê o raio de um círculo como um inteiro e imprime o diâmetro, a circunferência e a área do círculo. Utilize o valor constante 3,14159 para todos os cálculos em instruções de saída. [Neste capítulo, discutimos apenas as constantes e variáveis de inteiro. No Capítulo 4, discutimos números de ponto flutuante, isto é, valores que podem ter pontos de fração decimal.]
- 2.21 Escreva um programa que imprime uma caixa, uma oval, uma seta e um losango da seguinte maneira:

```
*****      ***      *      *
*   *   *   *   *   ***   *   *
*   *   *   *   *   *   *   *
*   *   *   *   *   *   *   *
*   *   *   *   *   *   *   *
*   *   *   *   *   *   *   *
*****      ***      *      *
```

- 2.22 O que o seguinte código imprime?
- ```
cout << "***\n**\n***\n***\n*****" << endl;
```
- 2.23 Escreva um programa que lê cinco inteiros e determina e imprime o maior e o menor inteiro no grupo. Utilize somente as técnicas de programação que você aprendeu neste capítulo.
- 2.24 Escreva um programa que lê um inteiro e determina e imprime se este é par ou ímpar. [Este é o problema 2.] Um número par é um múltiplo de dois. Qualquer múltiplo de dois deixa um resto de zero quando dividido por 2.]
- 2.25 Escreva um programa que lê dois inteiros e determina e imprime se o primeiro é maior que o segundo. [Este é o problema 2.]
- 2.26 Exiba o seguinte padrão de tabuleiro com oito instruções de saída e, em seguida, exiba o mesmo padrão utilizando o menor número de instruções possível.

```



```

- 2.27 Eis uma pequena antecipação do que está por vir. Neste capítulo, você aprende sobre tipos de caracteres. Para representar letras maiúsculas, minúsculas e uma variedade considerável de símbolos especiais. O C++ utiliza inteiros pequenos internamente para representar cada caractere diferente. O conjunto de caracteres que um computador utiliza e das correspondentes representações forma de inteiro desses caracteres é chamado de [caractere](#) desse computador. Você pode imprimir um caractere colocando esse caractere entre aspas simples, como em

```
cout << 'A'; // imprime um A maiúsculo
```

Você pode imprimir o equivalente inteiro de um caractere utilizando a seguinte maneira:

```
cout << static_cast < int >('A'); // imprime 'A' como um inteiro
```

Isso é chamado de [operação de coerção](#) (introduzimos formalmente as coerções no Capítulo 4). Quando a instrução precedente executar, ela imprimirá o valor 65 (em sistemas que utilizam a tabela ASCII). Escreva um programa que imprime o número inteiro equivalente de um caractere digitado no teclado. Teste seu programa várias vezes utilizando letras maiúsculas, minúsculas, dígitos e caracteres especiais, como

- 2.28 Escreva um programa que insere um inteiro de cinco dígitos, separa o inteiro em seus dígitos individuais e imprime os dígitos separados entre si por três espaços. [Use operadores de divisão de inteiros e módulo.] Por exemplo, se o usuário digitar 42339, o programa deve imprimir:

```
4 2 3 3 9
```

- 2.29 Utilizando apenas as técnicas aprendidas neste capítulo, escreva um programa que calcula os quadrados e cubos dos inteiros de 0 a 10. Use tabulações para imprimir as seguintes tabelas de valores, elegantemente formatadas:

| inteiro | quadrado | cubo |
|---------|----------|------|
| 0       | 0        | 0    |
| 1       | 1        | 1    |
| 2       | 4        | 8    |
| 3       | 9        | 27   |
| 4       | 16       | 64   |
| 5       | 25       | 125  |
| 6       | 36       | 216  |
| 7       | 49       | 343  |
| 8       | 64       | 512  |
| 9       | 81       | 729  |
| 10      | 100      | 1000 |



Você verá uma coisa nova.  
Duas coisas. E eu as chamo  
Coisa Um e Coisa Dois.  
Dr. Theodor Seuss Geisel

Nada pode ter valor sem ser um  
objeto útil.  
Karl Marx

Seus servidores públicos  
prestam-lhe bons serviços.  
Adlai E. Stevenson

Saber responder a quem fala,  
Para responder a quem envia  
uma mensagem.  
Amenemope

## Introdução a classes e objetos

### OBJETIVOS

Neste capítulo, você aprenderá:

O que são classes, objetos, funções-membro e membros de dados.

A definir uma classe e utilizá-la para criar um objeto.

A definir funções-membro em uma classe para implementar os comportamentos da classe.

A declarar membros de dados em uma classe para implementar os atributos da classe.

A chamar uma função-membro de um objeto para fazê-la realizar sua tarefa.

As diferenças entre membros de dados de uma classe e as variáveis locais de uma função.

Como utilizar um construtor para assegurar que os dados de um objeto sejam inicializados quando o objeto for criado.

Como projetar uma classe para separar sua interface de sua implementação e encorajar reutilização.

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P<br>á<br>m<br>S | <ul style="list-style-type: none"> <li>3.1 Introdução</li> <li>3.2 Classes, objetos, funções-membro e membros de dados</li> <li>3.3 Visão geral dos exemplos do capítulo</li> <li>3.4 Definindo uma classe com uma função-membro</li> <li>3.5 Definindo uma função-membro com um parâmetro</li> <li>3.6 Membros de dados, funções set e funções get</li> <li>3.7 Inicializando objetos com construtores</li> <li>3.8 Colocando uma classe em um arquivo separado para reusabilidade</li> <li>3.9 Separando a interface da implementação</li> <li>3.10 Validando dados com funções set</li> <li>3.11 Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional)</li> <li>3.12 Síntese</li> </ul> |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

### 3.1 Introdução

No Capítulo 2, você criou programas simples que exibiam mensagens para o usuário, informações obtidas do usuário, cálculos realizados e tomadas de decisão. Neste capítulo, você começará escrevendo programas que empregam os conceitos básicos da programação de objetos que introduzimos na Seção 1.17. Um recurso comum de cada programa do Capítulo 2 era que todas as instruções que realizavam tarefas localizavam-se na função principal, os programas que você desenvolve neste livro contendo uma única função ou mais classes, cada uma contendo membros de dados e funções-membro. Se você se tornar parte de uma equipe de desenvolvimento na indústria, poderá trabalhar em sistemas de software com centenas, ou até milhares, de classes. Neste capítulo, desenvolvemos estrutura simples e bem projetada para organizar programas orientados a objetos em C++.

Primeiro, motivamos a noção de classes com um exemplo do mundo real. Em seguida, apresentamos uma sequência cuidadosamente planejada de exercícios que introduzirá o leitor ao mundo das classes, dedicando tempo a discussões sobre suas estruturas, classes, funções e membros.

Este estudo de caso é integrado ao modo de avaliação de teste da classe dedicada ao estudo de suas estruturas, classes, funções e membros. Ele também serve como quadro de notas de testes dos alunos. Esse estudo de caso é aprimorado nos próximos capítulos, culminando com a versão apresentada no Capítulo 7, “Arrays e vetores”.

### 3.2 Classes, objetos, funções-membro e membros de dados

Vamos iniciar com uma analogia simples para ajudar a reforçar o entendimento da Seção 1.17 de classes e seu conteúdo. Suponha queira guiar um carro e fazê-lo andar mais rápido pisando no pedal acelerador. O que deve acontecer antes que você possa fazer isso? Bem, antes de você poder dirigir um carro, alguém tem de projetá-lo e construí-lo. Em geral, um carro inicia com os desenhos de engenharia, semelhantes às plantas utilizadas para projetar uma casa. Esses desenhos incluem o design de um pedal acelerador que o motorista usará para fazer o carro andar mais rápido. De certo modo, o pedal ‘oculta’ os complexos mecanismos que realmente fazem o carro ir mais rápido, assim como o pedal do freio ‘oculta’ os mecanismos que diminuem a velocidade do carro, o volante ‘oculta’ os mecanismos que mudam a direção do carro e assim por diante. Isso permite que pessoas com pouco ou nenhum conhecimento sobre como os carros são projetados possam dirigí-los facilmente, simplesmente utilizando o acelerador, o freio, o volante, o mecanismo de troca de marcha e outras ‘interfaces’ simples e amigáveis ao usuário para os complexos mecanismos internos do carro.

Infelizmente, você não pode dirigir os desenhos de engenharia de um carro — antes que você possa dirigir um carro, ele precisa ser construído a partir dos desenhos de engenharia que o descrevem. Um carro pronto terá um acelerador real para fazer o carro andar rapidamente. Mas mesmo isso não é suficiente — o carro não acelerará sozinho, então o motorista deve pressionar o acelerador.

Realmente queremos utilizar esse exemplo de carro para introduzir os conceitos de programação orientada a objeto desta seção. Vamos utilizar um programa pequeno para ilustrar o conceito de classe. A função descrição mecanismos que realmente realizam suas tarefas. A função oculta de seu usuário as tarefas complexas que ele realiza, assim como o acelerador de um carro oculta do motorista os complexos mecanismos que fazem o carro andar mais rapidamente. Em C++, podemos criar uma unidade de programa chamada classe para abrigar uma função, assim como os desenhos de engenharia do carro abrigam o projeto de um pedal acelerador. Considerando a Seção 1.17, lembre-se de que uma função que pertence a uma classe é chamada função-membro. Em uma classe, você fornece uma ou mais funções-membro que são projetadas para realizar as tarefas da classe. Por exemplo, uma classe que representa uma conta bancária poderia conter uma função-membro para depositar dinheiro na conta, ou retirar dinheiro da conta e uma terceira para consultar o saldo atual da conta.

Assim como você não pode dirigir um desenho de engenharia de um carro, você não pode ‘dirigir’ uma classe. Assim como é impossível construir um carro a partir de desenhos de engenharia antes que o carro possa ser realmente dirigido, você deve criar uma classe antes de fazer um programa realizar as tarefas que a classe descreve. Essa é uma razão pela qual C++ é conhecido como linguagem de programação orientada a objetos. Observe também que os problemas são construídos a partir do mesmo desenho de engenharia; os problemas podem ser construídos a partir da mesma classe.

Ao dirigir um carro, o ato de pressionar o acelerador envia uma mensagem para o carro realizar uma tarefa — isto é, fazer o carro andar mais rapidamente. De modo semelhante, quando se manda uma mensagem para um objeto — cada mensagem é conhecida como uma função-membro — diz para uma função-membro do objeto realizar sua tarefa. Isso é frequentemente chamado de ‘delegar’.

Até aqui utilizamos a analogia do carro para introduzir classes, objetos e funções-membro. Além das capacidades de um carro, também tem muitos atributos, como a cor, o número de portas, a quantidade de gasolina no tanque, a velocidade atual e o total percorrido (isto é, a leitura do odômetro). Como as capacidades do carro, esses atributos são representados como parte do projeto do carro em seus diagramas de engenharia. Quando você dirige um carro, esses atributos estão sempre associados com o carro. C

mostrar que os atributos de um carro são membros da classe carro, que é o tipo de objeto quando ele é salvo em um programa. Esses atributos são especificados como parte da classe do objeto. Por exemplo, um objeto conta bancária tem um saldo que representa a quantia de dinheiro na conta. Cada objeto conta bancária sabe o saldo da conta que ele representa, mas os saldos de outras contas no banco. Os atributos são especificados pelos membros de dados da classe.

### 3.3 Visão geral dos exemplos do capítulo

O restante deste capítulo apresenta sete exemplos simples que demonstram os conceitos que introduzimos no contexto da analise de um carro. Esses exemplos, resumidos a seguir, constroem incrementalmente as classes e os conceitos:

1. O primeiro exemplo apresenta uma classe que simplesmente exibe uma mensagem de boas-vindas quando é chamado. Então mostramos como criar um objeto dessa classe e chamar a função-membro para ele a mensagem de boas-vindas.
2. O segundo exemplo modifica o primeiro, permitindo que a função-membro receba um nome do curso como argumento. Então, a função-membro exibe o nome do curso como parte da mensagem de boas-vindas.
3. O terceiro exemplo mostra como armazenar o nome do curso. Para este terceiro exemplo, também mostramos como utilizar funções-membro para configurar o nome do curso no objeto e obter o nome do curso do objeto.
4. O quarto exemplo demonstra como usar a classe GradeBook para inicializar quando o objeto é criado. Esse exemplo também demonstra que cada objeto mantém seu próprio membro de dados nome do curso.
5. O quinto exemplo modifica o quarto demonstrando como colocar uma classe separado para permitir a reusabilidade de software.
6. O sexto exemplo modifica o quinto demonstrando o bom princípio de engenharia de software de fazer uma separação entre interface da classe e sua implementação. Isso torna a classe mais fácil de modificar sem afetar nenhum classe — isto é, qualquer classe ou função que chame as funções-membro dos objetos da classe a partir de fora dos objetos.
7. O último exemplo aprimora a classe introduzindo a validação de dados, que assegura que os dados em um objeto se adaptam a um formato particular ou estejam dentro de um intervalo de valores adequados. Por exemplo, um objeto exigirá um valor de mês no intervalo 1–12. Nesse exemplo, a função-membro que configura o nome do curso de um objeto assegura que o nome do curso tenha 25 ou menos caracteres. Se não, a função-membro utiliza somente os primeiros 25 caracteres do nome do curso e exibe uma mensagem de advertência.

Observe que os exemplos neste capítulo realmente não processam nem armazenam notas. Começamos a processar as notas com a classe GradeBook no Capítulo 4 e as armazenamos com a classe GradeBook no Capítulo 7, “Arrays e vetores”.

**Exemplo 3.1 Definindo uma classe com uma função-membro** Iniciamos com um exemplo (Figura 3.1) que define uma classe que representa um livro de notas que um instrutor pode utilizar para manter as notas que os alunos tiraram nas provas. A classe não cria um objeto. Esse é o primeiro em uma série de exemplos graduados que preparam o estudante para o uso profissional no Capítulo 7, “Arrays e vetores”. A figura apresenta a classe e sua função-membro para exibir na tela uma mensagem de boas-vindas ao instrutor para o programa de livro de notas.

Primeiro descrevemos como definir uma classe e uma função-membro. Depois explicamos como um objeto é criado e como é uma função-membro de um objeto. Os primeiros poucos exemplos da lista a seguir mostra a utilização no mesmo arquivo. Mais adiante no capítulo, introduzimos maneiras mais sofisticadas de estruturar programas para alcançar melhor engenharia de software.

```

1 // Figura 3.1: fig03_01.cpp
2 // Define a classe GradeBook com uma função-membro displayMessage;
3 // Cria um objeto GradeBook e chama sua função displayMessage.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // Definição da classe GradeBook
9 class GradeBook
10 {
11 public :
12 // função que exibe uma mensagem de boas-vindas ao usuário do GradeBook
13 void displayMessage()
14 {
15 cout << "Welcome to the Grade Book!"<< endl;
16 } // fim da função displayMessage
17 }; // fim da classe GradeBook
18
19 // a função main inicia a execução do programa
20 int main()
21 {
22 GradeBook myGradeBook;// cria um objeto GradeBook chamado myGradeBook
23 myGradeBook.displayMessage();chama a função displayMessage do objeto
24 return 0; // indica terminação bem-sucedida
25 } // fim de main

```

Welcome to the Grade Book!

Figura 3.1 Definindo a classe GradeBook com uma função-membro, criando um objeto GradeBook chamando sua função-membro.

### ClasseGradeBook

Antes que a função `displayMessage()` (linhas 20–25) possa criar um objeto da classe, precisamos informar ao compilador quais funções-membro e membros de dados pertencem à classe. Isso é conhecido como **corpo da classe**. As linhas 9–17 contém uma função-membro `displayMessage()` (linhas 13–16) que exibe uma mensagem na tela (linha 15). Lembre-se de que uma classe é como uma planta arquitetônica — então precisamos **escrever a estrutura da classe** para que sua função-membro `displayMessage()` (linha 23) para alcançar a linha 15 executar e exibir a mensagem de boas-vindas. Logo explicaremos as linhas 22–23 em detalhes.

A definição de classe inicia na linha 9 com o **palavra-chave** `class`, seguida pelo nome da classe. Por convenção, o nome de uma classe definida pelo usuário inicia com uma letra maiúscula e, por legibilidade, cada palavra subsequente no nome de classe inicia com uma letra maiúscula. Esse estilo de uso de letras maiúsculas e minúsculas é frequentemente referido como **camel case** (caixa alta e baixa), porque o padrão de letras maiúsculas e minúsculas se assemelha à silhueta de um camelo.

O **corpo** de cada classe é colocado entre um par de chaves, `{` e `}`, e precede as **linhas** (10 e 17). A definição de classe termina com um ponto-e-vírgula (linha 17).



### Erro comum de programação 3.1

Esquecer do ponto-e-vírgula no fim de uma definição de classe é um erro de sintaxe.

Lembre-se de que a função `displayMessage()` sempre é chamada automaticamente quando você executa um programa. A maioria das funções não é chamada automaticamente. Como logo verá, você deve chamar `displayMessage()` explicitamente para instruir a realizar sua tarefa.

A linha 11 contém o **especificador de acesso**: `public:`. A palavra-chave `public` é chamada de **especificador de acesso**. As linhas 13–16 definem a função-membro `displayMessage()`. Essa função-membro aparece depois do especificador de acesso para indicar que a função está ‘disponível ao público’ — isto é, pode ser chamada por outras funções no programa e por funções-membro de outras classes. Os especificadores de acesso são sempre **seguidos por dois pontos**, quando nos referirmos ao especificador de acesso omitiremos os dois-pontos como fizemos nesta frase. A Seção 3.6 introduz um segundo especificador de acesso `private` (novamente, omitimos os dois-pontos em nossas discussões, mas os incluímos em nossos programas).

Cada função em um programa realiza uma tarefa e pode retornar um valor quando ela completa sua tarefa — por exemplo, uma função poderia realizar um cálculo e, então, retornar o resultado desse cálculo. Quando você define uma função, deve especificar o tipo de retorno para indicar o tipo do valor retornado pela função quando ela completar sua tarefa. Na linha 13, a palavra-chave esquerda do nome `displayMessage` é o tipo de retorno da função. O tipo de retorno `void` indica que `displayMessage` realizará uma tarefa, mas não retornará (isto é, devolverá) dados para esse exemplo, como veremos em breve.) quando ela completar sua tarefa. (Na Figura 3.5, veremos um exemplo de uma função que retorna um valor.)

O nome da função-membro `displayMessage` segue o tipo de retorno. Por convenção, os nomes de função iniciam com a primeira letra minúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula. Os parênteses depois do nome de membro indicam que isso é uma função. Um conjunto vazio de parênteses, como mostrado na linha 13, indica que essa função não requer dados adicionais para realizar sua tarefa. Você verá um exemplo de uma função-membro que requer dados adicionais na Seção 3.5. A linha 13 é comumente referida como `definição`. O corpo de cada função é delimitado pelas chaves esquerda e direita {}, como nas linhas 14 e 16.

O corpo de uma função contém instruções que realizam a tarefa da função. Nesse caso, o corpo da função-membro `displayMessage` (na linha 15) que exibe "Welcome to the Grade Book!" Depois que essa instrução executar, a função terá

### Erro comum de programação 3.2

Retornar um valor de uma função cujo tipo de retorno foi declarado de maneira incorreta.

### Erro comum de programação 3.3

Definir uma função dentro de outra função é um erro de sintaxe.

#### Testando a classe `GradeBook`

Em seguida, gostaríamos de utilizar a classe em nosso programa. Como você aprendeu no Capítulo 2, a função `main` controla a execução de cada programa. As linhas 20–25 da Figura 3.3 contêm a função `main` que controla a execução do nosso programa.

Nesse programa, gostaríamos de chamar a função-membro `displayMessage` da classe `GradeBook` para exibir a mensagem de boas-vindas. Em geral, você não pode chamar uma função-membro de uma classe até criar um objeto dessa classe. (Como você aprendeu na Seção 10.7, as funções são membros de uma exceção.) A linha 22 cria um `objeto da classe` `GradeBook`. Observe que o tipo da variável é a classe que definimos nas linhas 9–17. Quando declaramos variáveis do tipo fizemos no Capítulo 2, o compilador não sabe automaticamente que é o tipo fundamental. Quando escrevemos a linha 22, porém, o compilador

de que tipo `GradeBook`, incluindo a definição de classe, como fizemos nas linhas 9–17. Se omitirmos essas linhas, o compilador emitirá uma mensagem de erro como: 'undeclared identifier 'GradeBook'' no Microsoft Visual C++ (NET) ou 'GradeBook': undeclared (no GNU C++). Cada nova classe que você cria torna-se um novo tipo que pode ser utilizado para criar objetos. Os programadores podem definir novos tipos de classe conforme necessário; essa é uma das extensões mais úteis do C++ é conhecido como uma extensível

A linha 23 chama a função-membro `displayMessage` (definida nas linhas 13–16) utilizando a variável criada na linha 22. O operador `ponto()`, o nome da função `displayMessage` e um conjunto vazio de parênteses. Essa chamada faz com que a função `displayMessage` realize sua tarefa. No começo da linha 23, indica que deve utilizar o objeto `book` que foi criado na linha 22. Os parênteses vazios na linha 13 indicam que a função-membro não requer dados adicionais para realizar sua tarefa. (Na Seção 3.5, você verá como passar dados para uma função-membro.) Quando a função continua executando na linha 24, o que indica que suas tarefas com sucesso. Esse é o final do programa.

#### Diagrama de classes UML para a classe `GradeBook`

A partir da Seção 1.17, lembre-se de que a UML é uma linguagem gráfica utilizada por programadores para representar sistemas orientados a objetos de maneira padronizada. Na UML, cada classe é modelada em um diagrama de classes como um retângulo com três compartimentos. A Figura 3.2 apresenta um diagrama UML para a classe `GradeBook`. A Figura 3.1. O compartimento

superior contém o nome da classe, que é `GradeBook`. O compartimento central contém os atributos da classe. A Figura 3.2 não tem nenhum atributo. (A Seção 3.6 apresenta um exemplo de classe que tem um atributo.) O compartimento inferior contém as operações da classe que correspondem às funções-membro em C++. A UML mostra as operações listando o nome da operação seguido por um conjunto de parênteses. A classe `GradeBook` tem uma operação `displayMessage`. Então o compartimento inferior da Figura 3.2 lista uma operação com esse nome. A função-membro não requer informações adicionais para realizar suas tarefas, então os parênteses estão vazios, da mesma forma que estão no cabeçalho da função-membro na linha 13 da Figura 3.1. O símbolo de adição (+) de operação indica que `displayMessage` é uma operação pública na UML (isto é, uma função). Embora utilizamos freqüentemente diagramas de classes UML para resumir os atributos e operações de classe.

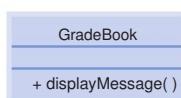


Figura 3.2 Diagrama de classes UML indicando que a classe GradeBook tem uma operação pública displayMessage().

### 3.5 Definindo uma função-membro com um parâmetro

Em nossa analogia do carro da Seção 3.2, mencionamos que pressionar o acelerador de um carro envia uma mensagem para ele uma tarefa — fazer o carro andar mais rapidamente. Mas quanto o carro deve acelerar? Como você sabe, quanto mais pressionar

mais a gasolina é usada. Essas mensagens adicionais são conhecidas como parâmetros, que fornecem informações adicionais que ajudam a executar a tarefa. As mensagens adicionais são enviadas para a função-membro para realizá-la com sucesso. De maneira semelhante, uma função-membro pode exigir um ou mais parâmetros que representam dados adicionais de que ela precisa para realizar sua tarefa. Uma chamada de função fornece valores — chamados de argumentos — para cada um dos parâmetros da função. Por exemplo, para fazer um depósito em uma conta bancária, suponha que uma função-membro deposit de uma classe contasse especificue um parâmetro que representa a quantia do depósito. Quando a função-membro é chamada, o valor de um argumento que representa a quantia de depósito é copiado para o parâmetro da função-membro. A função-membro então adiciona essa quantia ao saldo da conta.

Definindo e testando a classe GradeBook

Nosso próximo exemplo (Figura 3.3) redefine a classe 14–23) com uma função-membro displayMessage(linhas 18–22) que exibe o nome do curso como parte da mensagem de boas-vindas. Mesmo que a função-membro parâmetro courseName (na linha 18) que representa o nome do curso a ser enviado para a saída.

Antes de discutir os novos recursos da classe, vamos como a nova classe é definida (linhas 26–40). A linha 28 cria uma variável de tipo chamada nameOfCourse que será utilizada para armazenar o nome do curso inserido pelo usuário. Uma variável de tipo string representa uma string de caracteres. De acordo com "A C++ Programming" Uma string é, de fato, um objeto da classe C++ Standard Library. Essa classe é tipicamente definida em <iostream> e o nome string , como cout , pertence ao namespace . Para permitir que a linha 28 compile, a linha 9 inclui o arquivo de cabeçalho <iostream>. Observe que a declaração na linha 10 permite simplesmente substituir a linha 28 em vez de . Por enquanto, você pode pensar em strings como variáveis de outros tipos. Você aprenderá as capacidades adicionais de string na Seção 3.10.

A linha 29 cria um objeto da classe GradeBook. A linha 32 exibe um prompt que pede para usuário inserir o nome de um curso. A linha 33 lê o nome do usuário e armazena-o utilizando a função de biblioteca para realizar a entrada. Antes de explicarmos essa linha de código, vamos explicar por que simplesmente não podemos escrever

cin >> nameOfCourse;

para obter o nome do curso. Em nosso exemplo de execução de programa, utilizemos o nome do curso

"Introduction to C++ Programming" que contém múltiplas palavras. (Lembre-se de que destacamos a entrada fornecida pelo usuário em negrito.) Quando cin é utilizado com o operador de extração de fluxo, ele lê os caracteres até o primeiro caractere de espaço em branco ser alcançado. Portanto, somente este caractere é lido pela instrução precedente. O restante do nome do curso teria de ser lido por operações de entrada subsequentes.

Nesse exemplo, gostaríamos que o usuário digitasse o nome completo para que o programa pudesse armazenar o nome inteiro de um curso em uma variável chamada de função cin, nameOfCourse na linha 33 lê caracteres (incluindo os caracteres de espaço em branco que separam as palavras na entrada) do objeto de fluxo de entrada (padrão é o teclado) até o caractere nova linha ser encontrado, colocando os caracteres na variável nameOfCourse e descarta o caractere de nova linha. Observe que, dentro de um programa, um caractere de nova linha é inserido no fluxo de entrada. Note também que o deve ser declarado no programa

para a linha 28 funcionar corretamente. O argumento nameOfCourse em parênteses é o argumento que é passado para a função displayMessage para que ela possa realizar sua tarefa. Quando o valor é fornecido, torna-se o valor do parâmetro da função-membro displayMessage na linha 18. Ao executar esse programa, note que a saída da função-membro displayMessage gerada como parte da mensagem de boas-vindas do nome do curso que você digitou (em nosso exemplo de execução "Introduction to C++ Programming")

Mais sobre argumentos e parâmetros

Para especificar que uma função requer dados para realizar sua tarefa, você coloca informações adicionais na função que está localizada entre os parênteses depois do nome de função. A lista de parâmetros pode conter qualquer número d

```

1 // Figura 3.3: fig03_03.cpp
2 // Define a classe GradeBook com uma função-membro que aceita um parâmetro;
3 // Cria um objeto GradeBook e chama sua função-membro displayMessage.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <string> // o programa utiliza classe de string padrão C++
10 using std::string;
11 using std::getline;
12
13 // definição da classe GradeBook
14 class GradeBook
15 {
16 public :
17 // função que exibe uma mensagem de boas-vindas ao usuário do GradeBook
18 void displayMessage(string courseName)
19 {
20 cout << "Welcome to the grade book for\n" << courseName << endl;
21 << endl;
22 } // fim da função displayMessage
23 }; // fim da classe GradeBook
24
25 // a função main inicia a execução do programa
26 int main()
27 {
28 string nameOfCourse; // strings de caracteres para armazenar o nome do curso
29 GradeBook myGradeBook;// cria um objeto GradeBook chamado myGradeBook
30
31 // prompt para entrada do nome do curso
32 cout << "Please enter the course name:" << endl;
33 getline(cin, nameOfCourse); // lê o nome de um curso com espaços em branco
34 cout << endl; // gera saída de uma linha em branco
35
36 // chama a função displayMessage de myGradeBook
37 // e passa nameOfCourse como um argumento
38 myGradeBook.displayMessage(nameOfCourse);
39 return 0; // indica terminação bem-sucedida
40 } // fim de main

```

Please enter the course name:  
CS101 Introduction to C++ Programming

Welcome to the grade book for  
CS101 Introduction to C++ Programming!

**Figura 3.3** Definindo a classe GradeBook com uma função-membro que aceita um parâmetro.

metros, incluindo nenhum (representado por parênteses vazios como na Figura 3.1, linha 13) para indicar que uma função não tem nenhum parâmetro. A lista de parâmetros da função `displayMessage` (Figura 3.3, linha 18) declara que a função requer um parâmetro. Todo parâmetro deve especificar um tipo e um identificador. Neste caso, os nomes não indicam que a função-membro `displayMessage` requer uma string para realizar sua tarefa. O corpo da função-membro utiliza o parâmetro `courseName` para acessar o valor que é passado para a função na chamada de `displayMessage`. As linhas 12, 13 e 14 item o valor do parâmetro `courseName` como parte da mensagem de boas-vindas. Observe que o nome da variável de argumento (linha 18) pode ter ou não o mesmo nome da variável de argumento (linha 38) — você aprenderá por que isso acontece no Capítulo 6, “Finalizando a introdução à recursão”.

Uma função pode especificar múltiplos parâmetros separando cada parâmetro do próximo com uma vírgula (veremos um exemplo em figura 6.4–6.5). O número e a ordem de argumentos em uma chamada de função devem corresponder ao número e à ordem de parâmetros na lista de parâmetros do cabeçalho da função-membro chamada. Além disso, os tipos de argumento na chamada de função devem corresponder aos tipos dos parâmetros correspondentes no cabeçalho de função. (Como você aprenderá nos capítulos subsequentes, o tipo de um argumento e o tipo do seu parâmetro correspondente nem sempre precisam ser idênticos, mas devem ser ‘consistentes’.) No nosso exemplo, a argumento chamada de função-membro `courseName` corresponde exatamente ao parâmetro definido na função-membro `courseName`.



#### Erro comum de programação 3.4

Colocar um ponto-e-vírgula após o parêntese direito que envolve a lista de parâmetros de uma definição de função é um erro de sintaxe.



#### Erro comum de programação 3.5

Definir um parâmetro de função novamente como uma variável local na função é um erro de compilação.



#### Boa prática de programação 3.1

Para evitar ambigüidade, não utilize os mesmos nomes para os argumentos passados para uma função e os parâmetros correspondentes na definição de função.



#### Boa prática de programação 3.2

Escolher nomes significativos para funções e parâmetros torna os programas mais legíveis e ajuda a evitar utilização excessiva de comentários.

Diagrama da classe UML atualizado para a classe GradeBook

O diagrama da classe UML da Figura 3.4 modela a classe GradeBook da Figura 3.3. Como a classe GradeBook definida na Figura 3.1, essa classe GradeBook contém a função-membro `displayMessage`. Entretanto, essa versão descreve um parâmetro. A UML modela um parâmetro listando o nome de parâmetro, seguido por dois-pontos e o tipo de parâmetro entre os parênteses que se seguem ao nome da operação. A UML tem seus próprios tipos de dados semelhantes aos da C++. A UML é independente do C++. Por exemplo, o tipo UML `String` responde ao tipo C++ `std::string`. A função-membro `displayMessage` da classe GradeBook (Figura 3.3; linhas 18–22) tem um parâmetro `courseName`, então a Figura 3.4 usa o nome `courseName : String` entre os parênteses que se seguem ao nome da operação. Observe que essa versão da classe não tem nenhum membro de dados.

### 3.6 Membros de dados, funções set e funções get

No Capítulo 2, declararamos todas as variáveis de um programa. As variáveis declaradas em um corpo da definição de função são conhecidas como **variáveis locais** e só podem ser utilizadas a partir da linha de sua declaração na função até a chave direita de fechamento imediatamente seguinte à definição de função. Uma variável local deve ser declarada antes de poder ser utilizada em uma função. Uma variável local não pode ser acessada fora da função em que é declarada. Quando uma função termina, os valores de suas variáveis locais são perdidos. (Você verá uma exceção a isso no Capítulo 6 quando discutirmos as variáveis locais partindo da Seção 3.2, lembre-se de que um objeto tem atributos que são portados com esse objeto quando ele é utilizado em um programa.) Esses atributos existem por toda a vida do objeto.

Uma classe normalmente consiste em uma ou mais funções-membro que manipulam os atributos que pertencem a um objeto particular da classe. Os atributos são representados como variáveis em uma definição de classe. Essas variáveis são chamadas de

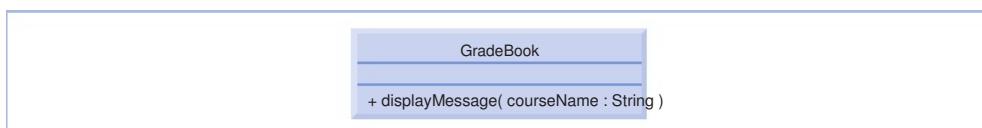


Figura 3.4 Diagrama de classes UML indicando que a classe GradeBook tem uma operação `displayMessage` com um parâmetro `courseName` do tipo UMLString.

de dados e são declaradas dentro de uma definição de classe, mas fora dos corpos das definições de função-membro da classe. O exemplo de código a seguir demonstra uma classe que contém um membro de dados para representar o nome do curso de um projeto.

A classe GradeBook tem um membro de dados, uma função set e uma função get.

No nosso próximo exemplo, a classe GradeBook (Figura 3.5) mantém o nome do curso como um membro de dados para que ele possa ser utilizado ou modificado a qualquer hora durante a execução de um programa. A classe GradeBook contém as funções-membro setCourseName e displayMessage. A função-membro setCourseName armazena um nome do curso em um membro de dados GradeBook — a função-membro getCourseName obtém desse membro de dados o nome do curso. A função-membro displayMessage — que agora não especifica nenhum parâmetro — ainda exibe uma mensagem de boas-vindas que inclui o nome do curso. Entretanto, como você verá, a função-membro agora obtém o nome do curso chamando outra função na mesma classe — CourseName.

```

1 // Figura 3.5: fig03_05.cpp
2 // Define a classe GradeBook que contém um membro de dados courseName
3 // e funções-membro para configurar e obter seu valor;
4 // Cria e manipula um objeto GradeBook com essas funções.
5 #include <iostream>
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 #include <string> // o programa utiliza classe de string padrão C++
11 using std::string;
12 using std::getline;
13
14 // definição da classe GradeBook
15 class GradeBook
16 {
17 public:
18 // função que configura o nome do curso
19 void setCourseName(string name)
20 {
21 courseName = name; // armazena o nome do curso no objeto
22 } // fim da função setCourseName
23
24 // função que obtém o nome do curso
25 string getCourseName()
26 {
27 return courseName; // retorna o courseName do objeto
28 } // fim da função getCourseName
29
30 // função que exibe uma mensagem de boas-vindas
31 void displayMessage()
32 {
33 // essa instrução chama getCourseName para obter o
34 // nome do curso que esse GradeBook representa
35 cout << "Welcome to the grade book for\n" << getCourseName() << endl;
36 } // fim da função displayMessage
37 private:
38 string courseName; // nome do curso para esse GradeBook
39 }; // fim da classe GradeBook
40
41

```

Figura 3.5 Definindo e testando a classe GradeBook com um membro de dados e as funções set e get.

(continua)

```

42 // a função main inicia a execução do programa
43 int main()
44 {
45 string nameOfCourse; // strings de caracteres para armazenar o nome do curso
46 GradeBook myGradeBook;// cria um objeto GradeBook chamado myGradeBook
47
48 // exibe valor inicial de courseName
49 cout << "Initial course name is: " << myGradeBook.getCourseName()
50 << endl;
51
52 // solicita, insere e configura o nome do curso
53 cout << "\nPlease enter the course name." << endl;
54 getline(cin, nameOfCourse); // lê o nome de um curso com espaços em branco
55 myGradeBook.setCourseName(nameOfCourse);configura o nome do curso
56
57 cout << endl; // gera saída de uma linha em branco
58 myGradeBook.displayMessage(); // exibe a mensagem com o novo nome do curso
59 return 0; // indica terminação bem-sucedida
60 } // fim de main

```

Initial course name is:

Please enter the course name:  
CS101 Introduction to C++ Programming

Welcome to the grade book for  
CS101 Introduction to C++ Programming!

Figura 3.5 Definindo e testando a classe GradeBook com um membro de dados e as funções get

(continuação)



### Boa prática de programação 3.3

Coloque uma linha em branco entre definições de função-membro para aprimorar a legibilidade do programa.

Um instrutor típico dá mais de um curso, cada um com seu próprio nome. A linha 39 declara o tipo string . Como a variável é declarada na definição de classe (linhas 15–40), mas fora dos corpos das definições de função-membro classe (linhas 19–22, 25–28 e 31–37), a linha 39 é uma declaração para um membro de dados. Cada instância (isto é, objeto) da GradeBook contém uma cópia de cada um dos membros de dados da classe. Por exemplo, se申两 objects tem sua própria cópia de name (uma por objeto), como veremos no exemplo da Figura 3.7, um benefício de tornar um membro de dados é que todas as funções-membro da classe podem manipular quaisquer membros de dados que aparecerem na definição de classe nesse caso,

Especificadores de acesso public e private

A maioria das declarações de membro de dados aparece depois do rótulo de especificador de acesso public ou private. A palavra-chave private é um especificador de acesso. Variáveis ou funções declaradas depois de private são acessíveis somente a funções-membro da classe para a qual elas são declaradas. O membro de dados name pode ser utilizado somente em funções-membro da classe GradeBook (de cada objeto) da classe GradeBook. O membro de dados name, porque é private, não pode ser acessado por funções fora da classe (como a função main do programa). Quando tentarmos acessar o membro de dados name de dentro de uma função externa ao escopo da classe GradeBook, obtendremos uma mensagem semelhante a

cannot access private member declared in class 'GradeBook'



### Observação de engenharia de software 3.1

Como regra geral, os membros de dados devem ser declarados como membros-membro devem ser declarados que é apropriado declarar certas funções-membro que precisam ser acessadas somente por outras funções-membro da classe.)



### Erro comum de programação 3.6

Uma tentativa de uma função, que não seja um membro de uma classe, pode ser feita como veremos no Capítulo 10), de acessar um membro dessa classe é um erro de compilação.

O acesso-padrão de membros de classe é todos os membros depois do cabeçalho de classe e antes do primeiro especificador de acesso. Os especificadores de acesso podem ser repetidos, mas isso é desnecessário e pode ser confuso.



### Boa prática de programação 3.4

Apesar de os especificadores de acesso poderem ser repetidos e combinados, liste todos os membros de uma classe primeiro em um grupo e, então, liste todos os membros de outros grupos. Isso focaliza a atenção do cliente na interface da classe, em vez de na implementação da classe.



### Boa prática de programação 3.5

Se escolher listar os membros primeiro em uma definição de classe, utilize o especificador de acesso apesar de ele ser assumido por padrão. Isso melhora a clareza do programa.

Declarar membros de dados com o especificador de acesso `private` é a maneira correta de declarar membros de dados. Quando um programa cria (instancia) um objeto da classe membro de dados, esse membro é encapsulado (ocultado) no objeto e pode ser acessado apenas por funções-membro da classe do objeto. As funções-membro `getCourseName()`, `setCourseName()` e `payMessage()` manipulam o membro de dados `courseName` diretamente (uma operação que poderia fazer isso se necessário).



### Observação de engenharia de software 3.2

Aprenderemos no Capítulo 10, “Classes: Um exame mais profundo, parte 2”, que funções e classes declaradas por uma classe `friend` podem acessar os membros da classe.



### Dica de prevenção de erro 3.1

Transformar os membros de dados de uma classe em funções-membro da classe facilita a depuração porque os problemas com manipulações de dados são localizados para as funções-membro da classe ou para os

As funções-membro `getCourseName()` e `setCourseName()` (definida nas linhas 19–22) não retorna nenhum dado quando completa sua tarefa, então seu tipo de retorno é `void`. A função-membro recebe um parâmetro que representa o nome do curso que será passado para ele como um argumento (como veremos na linha 21). Atribui o membro de dados `courseName` a esse argumento. Nesse exemplo, `tCourseName` não tenta validar o nome do curso — isto é, a função não verifica se o nome do curso obedece a algum formato particular ou segue alguma outra regra relacionada com o que seria um nome do curso ‘válido’. Suponha, por exemplo, que uma universidade possa imprimir listas de alunos contendo nomes de curso de apenas 25 caracteres ou menos. Nesse caso, talvez queiramos que `GradeBook` assegure que seu membro de dados `courseName` contenha mais de 25 caracteres. Discutimos as técnicas básicas de validação na Seção 3.10.

A função-membro `getCourseName()` (definida nas linhas 25–28) retorna um objeto `GradeBook` particular. A função-membro tem uma lista de parâmetros vazia, portanto não requer dados adicionais para realizar sua tarefa. A função especifica que ela retorna `GradeBook`. Quando uma função que especifica um tipo de retorno é chamada para completar sua tarefa, a função retorna um resultado para sua função chamadora. Por exemplo, ao utilizar a função `scanf()` e solicitar o saldo da sua conta, você espera o ATM devolver um valor que representa seu saldo. De modo semelhante, quando a instrução chama a função-membro `getCourseName()` para obter um objeto `GradeBook`, a instrução espera receber o nome de curso do (nesse caso, `tString`, como especificado pelo tipo de retorno da função). Se a função retornar um resultado que não é de tipo quadrado de

seu argumento, a instrução

retorna a partir da função e inicializa a variável com o valor. Se você tiver uma função que retorna o maior de três argumentos de inteiro, a instrução

```
int biggest = maximum(27, 114, 51);
return biggest;
```



### Erro comum de programação 3.7

Esquecer de retornar um valor de uma função que supostamente deve retornar um valor é um erro de compilação.

Observe que ambas as instruções nas linhas 21 e 27 utilizam a linha 30, embora ela não tenha sido declarada em nenhuma das funções-membro. Podemos usar funções-membro da classe porque `courseName` é um membro de dados da classe. Observe também que a ordem em que funções-membro são definidas não determina quando elas são executadas em tempo de execução. Então a função `name` deveria ser definida antes da função `message`.

#### Função-membro `displayMessage`

A função-membro `displayMessage` (linhas 31–37) não retorna dados quando ela completa sua tarefa. Isso é porque seu tipo de retorno é `void`. A função não recebe parâmetros, então sua lista de parâmetros está vazia. As linhas 35–36 geram saída de uma mensagem de boas-vindas que inclui o valor do membro `courseName`. A linha 35 chama a função `getName` para obter o valor de `courseName`. Observe que a função `displayMessage` também poderia acessar diretamente o membro de dados `courseName`.

**funções-membros sempre retornam valores** — isso é fundamental. Explicaremos em breve a razão pela qual escolhemos chamar a

#### Testando a classe `GradeBook`

A função `main` (linhas 43–60) cria um objeto da classe e utiliza cada uma de suas funções-membro. A linha 46 cria um objeto `GradeBook`, chamado `myGradeBook`. As linhas 49–50 exibem o nome inicial de curso chamado de função-membro `courseName` do objeto. Observe que a primeira linha da saída não mostra um nome de curso porque o objeto `myGradeBook` não tem dados `courseName` (uma `String`) — está inicialmente vazio — por padrão, o valor inicial é chamado `vazio`, isto é, uma string que não contém nenhum caractere. Nada aparece na tela quando uma string vazia é exibida.

A linha 53 exibe um prompt que pede para o usuário inserir o nome de um curso. A variável `courseName` (definida na linha 45) é configurada como o nome do curso inserido pelo usuário, que é obtido na linha 44 chamada à função `CourseName`. A linha 55 chama a função `displayMessage` do objeto `myGradeBook` fornecendo `courseName` como o argumento da função. Quando a função é chamada, o valor do argumento é copiado para o parâmetro `courseName` (linhas 19–22). Então o valor do parâmetro é atribuído ao membro de dados. A linha 57 pula uma linha na saída; então a linha 58 chama a função `displayMessage` do objeto `myGradeBook` para exibir a mensagem de boas-vindas contendo o nome do curso.

#### Engenharia de software com as funções `set` e `get`

Os membros de `dados` de uma classe só podem ser manipulados por funções-membro dessa classe (e por ‘amigos’ da classe, como veremos no Capítulo 10, “Classes: um exame mais profundo, parte 2”). Portanto, um cliente de um objeto — isto é, qualquer classe ou função que chame as funções-membro do objeto de fora do objeto — não pode manipular os membros. As funções-membro só podem ser usadas para solicitar os serviços dessa classe para objetos particulares dela própria. Essa é a razão principal das instruções na função `displayMessage` (linhas 43–60) chamarem as funções-membro `getName` e `displayMessage` em um objeto `GradeBook`. As classes costumam fornecer funções-membro `set` e `get` para permitir a clientes `dados` — isto é, atribuir valores a `dados` (ou obter valores de) membros de `dados`. Usar essas funções-membro não presta garantias, mas essa convenção de atribuição de nomes é comum. Nesse exemplo, a função-membro `getCourseName` é chamada para obter o nome de curso (que é o valor do membro de dados `courseName`). Observe que as funções `set` também são às vezes chamadas de `mutadoras` (porque modificam valores), e as funções `get` às vezes chamadas de `acessoras` (porque acessam valores).

Lembre-se de que declarar membros de dados com o especificador `private` impõe uma restrição de acesso que está tentando modificar ou obter os dados de um objeto, mas o cliente não sabe como o objeto realiza essas operações. Em alguns casos, a classe pode representar internamente uma parte de dados de uma maneira, mas expõe esses dados para clientes de maneira diferente. Por exemplo, suponha que uma classe represente a hora do dia como um membro de dados que armazena o número de segundos desde a meia-noite. Entretanto, quando um cliente chama a função-membro `getTime`, o objeto poderia retornar o tempo com horas, minutos e segundos, ou `mm:ss`. De modo semelhante, suponha que a classe fornece uma função-membro `Time` que aceita um parâmetro no formato `MM:SS`. Utilizando as

explicações estruturais apresentadas no Capítulo 2, se `Time` converter isso em um número de segundos, esse número é armazenado em seu membro de dados. Ela também pode verificar se o valor que ela recebe representa uma hora válida (por exemplo, “45” é válida, mas “85:70” não é). As funções `set` permitem a um cliente interagir com um objeto, mas os dados do objeto permanecem seguramente encapsulados (isto é, ocultos) no próprio objeto.

As funções-membro de uma classe também devem ser utilizadas por outras funções-membro dentro da classe para manipular os dados da classe, embora essas funções-membro possam manipular os dados diretamente. Na Figura 3.5, as funções-membro `setCourseName` e `getCourseName` são funções-membro portanto acessíveis aos clientes da classe, bem como à própria classe. A função-membro `displayMessage` chama a função-membro `courseName` para obter o valor do membro de dados para propósitos de exibição, mas não que possa acessar diretamente — acessar um membro de dados via sua função-membro é uma classe melhor e mais robusta (isto é, uma classe mais fácil de manter e com menos probabilidade de parar c

funcionar). Se decidirmos alterar o membro de dados de uma maneira, a definição da classe não exigirá modificação — apenas os corpos das funções que manipulam diretamente o membro de dados precisam mudar. Por exemplo, suponha que decidíssemos representar o nome do curso como dois membros de dados (`courseName` e `courseTitle`) (por exemplo, "Introduction to C++ Programming"). A função-membro `displayMessage()` ainda pode emitir uma única chamada para a função-membro `getCourseName()` com o fim de obter o nome inteiro do curso exibido como parte da mensagem de boas-vindas. Nesse caso, `getCourseName()` precisaria construir e retornar `courseName` e `courseTitle`. A função-membro `displayMessage()` continuaria a exibir o título completo (`Introduction to C++ Programming`) porque ela não é afetada pela alteração nos membros de dados da classe. Os benefícios da separação de responsabilidades de uma classe se tornarão claros quando discutirmos validação na Seção 3.10.



### Boa prática de programação 3.6

Tente sempre localizar os efeitos de alterações em membros de dados de uma classe acessando e manipulando os membros de dados por meio de suas funções. Alterações no nome de um membro de dados ou tipo de dados utilizado para armazenar um membro de dados afetam então apenas as funções correspondentes, mas não os chamadores dessas funções.



### Observação de engenharia de software 3.3

É importante escrever programas que sejam comprehensíveis e fáceis de manter. A mudança é a regra em vez da exceção. Os programadores devem antecipar que seu código será modificado.



### Observação de engenharia de software 3.4

O designer de classes não precisa fornecer ~~especificações~~ para item de dados; essas capacidades devem ser fornecidas somente quando apropriado. Se um serviço é útil para o código-cliente, esse serviço deve, em geral, ser fornecido na interface `public` da classe.

O diagrama de classes UML GradeBook com um membro de dados e as funções `set` e `get`.

A Figura 3.6 contém um diagrama de classes UML atualizado da Figura 3.5. Esse diagrama modela o membro de dados `courseName` da classe `GradeBook` como um atributo no compartimento do meio da classe. A UML representa os membros de dados como atributos listando o nome do atributo, seguido por um caractere de dois-pontos e pelo tipo do atributo. O tipo do atributo é `String`, que corresponde à `string` em C++. O membro de dados em C++, então

é equivalente ao especificado na classe. A classe GradeBook tem três funções implementadas no diagrama de classes: lista três operações no terceiro compartimento. Lembre-se de que cada operação indica que a operação `set` no C++. A operação `getCourseName()` tem um parâmetro chamado `name`. A UML indica o tipo de retorno de uma operação colocando dois-pontos e o tipo de retorno depois dos parênteses que se seguem ao nome da operação. A função `getCourseName()` da classe `GradeBook` (Figura 3.5) tem um tipo de retorno `CString`, portanto o diagrama de classes mostra um tipo de retorno `String` na UML. Observe que as operações `setCourseName()` e `displayMessage()` não retornam valores (isto é, elas retornam `void`), então o diagrama de classes UML não especifica um tipo de retorno depois dos parênteses dessas operações. A UML não void como o C++ utiliza quando uma função não retorna um valor.

## 3.7 Inicializando objetos com construtores

Como mencionado na Seção 3.6, quando um objeto (Figura 3.5) é criado, seu membro de dados `courseName` é inicializado, por padrão, como string vazia. E se você quiser fornecer o nome de um curso quando criar um objeto da classe que você declara pode fornecer uma função que pode ser utilizada para inicializar um objeto de uma classe quando o objeto é criado. Um construtor é uma função-membro especial que deve ser definida com o mesmo nome da classe, de modo que o

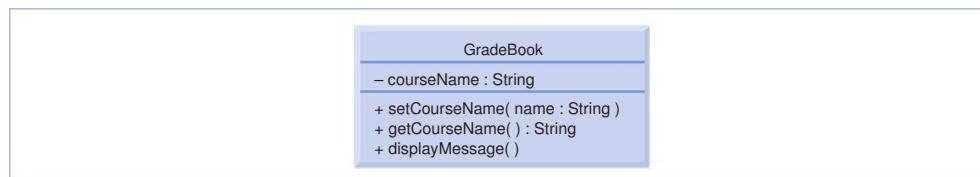


Figura 3.6 Diagrama de classes UML para a classe `GradeBook` com um atributo privado `courseName`, operações públicas `setCourseName()`, `getCourseName()` e `displayMessage()`.

lador possa diferenciá-lo de outras funções-membro da classe. Uma diferença importante entre construtores e outras funções é que os construtores não podem retornar valores, então eles não podem especificar o valor de retorno (sem mesmo dizer que os construtores são deploráveis). O termo ‘construtor’ é, freqüentemente, abreviado como ‘ctor’ na literatura — mas preferimos não utilizar essa abreviação.

O C++ requer uma chamada de construtor para cada objeto que é criado, o que ajuda a assegurar que o objeto é inicializado adequadamente antes de ser utilizado em um programa — a chamada de construtor ocorre implicitamente quando o objeto é criado. Em qualquer classe que não inclua um construtor explicitamente, o compilador fornece um construtor sem parâmetros. Por exemplo, quando a linha 46 da Figura 3.5 cria o objeto padão é chamado, porque a declaração de GradeBook não especifica nenhum argumento de construtor. O construtor-padrão fornecido pelo compilador cria um objeto GradeBook sem fornecer nenhum valor inicial para os membros de dados. Para todos os dados que são objetos de outras classes, o construtor-padrão chama o construtor-padrão de cada membro de dados para assegurar que o membro de dados seja inicializado de maneira adequada. De fato, essa é a razão pela qual a memória (máquina) da figura 3.5 foi inicializada como uma string vazia — o construtor-padrão da classe string é inicializada como a string vazia. Na

Série de 100 páginas, a Figura 3.7 nos mostra como inicializar múltiplos objetos da classe GradeBook (linhas 10–19). Nesse caso, o argumento `name` é passado para o construtor de GradeBook (linhas 17–20) e utilizado para inicializar `courseName`. A Figura 3.7 define uma classe modificada contendo um construtor com um parâmetro `string` que recebe o nome do curso inicial.

```

1 // Figura 3.7: fig03_07.cpp
2 // Instanciando múltiplos objetos da classe GradeBook e utilizando
3 // o construtor GradeBook para especificar o nome do curso
4 // quando cada objeto GradeBook é criado.
5 #include <iostream>
6 using std::cout;
7 using std::endl;
8
9 #include <string> // o programa utiliza classe de string padrão C++
10 using std::string;

11 // Definição da classe GradeBook
12 class GradeBook
13 {
14 {
15 public :
16 // o construtor inicializa courseName com a string fornecida como argumento
17 GradeBook(string name)
18 {
19 setCourseName(name); // chama a função set para inicializar courseName
20 } // fim do construtor GradeBook
21
22 // função para configurar o nome do curso
23 void setCourseName(string name)
24 {
25 courseName = name; // armazena o nome do curso no objeto
26 } // fim da função setCourseName
27
28 // função para obter o nome do curso

290 string getCourseName()
30 return courseName; // retorna courseName do objeto
31 } // fim da função getCourseName
32
33 // exibe uma mensagem de boas-vindas para o usuário GradeBook
34 void displayMessage()
35

```

Figura 3.7 Instanciando múltiplos objetos da classe GradeBook utilizando o construtor GradeBook para especificar o nome do curso quando é criado o objeto GradeBook  
(continua)

```

36 {
37 // chama getCourseName para obter o courseName
38 cout << "Welcome to the grade book for\n" << getCourseName()
39 << endl;
40 } // fim da função displayMessage
41 private :
42 string courseName; // nome do curso para esse GradeBook
43 }; // fim da classe GradeBook
44
45 // a função main inicia a execução do programa
46 int main()
47 {
48 // cria dois objetos GradeBook
49 GradeBook gradeBook1("CS101 Introduction to C++ Programming");
50 GradeBook gradeBook2("CS102 Data Structures in C++");
51
52 // exibe valor inicial de courseName para cada GradeBook
53 cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
54 << endl;
55 << endl;
56 return 0; // indica terminação bem-sucedida
57 } // fim de main

```

gradeBook1 created for course: CS101 Introduction to C++ Programming  
 gradeBook2 created for course: CS102 Data Structures in C++

**Figura 3.7** Instanciando múltiplos objetos da classe `GradeBook` utilizando o construtor `GradeBook` para especificar o nome do curso quando é criada o objeto `GradeBook`

#### Definindo um construtor

As linhas 17–20 da Figura 3.7 definem um construtor para `GradeBook`. O construtor tem o mesmo nome que sua classe, `GradeBook`. Um construtor especifica em sua lista de parâmetros os dados que ele requer para realizar sua tarefa. Quando você cria um novo objeto, coloca esses dados entre os parênteses que se seguem ao nome de objeto (como fizemos nas linhas 49–50). A linha 17 indica que o construtor `GradeBook` tem um parâmetro `string` chamado `courseName`. Observe que a linha 17 não especifica um tipo de retorno, porque os construtores não podem retornar valores (ou até mesmo

A linha 19 no corpo do construtor passa o parâmetro para o `setCourseName`, que atribui um valor para membro de `dados`. A função-membro `setCourseName` (linhas 23–26) simplesmente atribui seu parâmetro ao membro de `dados`. Entendendo talvez você se pergunte por que nos incomodamos tanto com essa chamada para a linha 19 — o construtor certamente poderia realizar a atribuição. Na Seção 3.10, modificaremos `setCourseName` para realizar a validação (assegurando que o argumento tem 25 ou menos caracteres de comprimento). Nesse ponto, os benefícios de chamar `setCourseName` partir do construtor se tornarão claros. Observe que tanto o construtor (linha 17) como a função `setCourseName` (linha 23) utilizam um parâmetro `courseName`. Pode utilizar os mesmos nomes de parâmetro em funções diferentes porque os parâmetros são locais a cada função; um não interfere no outro.

#### Testando a classe `GradeBook`

As linhas 46–57 da Figura 3.7 definem a classe `GradeBook` e demonstra a inicialização de objetos `GradeBook` utilizando um construtor. A linha 49 não inicializa um objeto `GradeBook`. Quando essa

linha executa, o código (`GradeBook` linhas 17–20) é chamado implicitamente por C++ “OS101 Introduction to C++ Programming” para inicializar o nome de curso. A linha 50 repete esse processo para o segundo objeto `gradeBook2`, dessa vez passando o argumento “CS102 Data Structures in C++” para inicializar o nome do curso. As linhas 53–54 utilizam a função-membro `getCourseName` de cada objeto para obter os nomes de curso e mostra que eles, de fato, foram inicializados quando os objetos foram criados. A saída confirma que cada objeto tem sua própria cópia de membro de dados `courseName`.

Duas maneiras de fornecer um construtor-padrão para uma classe

Qualquer construtor que não aceita argumentos é chamado construtor-padrão. Uma classe obtém um construtor-padrão em uma maneira:

- O compilador cria implicitamente um construtor-padrão em uma classe que não define um construtor. Esse construtor-padrão não inicializa os membros de dados da classe, mas chama o construtor-padrão para cada membro de dados que é um objeto de outra classe. [Uma variável não inicializada contém, em geral, um valor ‘lixo’ (por exemplo, uma variável inicializada que poderia ter sido 666, que provavelmente é um valor incorreto para essa variável na maioria dos programas).]

- O programador define explicitamente um construtor que não aceita argumentos. Tal construtor-padrão realizará a inicialização especificada pelo programador e chamará o construtor-padrão para cada membro de dados que for um objeto de outra classe.

Se o programador definir um construtor com argumentos, C++ não criará implicitamente um construtor-padrão para essa classe. Observe que para cada versão da classe, Figura 3.1, Figura 3.3 e Figura 3.5 o compilador definiu implicitamente um construtor-padrão.



#### Dica de prevenção de erro 3.2

A menos que nenhuma inicialização de membros de dados da classe seja necessária (quase nunca), forneça um construtor para assegurar que os membros de dados da classe sejam inicializados com valores significativos quando cada novo objeto da classe for criado.



#### Observação de engenharia de software 3.5

Os membros de dados podem ser inicializados em um construtor da classe ou seus valores podem ser configurados mais tarde depois que o objeto for criado. Entretanto, é uma boa prática de engenharia de software assegurar que um objeto seja completamente inicializado antes de o código-cliente invocar as funções-membro do objeto. Em geral, você não deve contar com o código-cliente para assegurar que um objeto seja inicializado adequadamente.

Adicionando o construtor ao diagrama de classes UML da [CaseBook](#)

O diagrama de classes UML da Figura 3.8 modela a Figura 3.7, que tem um construtor com um parâmetro tipo `string` (representado por `String` na UML). Assim como operações, a UML modela construtores no terceiro compartimento de uma classe em um diagrama de classes. Para distinguir entre um construtor e operações de uma classe, a UML coloca o prefixo ‘constructor’ entre aspas francesas (‘« e ») antes do nome do construtor. É comum listar o construtor da classe antes de outras operações no terceiro compartimento.

### 3.8 Colocando uma classe em um arquivo separado para reusabilidade

Desenvolvemos a classe até o ponto em que precisávamos, por enquanto de uma perspectiva de programação, então vamos considerar algumas questões de engenharia de software. Um dos benefícios de criar definições de classes é que, quando empregadas adequadamente, nossas classes podem ser reutilizadas por programadores — potencialmente em todo o mundo. Por exemplo, podemos reutilizar o tipo `String` da C++ Standard Library em qualquer programa C++ por meio da inclusão do arquivo de cabeçalho no programa (e, como veremos, por meio da capacidade de vincular ao código-objeto da biblioteca).

Infelizmente, os programadores que desejam utilizar essa prática simplesmente incluir o arquivo da Figura 3.7 em outro programa. Como você aprendeu no [Capítulo 2](#), a definição de cada programa deve ter exatamente uma função principal. Outros programadores incluirem o código da Figura 3.7, obterão uma bagagem extra — nossa função `main` — e os programas terão, portanto, duas funções. Seles tentarem compilar os programas, o compilador indicará um erro porque, novamente, cada programa pode ter apenas uma função. Tente compilar um programa com duas funções e o Microsoft Visual C++ .NET produz o erro

`error C2084: function 'int main(void)' already has a body`

quando o compilador tentar compilar a segunda definição contra. De modo semelhante, o compilador GNU C++ produz o erro

`redefinition of 'int main()'`

Esses erros indicam que um programa só pode ter uma definição de classe por arquivo, e que a definição de classe não pode ser dividida entre arquivos.

Arquivos de cabeçalho

Cada um dos exemplos anteriores no capítulo consiste em um único arquivo de código-fonte. No entanto, também é comum dividir o código-fonte em arquivos separados. Ao construir um programa C++ orientado a objetos, é comum definir o código-fonte reutilizável (como uma classe) em um arquivo que, por convenção, tem uma extensão de nome de arquivo — conhecido como [arquivo de cabeçalho](#). Os programas utilizam as diretivas de pré-processador para incluir arquivos de

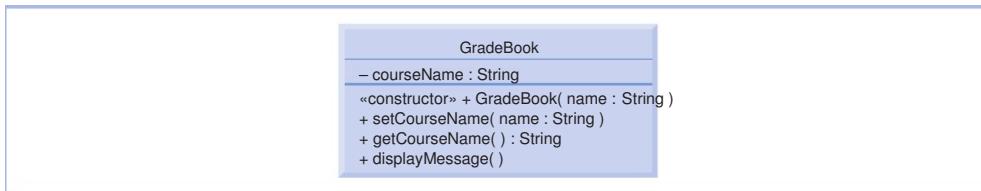


Figura 3.8 Diagrama de classes UML indicando que a classe GradeBook tem um construtor com um parâmetro name do tipo UMLString .

cabeçalho e tirar proveito de componentes de softwares reutilizáveis da C++ Standard Library e os tipos definidos pelo usuário, como GradeBook . No próximo exemplo, preparamos o código da Figura 3.7 (ver o Diagrama UML na Figura 3.8) no arquivo GradeBook.h (Figura 3.9). Ao examinar o arquivo de cabeçalho na Figura 3.9, note que ele contém todas as definições de classe (Figura 3.8) e as linhas 3–8, o que permite que a classe GradeBook utilize a classe std::string . A função main que utiliza a classe GradeBook é definida no arquivo de código-fonte (Figura 3.10) nas linhas 10–21. Para ajudá-lo a se preparar para os programas maiores que você encontrará mais adiante neste livro e na indústria, costumamos utilizar um arquivo de código-fonte se contendo a função main para testar nossas classes (isso é chamado de Você logo aprenderá como um arquivo de código-fonte compõe utilizar a definição de classe localizada em um arquivo de cabeçalho para criar objetos de uma classe).

```

1 // Figura 3.9: GradeBook.h
2 // Definição de classe GradeBook em um arquivo main separado.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string> // a classe GradeBook utiliza a classe de string padrão C++
8 using std::string;
9
10 // definição da classe GradeBook
11 class GradeBook
12 {
13 public :
14 // o construtor inicializa courseName com a string fornecida como argumento
15 GradeBook(string name)
16 {
17 setCourseName(name); // chama a função set para inicializar courseName
18 } // fim do construtor GradeBook
19
20 // função para configurar o nome do curso
21 void setCourseName(string name)
22 {
23 courseName = name; // armazena o nome do curso no objeto
24 } // fim da função setCourseName
25
26 // função para retornar nome do curso
27 string getCourseName()
28 {
29 return courseName;/ retorna courseName do objeto
30 } // fim da função getCourseName
31
32 // exibe uma mensagem de boas-vindas para o usuário GradeBook
33 void displayMessage()

```

Figura 3.9 A definição da classe GradeBook

(continua)

```

34 {
35 // chama getCourseName para obter o courseName
36 cout << "Welcome to the grade book for\n" << getCourseName()
37 << endl;
38 } // fim da função displayMessage
39 private :
40 string courseName; // nome do curso para esse GradeBook
41 }; // fim da classe GradeBook

```

Figura 3.9 A definição da classe GradeBook

(continuação)

Incluindo um arquivo de cabeçalho que contém uma classe definida pelo usuário ([Figura 3.9](#)) não pode ser utilizado para iniciar a execução de programa, porque ele não contém uma função. Se você tentar compilar [enquanto](#) por conta própria para criar um aplicativo executável, o Microsoft Visual C++ .NET produzirá a mensagem de erro de linker:

error LNK2019: unresolved external symbol \_main referenced in function \_mainCRTStartup

Executar GNU C++ no Linux produz uma mensagem de erro de linker contendo:

undefined reference to 'main'

Esse erro indica que o linker não pôde localizar [programa](#). Para testar a classe definida na Figura 3.9, você deve escrever um arquivo de código-fonte separado ([enquanto Figura 3.10](#)) que instancia e utiliza os objetos da classe.

A partir da discussão da Seção 3.4, lembre-se de que, enquanto o compilador sabe o que são tipos de dados fundamentais com o compilador não sabe o que é porque ele é um tipo definido pelo usuário. De fato, o compilador nem mesmo conhece as classes na C++ Standard Library. Para ajudar a entender como utilizar uma classe, devemos fornecer explicitamente o compilador com a definição da classe — essa é a razão pela qual, por exemplo, [para otimizar a aplicação](#) incluir o arquivo de cabeçalho `GradeBook.h`. Isso permite ao compilador determinar a quantidade de memória que deve reservar para cada objeto da classe e assegura que um programa chame corretamente as funções-membro da classe.

```

1 // Figura 3.10: fig03_10.cpp
2 // Incluindo a classe GradeBook a partir do arquivo Gradebook.h para uso em main.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // inclui a definição de classe GradeBook
8
9 // a função main inicia a execução do programa
10 int main()
11 {
12 // cria dois objetos GradeBook
13 GradeBook gradeBook1("CS101 Introduction to C++ Programming");
14 GradeBook gradeBook2("CS102 Data Structures in C++");
15
16 // exibe valor inicial de courseName para cada GradeBook
17 cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
18 << endl;
19 cout << "gradeBook2 created for course: " << gradeBook2.getCourseName()
20 << endl;
21 } // fim de main

```

```

gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++

```

Figura 3.10 Incluindo a classe GradeBook a partir do arquivo GradeBook.h para utilizar em main

Para criar os objetos `GradeBook`, nas linhas 13–14 da Figura 3.10, o compilador deve saber o tamanho de um objeto `GradeBook`. Enquanto os objetos contêm conceitualmente membros de dados e funções-membro, os objetos C++, em geral contêm apenas dados. O compilador cria apenas uma cópia das funções-membro da classe e compartilha essa cópia entre todos da classe. Cada objeto, naturalmente, precisa de sua própria cópia dos membros de dados da classe, porque seu conteúdo pode ser diferente entre objetos (como dois objetos diferentes tendo dois membros de dados diferentes). O código de função-membro, porém, não é modificável, podendo ser compartilhado entre todos os objetos da classe. Portanto, o tamanho de um objeto depende da quantidade de memória necessária para armazenar os membros de dados da classe, fornecidos para o compilador acesso às informações de que ele precisa (Figura 3.9, linha 40) para determinar o tamanho de um objeto e determinar se os objetos da classe são utilizados corretamente (nas linhas 13–14 e 17–18 da Figura 3.10).

A linha 7 instrui o pré-processador C++ a substituir a diretiva por uma cópia do código-fonte da classe `GradeBook` antes de o programa ser compilado. Quando o arquivo de código-fonte é compilado, ele agora contém a definição de `GradeBook` (por causa da #include) e o compilador é capaz de determinar como os objetos que suas funções-membro são chamadas corretamente. Agora que a definição de classe está em um arquivo de cabeçalho (seja

mais), você pode incluir esse cabeçalho em seu programa que precise reutilizar essa classe.

Como arquivos de cabeçalho são localizados?

Note que o nome do arquivo de cabeçalho na linha 7 da Figura 3.10 está entre colchetes angulares (`<GradeBook.h>`). Normalmente, os arquivos de código-fonte e arquivos de cabeçalho definidos pelo usuário de um programa são colocados no mesmo diretório. Quando o pré-processador encontra um nome de arquivo de cabeçalho entre aspas (por exemplo, `"GradeBook.h"`), o pré-processador tenta localizar o arquivo de cabeçalho no mesmo diretório do arquivo que a diretiva. Se o pré-processador não puder localizar o arquivo de cabeçalho nesse diretório, ele o procura na(s) mesma(s) localização(s) de arquivos de cabeçalho C++ Standard Library. Quando o pré-processador encontra um nome de arquivo de cabeçalho entre colchetes angulares (por exemplo, `<GradeBook.h>`), ele assume que o cabeçalho faz parte da C++ Standard Library e não examina o diretório do programa que está sendo pré-processado.



### Dica de prevenção de erro 3.3

Para assegurar que o pré-processador possa localizar os arquivos de cabeçalho corretamente, as diretivas de pré-processador `#include` devem colocar os nomes de arquivos de cabeçalho definidos pelo usuário entre aspas (por exemplo, `"GradeBook.h"`) e colocar os nomes de arquivos de cabeçalho da C++ Standard Library entre colchetes angulares (por exemplo,

Questões de engenharia de software adicionais

Agora que a classe `GradeBook` foi definida em um arquivo de cabeçalho, a classe é reutilizável. Infelizmente, colocar uma definição de classe em um arquivo de cabeçalho como na Figura 3.9 ainda revela a inteira implementação da classe para os clientes da —GradeBook. Isto é simplesmente um arquivo de texto que qualquer pessoa pode abrir e ler. A sabedoria da engenharia de software convencional diz que, para utilizar um objeto de uma classe, o código-cliente precisa saber somente quais funções-membro chamar argumentos fornecer para cada função-membro e qual tipo de retorno esperar de cada função-membro. O código-cliente não precisa saber como essas funções são implementadas.

Se o código-cliente sabe como uma classe é implementada, o programador de código-cliente poderia escrever o código-cliente com base nos detalhes de implementação da classe. Idealmente, se essa implementação mudar, os clientes da classe não devem mudar. Ocultar os detalhes de implementação da classe facilita a alteração de implementação da classe, ao mesmo tempo que reduz a chance de erros, espera-se, elimina a alteração do código-cliente.

Na Seção 3.9, mostramos como dividir os dois arquivos de modo que

1. a classe seja reutilizável;
2. os clientes da classe saibam que funções-membro a classe fornece, como chamá-las e que tipos de retorno esperar;
3. os clientes não saibam como as funções-membro da classe foram implementadas.

## 3.9 Separando a interface da implementação

Na seção anterior mostramos como promover a reusabilidade de software separando uma definição de classe do código-cliente. Agora introduzimos outro princípio fundamental de engenharia de software — a interface da implementação.

Interface de uma classe

Interfaces definem e padronizam o modo como coisas, pessoas e sistemas interagem entre si. Por exemplo, os controles de um rádio servem como uma interface entre os usuários do rádio e seus componentes internos. Os controles permitem aos usuários realizar um conjunto limitado de operações (como alterar a estação, ajustar o volume e escolher entre estações AM e FM). Vários rádios podem implementar essas operações de diferentes modos — alguns fornecem botões, alguns fornecem sintonizadores e outros suportam

mandos de voz. A interface especifica que operações um rádio permite que os usuários realizem, mas não especifica como as opções estão implementadas dentro do rádio.

De maneira semelhante, a interface de uma classe descreve os serviços que os clientes de uma classe podem utilizar e como esses serviços, na verdade, a classe executa os serviços. A interface de uma classe consiste nas definições-membro (também conhecidas como `public` da classe). Por exemplo, a interface da classe GradeBook (Figura 3.9) contém um construtor e as funções-membro `getCourseName`, `getGrade` e `displayMessage`. Os clientes de GradeBook, por exemplo, na Figura 3.10, utilizam essas funções para solicitar os serviços da classe. Como logo veremos, você pode especificar a interface de uma classe escrevendo uma definição de classe que lista apenas os nomes de função-membro, tipos de retorno e tipos de parâmetro.

Separando a interface da implementação

Em nossos primeiros exemplos, todas as definições de classe continham as definições completas das funções-membro e as declarações de seus membros de dados. Entretanto, é uma melhor engenharia de software definir funções-membro fora da definição de classe, para que os detalhes da sua implementação possam ficar ocultos do código-cliente. Essa prática assegura:

páginas de cliente, provavelmente, o código da interface implementada dentro da implementação da classe. Se eles precisassem fazer isso, o resultado seria um grande código-fonte com muitas definições de classe.

O programa das figuras 3.11–3.13 separa a interface das implementações dividindo a definição de classe da Figura 3.9 em dois arquivos — o arquivo de cabeçalho (Figura 3.11) em que a classe é definida e o arquivo de código-fonte (Figura 3.12) em que funções-membro são definidas. Por convenção, as definições de função-membro são colocadas em um arquivo de código-fonte com o mesmo nome de base, por exemplo, cabeçalho da classe, mas com uma extensão `.h`. O arquivo de código-fonte (Figura 3.13) define a função `main` (o código-cliente). O código e a saída da Figura 3.13 são idênticos aqueles da Figura 3.10. A Figura 3.14 mostra como esse programa de três arquivos é compilado a partir das perspectivas do programador da classe e do código-cliente — explicaremos essa figura em detalhes.

`GradeBook.h` Definindo a interface de uma classe com protótipos de função

O arquivo de cabeçalho `GradeBook.h` (Figura 3.11) contém outra versão de definição de classe (Figura 3.9). Essa versão é semelhante à da Figura 3.9, mas as definições de função na Figura 3.9 são substituídas (linhas 15–16).

que descrevem a interface da classe sem revelar as implementações da função-membro da classe. Um protótipo de função é uma declaração de função que informa ao compilador o nome da função, seu tipo de retorno e os tipos de seus parâmetros. Observe o arquivo de cabeçalho que especifica também o membro de dados (linha 17). Novamente, o compilador deve conhecer os membros de dados da classe para determinar quanta memória reservar para cada objeto da classe. Incluir o arquivo de cabeçalho `GradeBook.h` (linha 8 da Figura 3.13) fornece ao compilador as informações de que ele precisa para assegurar que o código-cliente chame corretamente as funções da classe.

O protótipo de função na linha 12 (Figura 3.12) indica que o construtor requer três parâmetros: o nome do curso, o nome do aluno e a nota. O protótipo de função da função `getCourseName` (linha 13) indica que a função não requer um parâmetro e não retorna um valor (isto é, seu tipo de retorno é `void`). O protótipo de função da função `getGrade` (linha 14) indica que a função não requer parâmetros e retorna uma `string`.

Por fim, o protótipo da função `displayMessage` (linha 15) especifica que a função não requer parâmetros e não retorna um valor. Esses protótipos de função são os mesmos que os cabeçalhos de função correspondentes na Figura 3.9, exceção feita de que os nomes de parâmetro (que são opcionais em protótipos) não são incluídos e cada protótipo de função deve terminar com um ponto-e-vírgula.



### Erro comum de programação 3.8

Esquecer de colocar o ponto-e-vírgula no final de um protótipo de função é um erro de sintaxe.



### Boa prática de programação 3.7

Embora nomes de parâmetro em protótipos de função sejam opcionais (eles são ignorados pelo compilador), muitos programadores utilizam esses nomes para propósitos de documentação.



### Dica de prevenção de erro 3.4

Os nomes de parâmetro em um protótipo de função (que, novamente, são ignorados pelo compilador) podem ser enganosos se nomeados errados ou confusos forem utilizados. Por essa razão, muitos programadores criam protótipos de função copiando a primeira linha das definições de função correspondentes (quando o código-fonte das funções estiver disponível), acrescentando então um ponto-e-vírgula ao final de cada protótipo.

```

1 // Figura 3.11: GradeBook.h
2 // Definição da classe GradeBook. Esse arquivo apresenta a interface pública de
3 // GradeBook sem revelar as implementações de funções-membro de GradeBook
4 // que são definidas em GradeBook.cpp.
5 #include <string> // a classe GradeBook utiliza a classe de string padrão C++
6 using std::string;
7
8 // definição da classe GradeBook
9 class GradeBook
10 {
11 public :
12 GradeBook(string); // construtor que inicializa courseName
13 void setCourseName(string); // função que configura o nome do curso
14 string getCourseName(); // função que obtém o nome do curso
15 void displayMessage(); // função que exibe uma mensagem de boas-vindas
16 private :
17 string courseName; // nome do curso para esse GradeBook
18 }; // fim da classe GradeBook

```

Figura 3.11 Definição da classe GradeBook contendo os protótipos de função que especificam a interface da classe.

```

1 // Figura 3.12: GradeBook.cpp
2 // Definições de função-membro de GradeBook. Esse arquivo contém
3 // implementações das funções-membro prototipadas em GradeBook.h.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // inclui a definição de classe GradeBook
9
10 // o construtor inicializa courseName com a string fornecida como argumento
11 GradeBook::GradeBook(string name)
12 {
13 setCourseName(name); // chama a função set para inicializar courseName
14 } // fim do construtor GradeBook
15
16 // função para configurar o nome do curso
17 void GradeBook::setCourseName(string name)
18 {
19 courseName = name; // armazena o nome do curso no objeto
20 } // fim da função setCourseName
21
22 // função para obter o nome do curso
23 string GradeBook::getCourseName()
24 {
25 return courseName; // retorna courseName do objeto
26 } // fim da função getCourseName
27
28 // exibe uma mensagem de boas-vindas para o usuário GradeBook
29 void GradeBook::displayMessage()
30 {
31 // chama getCourseName para obter o courseName
32 cout << "Welcome to the grade book for\n" << getCourseName()
33 << "!" << endl;
34 } // fim da função displayMessage

```

Figura 3.12 As definições de função-membro GradeBook representam a implementação da classe GradeBook.

```

1 // Figura 3.13: fig03_13.cpp
2 // Demonstração de classe GradeBook depois de separar
3 // sua interface de sua implementação.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // inclui a definição de classe GradeBook
9
10 // a função main inicia a execução do programa
11 int main()
12 {
13 // cria dois objetos GradeBook
14 GradeBook gradeBook1("CS101 Introduction to C++ Programming");
15 GradeBook gradeBook2("CS102 Data Structures in C++");
16
17 // exibe valor inicial de courseName para cada GradeBook
18 cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
19 << endl
20 << "gradeBook2 created for course: " << gradeBook2.getCourseName()
21 << endl;
22 } // fim de main

```

gradeBook1 created for course: CS101 Introduction to C++ Programming  
 gradeBook2 created for course: CS102 Data Structures in C++

Figura 3.13 A demonstração da classe GradeBook depois de separar sua interface de sua implementação.

O arquivo de código GradeBook.cpp (Figura 3.12) define as funções-membro da classe GradeBook declaradas nas linhas 12–15 da Figura 3.11. As definições de função-membro aparecem nas linhas 11–34 e são quase idênticas às definições de membro nas linhas 15–38 da Figura 3.9.

Note que cada nome de função-membro nos cabeçalhos de função (linhas 11, 17, 23 e 29) é precedido pelo nome de classe e é conhecido como *operador de resolução de escopo binário*. Isso ‘amarra’ cada função-membro à definição de classe (agora separada), que declara as funções-membro e membros da classe GradeBook. Sendo assim, essa ‘amarração’ garante que essas funções não seriam reconhecidas pelo compilador como funções-membro da classe GradeBook, consideraria funções ‘livres’ ou ‘irrestritas’. Essas funções não podem acessar os dados da classe GradeBook, ou chamar as funções-membro da classe, sem especificar um objeto. Portanto, o compilador não seria capaz de compilar essas funções. Por exemplo, as linhas 19 e 25 que acessam a variável courseName causariam erros de compilação, porque courseName é declarada como uma variável local em cada função — o compilador não saberia que courseName é declarada como um membro de GradeBook.



### Erro comum de programação 3.9

Ao definir funções-membro de uma classe fora dessa classe, omitir o nome de classe e operador de resolução de escopo binário (que precede os nomes de função) causa erros de compilação.

Para indicar que as funções-membro fazem parte da classe GradeBook devemos primeiro incluir o arquivo de cabeçalho GradeBook.h (linha 8 da Figura 3.12). Isso permite acesso ao GradeBook.h para o GradeBook.cpp. Ao

1. a primeira linha de cada função-membro (linhas 11, 17, 23 e 29) corresponde ao seu protótipo no arquivo GradeBook.h. Por exemplo, o compilador assegura que não aceita parâmetros e retorna uma GradeBook.
2. cada função-membro conheça os membros de dados e outras funções-membro da classe — por exemplo, as linhas 19 e 25 acessar a variável courseName, porque ela foi declarada em GradeBook como um membro de dados da classe GradeBook. As linhas 13 e 32 podem chamar as funções getName() e getCourseName(), respectivamente, porque cada uma foi declarada como uma função-membro da classe GradeBook (porque essas chamadas obedecem aos protótipos correspondentes).

## Testando a classe GradeBook

A Figura 3.13 realiza as mesmas manipulações de objeto. Figura 3.10. Separar a interface da implementação de suas funções-membro não afeta a maneira como esse código-cliente utiliza a classe. Afeta apenas a maneira como o programa é compilado e vinculado, o que discutiremos detalhadamente em breve.

Como na Figura 3.10, a linha 8 da Figura 3.13 inclui o arquivo de cabeçalho compilador possa assegurar que os objetos da classe GradeBook sejam criados e manipulados corretamente no código-cliente. Antes de executar esse programa, os arquivos de código-fonte nas figuras 3.12 e 3.13 devem ser compilados, e, então, vinculados entre si — isto é, as chamadas de função-membro no código-cliente precisam ser associadas às implementações das funções-membro da classe — um trabalho realizado pelo link.

O processo de compilação e vinculação

O diagrama na Figura 3.14 mostra o processo de compilação e vinculação que resulta em um aplicativo executável que pode ser executado.

Freqüentemente, a interface e a implementação de uma classe serão criadas e compiladas por um programador e utilizadas por um programador separado que implementa o código-cliente da classe. Então, o diagrama mostra o que é requerido para separar a implementação da classe como código-fonte, e o que é requerido para gerar o código-cliente, pelo programador do código-cliente.

[A Figura 3.14 não é um diagrama UML.]

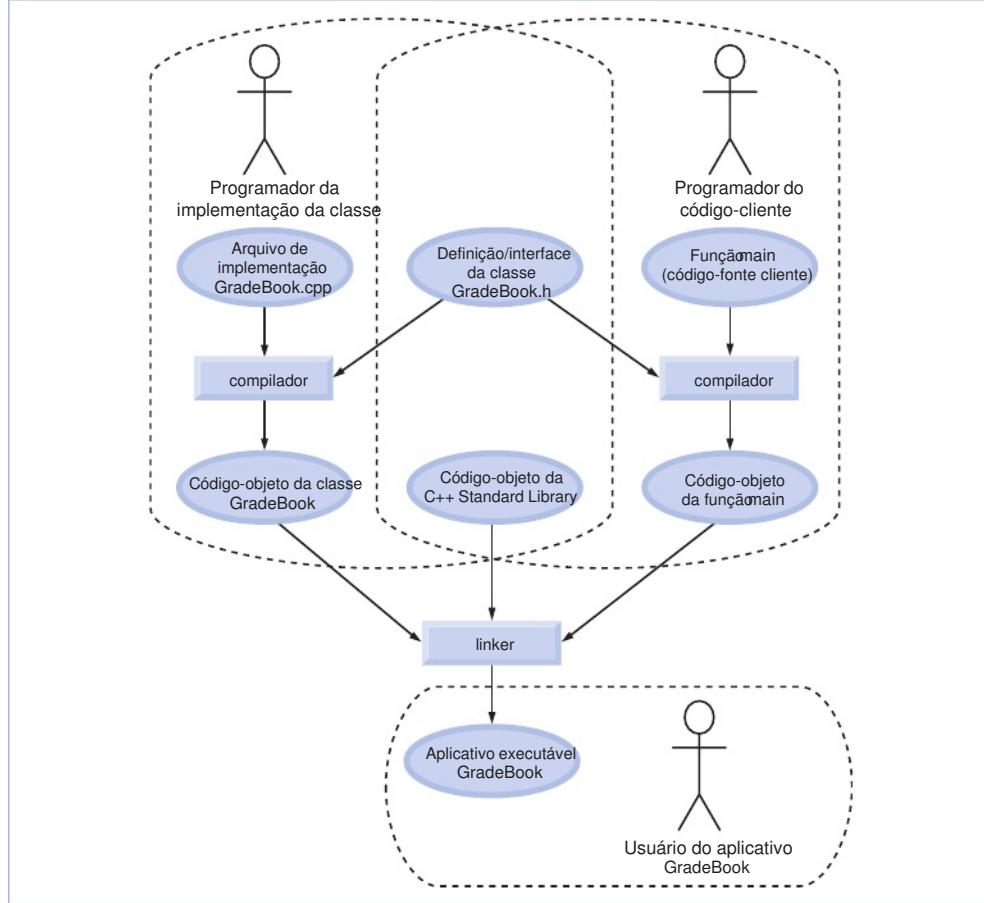


Figura 3.14 O processo de compilação e vinculação que produz um aplicativo executável.

Um programador de implementação de classe responsável por criar a classe `GradeBook` só precisa utilizar o arquivo de cabeçalho `GradeBook.h`. O arquivo de código-fonte (`GradeBook.cpp`) que inclui (`#include`) o arquivo de cabeçalho, e, depois, compila o arquivo de código-fonte para criar código-objeto. Para ocultar os detalhes de implementação de funções, o programador da implementação da classe forneceria ao programador do código-cliente o arquivo de cabeçalho que especifica a interface e os membros de dados da classe) e o código-objeto para as instruções de linguagem de máquina que representam as funções implementadas.

O código-cliente só precisa conhecer a interface para utilizar a classe e deve ser capaz de vincular seu código-objeto. Visto que a interface da classe faz parte da definição de classe, o arquivo de cabeçalho do código-cliente deve ter acesso a esse arquivo e incluir, assim, um arquivo de código-fonte do cliente. Quando o código-cliente é compilado, o compilador utiliza a definição da classe para assegurar que a função manipula objetos da classe `GradeBook`, portanto, o cliente permanece sem saber como as funções são implementadas.

Para criar o aplicativo executável para ser utilizado por instrutores, o último passo é vincular

1. o código-objeto da função (é, o código-cliente);
2. o código-objeto para implementações de função criado pelo programador da classe;
3. o código-objeto da C++ Standard Library para as classes C++ utilizadas pelo programador da implementação da classe e pelo programador do código-cliente.

A saída do linker é o aplicativo executável que instrutores podem utilizar para gerenciar as notas de seus alunos.

Para obter informações adicionais sobre como compilar programas de múltiplos arquivos de código-fonte, veja a documentação do seu compilador ou estude as publicações oferecemos para vários compiladores em [http://www.cppbooks.org/cpphtp5](#)

### 3.10 Validando dados com funções set

Na Secção 3.6, introduzimos `setCourseName` para permitir aos clientes de uma classe modificar o valor de um membro de dados. Na Figura 3.5, a classe define a função-membro `setCourseName` para simplesmente atribuir um valor recebido em seu parâmetro ao membro de dados `courseName`. Essa função-membro não assegura que o nome do curso obedeca a qualquer formato particular ou siga quaisquer outras regras relacionadas ao que é um nome do curso ‘válido’. Como declaramos anteriormente, supõe-se que uma universidade pode imprimir listagens de alunos contendo nomes de curso com apenas 25 caracteres ou menos. Se a universidade usa um sistema contendo objetos para gerar as listagens, poderíamos querer que esse sistema verifique se esse nome de curso é válido. No entanto, o código da classe GradeBook das figuras 3.15–3.17 aprimora a função-membro

Definição de classe GradeBook

Note que a definição da classe (Figura 3.15) — e, portanto, sua interface — é idêntica à da Figura 3.11. Visto que a interface permanece inalterada, os clientes dessa classe não precisam ser alterados quando a definição de função-membro

```

1 // Figura 3.15: GradeBook.h
2 // Definição de classe GradeBook apresenta a interface public da
3 // classe. Definições de função-membro aparecem em GradeBook.cpp.
4 #include <string> // o programa utiliza classe de string padrão do C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public:
11 GradeBook(string); // construtor que inicializa um objeto GradeBook
12 void setCourseName(string); // função que configura o nome do curso
13 string getCourseName(); // função que obtém o nome do curso
14 void displayMessage(); // função que exibe uma mensagem de boas-vindas
15 private :
16 string courseName; // nome do curso para esse GradeBook
17 }; // fim da classe GradeBook

```

Figura 3.15 A definição da classe GradeBook

```

1 // Figura 3.16: GradeBook.cpp
2 // Implementações das definições de função-membro de GradeBook.
3 // A função setCourseName realiza a validação.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h"// inclui a definição de classe GradeBook
9
10 // construtor inicializa courseName com String fornecido como argumento
11 GradeBook::GradeBook(string name)
12 {
13 setCourseName(name); // valida e armazena courseName
14 } // fim do construtor GradeBook
15
16 // função que configura o nome do curso;
17 // assegura que o nome do curso tenha no máximo 25 caracteres
18 void GradeBook::setCourseName(string name)
19 {
20 if (name.length() <= 25) // se o nome tiver 25 ou menos caracteres
21 courseName = name; // armazena o nome do curso no objeto
22
23 if (name.length() > 25) // se o nome tiver mais de 25 caracteres
24 {
25 // configura courseName como os primeiros 25 caracteres do parâmetro name
26 courseName = name.substr(0, 25); // inicia em 0, comprimento de 25
27
28 cout << "Name '" << name << "' exceeds maximum length (25).\n"
29 << "Limiting courseName to first 25 characters.\n" << endl;
30 } // fim do if
31 } // fim da função setCourseName
32
33 // função para obter o nome do curso
34 string GradeBook::getCourseName()
35 {
36 return courseName; // retorna courseName do objeto
37 } // fim da função getCourseName
38
39 // exibe uma mensagem de boas-vindas para o usuário GradeBook
40 void GradeBook::displayMessage()
41 {
42 // chama getCourseName para obter o courseName
43 cout << "Welcome to the grade book for\n" << getCourseName()
44 << "!" << endl;
45 } // fim da função displayMessage

```

**Figura 3.16** As definições de função-membro para a classe GradeBook com uma função que valida o comprimento do membro de dados courseName

modificada. Isso permite que os clientes tirem proveito da classe simplesmente vinculando o código-cliente ao código-objeto da GradeBook atualizada.

Validando o nome do curso com a função-membro setCourseName GradeBook

O aprimoramento para a classe GradeBook está na definição de setCourseName (Figura 3.16, linhas 18–31). A instrução if nas linhas 20–21 determina se o parâmetro é um nome válido de curso (isto é, uma string de 25 ou menos caracteres). Se o nome do curso for válido, a linha 21 armazenará o nome do curso no membro de dados courseName. Observe a expressão length() na linha 20. Essa é uma chamada de função-membro idêntica à função strlen() da classe `<cctype>` da C++ Standard.

Library define uma função `length` que retorna o número de caracteres em um `objeto`. O `objeto` é um `objeto` string, portanto a chamada `length()` retorna o número de caracteres desse `objeto`. Se esse valor é menor que 25, é legal a validade e a linha 21 é executada.

A instrução nas linhas 23–30 trata o caso em que se recebe um nome inválido de curso (isto é, um nome que tem mais de 25 caracteres de comprimento). Mesmo se o `objeto`, ainda queremos deixar o `objeto` um `estado consistente`: isto é, um estado em que o membro de dados `courseName` tenha um valor válido (isto é, uma string de 25 ou menos caracteres). Portanto, truncamos (isto é, encurtamos) o nome do curso especificado e atribuímos os 25 primeiros caracteres ao membro de dados `courseName` (infelizmente, isso poderia truncar horrivelmente o nome do curso). A classe padrão fornece a função `memorizar` (abbreviação de 'substring') que retorna um novo `objeto` mediante a cópia de parte de um `objeto` existente. A chamada na linha 26 (`name(0, 25)`) passa dois inteiros para a função-membro `substr` de `name`. Esses argumentos indicam a `parte` que deve retornar. O primeiro argumento especifica a posição inicial `srcinal` a partir da qual os caracteres são copiados — considera-se que o primeiro caractere em cada string está na posição 0. O segundo argumento especifica o número de caracteres a ser copiado. Portanto, a chamada na linha 26 retona

```

1 // Figura 3.17: fig03_17.cpp
2 // Cria e manipula um objeto GradeBook; ilustra a validação.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // inclui a definição de classe GradeBook
8
9 // a função main inicia a execução do programa
10 int main()
11 {
12 // cria dois objetos GradeBook;
13 // nome inicial de curso de gradeBook1 é muito longo
14 GradeBook gradeBook1("CS101 Introduction to Programming in C++");
15 GradeBook gradeBook2("CS102 C++ Data Structures");
16
17 // exibe courseName de cada GradeBook
18 cout << "gradeBook1's initial course name is: "
19 << gradeBook1.getCourseName()
20 << endl;
21 cout << "gradeBook2's initial course name is: "
22 << gradeBook2.getCourseName() << endl;
23
24 // modifica courseName do myGradeBook (com uma string de comprimento válido)
25 gradeBook1.setCourseName("CS101 C++ Programming");
26
27 // exibe courseName de cada GradeBook
28 cout << "gradeBook1's course name is: "
29 << gradeBook1.getCourseName()
30 << endl;
31 cout << "gradeBook2's course name is: "
32 << gradeBook2.getCourseName() << endl;
33 return 0; // indica terminação bem-sucedida
34 } // fim de main

```

Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).

gradeBook1's initial course name is: CS101 Introduction to Pro  
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming  
gradeBook2's course name is: CS102 C++ Data Structures

Figura 3.17 Criando e manipulando um objeto `GradeBook` que o nome do curso é limitado a 25 caracteres de comprimento.

substring de 25 caracteres que inicia na posição 0 (isto é, os primeiros 25 caracteres). Por exemplo, se armazena o valor "CS101 Introduction to Programming in C++"; substr retorna "CS101 Introduction to Pro". Depois da chamada para substr, a linha 26 atribui a substring retornado ao membro de dados courseName. Dessa maneira, a função-membro setCourseName garante que courseName sempre atribuída uma string contendo 25 ou menos caracteres. Se a função-membro tem de truncar o nome do curso para torná-lo válido, as linhas 28–29 exibem uma mensagem de advertência.

Observe que a instrução nas linhas 23–30 contém duas instruções de corpo — uma para os comentários 25 caracteres do parâmetro para imprimir uma mensagem anexa para o usuário. Queremos que essas duas instruções executem quando name for muito longo, então as colocamos entre chaves. Na discussão no Capítulo 2, lembre-se de que isso cria um bloco. Você aprenderá mais sobre como colocar múltiplas instruções no corpo de uma instrução de controle no Capítulo 4.

Observe que a instrução nas linhas 28–29 também poderia aparecer sem o operador de inserção de fluxo no início da segunda linha da instrução, como em:

```
cout << "Name \"<< name << "\" exceeds maximum length (25).\n"
 "Limiting courseName to first 25 characters.\n" << endl;
```

O compilador C++ combina literais string adjacentes, mesmo se eles aparecerem em linhas separadas de um programa. Portanto, o compilador C++ combinará "exceeds maximum length (25).\n" e "Limiting courseName to first 25 characters.\n" em um único literal string para produzir saída idêntica à das linhas 28–29 na Figura 3.16. Esse comportamento permite imprimir strings longas dividindo-as ao longo das linhas no programa sem incluir operações de inserção de fluxo adicionais.

Testando a classe GradeBook

A Figura 3.17 demonstra a versão modificada da classe (Figuras 3.15–3.16) destacando a validação. A linha 14 cria um objeto GradeBook gradeBook1. Lembre-se de que o construtor chama a função-membro setCourseName para inicializar o membro de dados courseName. Nas versões anteriores da classe, o benefício desse construtor não era evidente. Agora, porém, o construtor tira proveito da validação fornecida pelo construtor simplesmente chamando setCourseName em vez de duplicar seu código de validação. Quando a linha 14 da Figura 3.17 passa um nome inicial de curso de "Introduction to Programming in C++" para o construtor, o construtor passa esse valor para onde ocorre a inicialização real. Como esse nome do curso contém mais que 25 caracteres, o resultado é truncado ("Introduction Pro") (a parte truncada é destacada em itálico na linha 14). Note que a saída na Figura 3.17 contém a mensagem de advertência gerada pelas linhas 28–29 da Figura 3.16. A linha 15 cria outro objeto e chama gradeBook2 — o nome do curso válido passado para o construtor tem exatamente 25 caracteres.

As linhas 18–21 da Figura 3.17 exibem o nome gradeBook1 destacados, inseridos em itálico na saída de programa (e o nome do objeto gradeBook2 na linha 24 chama a função setCourseName diretamente, para mudar o nome do curso no objeto Boop para um nome mais curto que não precise ser truncado. Então, as linhas 27–30 geram novamente a saída dos nomes de curso dos objetos).

Notas adicionais sobre as funções

Uma função public como setCourseName deve verificar qualquer tentativa de modificação do valor de um membro de dados (por exemplo, courseName) para assegurar que o novo valor seja apropriado a esse item de dados. Por exemplo, uma tentativa de (se) o dia do mês como 37 deve ser rejeitada, uma tentativa de uma pessoa como zero ou um valor negativo deve ser rejeitada, uma tentativa de uma nota de uma prova como 185 (quando o intervalo adequado é de zero a 100) deve ser rejeitada etc.



#### Observação de engenharia de software 3.6

Tornar os membros de dados controlar o acesso, especialmente acesso de gravação, para aqueles membros de dados via funções-membro ajuda a assegurar a integridade de dados.



#### Dica de prevenção de erro 3.5

Os benefícios da integridade de dados não são automáticos simplesmente porque os membros de dados se tornaram programador deve fornecer teste de validade apropriado e informar os erros.



#### Observação de engenharia de software 3.7

As funções-membro integradas nos valores de dados devem verificar se os novos valores projetados são adequados; se não forem, as funções-membro devem colocar os membros de dados num estado apropriado.

As funções de uma classe podem retornar valores para os clientes da classe indicando que foram feitas tentativas de atribuição de dados inválidos a objetos da classe. Um cliente da classe pode testar o valor de determinado membro para determinar se a tentativa

do cliente de modificar o objeto foi bem-sucedida e executar uma ação apropriada. No Capítulo 16, demonstramos como os clientes de uma classe podem ser notificados por meio do mecanismo de tratamento de exceções quando uma tentativa de modificar um com um valor impróprio for feita. Para manter o programa das figuras 3.15–3.17 simples neste ponto inicial no livro, na Figura 3.16 imprime apenas uma mensagem apropriada na tela.

### 3.11 Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional)

Agora começamos a projetar o sistema ATM que introduzimos no Capítulo 2. Nesta seção, identificamos as classes que são necessárias para construir o sistema ATM analisando os substantivos simples e os substantivos compostos que aparecem no documento de requisitos. Introduzimos diagramas de classes UML para modelar os relacionamentos entre essas classes. Este é um primeiro passo importante na definição da estrutura do nosso sistema.

**Identificando as classes em um projeto**  
Iniciamos nosso processo UOD identificando as classes necessárias para construir o sistema ATM. Por fim, descrevemos essas classes utilizando diagramas de classes UML e implementamos essas classes em C++. Inicialmente, vamos revisar o documento de requisitos da Seção 2.8 e localizar substantivos e substantivos compostos para nos ajudar a identificar classes que compreendem o sistema. Podemos decidir que alguns desses substantivos e substantivos compostos são atributos de outras classes no sistema. Também podemos concluir que alguns substantivos não correspondem a partes do sistema e, portanto, simplesmente não devem ser modelados. As adicionais podem tornar-se visíveis à medida que avançamos no processo de projeto.

A Figura 3.18 lista os substantivos e substantivos compostos encontrados no documento de requisitos. Listamos esses substantivos da esquerda para a direita na ordem em que aparecem no documento de requisitos. Listamos somente a forma singular de cada substantivo simples ou substantivo composto.

Criamos classes apenas para os substantivos simples e os substantivos compostos que têm importância no sistema ATM. Não precisamos modelar o ‘banco’ como uma classe, porque o banco não é uma parte do sistema ATM — o banco simplesmente quer que o ATM seja construído. O ‘cliente’ e o ‘usuário’ também representam entidades fora do sistema — são importantes porque interagem com o nosso sistema ATM, mas não precisamos modelá-los como classes no software ATM. Lembre-se de que modelamos um usuário (isto é, um cliente de banco) como o ator no diagrama de casos de uso da Figura 2.18.

Não modelamos a ‘cédula \$ 20’ nem o ‘envelope de depósito’ como classes. Esses são objetos físicos no mundo real, mas não fazem parte do que é automatizado. Podemos representar adequadamente a presença de contas no sistema utilizando um atributo da classe que modela o dispensador de cédulas. (Atribuímos atributos a classes na Seção 4.13.) Por exemplo, o dispensador de cédulas n

~~emulca a operação de depósito dentro de sua classe. Podemos ignorar o detalhe de que a classe de depósito não faz parte do sistema de fazer um depósito de dentro de seu escopo. O documento de requisitos só indica que a classe de depósito deve ser capaz de fazer a operação realizada pela classe que modela a abertura para depósito — é suficiente para representar a presença de um envelope no sistema.~~

Em nosso sistema ATM simplificado parece mais apropriado representar as várias quantias de ‘dinheiro’, incluindo o ‘saldo’ de uma conta, como atributos de outras classes. Da mesma forma, os substantivos ‘número de conta’ e ‘PIN’ representam partes significativas de informações no sistema ATM. Esses são atributos importantes de uma conta bancária. Mas não exibem comportamentos. Portanto, você pode modelá-los mais apropriadamente como atributos de uma classe de conta.

Embora o documento de requisitos costume descrever uma ‘transação’ em um sentido geral, não modelamos a noção ampla de transação financeira nesse momento. Em vez disso, modelamos os três tipos de transações (isto é, ‘consulta de saldo’, ‘saque’ e ‘depósito’) como classes individuais. Essas classes possuem atributos específicos necessários para executar as transações que elas representam.

| Substantivos e substantivos compostos no documento de requisitos |                         |                 |
|------------------------------------------------------------------|-------------------------|-----------------|
| banco                                                            | dinheiro                | dinheiro        |
| ATM                                                              | tela                    | número de conta |
| usuário                                                          | teclado                 | PIN             |
| cliente                                                          | dispensador de cédulas  | banco           |
| transação                                                        | cédula de \$20/dinheiro | dados do banco  |
| conta                                                            | abertura para depósito  | consultar saldo |
| saldo                                                            | envelope de depósito    | retirada/saque  |
|                                                                  |                         | depósito        |

Figura 3.18 Substantivos e substantivos compostos no documento de requisitos.

exemplo, uma retirada precisa saber quanto dinheiro o usuário quer sacar. Uma consulta de saldo, porém, não requer dados adicionais. Além disso, as três classes de transação exibem comportamentos únicos. Uma retirada inclui liberar dinheiro para o usuário, enquanto depósito envolve receber envelopes de depósito. [No Seção 10.3, 'fatoramos' recursos comuns a todas as transações em uma classe 'transaction' geral utilizando os conceitos orientados a objetos de classes abstratas e herança.]

Determinamos as classes para nosso sistema com base nos substantivos e substantivos compostos restantes da Figura 3.18. Deles referenciamos um ou mais objetos, como os seguintes:

- ATM
- telas de criação
- teclado de depósito
- dispensador de dinheiro
- abertura para depósito
- conta
- banco de dados
- consulta de saldo
- retirada/saque
- depósito

É provável que os elementos dessa lista sejam as classes necessárias para implementar nosso sistema.

Agora podemos modelar as classes em nosso sistema com base na lista que criamos. Escrevemos os nomes de classe no projeto em letras maiúsculas — uma convenção UML — assim como quando escrevemos o código C++ real que implementa projeto. Se o nome de uma classe contiver mais de uma palavra, unimos as palavras e colocamos a letra inicial de cada palavra maiúscula (por exemplo, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` e `Deposit`). Utilizando essa convenção, criamos as classes `ATM`, `BankDatabase`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` e `Deposit`. Construiremos nosso sistema utilizando todas essas classes como blocos de construção. Entretanto, antes de iniciarmos a construção do sistema, devemos obter um melhor entendimento de como as classes se relacionam.

#### Modelando classes

A UML permite modelar [classes](#) no sistema ATM e seus inter-relacionamentos. A Figura 3.19 representa a classe ATM. Na UML, cada classe é modelada como um retângulo com três compartimentos. O compartimento superior contém o nome da classe centralizado horizontalmente e em negrito. O compartimento do meio contém os atributos da classe. (Discutimos os atributos na Seção 4.13 e Seção 5.11.) O compartimento inferior contém as operações da classe (discutidas na Seção 6.22). Na Figura 3.19, os compartimentos do meio e inferior estão vazios, porque ainda não determinamos os atributos e as operações dessa classe.

Os diagramas de classes também mostram os relacionamentos entre as classes do sistema. A Figura 3.20 mostra como nossa classe ATM se relaciona com a classe Withdrawal. Por ora, escolhemos modelar apenas esse subconjunto de classes para simplificar. Apresentaremos um diagrama de classes mais completo a seguir nesta seção. Observe que os retângulos que representam classes nesse diagrama são subdivididos em compartimentos. A UML permite a supressão de atributos e operações de classe dessa maneira, quando apropriado, para criar diagramas mais legíveis. Diz-se que esse diagrama é [um](#) diagrama em que algumas informações, como o conteúdo do segundo e terceiro compartimentos, não são modeladas. Colocaremos informações nesses compartimentos nas seções 4.13 e 6.22.

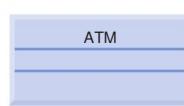


Figura 3.19 Representando uma classe na UML utilizando um diagrama de classes.

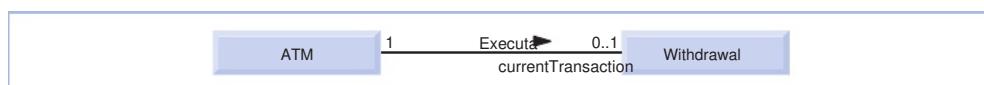


Figura 3.20 Diagrama de classes que mostra uma associação entre classes.

Na Figura 3.20, a linha sólida que conecta as duas classes representa a associação entre classes. Os números próximos de cada extremidade da linha indicam quantos objetos de cada classe participam da associação. Nesse caso, seguir a linha de uma extremidade a outra revela que, a qualquer momento, um objeto tem uma associação com zero ou um objeto zero se o usuário atual não estiver realizando atualmente uma transação ou tiver solicitado um tipo de transação diferente, e um se o usuário tiver solicitado uma retirada. A UML pode modelar muitos tipos de multiplicidade. A Figura 3.21 lista e explica os tipos de multiplicidade.

Uma associação pode ser nomeada. Por exemplo, a palavra `withdrawal` na linha que conecta a classe `User` à classe `Withdrawal` na Figura 3.20 indica o nome dessa associação. Essa parte do diagrama é chamada de execução da classe e o objeto da classe `Withdrawal`. Observe que os nomes de associação são direcionais, como indicado pela ponta da seta preenchida — então seria inadequado, por exemplo, ler a associação anterior da direita para a esquerda como `user.withdrawal` ou um objeto da classe `user` executa um objeto da classe `Withdrawal`.

A palavra `currentTransaction` ao lado de `withdrawal` na linha de associação da Figura 3.20 é porque identifica o papel que o objeto `withdrawal` desempenha em seu relacionamento. A palavra `come` de papel adiciona significado a uma associação

~~para classe e sistema existindo. O papel de uma classe é sempre uma associação de uma parte de uma classe para a outra. Por exemplo, uma classe pode sempre relacionar com alunos. A mesma pessoa pode assumir o papel de 'colega' ao relacionar-se com outro professor e de 'treinador' ao alunos atletas. Na Figura 3.20, o nome de papel indica que o objeto `withdrawal` que participa da associação de `executes` com um objeto da classe `transaction` apresenta a transação sendo atualmente processada pelo ATM. Em outros contextos, um objeto `Withdrawal` pode assumir outros papéis (por exemplo, a transação anterior). Observe que não especificamos um nome de papel para extremidades da associação. Os nomes de papel em diagrama de classes costumam ser omitidos quando o significado de uma associação é claro sem eles.~~

Além de indicar relacionamentos simples, as associações podem especificar relacionamentos mais complexos, como objetos de uma classe que são compostos de objetos de outras classes. Considere um caixa automático do mundo real. Que 'peças' um fabricante monta para construir um ATM funcional? Nossa documentação de requisitos informa que o ATM é composto de uma tela, um teclado, dispensador de cédulas e uma abertura para depósito.

Na Figura 3.22, ~~os~~ losangos sólidos anexados às linhas de associação indicam que a classe `ATM` tem um relacionamento de composição com as classes `Keypad`, `CashDispenser` e `DepositSlot`. A composição implica um relacionamento do todo/parte. A classe que tem o símbolo de composição (o losango sólido) em sua extremidade da linha de associação é o todo (caso ~~que~~ ~~que~~ as classes na extremidade das linhas de associação são as partes). Keypad, cashDispenser e DepositSlot. As composições na Figura 3.22 indicam que um ~~objeto~~ ~~parte~~ pode partir de um objeto da classe `ATM` para um objeto da classe `cashDispenser` ou `depositSlot`. O ATM 'tem uma' tela, um te-

~~relacionamento 'tem um'~~  
~~de deposito~~  
De acordo com a especificação UML, os relacionamentos de composição têm as seguintes propriedades:

1. Somente uma classe no relacionamento pode representar o todo (isto é, o losango pode ser colocado somente no final da linha de associação). Por exemplo, a tela é parte do ATM ou o ATM é parte da tela, mas a tela e o ATM não podem representar todo no relacionamento.

| Símbolo | Significado                                  |
|---------|----------------------------------------------|
| 0       | Nenhuma                                      |
| 1       | Um                                           |
| m       | Um valor de inteiro                          |
| 0..1    | Zeroum                                       |
| m..n    | moun                                         |
| m..n    | Pelo menos mas não mais do que n             |
| *       | Qualquer inteiro não negativo (zero ou mais) |
| 0..*    | Zero ou mais (idêntico a *)                  |
| 1..*    | Um ou mais                                   |

Figura 3.21 Tipos de multiplicidade.

2. As partes no relacionamento de composição só existem enquanto o todo existir, e o todo é responsável pela criação e desto de suas partes. Por exemplo, o ato de construir um ATM inclui manufaturar suas partes. Além disso, se o ATM é destruído, tela, teclado, dispensador de cédulas e abertura para depósito também são destruídos.
3. Uma parte pode pertencer a somente um todo de cada vez, embora a parte possa ser removida e anexada a outro todo, então assume a responsabilidade pela parte.

Os losangos sólidos em nossos diagramas de classes indicam relacionamentos de composição que satisfazem essas três propriedades. Se um relacionamento ‘tem um’ não satisfaz um ou mais desses critérios, a UML especifica que os losangos sem preenchimento anexados às extremidades de linhas de associação representam uma forma mais fraca de composição. Por exemplo, um computador pessoal e um monitor de computador participam de um relacionamento de agregação — o computador ‘tem um’ monitor, mas as duas partes podem existir independentemente e o mesmo monitor pode ser anexado a múltiplos computadores de um violando assim a segunda e a terceira propriedades de composição.

A Figura 3.23 mostra um diagrama de classes para o sistema ATM. Esse diagrama modela a maioria das classes que identificamos anteriormente. No entanto, este diagrama não inclui classes como *BankDatabase*, *BankManager* e *BankObject*, que aparecerão no final desse capítulo. [Note que este diagrama para manter o diagrama simples. No Capítulo 13, expandimos nosso diagrama de classes para incluir todas as classes do sistema ATM.]

A Figura 3.23 apresenta um modelo gráfico da estrutura do sistema ATM. Esse diagrama de classes inclui as classes *Account*, *BankDatabase*, *CashDispenser*, *DepositSlot*, *Keypad* e *Screen*. Diversas associações que não estavam presentes nas figuras 3.20 ou 3.22. O diagrama de classes mostra que a classe *Account* participa de um relacionamento de composição com zero ou mais objetos *BankDatabase* — um objeto que autentica usuários em um banco de dados. Na Figura 3.23, também modelamos o fato de que o banco de dados do banco contém informações sobre muitas contas — um ou mais objetos *BankDatabase* armazenam muitas contas. De maneira semelhante, a classe *BankDatabase* participa de um relacionamento de composição com zero ou mais objetos *BankObject* — um banco de dados armazena muitos objetos de banco de dados. [Nota: A partir da Figura 3.21, lembre-se de que o valor de multiplicidade \* é idêntico ao 0..\*. Incluímos 0..\* em nossos diagramas de classes para tornar isso mais claro.]

A Figura 3.23 também indica que, se o usuário estiver fazendo um depósito, o valor da conta é alterado diretamente. Poderíamos ter criado uma associação direta entre a classe *Account* e a classe *DepositSlot*. O documento de requisitos, porém, determina que o ‘ATM deve interagir com o banco de dados de informações de contas do banco’ para realizar transações. Uma conta bancária contém informações sensíveis e os engenheiros de sistema sempre considerar a segurança de dados pessoais ao projetar um sistema que pode manipular uma conta diretamente. Todas as outras partes do sistema devem interagir com o banco de dados para recuperar ou atualizar informações da conta (por exemplo, o saldo da conta).

O diagrama de classes na Figura 3.23 também modela associações entre a classe *CashDispenser* e *Keypad*. Uma transação de retirada inclui solicitar ao usuário a escolha de uma quantia em dinheiro a ser retirada e receber a informação numérica. Essas ações requerem o uso da tela e do teclado, respectivamente. Além disso, liberar dinheiro para o usuário requer ao dispensador de cédulas.

As classes *BalanceInquiry* e *Deposit*, embora não mostradas na Figura 3.23, fazem parte de diversas associações com as outras classes do sistema ATM. Como cada uma dessas classes associa-se com as classes *BankDatabase* e *BankObject*, o objeto da classe *BalanceInquiry* também se associa com um objeto de banco de dados para obter o saldo de uma conta para o usuário.

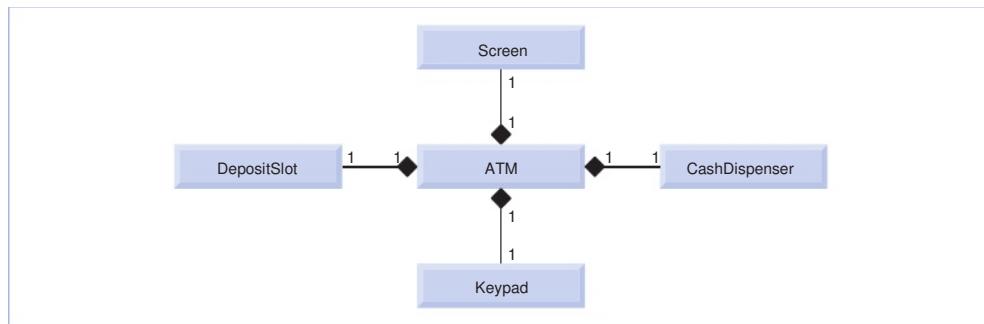


Figura 3.22 Diagrama de classes mostrando os relacionamentos de composição.

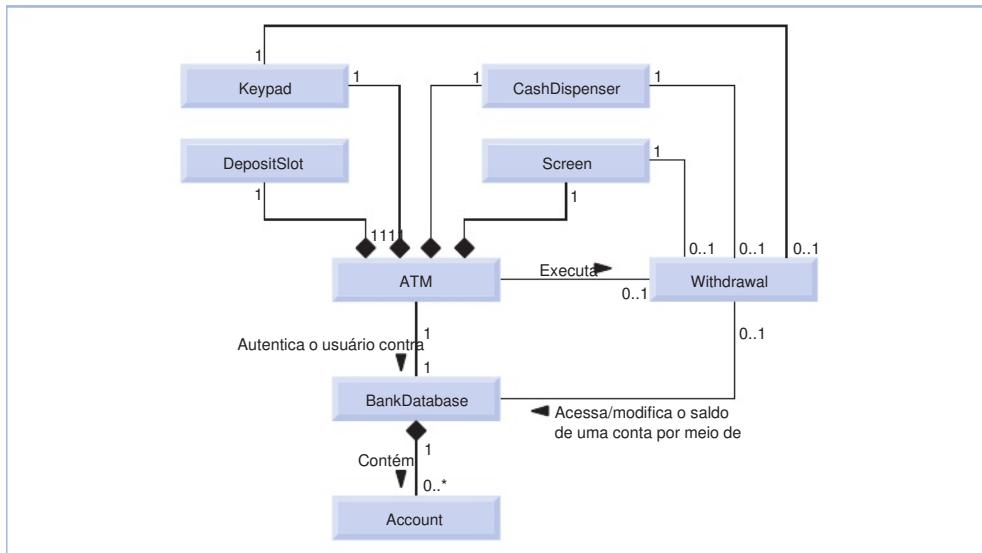


Figura 3.23 Diagrama de classes para o modelo do sistema ATM.

A classe `Deposit` associa-se com as classes `Keypad`, `DepositSlot`, `CashDispenser` e `Screen`. Semelhantemente aos saques, as transações de depósito exigem o uso da tela e do teclado para exibir prompts e receber entrada, respectivamente. Para receber envelopes de depósito, urda classe `Deposit` acessa a abertura para depósito.

e a `Autenticação`. Na Seção 6.18, determinamos se o sistema ATM terá depósitos e saques. As classes `Deposit` e `Withdrawal` são adicionadas ao diagrama de classes para examinar como o sistema muda ao longo do tempo. Na Seção 6.22, determinaremos as operações das classes em nosso sistema.

Exercícios de revisão do estudo de caso de engenharia de software

- 3.1 Suponha que tivéssemos uma classe `Car` que representasse um carro. Pense em algumas peças diferentes que um fabricante ligaria para montar um carro inteiro. Crie um diagrama de classes (semelhante ao da Figura 3.22) que modela alguns relacionamentos de composição para a classe `Car`.
- 3.2 Suponha que temos uma classe `Computer` que representa um documento eletrônico em um computador independente conectado em rede e representado pela classe `Computer`. Que tipo de associação existe entre `Computer` e `Network`?
  - a) A classe `Computer` tem um relacionamento de um para um com a classe `Network`.
  - b) A classe `Computer` tem um relacionamento de muitos para muitos com a classe `Network`.
  - c) A classe `Computer` tem um relacionamento de um para muitos com a classe `Network`.
  - d) A classe `Computer` tem um relacionamento de muitos para muitos com a classe `Network`.
- 3.3 Determine se a seguinte sentença é verdadeira ou falsa, e explique por quê: Diz-se que um diagrama UML em que o segundo e o terceiro compartimentos da classe não são modelados é um diagrama elidido.
- 3.4 Modifique o diagrama de classes da Figura 3.23 para incluir a classe `Withdrawal`.

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 3.1 [Nota: As respostas do aluno podem variar.] A Figura 3.24 apresenta um diagrama de classes que mostra alguns relacionamentos de composição de uma classe.
- 3.2 c. [Nota: Em uma rede de computadores, esse relacionamento poderia ser de muitos para muitos.] Verdadeira.
- 3.4 A Figura 3.25 apresenta um diagrama de classes para o ATM que inclui a classe `Withdrawal` (como na Figura 3.23). Observe que o ATM não acessa `CashDispenser` mas acessa `DepositSlot`.

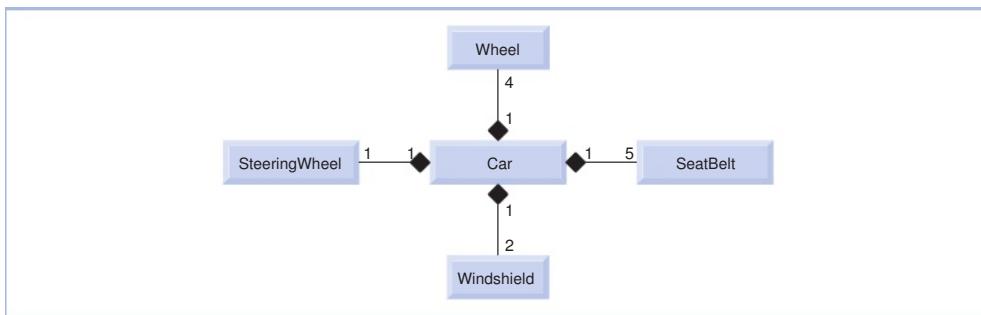


Figura 3.24 Diagrama de classes mostrando relacionamentos de composição de uma classe

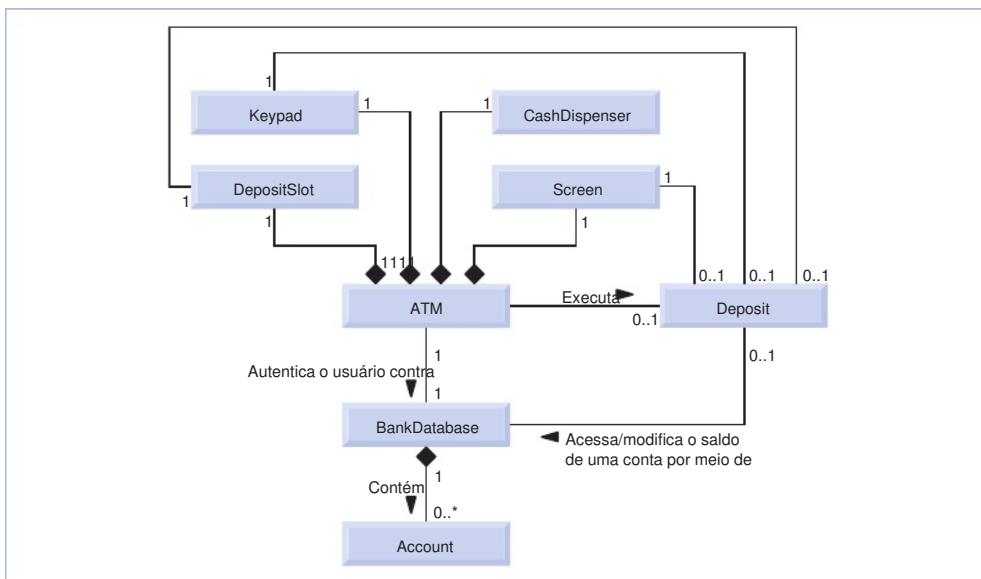


Figura 3.25 Diagrama de classes para o modelo do sistema ATM incluindo Depósito.

### 3.12 Síntese

Neste capítulo, você aprendeu a criar classes definidas pelo usuário e a criar e utilizar objetos dessas classes. Em particular, decidiu como usar membros de dados de uma classe para manter dados para cada objeto da classe. Definimos também as funções-membro que operam sobre os membros de dados. Você aprendeu a chamar as funções-membro de um objeto para solicitar os serviços que ele fornece e a passar dados para essas funções-membro como argumentos. Discutimos a diferença entre uma variável local de uma função-membro e um membro de uma classe. Também mostramos como utilizar um construtor para especificar os valores iniciais para os membros de dados de uma classe. Você aprendeu a separar a interface de uma classe de sua implementação para promover boa engenharia de software. Apresentamos também um diagrama que mostra os arquivos que os programadores da implementação da classe e os programadores do código precisam para compilar o código que eles escrevem. Demonstramos como utilizá-las para validar os dados de um objeto e assegurar que os objetos sejam mantidos em um estado consistente. Além disso, os diagramas de classes UML foram introduzidos para modelar classes e seus construtores, funções-membro e membros de dados. No próximo capítulo iniciamos nossa introdução às instruções de controle, que especificam a ordem em que as ações de uma função são realizadas.

### Resumo

- Realizar uma tarefa em um programa requer uma função. A função oculta de seu usuário as tarefas complexas que ela realiza.
- Uma função em uma classe é conhecida como uma função-membro e realiza uma das tarefas da classe.
- Você deve criar um objeto de uma classe antes de um programa realizar as tarefas que a classe descreve. Essa é uma razão pela qual C conhecido como uma linguagem de programação orientada a objetos.
- Toda mensagem enviada para um objeto é uma chamada de função-membro que instrui o objeto a realizar uma tarefa.
- Um objeto tem atributos que são portados com o objeto quando ele é utilizado em um programa. Esses atributos são especificados como membros de dados na classe do objeto.
- Uma definição de classe contém os membros de dados e funções-membro que definem os atributos e os comportamentos da classe, respectivamente.
- Uma definição de classe inicia com a palavra-chave `class` imediatamente pelo nome de classe.
- Por convenção, o nome de uma classe definida pelo usuário inicia com uma letra maiúscula e, por legibilidade, cada palavra subsequente é nome de classe inicia com uma letra maiúscula.
- O corpo de toda classe é incluído entre as chaves `{ }`  (um ponto-e-vírgula).
- As funções-membro que aparecem depois do ~~especificador de classe~~ servem para outras funções em um programa e por funções-membro de outras classes.
- Os especificadores de acesso são sempre seguidos por dois-pontos (`:`)
- A palavra-chave `new` é um tipo de retorno especial que indica que uma função realizará uma tarefa, mas não retornará nenhum dado para sua função chamadora quando completar sua tarefa.
- Por convenção, os nomes de função-membro iniciam com a primeira letra minúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula.
- Um conjunto vazio de parênteses depois do nome de uma função indica que a função não requer dados adicionais para realizar sua tarefa.
- O corpo de todas as funções é delimitado por uma chave esquerda e uma chave direita (`{ }` ).
- Em geral, você não pode chamar uma função-membro até que você crie um objeto de sua classe.
- Cada nova classe que você cria torna-se um novo tipo em C++ que pode ser utilizado para declarar variáveis e criar objetos. Essa é uma razão pela qual C++ é conhecido como uma linguagem extensível.
- ~~Chamada de função pode requerer argumentos para cada argumento possuir dados adicionais de que ele precisa para realizar sua tarefa.~~
- Uma função-membro é chamada utilizando o nome de objeto seguido ~~pelum opção de nome~~ por um conjunto de parênteses contendo os argumentos da função.
- Uma variável de ~~classe~~ da C++ Standard Library representa uma string de caracteres. Essa classe é definida no arquivo de cabeçalho `<string>` e o nome pertence ao `namespace`.
- A função `getline` (do cabeçalho `<iostream>`) lê caracteres de seu primeiro argumento até que um caractere nova linha seja encontrado, então, coloca os caracteres (não incluindo o caractere nova linha) especificados como seu segundo argumento. O caractere de nova linha é descartado.
- Uma lista de parâmetros pode conter qualquer número de parâmetros, incluindo nenhum (representado por parênteses vazios) para indicar que uma função não requer parâmetros.
- O número de argumentos em uma chamada de função deve corresponder ao número de parâmetros na lista de parâmetros do cabeçalho da função chamada. Além disso, os tipos de argumento na chamada de função devem ser consistentes com os tipos dos parâmetros correspondentes no cabeçalho de função.
- As variáveis declaradas no corpo de uma função são variáveis locais e só podem ser utilizadas a partir de sua declaração na função até a clausula `}`  imediatamente seguinte. Quando uma função termina, os valores de suas variáveis locais são perdidos.
- Uma variável local deve ser declarada antes de poder ser utilizada em uma função. Uma variável local não pode ser acessada fora da função que é declarada.
- Normalmente, os membros de dados ~~de uma classe~~ variáveis ou funções declaradas ~~que~~ são acessíveis apenas às funções-membro da classe em que elas são declaradas.
- Quando um programa cria (instancia) um objeto de uma classe, esses membros são ~~ocultos~~ no objeto e podem ser acessados apenas por funções-membro da classe do objeto.
- Quando uma função que especifica um tipo de ~~regras de herança~~ completa sua tarefa, a função retorna um resultado para sua função chamadora.

- Por padrão, o valor inicial de **string** é uma string vazia — isto é, uma string que não contém caracteres. Nada aparece na tela quando uma string vazia é exibida.
- As classes freqüentemente fornecem **funções-membro** para permitir aos clientes **acessar** ou **obter** membros de dados **privados**. Os nomes dessas funções-membro normalmente **iniciam** com **símbolos**.
- Fornecer funções **set** e **get** permite aos clientes de uma classe acessar indiretamente os dados ocultos. O cliente sabe que está tentando modificar ou obter os dados de um objeto, mas não sabe como o objeto realiza essas operações.
- As funções **set** e **get** de uma classe também devem ser utilizadas por outras funções-membro dentro da **classe** para manipular os dados da classe, embora essas funções-membro possam **acessar** diretamente. Se a representação de dados da classe é alterada, as funções-membro que acessam os dados apenas **precisarão** de **modificação** — somente os **códigos** das funções **get** e **set** que manipulam diretamente o membro de dados precisarão mudar.
- Uma função **public** deve verificar qualquer tentativa de modificação do valor de um membro de dados para assegurar que o novo valor seja apropriado àquele item de dados.
- Cada classe que você declara deve fornecer um construtor para inicializar um objeto da classe quando o objeto é criado. Um construtor é uma função-membro especial que deve ser definida com o mesmo nome da classe, de modo que o compilador possa diferenciá-lo de outras funções-membro da classe.
- Uma diferença entre construtores e funções é que os construtores não podem retornar valores, portanto não podem especificar um tipo de retorno (nem mesmo void). Normalmente, os construtores são declarados.
- O C++ requer uma chamada de construtor no momento em que cada objeto é criado, o que ajuda a assegurar que cada objeto é inicializado de ser utilizado em um programa.
- Um construtor que não aceita argumentos é um construtor-padrão. Em qualquer classe que não inclui um construtor, o compilador fornece construtor-padrão. O programador de classe também pode definir um construtor-padrão explicitamente. Se o programador definir um construtor para uma classe, o C++ não criará um construtor-padrão.
- As definições de classe, quando empacotadas adequadamente, podem ser reutilizadas por programadores em todo o mundo.
- É comum definir uma classe em um arquivo de cabeçalho que tenha uma extensão de nome do arquivo.
- Se a implementação da classe mudar, os clientes da classe não devem precisar mudar.
- As interfaces definem e padronizam as maneiras como coisas, pessoas e sistemas interagem.
- A interface de uma classe descreve as **funcões** e **membros** conhecidas como **serviços**. São disponibilizados para os clientes da classe. A interface descreve os clientes podem utilizar esses serviços, mas não especifica como a classe executa os serviços.
- Um princípio fundamental da boa engenharia de software é separar a interface da implementação. Isso torna os programas mais fáceis de modificar. Alterações na implementação da classe não afetam o cliente contanto que a interface da classe permaneça inalterada.
- Um protótipo de função contém o nome de uma função, seu tipo de retorno e o número, tipos e a ordem dos parâmetros que a função espera receber.
- Uma vez que uma classe é definida e suas funções-membro são declaradas (via protótipos de função), as funções-membro devem ser definidas em um arquivo de código-fonte separado.
- Para cada função-membro definida fora de sua definição de classe correspondente, o nome de função deve ser precedido pelo nome de classe pelo operador de resolução de escopo binário (
- A função-membro **length** da classe **string** retorna o número de caracteres em um objeto **string**.
- A função-membro **substr** da classe **string** (abreviação de 'substring') retorna um novo objeto copiando parte de um objeto **string** existente. O primeiro argumento da função especifica a posição na string a partir da qual caracteres são copiados. Seu segundo argumento especifica o número de caracteres a copiar.
- Na UML, cada classe é modelada em um diagrama de classes como um retângulo com três compartimentos. O compartimento superior contém o nome da classe, centralizado horizontalmente em negrito. O compartimento do meio contém os atributos da classe (membros de dados).
- A UML tem operações visando ao gerenciamento de classes. As operações de classe (funções-membro e construtores) em geral são adição (+) e subtração (-).
- A UML modela um parâmetro de uma operação listando o nome do parâmetro, seguido por um caractere de dois-pontos e o tipo de parâmetro entre os parênteses depois do nome de operação.
- A UML tem seus próprios tipos de dados. Nem todos os tipos de dados UML têm os mesmos nomes que os tipos C++ correspondentes. O tipo **String** corresponde ao tipo **string**.
- A UML representa os membros de classe como atributos listando o nome do atributo, seguido por um caractere de dois-pontos e o tipo de atributo. Os atributos privados são precedidos por um símbolo de subtração (-).

- A UML indica o tipo de retorno de uma operação colocando dois-pontos e o tipo de retorno depois dos parênteses que se seguem ao nome da operação.
- Os diagramas de classes UML não especificam tipos de retorno para operações que não retornam valores.
- A UML modela os construtores como operações em um terceiro compartimento do diagrama de classes. Para distinguir entre um construtor e operações de uma classe, a UML coloca a palavra 'constructor' entre aspas francesas (« e ») antes do nome do construtor.

### Terminologia

|                                                     |                                           |
|-----------------------------------------------------|-------------------------------------------|
| argumento                                           | grafia camel                              |
| arquivo de cabeçalho                                | implementação de uma classe               |
| arquivo de cabeçalho                                | instância de uma classe                   |
| arquivo de código-fonte                             | interface de uma classe                   |
| <b>atributos</b> , « e » (UML)                      | <b>atributos</b> , « e » (UML)            |
| cabeçalho de função                                 | implementação de função/membro            |
| chamada de função                                   | lista de parâmetro                        |
| chamada de função-membro                            | membro de dados                           |
| cliente de um objeto ou classe                      | mensagem (envio a um objeto)              |
| código-objeto                                       | ocultamento de dados                      |
| compartimento em um diagrama de classes (UML)       | operação (UML)                            |
| constructor                                         | operador de resolução de escopo binário ( |
| construtor-padrão                                   | operador ponto (                          |
| corpo de uma definição de classe                    | parâmetro                                 |
| definição de classe                                 | parâmetro de operação (UML)               |
| definir uma classe                                  | precisão                                  |
| diagrama de classes (UML)                           | precisão-padrão                           |
| engenharia de software                              | programador de código-cliente             |
| especificador de acesso                             | programador de implementação de classe    |
| especificador de <b>acesso</b>                      | protótipo de função                       |
| especificador de <b>abílio</b>                      | separar interface da implementação        |
| estado consistente                                  | <b>services</b> public de uma classe      |
| função chamadora (chamador)                         | sinal de subtração (UML)                  |
| função de acesso                                    | sinal de adição (UML)                     |
| função <b>get</b>                                   | string vazia                              |
| função <b>getline</b> da biblioteca <b>string</b>   | string , classe                           |
| função modificadora                                 | tipo de retorno                           |
| função <b>set</b>                                   | validação                                 |
| função-membro                                       | validade, verificação                     |
| função-membro <b>length</b> da classe <b>string</b> | variável local                            |
| função-membro <b>substr</b> da classe <b>string</b> | void, tipo de retorno                     |

### Exercícios de revisão

3.1 Preencha as lacunas em cada uma das seguintes sentenças:

- Uma casa está para uma planta arquitetônica assim como um(a) \_\_\_\_\_ está para uma classe.
- Toda definição de classe contém a palavra-chave \_\_\_\_\_ seguida imediatamente do nome da classe.
- \_\_\_\_\_ é a palavra-chave que define uma classe em um arquivo.
- Quando cada objeto de uma classe mantém sua própria cópia de um atributo, a variável que representa o atributo também é conhecida como um(a) \_\_\_\_\_.
- A palavra-chave um(a) \_\_\_\_\_.
- O tipo de retorno \_\_\_\_\_ indica que uma função realizará uma tarefa, mas não retornará nenhuma informação quando completar sua tarefa.
- A função \_\_\_\_\_ da biblioteca **string** lê caracteres até um caractere nova linha ser encontrado, então copia esses caracteres para a string especificada.

- i) Quando uma função-membro é definida fora da definição de classe, o cabeçalho de função deve incluir o nome de classe e o(a) \_\_\_\_\_, seguido(a) pelo nome de função para 'associar' a função-membro à definição de classe.
- j) O arquivo de código-fonte e quaisquer outros arquivos que utilizam uma classe podem incluir o arquivo de cabeçalho da classe via uma diretiva de pré-processador .
- 3.2 Determine se cada uma das seguintes afirmações é verdadeira ou falsa. Se falsa, explique por quê.
- Por convenção, os nomes de função iniciam com uma letra maiúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula.
  - Os parênteses vazios depois de um nome de função em um protótipo de função indicam que a função não requer parâmetros para realizar sua tarefa.
  - Os membros de dados ou funções-membro declarados compõem a especificação de classes e funções-membro da classe em que eles são declarados.
  - As variáveis declaradas no corpo de uma função-membro particular são conhecidas como membros de dados e podem ser utilizadas em outras partes do programa.
  - Onde os tipos de dados e funções-membro são definidos por uma chave esquerda e uma chave direita ( )
  - Qualquer arquivo de código-fonte pode ser utilizado para executar um programa.
  - Os tipos de argumentos em uma chamada de função devem corresponder aos tipos dos parâmetros correspondentes na lista de parâmetros do protótipo de função.
- 3.3 Qual a diferença entre uma variável local e um membro de dados?
- 3.4 Explique o propósito de um parâmetro de função. Qual a diferença entre um parâmetro e um argumento?

### Respostas dos exercícios de revisão

- 3.1 a) objeto. b) class. c) h d) tipo, nome. e) membro de dados. f) especificador de escopo. g) operador de resolução de escopo binário. h) inclui.
- 3.2 a) Falsa. Por convenção, os nomes de função iniciam com uma letra minúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula. b) Verdadeira. c) Verdadeira. d) Falsa. Essas variáveis são chamadas variáveis locais e só podem ser utilizadas nas funções-membro em que são declaradas. e) Verdadeira. f) Verdadeira. g) Verdadeira.
- 3.3 A variável local é declarada no corpo de uma função e só pode ser utilizada a partir do ponto em que é declarada até a chave direita imediatamente seguinte. Um membro de dados é declarado em uma definição de classe, mas não no corpo de qualquer das funções-membro da classe. Todo objeto (instância) de uma classe tem uma cópia separada dos membros de dados da classe. Além disso, os membros de dados são acessíveis a todas as funções-membro da classe.
- 3.4 Um parâmetro representa informações adicionais que uma função requer para realizar sua tarefa. Cada parâmetro requerido por uma função é especificado no cabeçalho de função. Um argumento é o valor fornecido na chamada de função. Quando a função é chamada, o valor de argumento é passado no parâmetro de função para que a função possa realizar sua tarefa.

### Exercícios

- 3.5 Explique a diferença entre um protótipo de função e uma definição de função.
- 3.6 O que é um construtor-padrão? Como os membros de dados de um objeto são inicializados se uma classe tiver apenas um construtor-padrão implicitamente definido?
- 3.7 Explique o propósito de um membro de dados.
- 3.8 O que é um arquivo de cabeçalho? O que é um arquivo de código-fonte? Discuta o propósito de cada.
- 3.9 Explique como um programa poderia utilizar a classe `String` para gerir uma declaração.
- 3.10 Explique por que uma classe poderia fornecer uma função para um membro de dados.
- 3.11 (Modificando a classe Book) Modifique a classe `Book` (Figuras 3.11–3.12) como mostrado a seguir:
- Inclua um segundo membro de dados que representa o nome do instrutor de curso.
  - Forneca uma função para alterar o nome do instrutor e outra para recuperá-lo.
  - Modifique o construtor para especificar dois parâmetros — um para o nome do curso e um para o nome do instrutor.
  - Modifique a função `display()` de tal maneira que ele primeiro gere a saída da mensagem de boas-vindas e o nome do curso, depois gere a saída “`is presented by:` ” seguido pelo nome do instrutor.
- Utilize sua classe modificada em um programa de teste que demonstra as novas capacidades da classe.
- 3.12 (Classe Account) Crie uma classe `Account` que um banco poderia utilizar para representar contas bancárias dos clientes. Sua classe deve incluir um membro de dados que represente o saldo. Nossos capítulos subsequentes, utilizaremos números que contêm pontos de fração decimal (por exemplo, 2,75) — chamados valores de ponto flutuante — para representar quanti-

em dólar.] Sua classe deve fornecer um construtor que recebe um saldo inicial e o utiliza para inicializar o membro de dados. O construtor deve validar o saldo inicial para assegurar que ele seja maior que ou igual a 0. Se não, o saldo deve ser configurado como 0 e o construtor deve exibir uma mensagem de erro, indicando que o saldo inicial era inválido. A classe deve fornecer três funções-membro. A função-membro `debit` deve adicionar uma quantia ao saldo atual. A função-membro `credit` deve assegurar que a quantia de débito não excede o saldo. Se exceder, o saldo deve permanecer inalterado e a função deve imprimir uma mensagem que indica que o débito excedeu o saldo. A função-membro `balance` deve retornar o saldo atual. Crie um programa que cria duas instâncias da classe e demonstra as funções-membro da classe.

- 3.13 (Classe `Invoice`) Crie uma classe `Invoice` que uma loja de suprimentos de informática possa utilizar para representar uma fatura de um item vendido na loja. A classe `Invoice` deve incluir quatro partes das informações como membros de dados — um número identificador (tipo `int`), uma descrição (tipo `string`), a quantidade comprada de um item (tipo `int`) e o preço por item (tipo `int`). [Nota: Nos capítulos subsequentes, utilizaremos números que contêm pontos de fração decimal (por exemplo, 2,75) — chamados valores de ponto flutuante — para representar quantias em dólar.] Sua classe deve ter um construtor que inicializa os quatro membros

getdados. A função-membro `getdados` deve solicitar que o usuário forneça informações sobre a fatura. A função-membro `print` deve retornar a fatura como uma string. Se a quantidade não for positiva, ela deve ser zero. Se o preço por item não for positivo, ele deve ser configurado como zero. Escreva um programa de teste que demonstre as capacidades da classe.

- 3.14 (Classe `Employee`) Crie uma classe `Employee` que inclua três partes de informações como membros de dados — um nome (tipo `string`), um sobrenome (tipo `string`) e um salário mensal (tipo `float`). Nos capítulos subsequentes, utilizaremos números que contêm pontos de fração decimal (por exemplo, 2,75) — chamados valores de ponto flutuante — para representar quantias em dólar. Sua classe deve ter um construtor que inicialize os três membros desse tipo. Forneça uma função-membro `print` que exiba o nome e o salário de cada objeto. Escreva um programa de teste que demonstre as capacidades da classe `Employee`. Crie dois objetos `Employee` e exiba o salário de cada objeto. Então incremente o salário de 10% e exiba novamente o salário anual de cada.

- 3.15 (Classe `Date`) Crie uma classe `Date` que inclua três partes de informações como membros de dados — um mês (tipo `dia` (tipo `int`)) e um ano (tipo `int`). Sua classe deve ter um construtor com três parâmetros que utilize os parâmetros para inicializar os três membros de dados. Para o propósito desse exercício, assuma que os valores fornecidos para o ano e o dia são corretos, mas certifique-se de que o valor de mês esteja no intervalo 1–12; se não estiver, configure o mês como 1. Forneça uma função-membro `print` que exiba o dia, o mês e o ano separados por barras normais (/). Escreva um programa de teste que demonstre as capacidades da classe.



Vamos todos dar um passo para a frente.  
Lewis Carroll

A roda já deu uma volta completa.  
William Shakespeare

Quantas maçãs não caíram na cabeça de Newton antes de ele ter percebido a dica!  
Robert Frost

Toda a evolução que conhecemos procede do vago para o definido.  
Charles Sanders Peirce

## Instruções de controle: parte 1

### OBJETIVOS

Neste capítulo, você aprenderá:

Técnicas básicas para solução de problemas.

A desenvolver algoritmos por meio do processo de refinamento passo a passo de cima para baixo.

A utilizar as instruções de seleção `if ...else` para escolher entre ações alternativas.

Como utilizar a instrução de repetição `while` para executar instruções em um programa repetidamente.

Repetição controlada por contador e repetição controlada por sentinelas.

Como utilizar os operadores de incremento, decremento e atribuição.

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P<br>á<br>m<br>S | <ul style="list-style-type: none"> <li>4.1 Introdução</li> <li>4.2 Algoritmos</li> <li>4.3 Pseudocódigo</li> <li>4.4 Estruturas de controle</li> <li>4.5 Instrução de seleçãoif</li> <li>4.6 A instrução de seleção duplaif ...else</li> <li>4.7 A instrução de repetiçãowhile</li> <li>4.8 Formulando algoritmos: repetição controlada por contador</li> <li>4.9 Formulando algoritmos: repetição controlada por sentinelas</li> <li>4.10 Formulando algoritmos: instruções de controle aninhadas</li> <li>4.11 Operadores de atribuição</li> <li>4.12 Operadores de incremento e decremento</li> <li>4.13 Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional)</li> <li>4.14 Síntese</li> </ul> |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

## 4.1 Introdução

Antes de escrever um programa para resolver um problema, devemos ter um entendimento completo do problema e uma abordagem cuidadosamente planejada para resolvê-lo. Ao escrever um programa, também devemos entender os tipos de blocos de construção que estão disponíveis e empregar técnicas comprovadas de construção de programas. Neste capítulo e no Capítulo 5, “Instruções de controle de parte 2”, discutimos essas questões na apresentação da teoria e princípios da programação estruturada. Os conceitos apresentados são cruciais para construir classes eficientes e manipular objetos.

Neste capítulo introduzimos as instruções `while` do C++, três dos blocos de construção que permitem aos programadores especificar a lógica requerida para que funções-membro realizem suas tarefas. Dedicamos uma parte deste capítulo a discussões sobre outras maneiras de combinar instruções de controle para resolver um problema semelhante. Introduzimos operadores de atribuição do C++ e exploramos os operadores de incremento e decremento do C++. Esses operadores adicionais simplificam muitas instruções de programa.

## 4.2 Algoritmos

Qualquer problema de computação solucionável pode ser resolvido pela execução de uma série de ações em uma ordem específica para resolver um problema em termos de

1. ações a executar e
2. ordem em que essas ações executam

é chamado **algoritmo**. O exemplo a seguir demonstra que é importante especificar corretamente a ordem em que as ações executam.

Considere o ‘algoritmo cresça e brilhe’ seguido por um executivo júnior para sair da cama e ir trabalhar: (1) levantar-se da cama, (2) tirar o pijama, (3) tomar banho, (4) vestir-se, (5) tomar o café da manhã, (6) dirigir o carro até o trabalho. Essa rotina leva o executivo bem preparado para tomar decisões críticas. Suponha que os mesmos passos sejam seguidos em uma ordem um pouco diferente: (1) levantar-se de cama, (2) tirar o pijama, (3) vestir-se, (4) tomar banho, (5) tomar o café da manhã, (6) dirigir o carro até o trabalho.

Nesse segundo caso, o executivo provavelmente não vai sair de casa. Especificar a ordem em que as ações (ações) são executadas é crucial para o sucesso de um algoritmo.

## 4.3 Pseudocódigo

Pseudocódigo (ou código ‘fictício’) é uma linguagem artificial e informal que ajuda os programadores a desenvolver algoritmos sem preocupação com os rigorosos detalhes da sintaxe de linguagem C++. O pseudocódigo que apresentamos aqui é particularmente útil para desenvolver algoritmos que serão convertidos em partes estruturadas de programas C++. O pseudocódigo é similar à língua C e é conveniente e amigável ao usuário embora não seja uma linguagem de programação de computador real.

O pseudocódigo não é executado nos computadores. Mais exatamente, ele ajuda o programador a ‘estudar’ um programa, tentar escrevê-lo em uma linguagem de programação como C++. Este capítulo fornece vários exemplos de como utilizar o pseudocódigo para desenvolver programas C++.

O estilo do pseudocódigo que apresentamos consiste puramente em caracteres, de modo que os programadores possam o pseudocódigo convenientemente, utilizando um programa editor qualquer. O computador pode produzir uma nova cópia independente de um programa em pseudocódigo sob demanda. Um programa em pseudocódigo cuidadosamente preparado pode ser convertido em um programa C++ correspondente. Em muitos casos, isso simplesmente requer a substituição de instruções e código por equivalentes em C++.

O pseudocódigo normalmente ~~descreve apenas~~<sup>descreve</sup> ações que fazem com que ações específicas ocorram depois que um programador converte um programa do pseudocódigo em C++ e o programa é executado em um computador. As declarações têm inicializadores ou não envolvem chamadas de construtor) não são instruções executáveis. Por exemplo, a declaração

```
int i;
```

informa ao compilador o tipo de variável a reservar espaço na memória para a variável. Essa declaração não faz com que qualquer ação — como entrada, saída ou cálculo — ocorra quando o programa é executado. Em geral, não incluímos declarações de variáveis no nosso pseudocódigo. Entretanto, alguns programadores optam por listar as variáveis e mencionar seus propósitos dos programas em pseudocódigo.

Agora examinamos um exemplo em pseudocódigo que pode ser escrito para ajudar um programador a criar o programa de a Figura 2.5. Esse pseudocódigo (Figura 4.1) corresponde ao algoritmo que insere dois inteiros a partir do usuário, adiciona esse e exibe sua soma. Embora mostremos a listagem completa em pseudocódigo aqui, mostraremos como criar pseudocódigo a declaração de um problema mais adiante no capítulo.

As linhas 1–2 correspondem às instruções nas linhas 13–14 da Figura 2.5. Note que as instruções em pseudocódigo são simplesmente instruções em linguagem natural que expressam qual é a tarefa a ser realizada em C++. Da mesma forma, as linhas 4–5 correspondem às instruções nas linhas 16–17 da Figura 2.5 e as linhas 7–8 correspondem às instruções nas linhas 19 e 21 da Figura 2.5.

Há alguns aspectos importantes do pseudocódigo na Figura 4.1. Note que o pseudocódigo corresponde ao código somente da main(). Isso ocorre porque o pseudocódigo é normalmente utilizado para algoritmos, não para programas completos. Nesse caso, o pseudocódigo é utilizado para representar o algoritmo. A função em que esse código é colocado não é importante para o algoritmo. Pela mesma razão, a linha 23 da Figura 2.5 (~~a não é incluída no pseudocódigo — essa declaração no final de cada função não é importante para o algoritmo~~)

~~— essa declaração no final de cada função não é importante para o algoritmo~~

porque essas declarações de variável não são instruções executáveis.

Normalmente, instruções em um programa são executadas uma após a outra na ordem ~~em que~~<sup>em que</sup> são escritas. Isso é chamado de sequência. Várias instruções C++ que discutiremos em breve permitem ao programador especificar que a próxima instrução a ser executada deve ser diferente da próxima instrução na sequência. Isso é chamado de transferência de controle.

Durante a década de 1960 tornou-se claro que a utilização indiscriminada de transferências de controle era a raiz de muitas dades experimentadas por grupos de desenvolvimento de software. A culpa pertence ao programador que especificava uma transferência de controle para um de vários possíveis destinos em um programa (criando o que é freqüentemente ‘código espaguete’). A noção de programação estruturada tornou-se quase sinônimo de ‘eliminação do

A pesquisa de Böhm é uma prova que poderiam ser escritos programas sem transferências de controle. Isso levou a mudanças de estilo para programadores mudar seus estilos para ‘programação estruturada’. Mas foi só a segunda metade de 1970 que os programadores começaram a tratar programação estruturada seriamente. Os resultados foram impressionantes, pois grupos de desenvolvimento informaram que tinham reduzido o tempo de desenvolvimento, que a entrega de sistemas ocorria com mais freqüência dentro de prazos.

- 1 Solicite que o usuário insira o primeiro inteiro
- 2 Insira o primeiro inteiro
- 3
- 4
- 5 Solicite que o usuário insira o segundo inteiro
- 6
- 7 Some o primeiro e o segundo inteiros, armazene o resultado
- 8 Exiba o resultado

Figura 4.1 Pseudocódigo para o programa de adição da Figura 2.5.

<sup>1</sup> Böhm, C., & G. Jacopini, “Flow diagrams, turing machines, and languages with only two formation rules”, No. 5, maio de 1966, p. 366–371.

que a conclusão do projeto de software ocorria com mais freqüência dentro do orçamento. A chave para esses sucessos é que os estruturados são mais claros, mais fáceis de depurar, testar e modificar e menos propensos a conter bugs.

O trabalho de Böhm e Jacopini demonstrou que todos os programas poderiam ser escritos<sup>2</sup> em termos de somente três estruturas de controle: estrutura de seleção, estrutura de repetição. O termo ‘estruturas de controle’ vem do campo da ciência da computação. Quando introduzirmos as implementações de estruturas de controle do C++, iremos nos eles seguindo a terminologia do documento ‘padrões de controle’.

#### Estrutura de seqüência em C++

A estrutura de seqüência é construída no C++. A menos que instruído de outra maneira, o computador executa as instruções C depois da outra na ordem em que elas são escritas — isto é, em seqüência. Modeling Language (UML) da Figura 4.2 ilustra uma típica estrutura de seqüência em que dois cálculos são realizados na ordem. O C++ permite ter ações quaisquer em uma estrutura de seqüência. Como veremos logo, uma única ação pode ser colocada em qualquer lugar e também podemos colocar várias ações em seqüência.

Tais estruturas podem aparecer em muitas linguagens de programação, dentro de laços ou condicionais de variáveis. Para calcular uma média, o total cuja média está sendo calculada é dividido pelo número de notas. Uma variável contadora seria utilizada para monitorar o número de valores cuja média está sendo calculada. Você verá instruções semelhantes no programa da Seção 4.8.

Os diagramas de atividades são parte da UML. Um diagrama de atividades é também chamado de modelo de trabalho de uma parte de um sistema de software. Esses fluxos de trabalho podem incluir uma parte de um algoritmo, como a estrutura de seqüência na Figura 4.2. Os diagramas de atividades são compostos de símbolos de uso especial, como (um retângulo com seus lados esquerdo e direito substituídos por arcos que se curvam para trás), símbolos são conectados por transições que representam o fluxo da atividade.

Como ocorre com o pseudocódigo, diagramas de atividades ajudam os programadores a desenvolver e representar algoritmos embora muitos programadores prefiram o pseudocódigo. Os diagramas de atividades mostram claramente como operam as estruturas de controle.

Considere o diagrama de atividades de estrutura de seqüência da Figura 4.2. Ele representa as ações a realizar. Cada estado da ação contém uma ação, por exemplo, “adicionar grade a total” ou “adicionar 1 a counter” — que especifica uma ação particular a realizar. Outras ações poderiam incluir cálculos ou operações de entrada e saída. As transições entre os estados representam em que ocorrem as ações representadas pelos estados de ação — o programa que implementa as atividades ilustradas pelo diagrama de atividades é, então, adiciona a total, e, então, adiciona counter.

O círculo sólido localizado na parte superior do diagrama de atividades representa a ação inicial — o início do fluxo de trabalho antes de o programa realizar as atividades modeladas. O círculo sólido cercado por um círculo vazio que aparece na parte final do diagrama de atividades representa o estado final — o fim do fluxo de trabalho depois que o programa realiza suas atividades.

A Figura 4.2 também inclui retângulos com os cantos superiores direitos dobrados. As ações são chamadas de observações explicatórias que descrevem o propósito dos símbolos no diagrama. As notas podem ser utilizadas em qualquer parte da UML — não só em diagramas de atividades. A Figura 4.2 utiliza notas da UML para mostrar o código C++ associado ao estado de ação no diagrama de atividades. A nota conecta cada nota ao elemento que a nota descreve. Os diagramas de atividades normalmente não mostram o código C++ que implementa a atividade. Aqui, utilizamos notas para esse propósito a fim de ilustrar como os diagramas se relacionam ao código C++. Para informações adicionais sobre a UML, veja nosso estudo de caso opcional, que nas seções “Estudo de caso de engenharia de software” no final dos capítulos 7, 9, 10, 12 e 13 ou visite

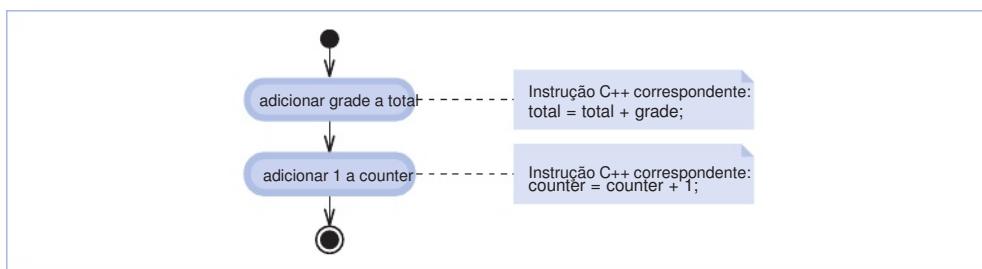


Figura 4.2 Diagrama de atividades da estrutura de seqüência.

<sup>2</sup> Esse documento é mais especificamente intitulado UML Activity Diagrams Version 2.0, IEC/ISO/IEEE 19758-2:2003 Programming languages – e está disponível para download (por uma taxa) em [www.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+14882%2D2003](http://www.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+14882%2D2003)

### Instruções de seleção em C++

O C++ fornece três tipos de instruções de seleção (discutidos neste capítulo e no Capítulo 5): A instrução de seleção `if` (Capítulo 5) se uma ação se uma condição (predicado) for verdadeira, ou pula a ação se a condição for falsa. A instrução de seleção `else` realiza uma ação se uma condição for verdadeira, ou realiza uma ação diferente se a condição for falsa. A instrução de seleção `switch` (Capítulo 5) realiza uma de muitas ações diferentes, dependendo do valor de uma expressão do tipo inteiro.

A instrução de seleção `if` instruções de uma única seleção porque seleciona ou ignora uma única ação (ou, como veremos a seguir, um único grupo de ações). A instrução `else` instruções de seleção dupla porque seleciona entre duas ações diferentes (ou grupos de ações). A instrução de seleção `switch` instruções de seleção múltipla uma vez que seleciona entre muitas ações diferentes (ou grupos de ações).

### Instruções de repetição em C++

O C++ fornece três tipos de instruções de repetição (também chamadas de loops) que permitem aos programas realizar instruções repetidamente se uma condição (chamada de loop) permanecer verdadeira. As instruções

de repetição são as seguintes: `while`, `for` (Capítulo 5 apresenta mais detalhes). As instruções `for` e `while` realizam a ação (ou grupo de ações) no seu corpo uma vez ou mais. Se a condição de continuação de loop for iniciada falsa, a ação (ou grupo de ações) não será executada. A instrução `do` é similar ao `while` (ou grupo de ações) no seu corpo pelo menos uma vez.

Cada uma das palavras-chave `switch`, `while`, `do` e `for` é uma palavra-chave C++. Essas palavras são reservadas pela linguagem de programação C++ para implementar vários recursos, como instruções de controle do C++. As palavras-chave não devem ser usadas como identificadores, como nomes de variável. A Figura 4.3 contém uma lista completa de palavras-chave do C++.



### Erro comum de programação 4.1

Utilizar uma palavra-chave como um identificador é um erro de sintaxe.

### Palavras-chave do C++

#### Palavras-chave comuns às linguagens de programação C e C++

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

#### Palavras-chave do C++ somente

|          |              |          |                  |             |
|----------|--------------|----------|------------------|-------------|
| and      | and_eq       | asm      | bitand           | bitor       |
| bool     | catch        | class    | compl            | const_cast  |
| delete   | dynamic_cast | explicit | export           | false       |
| friend   | inline       | mutable  | namespace        | new         |
| not      | not_eq       | operator | or               | or_eq       |
| private  | protected    | public   | reinterpret_cast | static_cast |
| template | this         | throw    | true             | try         |
| typeid   | typename     | using    | virtual          | wchar_t     |
| xor      | xor_eq       |          |                  |             |

Figura 4.3 Palavras-chave do C++.



### Erro comum de programação 4.2

Escrever uma palavra-chave com alguma letra maiúscula é um erro de sintaxe. Todas as palavras-chave do C++ têm apenas letras minúsculas.

Resumo de instruções de controle em C++

O C++ contém somente três tipos de estruturas de controle, que daqui para a frente chamaremos instruções de controle: a instância de sequência, instruções de seleção (três tipos: if, switch) e instruções de repetição (três tipos: for, while).

Todo programa C++ combina quantas dessas instruções de controle for apropriado para o algoritmo que o programa implementa. Ocorre com a instrução de sequência da Figura 4.2, você pode modelar cada instrução de controle como um diagrama de atividade. Cada diagrama contém um estado inicial e um estado final, que representam um ponto de entrada e um ponto de saída da instrução de controle, respectivamente. **Essas instruções de controle de entrada única/saída facilitam a construção de programas** — as instruções de controle são vinculadas conectando-se o ponto de saída de uma instrução ao ponto de entrada da seguinte. Isso é

Aprendendo como se encaixa uma lógica de controle dentro de outra, torna-se mais fácil entender o que é uma instrução de controle — chamada de **instrução de controle** em que uma instrução de controle está contida dentro de outra. Portanto, algoritmos nos programas C++ são construídos a partir de apenas três tipos de instruções de controle, combinadas apenas de duas maneiras. Isso é a essência da simplicidade.



### Observação de engenharia de software 4.1

Qualquer programa C++ que criemos pode ser construído a partir de apenas sete tipos de instruções de controle diferentes (sequência, if, ...else, switch, do...while, for) combinados apenas de duas maneiras (empilhamento de instruções de controle e aninhamento de instruções de controle).

## 4.5 Instrução de seleção if

Os programas utilizam instruções de seleção para escolher entre cursos alternativos de ações. Por exemplo, suponha que a aprovação de um exame seja 60. A instrução em pseudocódigo

```
Se a nota do aluno for maior que ou igual a 60
 Imprima 'Passed'
```

determina se a condição 'a nota do aluno for maior que ou igual a 60' é verdadeira (verdadeira) ou falsa (falsa). Se a condição for

Passado é verdadeiro, a instrução é executada. Se a condição for falsa, a instrução é ignorada. A instrução é ignorada se a condição não é realizada. Observe que a segunda linha dessa instrução de seleção está recuada. Esse recuo é opcional, mas é recomendado para enfatizar a estrutura inerente de programas estruturados. Quando você converte o pseudocódigo em código C++, o compilador C++ ignora todos os caracteres de espaço em branco (como espaços em branco, tabulações e nova linha) utilizados para recuo e espaçamento ver



### Boa prática de programação 4.1

Aplicar consistentemente convenções razoáveis de recuo ao longo de todos os seus programas melhora significativamente a legibilidade do programa. Sugerimos três espaços por recuo. Algumas pessoas preferem utilizar tabulações, mas estas podem variar de um editor para outro, fazendo com que um programa escrito em um editor seja alinhado diferentemente quando utilizado com outro.

A instrução de pseudocódigo precedente pode ser escrita em C++ como

```
if (grade >= 60)
 cout << "Passed";
```

Note que o código C++ apresenta uma íntima correspondência com o pseudocódigo. Essa é uma das propriedades do pseudocódigo que torna essa ferramenta de desenvolvimento de programas tão útil.

A Figura 4.4 ilustra a **instrução de seleção** única seleção. Ele contém o que talvez seja o símbolo mais importante em um diagrama de atividades — o losango ou decisão que indica que uma decisão deve ser tomada. O símbolo de decisão indica que o fluxo

de trabalho continuará por um caminho determinado pelo símbolo associado às quais podem ser verdadeiras ou falsas. Cada seta de transição que sai de um símbolo de decisão tem uma condição de guarda (especificada entre colchetes ao lado da seta de transição). Se uma condição de guarda for verdadeira, o fluxo de trabalho entra no estado de ação para o qual a seta aponta. Na Figura 4.4, se a nota for maior que ou igual a 60, o programa imprime 'Passed' na tela, e, em seguida, para o estado final dessa atividade. Se a nota for menor que 60, o programa se dirige imediatamente para o estado final sem essa mensagem.

Aprendemos no Capítulo 1 que as decisões podem ser baseadas em condições contendo operadores relacionais ou de igualdade. Em C++, uma decisão pode ser baseada em qualquer expressão — se a expressão é avaliada como zero, ela é tratada como falsa. Se a expressão é avaliada como não-zero, é tratada como verdadeira. O comando `if` parece dizer que podemos armazenar apenas os valores `true` e `false` — cada um deles é uma palavra-chave C++.

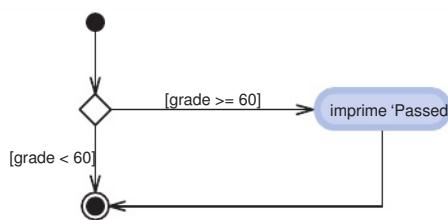


Figura 4.4 Diagrama de atividades de uma instrução de seleção única

**Dica de portabilidade 4.1**

Para compatibilidade com versões anteriores do C, que utilizavam inteiros para valores booleanos, também pode ser representado por qualquer valor não-zero (compiladores normalmente utilizam 1) ou false (que também pode ser representado como o valor zero).

Observe que a instrução é uma instrução de entrada única/saída única. Veremos que os diagramas de atividades para as instruções de controle restantes também contêm estados iniciais, setas de transição, estados de ação que indicam ações a realizar, símbolos (com condições de guarda associadas) que indicam decisões a serem tomadas e estados finais. Isso é consistente com o princípio de programação que temos enfatizado.

Podemos imaginar sete contêineres, cada um contendo somente diagramas de atividades UML vazios de um dos sete tipos de controles de controle. A tarefa do programador é, portanto, montar um programa dos diagramas de atividades de cada tipo de controle que o algoritmo demanda, combinar os diagramas de atividades de apenas duas maneiras possíveis (empilhar aninhamento) e, então, preencher os estados de ação e decisões com expressões de ação e condições de guarda de maneira para formar uma implementação estruturada para o algoritmo. Discutiremos agora a variedade de maneiras em que as ações podem ser escritas.

**A instrução de uma única seleção realiza uma ação quando a condição é verdadeira; quando a condição é falsa, a ação é pulada.** A instrução de seleção duplaif permite que o programador especifique uma ação a realizar, quando a condição é diferente de realizar quando a condição é verdadeira. Por exemplo, a instrução em pseudocódigo

Se a nota do aluno for maior que ou igual a 60

    Imprima 'Passed'

Caso contrário

    Imprima 'Failed'

imprime 'Passed' se a nota do aluno for maior que ou igual a 60 e imprime 'Failed' se a nota do aluno for menor que 60. Nos dois casos que ocorre a impressão, a próxima instrução em pseudocódigo na sequência é realizada.

A instrução Caso contrário do pseudocódigo precedente pode ser escrita em C++ como

```

if (grade >= 60)
 cout << "Passed";
else
 cout << "Failed";

```

Observe que o corpo também é recuado. Qualquer que seja a convenção de recuo que você escolher, deve aplicá-la consistente por todos os seus programas. É difícil ler programas que não obedecem às convenções de espaçamento uniforme.

**Boa prática de programação 4.2**

Recue as duas instruções do corpo de uma instrução.

**Boa prática de programação 4.3**

Se existem vários níveis de recuo, cada nível deve ser recuado pela mesma quantidade adicional de espaço.

A Figura 4.5 ilustra o fluxo de controle na instrução. Mais uma vez, observe que (além do estado inicial, setas de transição e estado final) os únicos outros símbolos no diagrama de atividades representam estados de ação e decisões. Continuaremos a

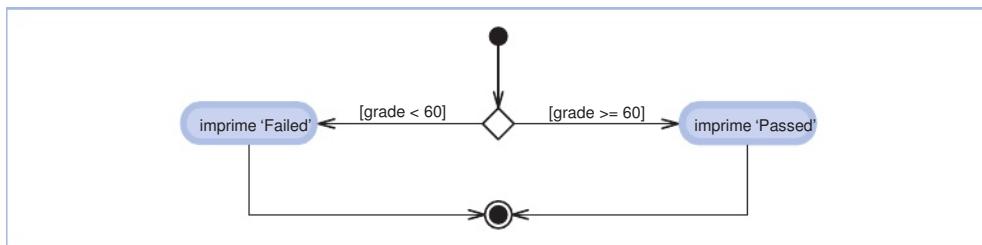


Figura 4.5 Diagrama de atividades de uma instrução de seleção dupla.

esse modelo de computação de ação/decisão. Imagine novamente um contêiner profundo de diagramas de atividades UML vazias instruções de seleção dupla — tantas quantas o programador pode precisar para empilhar e aninhar com os diagramas de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.

Operador condicional? (

O C++ fornece o operador condicional?:, que está intimamente relacionado com a instrução if. Operador condicional é o único operador ternário do C++ — ele aceita três operandos. Os operandos, juntos com o operador condicional, formam uma condição. O primeiro operando é uma condição, o segundo é o valor para a expressão condicional inteira se a condição for verdadeira, e o terceiro é o valor para a expressão condicional inteira se a condição for falsa.

`cout << ( grade >= 60 ? "Passed": "Failed" );`

contém uma expressão condicional? "Passed": "Failed", que é avaliada como `Passed` se a condição de `>= 60` for true, mas é avaliada como `Failed` se a condição for false. Portanto, a instrução com o operador condicional realiza essencialmente a mesma coisa que a instrução `if`. Como veremos, a precedência do operador condicional é baixa, portanto os parênteses na expressão anterior são requeridos.



#### Dica de prevenção de erro 4.1

Para evitar problemas de precedência (e por clareza), coloque as expressões condicionais (que aparecem em expressões maiores) entre parênteses.

Os valores em uma expressão condicional também podem ser ações a executar. Por exemplo, a seguinte expressão condicional também imprime "OU Failed":

`grade >= 60 ? cout << "Passed": cout << "Failed" ;`

A expressão condicional precedente é idêntica à seguinte: "Se a nota for maior ou igual a 60, imprima 'Passed'; caso contrário, imprima 'Failed'." Essa, também, é comparável à instrução `if`. As expressões condicionais podem aparecer em algumas localizações do programa onde as instruções `if` não podem.

Instruções `if ...else` aninhadas

As instruções `if ...else` aninhadas testam múltiplos casos colocando instruções de seleção dentro de outras instruções de seleção `if ...else`. Por exemplo, a seguinte instrução pseudocódigo imprime notas de exame maiores que ou igual a 90, B para notas no intervalo [90, 100], A para 80, F para 70, D para 60, C para 50 e E para 40:

Se a nota do aluno for maior que ou igual a 90

    Imprima 'A'

Caso contrário

    Se a nota do aluno for maior que ou igual a 80

        Imprima 'B'

    Caso contrário

        Se a nota do aluno for maior que ou igual a 70

            Imprima 'C'

        Caso contrário

            Se a nota do aluno for maior que ou igual a 60

                Imprima 'D'

            Caso contrário

                Imprima 'F'

Esse pseudocódigo pode ser escrito em C++ como

```
if (studentGrade >= 90) // 90 e acima obtém "A"
 cout << "A";
else
 if (studentGrade >= 80) // 80-89 obtém "B"
 cout << "B";
 else
 if (studentGrade >= 70) // 70-79 obtém "C"
 cout << "C";
 else
 if (studentGrade >= 60) // 60-69 obtém "D"
 cout << "D";
 else // menor que 60 obtém "F"
 cout << "F".
```

Se `studentGrade` for maior que ou igual a 90, as primeiras quatro condições da instrução `if` depois do primeiro teste executará. Depois que o programa pula para a estrutura `else` 'mais externa'. A maioria dos programadores em C++ prefere escrever a mesma forma como

```
if (studentGrade >= 90) // 90 e acima obtém "A"
 cout << "A";
else if (studentGrade >= 80) // 80-89 obtém "B"
 cout << "B";
else if (studentGrade >= 70) // 70-79 obtém "C"
 cout << "C";
else if (studentGrade >= 60) // 60-69 obtém "D"
 cout << "D";
else // menor que 60 obtém "F"
 cout << "F";
```

As duas formas são idênticas, exceto quanto ao espaçamento e recuo, que o compilador ignora. A última forma é popular por grande recuo de código para a direita. Esse recuo freqüentemente deixa pouco espaço em uma linha, forçando muitas quebras e diminuindo a legibilidade do programa.



#### Dica de desempenho 4.1

Uma instrução `if ...else` aninhada pode executar com muito mais rapidez que uma série de `if ...else`. Isso ocorre porque a possibilidade de saída prévia depois de uma das condições ser satisfeita.



#### Dica de desempenho 4.2

Em uma instrução `if ...else` aninhada, teste as condições que têm maior probabilidade de serem satisfeitas no começo da instrução.. Isso permitirá que a instrução `else` aninhada execute mais rapidamente e saia mais cedo do que ao testar primeiro os casos que ocorrem raramente.

O problema do `else` oscilante

O compilador C++ sempre associa a instrução `else` imediatamente anterior, a menos que instruído de outro modo pela colocação de chaves `{}`. Esse comportamento pode levar àquilo que é chamado de `else` oscilante. Por exemplo,

```
if (x > 5)
 if (y > 5)
 cout << "x and y are > 5" ;
```

```
else cout << "x is <= 5" ;
```

parece indicar que se `x` for maior do que `y`, a instrução aninhada determinará se `y` também é maior do que `x`. Se `y` for menor ou igual a `x`, a saída "x is <= 5" é gerada.

Cuidado! Essa instrução aninhada não é executada como parece. Na verdade, o compilador interpreta a instrução como

```
if (x > 5)
 if (y > 5)
 cout << "x and y are > 5" ;
 else
 cout << "x is <= 5" ;
```

em que o corpo do `if` é aninhado. A instrução `else` é maior do que `for`, a execução continuará testando se a segunda condição for verdadeira, a string adequada é exibida. Entretanto, se a segunda condição for falsa, é exibida, apesar de saber que

Para forçar a instrução aninhada a executar como foi originalmente concebida, devemos escrevê-la como a seguir:

```
if (x > 5)
{
 if (y > 5)
 cout << "x and y are > 5" ;
}
else
 cout << "x is <= 5" ;
```

As chaves `{ }`  indicam ao compilador que a segunda instrução no corpo da `if` está associado com a primeira. Os exercícios 4.23 e 4.24 investigam a possibilidade mais detalhadamente.

#### Blocos

A instrução de seleção `if` originalmente espera somente uma instrução no seu corpo. De maneira semelhante, as partes `if` e `else` esperam apenas uma instrução de corpo. Para incluir várias instruções no corpo de uma parte de uma `if` ou `else`, coloque as instruções entre chaves em conjunto de instruções entre chaves, chamado de bloco. Utilizamos o termo ‘bloco’ deste ponto em diante.



#### Observação de engenharia de software 4.2

Um bloco pode ser colocado em qualquer lugar em um programa em que uma única instrução pode ser colocada.

O exemplo a seguir inclui um bloco na `if` instrução:

```
if (studentGrade >= 60)
 cout << "Passed.\n";
else
{
 cout << "Failed.\n";
 cout << "You must take this course again.\n" ;
```

Nesse caso, se `studentGrade` é menor que 60, o programa executa ambas as instruções no corpo do `else`:

Failed.  
You must take this course again.

Note as chaves que cercam as duas instruções. Essas chaves são importantes. Sem as chaves, a instrução

`cout << "You must take this course again.\n" ;` ficaria fora do corpo da `else` e executaria independentemente de a nota ser ou não menor que 60. Esse é um exemplo de um erro de lógica.



#### Erro comum de programação 4.3

Esquecer uma ou ambas as chaves que delimitam um bloco pode levar a erros de sintaxe ou erros de lógica em um programa.



#### Boa prática de programação 4.4

Colocar sempre as chaves em uma instrução (ou qualquer instrução de controle) ajuda a evitar sua omissão acidental, especialmente ao adicionar instruções a uma cláusula `else` mais tarde. Para evitar omitir uma ou as duas chaves, alguns programadores preferem digitar as chaves de abertura e fechamento de blocos mesmo antes de digitar as instruções individuais dentro das chaves.

Exatamente como um bloco, uma instrução única pode ser colocada em qualquer lugar; também é possível não ter nenhuma ir — isso é chamado de `instrução nula` (ou `instrução vazia`). A instrução nula é representada colocando-se um ponto-e-vírgula (`;`) normalmente entraria uma instrução.



#### Erro comum de programação 4.4

Colocar um ponto-e-vírgula depois da condição em uma `if` causa um erro de lógica em instruções de uma única seleção `if` e a um erro de sintaxe em instruções de seleção dupla (quando a parte contém uma instrução de corpo real).

## 4.7 A instrução de repetição while

Uma **instrução de repetição** (também chamada **instrução de loop** ou simplesmente **loop**) permite ao programador especificar que um programa deve repetir uma ação enquanto alguma condição permanecer verdadeira. A instrução em pseudocódigo

```
Enquanto houver mais itens em minha lista de compras
 Comprar o próximo item e riscá-lo da minha lista
```

descreve a repetição que ocorre durante um passeio de compras. A condição, ‘enquanto houver mais itens em minha lista’ pode ser verdadeira ou falsa. Se ela for verdadeira, então a ação ‘Comprar o próximo item e riscá-lo da minha lista’ é realizada. A ação será realizada repetidamente enquanto a condição permanecer verdadeira. A instrução **while** na instrução de repetição constitui o **corpo** que pode ser uma instrução única ou um bloco. Por fim, a condição se tornaria falsa (quando o último item na lista de compras foi comprado e riscado da lista). Nesse ponto, a repetição termina e a primeira instrução em pseudocódigo depois da instrução de repetição é executada.

Como um exemplo da instrução de repetição considere um segmento de programa projetado para localizar a primeira repetição de seguir quando o valor de **product** é maior ou igual a 100. Quando a instrução de

```
int product = 3;
```

```
while (product <= 100)
 product = 3 * product;
```

Quando a instrução inicia a execução, o valor é 3. Cada repetição da instrução multiplica o produto por 3, então **product** assume os valores 9, 27, 81 e 243, sucessivamente. Quando **product** torna-se **<= 100**, a condição da instrução **<= 100** — torna-se **false**. Isso termina a repetição, portanto o valor é 243. Nesse ponto, a execução de programa continua com a próxima instrução depois da instrução

### Erro comum de programação 4.5

Não fornecer, no corpo de uma instrução, uma ação que consequentemente faz com que a condição se torne falsa normalmente resulta em um erro de lógica chamado **loop infinito**, qual a instrução de repetição nunca termina. Isso pode fazer um programa parecer ‘travado’ ou ‘congelado’ se o corpo do loop não contiver instruções que interagem com o usuário.

O diagrama de atividades UML da Figura 4.6 ilustra o fluxo de controle da instrução de repetição. Nessa figura, os símbolos no diagrama (além do estado inicial, setas de transição, um estado final e três notas) representam um estado de decisão de ação. Esse diagrama também introduz o UML que une dois fluxos de atividade a um único. O UML representa o símbolo de agregação e o símbolo de decisão como losangos. Nesse diagrama, o símbolo de agregação une as três etapas iniciais e do loop, assim ambos fluem para a decisão que determina se o loop deve iniciar (ou continuar) a execução. Símbolos de decisão e agregação podem ser separados pelo número de setas de transição ‘entrantes’ e ‘saintes’. Um símbolo de decisão contém uma seta de transição apontando para o losango e duas ou mais setas de transição apontando a partir do losango para possíveis transições a partir desse ponto. Além disso, cada seta de transição apontando de um símbolo de decisão contém uma condição de guarda ao lado dela. Um símbolo de agregação contém duas ou mais setas de transição apontando para o losango e uma seta de transição apontando a partir do losango, para indicar a conexão de múltiplos fluxos de atividades a fim de continuar a execução. Observe que, diferentemente do símbolo de decisão, o símbolo de agregação não tem uma contraparte no código C++. Nenhuma seta de transição associada com um símbolo de agregação contém condições de guarda.

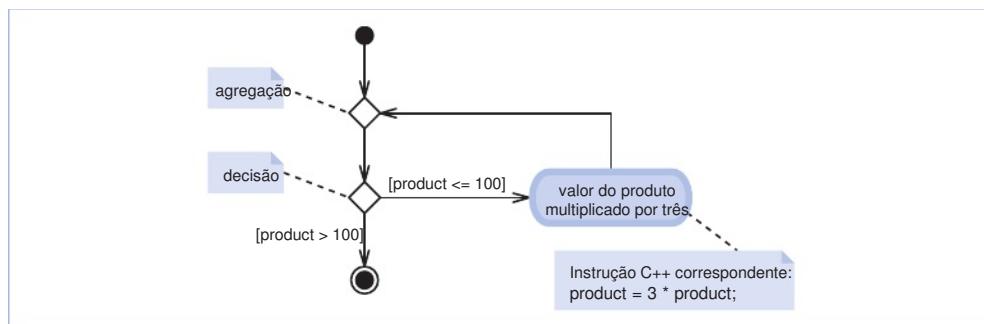


Figura 4.6 Diagrama de atividades UML da instrução de repetição.

O diagrama da Figura 4.6 mostra com clareza a ~~repetição ou instrução~~ que emerge do estado de ação aponta para o símbolo de agregação, que volta à decisão que é testada a cada passagem pelo condição de guarda ( $guarda > 100$ ) tornar-se verdadeira. Em seguida, a instrução (que muda seu estado final) e passa o controle para a próxima instrução na seqüência do programa.

Imagine um contêiner profundo de diagramas de atividades de instruções de repetição que o programador pode precisar empilhar e aninhar com os diagramas de atividades de outras instruções de controlar formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão pressões de ação e condições de guarda apropriadas ao algoritmo.



#### Dica de desempenho 4.3

Muitas das dicas de desempenho que mencionamos neste texto resultam apenas em pequenas melhorias, portanto o leitor pode ficar tentado a ignorá-las. Entretanto, uma pequena melhora de desempenho para um código que executa muitas vezes em um loop pode resultar em melhora significativa no desempenho geral.

### 4.8 Formulando algoritmos: repetição controlada por contador

Para ilustrar como os programadores desenvolvem algoritmos, esta seção e a Seção 4.9 resolvem duas variações de um problema: calcular a média da classe. Considere a seguinte declaração do problema:

Uma classe de dez alunos se submeteu a um questionário. As notas (inteiros no intervalo 0 a 100) para esse questionário estão disponíveis. Calcule e exiba o total de todas as notas e a média da classe no questionário.

A média de classe é igual à soma das notas divididas pelo número de alunos. O algoritmo para resolver esse problema em um código deve inserir cada uma das notas, calcular a média e imprimir o resultado.

Algoritmo em pseudocódigo com repetição controlada por contador

Vamos utilizar o pseudocódigo para listar as ações a executar e especificar a ordem em que essas ações devem ocorrer. Utilizar a ~~técnica de repetição controlada por contador~~ para inserir as notas uma por vez. Essa técnica utiliza uma variável chamada ~~contador~~ para controlar o número de vezes que um grupo de instruções executará (também conhecido como ~~loop~~) o número de

A repetição controlada por contador é frequentemente chamada de ~~vez~~ que o número de repetições é conhecido antes de o loop começar a executar. Nesse exemplo, a repetição termina quando o contador excede 10. Esta seção apresenta um pseudocódigo completamente desenvolvido (Figura 4.7), a mesma versão da Figura 4.9 que implementa

~~Sexta-feira, 10 de junho de 2011, 10:20:20~~ Sexta-feira, 10 de junho de 2011, 10:20:20. A seguir, o pseudocódigo para resolver o problema (Figura 4.7) é demonstrado. A Figura 4.10 demonstra o algoritmo em ação.



#### Observação de engenharia de software 4.3

A experiência tem mostrado que a parte mais difícil de resolver um problema em um computador é desenvolver o algoritmo para a solução. Uma vez que um algoritmo correto foi especificado, o processo de produção de um programa C++ funcional a partir do algoritmo é normalmente simples e direto.

Observe as referências no algoritmo em pseudocódigo da Figura 4.7 a ~~um total de variável utilizada~~ para acumular a soma de vários ~~valores~~. Uma variável utilizada para contar — nesse caso, o contador de notas indica qual das dez notas está em vias de ser inserida pelo usuário. Variáveis utilizadas para armazenar totais normalmente são iniciadas

- 1 Configura o total como zero
- 2 Configura o contador de notas como um
- 3
- 4 Enquanto contador de notas for menor ou igual a dez
- 5     Solicita que o usuário insira a próxima nota
- 6     Adicione a nota ao total
- 7     Adicione um ao contador de notas
- 8
- 9
- 10 Configure a média da classe como o total dividido por dez
- 11 Imprima o total das notas de todos os alunos da classe
- 12 Imprima a média da classe

Figura 4.7 Algoritmo em pseudocódigo que utiliza repetição controlada por contador para resolver o problema de média de classe.

como zero antes de serem utilizadas em um programa; caso contrário, a soma incluiria o valor anterior armazenado na pos memória do total.

#### Aprimorando a validação GradeBook

Antes de discutirmos a implementação do algoritmo de média da classe, vamos considerar um aprimoramento que fizemos para a classe GradeBook. Na Figura 4.16, nossa função membro `setCourseName` validaria o nome do curso primeiramente testando se o comprimento do nome do curso era menor que ou igual a 25 caracteres. Se isso fosse verdadeiro, o nome do curso seria configurado. Esse código era então seguido por uma instrução `if` que testava se o nome do curso tinha um comprimento maior que 25 caracteres (caso em que o nome do curso seria encurtado). Note que a ~~segunda condição é avaliada depois da construção da instância da classe~~. Essa condição é avaliada ~~antes da construção~~ e deve ser avaliada ~~com~~ a construção da classe. Esta situação é ideal para uma instrução `if`, portanto modificamos nosso código, substituindo ~~duas instruções~~ por ~~uma~~ `if` (linhas 21–28 da Figura 4.9).

#### Implementando a repetição controlada por contador na GradeBook

A classe GradeBook (Figura 4.8–Figura 4.9) contém um construtor (declarado na linha 11 da Figura 4.8 e definido nas linhas 12–15 da Figura 4.9) que atribui um valor à variável de instância `courseName` na linha 17 da Figura 4.8. As linhas 19–29,

```

1 // Figura 4.8: GradeBook.h
2 // Definição da classe GradeBook que determina a média de uma classe.
3 // As funções-membro são definidas no GradeBook.cpp
4 #include <iostream> // o programa utiliza a classe de string padrão do C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public :
11 GradeBook(string); // o construtor inicializa o nome do curso
12 void setCourseName(string); // função para configurar o nome do curso
13 string getCourseName(); // função para recuperar o nome do curso
14 void displayMessage(); // exibe uma mensagem de boas-vindas
15 void determineClassAverage(); // calcula a média das notas inseridas pelo usuário
16 private :
17 string courseName; // nome do curso para esse GradeBook
18 }; // fim da classe GradeBook

```

Figura 4.8 Problema para calcular a média de uma classe utilizando repetição controlada por contador: arquivo de GradeBook.h

```

1 // Figura 4.9: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que resolve o
3 // problema de média da classe com repetição controlada por contador.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // inclui a definição de classe GradeBook
10
11 // construtor inicializa courseName com String fornecido como argumento
12 GradeBook::GradeBook(string name)
13 {
14 setCourseName(name); // valida e armazena courseName
15 } // fim do construtor GradeBook

```

Figura 4.9 Problema para calcular a média de uma classe utilizando repetição controlada por contador: arquivo de GradeBook.cpp  
(continua)

```

16
17 // função para configurar o nome do curso;
18 // assegura que o nome do curso tenha no máximo 25 caracteres
19 void GradeBook::setCourseName(string name)
20 {
21 if (name.length() <= 25) // se o nome tiver 25 ou menos caracteres
22 courseName = name; // armazena o nome do curso no objeto
23 else // se o nome tiver mais que 25 caracteres
24 {
25 // configura courseName como os primeiros 25 caracteres do nome de parâmetro
26 courseName = name.substr(0, 25); // seleciona os primeiros 25 caracteres
27 cout << "Name '" << name << "' exceeds maximum length (25).\n"
28 << "Limiting courseName to first 25 characters.\n" << endl;
29 } // fim do if...else
30 } // fim da função setCourseName
31 // função para recuperar o nome do curso
32 string GradeBook::getCourseName()
33 {
34 return courseName;
35 } // fim da função getCourseName
36
37 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
38 void GradeBook::displayMessage()
39 {
40 cout << "Welcome to the grade book for\n" << getCourseName() << endl;
41 << endl;
42 } // fim da função displayMessage
43
44 // determina a média da classe com base em 10 notas inseridas pelo usuário
45 void GradeBook::determineClassAverage()
46 {
47 int total; // soma das notas inseridas pelo usuário
48 int gradeCounter; // número da nota a ser inserida a seguir
49 int grade; // valor da nota inserida pelo usuário
50 int average; // média das notas
51
52 // fase de inicialização
53 total = 0; // inicializa o total
54 gradeCounter = 1; // inicializa o contador de loops
55
56 // fase de processamento
57 while (gradeCounter <= 10) // faz o loop 10 vezes
58 {
59 cout << "Enter grade: " ; // solicita entrada
60 cin >> grade; // insere a próxima nota
61 total = total + grade; // adiciona grade a total
62 gradeCounter = gradeCounter + 1; // incrementa o contador por 1
63 } // fim do while
64
65 average = total / 10; // divisão de inteiros produz um resultado inteiro
66
67 // exibe o total e a média das notas
68 cout << "\nTotal of all 10 grades is " << total << endl;
69 cout << "Class average is " << average << endl;
70 } // fim da função determineClassAverage

```

Figura 4.9 Problema para calcular a média de uma classe utilizando repetição controlada por contador: arquivo de `GradeBook` (continuação)

```

1 // Figura 4.10: fig04_10.cpp
2 // Cria o objeto da classe GradeBook e invoca sua função determineClassAverage.
3 #include "GradeBook.h"// inclui a definição de classe GradeBook
4
5 int main()
6 {
7 // cria o objeto myGradeBook da classe GradeBook e
8 // passa o nome do cursor para o construtor
9 GradeBook myGradeBook("CS101 C++ Programming")
10
11 myGradeBook.displayMessage()// exibe a mensagem de boas-vindas
12 myGradeBook.determineClassAverage(); // calcula a média das 10 notas
13 return 0; // indica terminação bem-sucedida
14 } // fim de main

```

Welcome to the grade book for  
CS101 C++ Programming

Enter grade: 67  
Enter grade: 78  
Enter grade: 89  
Enter grade: 67  
Enter grade: 87  
Enter grade: 98  
Enter grade: 93  
Enter grade: 85  
Enter grade: 82  
Enter grade: 100

Total of all 10 grades is 846

Class average is 84

**Figura 4.10** Problema para calcular a média de uma classe utilizando repetição controlada por contador: criando um objeto GradeBook (Figura 4.8–Figura 4.9) e invocando sua função-membro determineClassAverage

32–35 e 38–42 da Figura 4.9 definem as funções-membro CourseName e displayMessage, respectivamente. As linhas 45–71 definem a função-membro determineClassAverage que implementa o algoritmo de média da classe descrito pelo pseudocódigo na Figura 4.7.

As linhas 47–50 declaram as variáveis locais courseName, gradeCounter e average como do tipo A variável courseName armazena a entrada de usuário. Note que as declarações precedentes aparecem dentro da função-membro.

Nas versões da classe GradeBook deste capítulo, simplesmente lemos e processamos um conjunto de notas. O cálculo da média é realizado na função-membro determineClassAverage utilizando variáveis locais — não preservamos nenhuma informação sobre as notas dos alunos nas variáveis de instância da classe. No Capítulo 7, “Arrays e vetores”, modificaremos a classe GradeBook para realizar vários cálculos sobre o mesmo conjunto de notas sem exigir que o usuário insira as notas múltiplas vezes.



#### Boa prática de programação 4.5

Separe as declarações de outras instruções nas funções com uma linha em branco para legibilidade.

As linhas 53–54 iniciam como 0 a variável gradeCounterComo 1. Observe que as variáveis gradeCounterSão inicializadas antes de ser utilizadas em um cálculo. Normalmente, as variáveis contadoras são inicializadas como zero ou um, depois de seu uso (apresentaremos exemplos de cada possibilidade). Uma variável não inicializada assume um valor indefinido — o último valor armazenado na posição da memória reservada para essa variável. Variáveis a entrada de usuário e a média calculada, respectivamente) não precisam ser inicializadas aqui — seus valores serão atribuídos que forem inseridos ou calculados mais tarde na função.



### Erro comum de programação 4.6

Não inicializar contadores e totais pode levar a erros de lógica.



### Dica de prevenção de erro 4.2

Inicialize cada contador e total, seja em sua declaração ou em uma instrução de atribuição. Normalmente, os totais são inicializados como 0. Os contadores normalmente são inicializados como 0 ou 1, dependendo de como eles são utilizados (mostraremos exemplos de quando utilizar 0 e quando utilizar 1).



### Boa prática de programação 4.6

Declare cada variável em uma linha separada com seu próprio comentário para tornar os programas mais legíveis.

A linha 57 indica que é permitido e correto fazer loop (também chamado *while loop*) entre as chaves que delimitam seu corpo (linhas 58–63).

A linha 59 exibe o `prompt` para inserir a instrução em pseudocódigo insira a próxima nota. A linha 60 é a nota inserida pelo usuário e a `essa variável` responde à instrução em pseudocódigo ‘Insira a próxima nota’. Lembre-se de que a `variável` foi inicializada anteriormente no programa, porque o programa obtém o valor de a partir do usuário durante cada iteração do loop. A linha 61 adiciona a nova nota inserida pelo usuário ao atribuir o resultado aque substitui o valor anterior.

A linha 62 adiciona `gradeCounter` para indicar que o programa processou uma nota e está pronto para inserir a próxima nota fornecida pelo usuário. Incrementando `gradeCounter` por fim faz com que `gradeCounter` exceda Nesse ponto o loop termina porque sua condição (linha 57) torna-se falsa.

Quando o loop termina, a linha 66 realiza o cálculo médio e atribui seu resultado à variável `averageGrade`. A função-membro `averageGrade` retorna o controle à função chamada `averageGrade` (isto é, Figura 4.10).

Demonstrando a classe `GradeBook`

A Figura 4.10 contém a classe `GradeBook`, que cria um objeto da classe `GradeBook` e demonstra suas capacidades. A linha

9 da Figura 4.10 da Figura 4.9 chama a função `averageGrade` para exibir a mensagem de boas-vindas para o usuário. A linha 12 então chama a função `memoria` para permitir ao usuário inserir dez notas, para as quais a função-membro então calcula e imprime a média — a função-membro re-

Observações sobre divisão de inteiros e truncamento

O cálculo da média feito pela função-membro `averageGrade` responde à chamada de função na linha 12 na Figura 4.10 produz um resultado de inteiro. A saída do programa indica que a soma dos valores de nota na execução de exemplo é 846, que dividido por 10, deve ser igual a 84,6 — um número com um ponto de fração decimal. Entretanto, o resultado do cálculo (linha 66 da Figura 4.9) é o inteiro 84, e não ambos inteiros. Dividir dois inteiros resulta em divisão de inteiros — qualquer parte fracionária do cálculo é perdida. Não é possível obter um resultado que inclui um ponto de fração decimal a partir do cálculo da média na próxima seção.



### Erro comum de programação 4.7

Assumir que divisão de inteiros arredonda (em vez de truncar) pode levar a resultados incorretos. Por exemplo,  $7 \div 4$ , que produz 1,75 na aritmética convencional, é truncado para 1 na aritmética de inteiros, em vez de arredondado para 2.

Na Figura 4.9, se a linha 66 utilizasse `sum / gradeCount` vez de 10 para o cálculo, a saída desse programa exibiria um valor incorreto, 76. Isso ocorre porque na iteração final da impressão foi incrementado para o valor 11 na linha 62.



### Erro comum de programação 4.8

Utilizar uma variável controlada por contador de um loop em um cálculo depois do loop costuma causar um erro de lógica comum chamado `erro off-by-one`. Em um loop controlado por contador que é incrementado por um a cada passagem pelo loop, o loop termina quando o valor do contador é um mais alto que seu último valor legítimo (isto é, 11 no caso de contar de 1 a 10).

## 4.9 Formulando algoritmos: repetição controlada por sentinelas

Vamos generalizar o problema da média da classe. Considere o seguinte problema:

Desenvolva um programa para calcular a média da classe que processe as notas de acordo com um número arbitrário de alunos toda vez que é executado.

No exemplo anterior de cálculo da média da classe, a declaração do problema especificou o número de alunos, assim o número (10) era conhecido antecipadamente. Neste exemplo, nenhuma indicação é dada de quantas notas o usuário irá inserir durante do programa. O programa deve processar um número arbitrário de notas. Como o programa pode determinar quando parar a execução? Como saber quando calcular e imprimir a média da classe?

Uma maneira de resolver esse problema é utilizar um **valor especial** (também chamado de **sentinela**) — um **valor fictício** ou **valor de flag** para indicar ‘final de entrada de dados’. O usuário digita notas até a última nota ter sido inserida. O usuário então digita o valor da sentinelas para indicar que a última nota foi inserida. A repetição controlada por sentinelas é freqüentemente chamada de **repetição indefinida**, uma vez que o número de repetições não é conhecido antes de o loop iniciar a execução.

Claramente, o valor da sentinelas deve ser escolhido de modo que não possa ser confundido com um valor de entrada legal. Portanto, uma execução do programa de média de classe poderia processar um fluxo de entradas como 95, 96, 75, 74, 89 e -1. O programa então calcularia e imprimiria a média de classe para as notas 95, 96, 75, 74 e 89 (-1 é o valor de sentinelas, ele não deve ser considerado no cálculo da média).

### Erro comum de programação 4.9

Escolher um valor de sentinelas que também seja um valor legítimo de dados é um erro de lógica.

Desenvolvendo o algoritmo em pseudocódigo com refinamento passo a passo de cima para baixo: a parte superior e o primeiro refinamento

Abordamos o programa de média de classe com **uma técnica chamada de topo**, uma técnica que é essencial para o desenvolvimento de programas bem estruturados. Iniciamos com **uma representação em pseudocódigo** — uma única instrução que fornece a função geral do programa:

Determine a média da classe para o questionário

O topo é, em efeito, uma **representação** do programa. Infelizmente, o topo (como nesse caso) raramente transmite detalhes

detalhados sobre o que é necessário para o programa. Então precisamos adicionar detalhamento. Dividimos o topo em um refinamento:

Inicialize as variáveis

Insira, some e conte as notas do exame

Calcule e imprima o total de todas as notas de aluno e a média da classe

Esse refinamento utiliza somente a estrutura de seqüência — os passos listados devem ser executados na ordem, um depois do outro.

### Observação de engenharia de software 4.4

Cada refinamento, bem como o topo, é uma especificação completa do algoritmo; somente o nível de detalhe varia.

### Observação de engenharia de software 4.5

Muitos programas podem ser divididos logicamente em três fases: uma fase de inicialização que inicializa as variáveis do programa; uma fase de processamento que insere os valores dos dados e ajusta as variáveis do programa (como contadores e totais) de maneira correspondente; e uma fase de término que calcula e gera a saída dos resultados finais.

Prosseguindo para o segundo refinamento

A “Observação de engenharia de software” anterior postula certamente que queremos precisar para o primeiro refinamento no pseudocódigo. Neste exemplo, precisamos de um total dos números, uma contagem de quantos números foram processados, uma variável para receber o valor de cada nota à medida que é inserida pelo usuário e uma variável para armazenar a média calculada. A instrução

Inicialize as variáveis

pode ser refinada desta maneira:

1 Inicialize total como zero

2 Inicialize o contador como zero

Somente as variáveis contadora e a entrada do usuário, respectivamente) não precisam ser inicializadas, uma vez que seus valores serão substituídos por zeros. As primeiras médias que são calculados ou inseridos.

A instrução em pseudocódigo

Insira, some e conte as notas do exame

requer uma instrução de repetição (isto é, um loop) que insere sucessivamente cada nota. Não conhecemos antecipadamente quais notas devem ser processadas, assim utilizaremos a repetição controlada por sentinela. O usuário insere as notas legítimas uma de cada vez. Depois de inserir a última nota legítima, o usuário insere o valor de sentinela. O programa faz um teste para o valor de sentinela e termina o loop quando o usuário insere o valor de sentinela. O segundo refinamento da instrução em pseudocódigo precedente é então

1 Insira a primeira nota (possivelmente o sentinela)

2 Enquanto o usuário não inserir o sentinela

    3 Adicione essa nota à soma total

    4 Adicione um ao contador de notas

    5 Solicite que o usuário insira a próxima nota

    6 Insira a próxima nota (possivelmente a sentinela)

No pseudocódigo, não utilizamos chaves em torno das instruções que formam a estrutura de repetição. Nós simplesmente cuamos as instruções **Enquanto** para mostrar que pertencem ao bloco. Novamente, o pseudocódigo é apenas um auxílio informal ao desenvolvimento de programa.

A instrução em pseudocódigo

Calcule e imprima o total de todas as notas de aluno e a média da classe

pode ser refinada desta maneira:

1 Se o contador não for igual a zero

    2 Configure a média como o total dividido pelo contador

    3 Imprima o total das notas de todos os alunos da classe

    4 Imprima a média da classe

caso contrário

    5 Imprima 'Nenhuma nota foi inserida'

Somos cuidadosos aqui para testar a possibilidade de divisão por zero, que normalmente não é detectado, faria com que o programa falhasse (frequentemente chamado de bug). O segundo refinamento completo do pseudocódigo para o problema da média da classe é mostrado na Figura 4.11.

```

1 1 Inicialize total como zero
2 2 Inicialize contador como zero
3
4 3 Solicite que o usuário insira a primeira nota
5 4 Insira a primeira nota (possivelmente o sentinela)
6
7 5 Enquanto o usuário não inserir o sentinela
8 6 Adicione essa nota à soma total
9 7 Adicione um ao contador de notas
10 8 Solicite que o usuário insira a próxima nota
11
12 9 Insira a próxima nota (possivelmente a sentinela)
13 10 Se o contador não for igual a zero
14 11 Configure a média como o total dividido pelo contador
15 12 Imprima o total das notas de todos os alunos da classe
16 13 Imprima a média da classe
17 14 caso contrário
18 15 Imprima 'Nenhuma nota foi inserida'

```

Figura 4.11 Algoritmo em pseudocódigo do problema de média da classe com repetição controlada por sentinela.



### Erro comum de programação 4.10

Uma tentativa de dividir por zero normalmente causa um erro fatal de tempo de execução.



### Dica de prevenção de erro 4.3

Ao realizar divisão por uma expressão cujo valor poderia ser zero, teste explicitamente para essa possibilidade e trate-a apropriadamente em seu programa (como imprimindo uma mensagem de erro) em vez de permitir que ocorra um erro fatal.

Nas figuras 4.7 e 4.11, incluímos algumas linhas completamente em branco e recuos no pseudocódigo para torná-lo mais legível. As linhas em branco separam os algoritmos em pseudocódigo em suas várias fases e o recuo enfatiza os corpos de instrução de.

O algoritmo em pseudocódigo na Figura 4.11 resolve o problema da média de classe mais geral. Esse algoritmo foi desenvolvido apenas dois níveis de refinamento. As vezes são necessários mais níveis.



### Observação de engenharia de software 4.6

Termine o processo de refinamento passo a passo de cima para baixo quando o algoritmo em pseudocódigo for especificado em detalhes suficientes para você ser capaz de converter o pseudocódigo em C++. Normalmente, implementar o programa C++ é então simples e direto.



### Observação de engenharia de software 4.7

Muitos programadores experientes escrevem programas sem jamais utilizar ferramentas de desenvolvimento de programa como pseudocódigo. Esses programadores acreditam que seu objetivo final é resolver o problema em um computador e que escrever pseudocódigo só retarda a produção de saídas finais. Embora esse método possa funcionar para problemas simples e familiares, ele pode levar a sérias dificuldades em projetos complexos e grandes.

Implementando a repetição controlada por sentinela na GradeBook

As figuras 4.12 e 4.13 mostram a classe C++ contendo a função `determineClassAverage()` que implementa o algoritmo em pseudocódigo da Figura 4.11 (essa classe é demonstrada na Figura 4.14). Embora cada nota inserida seja um int, o cálculo da média provavelmente produz um número com um ponto de fração decimal — ~~em outras~~ palavras, um número real ou ponto flutuante (por exemplo, 7,33, 0,0975 ou 1000,12345). Para representar tal número, portanto essa classe deve

~~utilizar double~~ A principal diferença entre essas duas classes é que a classe C++ usa tipos de dados numéricos de ponto flutuante na memória, números com maior magnitude e mais detalhes (isto é, mais dígitos à direita do ponto de fração decimal — também conhecido como precisão do número). Esse programa introduz um operador ~~de especialização~~ usado para forçar o cálculo da média a produzir um resultado numérico de ponto flutuante. Esses recursos serão explicados em detalhes quando discutirmos o progr

```

1 // Figura 4.12: GradeBook.h
2 // Definição da classe GradeBook que determina a média de uma classe.
3 // As funções-membro são definidas no GradeBook.cpp
4 #include <string> // o programa utiliza classe de string padrão C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public :
112 GradeBook(); // construtor que inicializa o nome do curso
13 string getCourseName(); // função para recuperar o nome do curso
14 void displayMessage(); // exibe uma mensagem de boas-vindas
15 void determineClassAverage(); // calcula a média das notas inseridas pelo usuário
16 private :
17 string courseName; // nome do curso para esse GradeBook
18 }; // fim da classe GradeBook

```

Figura 4.12 Problema para calcular média da classe utilizando repetição controlada por sentinela: arquivo de GradeBook

Neste exemplo, vemos que as instruções de controle podem ser empilhadas umas sobre as outras (na seqüência) assim como a criancinha empilha blocos de construção. A [instância](#) 70–75 da Figura 4.13 é imediatamente seguida por uma instrução `else` (linhas 78–90) na seqüência. Boa parte do código nesse programa é idêntica ao código na Figura 4.9, portanto nos concentraremos nos novos recursos e questões.

```

1 // Figura 4.13: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que resolve o
3 // problema de média de classe com repetição controlada por sentinela.
4 #include <iostream>
5 using std::cout;
6 using std::cin;

8 using std::endl; // assegura que o ponto de fração decimal seja exibido
9
10 #include <iomanip> // manipuladores de fluxo parametrizados
11 using std::setprecision; // configura a precisão da saída numérica
12
13 // inclui a definição da classe GradeBook de GradeBook.h
14 #include "GradeBook.h"
15
16 // construtor inicializa courseName com string fornecido como argumento
17 GradeBook::GradeBook(string name)
18 {
19 setCourseName(name); // valida e armazena courseName
20 } // fim do construtor GradeBook
21
22 // função para configurar o nome do curso;
23 // assegura que o nome do curso tenha no máximo 25 caracteres
24 void GradeBook::setCourseName(string name)
25 {
26 if (name.length() <= 25) // se o nome tiver 25 ou menos caracteres
27 courseName = name; // armazena o nome do curso no objeto
28 else // se o nome tiver mais que 25 caracteres
29 {
30 // configura courseName como os primeiros 25 caracteres do nome de parâmetro
31 courseName = name.substr(0, 25); // seleciona os primeiros 25 caracteres
32 cout << "Name '" << name << "'" exceeds maximum length (25).\\n"
33 << "Limiting courseName to first 25 characters.\\n" << endl;
34 } // fim do if...else
35 } // fim da função setCourseName
36
37 // função para recuperar o nome do curso
38 string GradeBook::getCourseName()
39 {
40 return courseName;
41 } // fim da função getCourseName
42
43 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
44 void GradeBook::displayMessage()
45 {
46 cout << "Welcome to the grade book for\\n" << getCourseName() << "\\n"
47 << endl;
48 } // fim da função displayMessage
49
50 // determina a média da classe com base em 10 notas inseridas pelo usuário
51 void GradeBook::determineClassAverage()

```

Figura 4.13 Problema para calcular média da classe utilizando repetição controlada por sentinela: arquivo de `GradeBook`  
(continua)

```

51 {
52 int total; // soma das notas inseridas pelo usuário
53 int gradeCounter; // número de notas inseridas
54 int grade; // valor da nota
55 double average; // número com ponto de fração decimal para a média
56
57 // fase de inicialização
58 total = 0; // inicializa o total
59 gradeCounter = 0; // inicializa o contador de loops
60
61 // fase de processamento
62 // solicita entrada e lê a nota do usuário
63 cout << "Enter grade or -1 to quit: " ;
64 cin >> grade; // insere nota ou valor de sentinelas
65
66 // faz um loop até ler o valor de sentinelas inserido pelo usuário
67 while (grade != -1) // enquanto a nota não é -1
68 {
69 total = total + grade; // adiciona grade a total
70 gradeCounter = gradeCounter + 1; // incrementa contador
71
72 // solicita entrada e lê a próxima nota fornecida pelo usuário
73 cout << "Enter grade or -1 to quit: " ;
74 cin >> grade; // insere nota ou valor de sentinelas
75 } // fim do while
76
77 // fase de término
78 if (gradeCounter != 0) // se usuário inseriu pelo menos uma nota...
79 {
80 // calcula a média de todas as notas inseridas
81 average = static_cast < double >(total) / gradeCounter;
82
83 // exibe o total e a média (com dois dígitos de precisão)
84 cout << "\nTotal of all " << gradeCounter << " grades entered is "
85 << total << endl;
86 cout << "Class average is " << setprecision(2) << fixed << average
87 << endl;
88 } // fim do if
89 else // nenhuma nota foi inserida, assim gera a saída da mensagem apropriada
90 cout << "No grades were entered" << endl;
91 } // fim da função determineClassAverage

```

Figura 4.13 Problema para calcular média da classe utilizando repetição controlada por sentinelas: arquivo de código `determineClassAverage.cpp` (continuação)

A linha 55 declara a variável `average`. Lembre-se de que utilizamos [um exemplo precedente](#) para armazenar a média da classe. Utilizar o exemplo atual permite armazenar o resultado do cálculo da média da classe como um número de ponto flutuante. A linha 59 inicializa a variável porque nenhuma nota foi ainda inserida. Lembre-se de que esse programa utiliza repetição controlada por sentinelas. Para manter um registro exato do número das notas inseridas, incrementa a variável `gradeCounter` quando o usuário inserir um valor de nota válido (isto é, não o valor de sentinelas) e continua o processamento da nota. Por fim, note que ambas as instruções de entrada (linhas 64 e 74) são precedidas de uma instrução de saída que solicita a entrada ao usuário.



#### Boa prática de programação 4.7

Solicite cada entrada de teclado ao usuário. O prompt deve indicar a forma da entrada e qualquer valor especial de entrada. Por exemplo, em um loop controlado por sentinelas, os prompts solicitando entrada de dados devem lembrar explicitamente ao usuário qual é o valor da sentinelas.

```

1 // Figura 4.14: fig04_14.cpp
2 // Cria o objeto da classe GradeBook e invoca sua função-membro determineClassAverage
3
4 // inclui a definição da classe GradeBook de GradeBook.h
5 #include "GradeBook.h"
6
7 int main()
8 {
9 // cria o objeto myGradeBook da classe GradeBook e
10 // passa o nome do cursor para o construtor
11 GradeBook myGradeBook("CS101 C++ Programming")
12
13 myGradeBook.displayMessage() // exibe a mensagem de boas-vindas
14 myGradeBook.determineClassAverage(); // calcula a média das 10 notas
15 return 0; // indica terminação bem-sucedida
16 } // fim de main

```

```

Welcome to the grade book for
CS101 C++ Programming

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67

```

**Figura 4.14** Problema para calcular média da classe utilizando repetição controlada por sentinelas: criando um objeto GradeBook (Figura 4.12–Figura 4.13) e invocando sua função-membro determineClassAverage

Lógica do programa para repetição controlada por sentinelas versus repetição controlada por contador

Compare a lógica do programa para a repetição controlada por sentinelas nesse aplicativo com a da repetição controlada por contador Figura 4.9. Na repetição controlada por contador, cada iteração (linhas 15–17 na Figura 4.9) lê um valor fornecido pelo usuário de acordo com o número especificado de iterações. Na repetição controlada por sentinelas, o programa lê o primeiro (linhas 63–64 da Figura 4.13) antevisão de falso. Esse valor determina se o fluxo do programa de controle deve entrar no corpo do while. Se a condição while for falsa, o usuário inseriu o valor de sentinelas, portanto é executado (isto é, nenhuma nota foi inserida). Se, por outro lado, a condição for verdadeira, o corpo inicia a execução e o loop adiciona o valor de atotal (linha 69). As linhas 73–74 no corpo do loop inserem então o próximo valor fornecido pelo usuário. Em seguida, o controle do programa alcança a chave direita de loop (linha 75), assim a execução continua com a teste da condição do while (linha 67). A condição utiliza a antevisão de falso inserida pelo usuário para determinar se o corpo do loop deve executar novamente. Observe que o valor é sempre a entrada do usuário imediatamente antes de o programa testar a condição while. Isso permite que o programa determine se o valor recém-inserido é a grade a ser inserida ou esse valor (isto é, adiciona à atotal). Se o valor de sentinelas for inserido, o loop termina e o programa não adiciona atotal a

Depois que o loop termina, a instrução (linhas 78–90) é executada. A condição na linha 78 determina se quaisquer notas foram inseridas. Se nenhuma nota foi inserida (linhas 80–90) da instrução executa e exibe a mensagem

grades were entered". Note o bloco if na Figura 4.19. Sem as chaves, as últimas três instruções no corpo do loop cairiam fora do loop, fazendo o computador interpretar esse código incorretamente, como segue:

```

// faz um loop até ler o valor de sentinelas inserido pelo usuário
while (grade != -1)
 total = total + grade; // adiciona grade a total
 gradeCounter = gradeCounter + 1; // incrementa counter

```

```
// solicita entrada e lê a próxima nota fornecida pelo usuário
cout << "Enter grade or -1 to quit: " ;
cin >> grade;
```

Isso causaria um loop infinito no programa se o usuário não inserisse a seta (na linha 64).



Erro comum de programação 4.11

Omitir as chaves que delimitam um bloco pode levar a erros de lógica, como loops infinitos. Para evitar esse problema, alguns programadores incluem o corpo de cada instrução de controle dentro de chaves, mesmo que o corpo contenha somente uma única instrução.

Precisão de número de ponto flutuante e requisitos de memória

As variáveis de tipo **real** representam números de ponto flutuante de precisão simples e têm sete dígitos significativos na maioria dos sistemas.

sistema de 32 bits, é só que a precisão só tem 15 dígitos significativos na maioria dos sistemas. O intervalo de valores requerido pela maioria dos programas, as variáveis de tipo `float` devem basta, mas você pode querer trabalhar com segurança'. Em alguns programas, até mesmo as variáveis do tipo `double` seriam inadequadas — esses programas estão além do escopo deste livro. A maioria dos programadores representa o ponto flutuante como `float`. De fato, o C++ trata todos os números de ponto flutuante que você digita no código-fonte de um programa (como 7,33 e 0,0975) como padrão. Esses valores no código-fonte são conhecidos como ponto flutuante. Consulte o Apêndice C, "Tipos fundamentais", para os intervalos de valores de

Os números de ponto flutuante costumam surgir como resultado de divisão. Na aritmética convencional, quando dividimos 10 por 3, o resultado é 3,333333..., com a seqüência de 3s repetindo-se infinitamente. O computador aloca apenas uma quantidade fixa para armazenar tal valor, portanto evidentemente o valor de ponto flutuante armazenado somente pode ser uma aproximação.



Erro comum de programação 4.12

Utilizar números de ponto flutuante de uma maneira que supõe que eles são representados exatamente (por exemplo, utilizando-os em comparações de igualdade) pode levar a resultados incorretos. Os números de ponto flutuante são representados apenas aproximadamente pela maioria dos computadores.

Embora os números de ponto flutuante não sejam sempre 100% precisos, eles têm numerosas aplicações. Por exemplo,

tal gíria de quando temos a temperatura de 98,6°F ou 36,6°C. A maioria das aplicações de software usa esse número de 98,6°F para a maioria dos aplicativos que medem temperaturas de corpo. Devido à natureza imprecisa de ponto flutuante, é o tipo preferido a ser usado porque as variáveis podem representar números de ponto flutuante com mais exatidão. Por essa razão, utilizaremos todo o livro.

Convertendo entre tipos fundamentais explícita e implicitamente

A variável `resultado` é declarada como tipo `double` (linha 55 da Figura 4.13) para capturar o resultado fracionário de nosso cálculo. Entretanto, as variáveis `idade` e `contagem` são ambas variáveis do tipo inteiro. Lembre-se de que dividir dois inteiros resulta em divisão de inteiro, em que qualquer parte fracionária do cálculo é perdida. A instrução:

```
average = total / gradeCounter;
```

o cálculo de divisão é realizado primeiro, então a parte fracionária do resultado é perdida. Para de ela ser atribuída a realizar um cálculo de ponto flutuante com valores de inteiro, devemos criar valores temporários que são números de ponto flutuante para o cálculo. O C++ [tomece de coerção unária](#) para realizar essa tarefa. A linha 81 utiliza o operador de coerção `cast<double>(total)`, para criar uma cópia de ponto flutuante seu operando entre parênteses. A utilização de um operador de coerção dessa maneira é chamada de [Operador armazenado](#). O valor armazenado em `total` é um inteiro.

O cálculo agora consiste em um valor de ponto flutuante (`resultado`) dividida pelo inteiro `Counter`. O compilador C++ sabe avaliar somente expressões em que os tipos de dados dos operandos são idênticos. Para asse-

os operadores de coerção de tipo. Por exemplo, em uma expressão contendo valores de tipos os dados operando valores `double`. No nosso exemplo, estamos tratando `double` (utilizando o operador de coerção unário), então o compilador gera `double`, permitindo que o cálculo seja realizado — o resultado da divisão de ponto flutuante é atribuído. No Capítulo 6, “Funções e uma introdução à recursão”, discutimos todos os tipos de dados fundamentais e sua ordem de promoção.



### Erro comum de programação 4.13

O operador de coerção pode ser utilizado para converter entre tipos numéricos fundamentais `double`, e entre tipos de classe relacionados (como discutiremos no Capítulo 13, “Programação orientada a objetos. Polimorfismo”). Aplicar uma coerção ao tipo errado pode causar erros de compilação ou erros de tempo de execução.

Os operadores de coerção estão disponíveis para o uso com todos os tipos de dados e também com os tipos de classe. O `static cast` é formado pela palavra-chave `static` seguida de colchetes angulares (torno de um nome de tipo de dados). O operador de coerção é **únario** — um operador que aceita apenas um operando. No Capítulo 2, estudamos os operadores aritméticos binários. O C++ também suporta versões **unárias** dos operadores binários (mador possa escrever expressões como `!x`). Os operadores de coerção têm precedência mais alta que outros operadores unários, como **únario** e **anário**. Essa precedência é mais alta do que **operadores lógicos**, e mais baixa que a de parênteses. Indicamos o operador de coerção com `cast` em nossos gráficos de precedência (ver, por exemplo, Figura 4.22).

Formatação de números de ponto flutuante

As capacidades de formatação na Figura 4.13 são brevemente discutidas aqui e explicadas em profundidade no Capítulo 15, “Saída de fluxo”. A chamada `cout` na linha 86 (com um argumento de variável `average`) deve ser impressa com dois dígitos à direita do ponto de fração decimal (por exemplo, 92,37). Essa chamada é referida como um **manipulador de fluxo parametrizado** (por causa das parênteses). Os programas que utilizam essas chamadas devem conter a diretiva de pré-processador (linha 10)

```
#include <iomanip>
```

A linha 11 especifica os nomes do arquivo de cabeçalho `iomanip.h` utilizados nesse programa. Observe que o manipulador de fluxo não parametriza (porque não é seguido por um valor ou expressão entre parênteses) e não requer o arquivo de cabeçalho `iomanip.h`. Se a precisão não é especificada, a saída dos valores de ponto flutuante tem normalmente seis dígitos de precisão (isto é, precisão-padrão) na maioria dos sistemas de 32 bits hoje), embora vejamos uma exceção para isso um pouco mais à frente.

O manipulador de fluxo (linha 86) indica que os valores de ponto flutuante devem ser enviados para a saída no formato de ponto fixo em oposição à notação científica. A notação científica é uma maneira de exibir um número como um número de ponto flutuante entre os valores de 1 e 10, multiplicado por uma potência de 10. Por exemplo, o valor 3.100 seria exibido em notação científica como  $3,1 \times 10^3$ . A notação científica é útil para exibir valores muito grandes ou muito pequenos. A formatação utilizar notação científica é discutida ainda mais no Capítulo 15. A formatação de ponto fixo, por outro lado, é utilizada para forçar um número flutuante a exibir um número específico de dígitos. Especificar a formatação de ponto fixo também força a impressão de zeros finais. Por exemplo, se o valor seja um número inteiro, como 88,00. Sem nenhuma formatação de ponto fixo, setprecision são utilizados em um programa, o valor impresso é o número de posições decimais indicado pelo valor passado a `precision` (por exemplo, `cout` (linha 86)), embora o valor na memória permaneça inalterado. Por exemplo, as saídas dos valores 87,946 e 67,543 são como 87,95 e 67,54, respectivamente. Observe que também é possível forçar um ponto flutuante a aparecer utilizando manipulador `setfpoint` especificado como `setfpoint(0)`; zeros finais não serão impressos. Como manipuladores de fluxo, `setfpoint` não são parametrizados e não requerem o arquivo de cabeçalho `<iomanip.h>`. Ambos podem ser encontrados no cabeçalho.

As linhas 86 e 87 da Figura 4.13 geram a saída da média da classe. Nesse exemplo, exibimos a média de classe arredondada para centésimo mais próximo e realizamos a saída com exatamente dois dígitos à direita do ponto de fração decimal. O manipulador parametrizado (linha 86) indica que o valor deve ser exibido com dois dígitos de precisão à direita do ponto de fração decimal — indicado por `precision(2)`. As três notas entraram durante a execução de exemplo do programa na Figura 4.14 totais 257, que produz as médias 85,66666.... O manipulador de fluxo parametrizado valor seja arredondado para o número especificado de dígitos. Nesse programa, a média é arredondada para a posição de centésimos e exibida como

## 4.10 Formulando algoritmos: instruções de controle aninhadas

Para o próximo exemplo, mais uma vez formulamos um algoritmo utilizando o pseudocódigo e o refinamento passo a passo de cima para baixo e escrevemos um programa C++ correspondente. Vimos que as instruções de controle podem ser empilhadas uma sobre a outra (em seqüência), assim como uma criança empilha blocos de construção. Nesse estudo de caso, examinaremos a outra estrutura de instruções de controle poderem ser combinadas, isto é, de controle dentro de outra.

Considere a seguinte declaração do problema:

Uma faculdade oferece um curso que prepara os candidatos a obter licença estadual para corretores de imóveis. No ano passado, dez alunos que concluíram esse curso prestaram o exame. A universidade quer saber como foi o desempenho dos seus alunos nesse exame. Você foi contratado para escrever um programa que resuma os resultados. Para tanto, você recebeu uma lista com dez desses alunos. Ao lado de cada nome é escrito 1 se o aluno passou no exame ou 2 se o aluno foi reprovado.

Seu programa deve analisar os resultados do exame assim:

1. Insira cada resultado de teste (isto é, 1 ou 2). Exiba a mensagem de solicitação 'Inserir resultado' toda vez que o programa solicitar outro resultado de teste.
2. Conte o número de cada tipo de resultado.
3. Exiba um resumo dos resultados do teste indicando o número de alunos aprovados e reprovados.
4. Se mais de oito alunos foram aprovados no exame, imprima a mensagem 'Aumentar a mensalidade escolar'.

Depois de ler a declaração do problema cuidadosamente, fazemos estas observações:

1. O programa deve processar resultados de teste para dez alunos. Um loop controlado por contador pode ser utilizado porque o número de resultados do teste é conhecido antecipadamente.
2. Cada resultado do teste é um número — 1 ou 2. Toda vez que o programa ler um resultado, deve determinar se o número é 1 ou 2. Em nosso algoritmo, testamos se o número é 1. Se o número não for um 1, supomos que ele seja um 2. (O Exercício considera as consequências dessa suposição.)
3. Dois contadores são utilizados para monitorar os resultados do exame — um para contar o número de alunos que foram aprovados no exame e outro para contar o número de alunos que foram reprovados no exame.
4. Depois que o programa processou todos os resultados, ele deve decidir se mais de oito alunos foram aprovados no exame.

Vamos prosseguir com o refinamento passo a passo de cima para baixo. Iniciamos com uma representação do pseudocódigo parte superior:

*Analice os resultados do exame e decida se a mensalidade escolar deve ser elevada*

Mais uma vez, é importante enfatizar que a parte superior da estrutura de controle é provavelmente necessária antes que o pseudocódigo possa transformar-se naturalmente em um programa C++.

Nosso primeiro refinamento é

*Inicialize as variáveis*

*Insira os 10 resultados dos exames e conte as aprovações e reprovações*

*Imprima um resumo dos resultados do exame e decida se a mensalidade escolar deve ser elevada*

Aqui, igualmente, embora tenhamos uma representação completa do programa inteiro, é necessário refinamento adicional. Empregamos variáveis específicas. Precisamos de contadores para registrar as aprovações e reprovações, de um contador para o processo de loop e de uma variável para armazenar a entrada do usuário. A última variável não é inicializada, porque seu valor

a partituração é só no final do código de iteração do loop.

*Inicialize as variáveis*

pode ser refinada desta maneira:

*Inicialize as aprovações como zero*

*Inicialize as reprovações como zero*

*Inicialize o contador de alunos como um*

Observe que somente os contadores são inicializados no início do algoritmo.

A instrução em pseudocódigo

*Insira os 10 resultados dos exames e conte as aprovações e reprovações*

requer um loop que sucessivamente insere o resultado de cada exame. Aqui sabemos com antecedência que há precisamente 10 resultados de exame, desse modo o loop controlado por contador é apropriado. Deveríamos, portanto, adicionar a instrução `if .. else` para determinar se cada resultado de exame é uma passagem ou uma falha e incrementar o contador apropriado. O refinamento da instrução em pseudocódigo precedente é então

*Enquanto o contador de alunos for menor ou igual a 10*

*Solicite que o usuário insira o próximo resultado de exame*

*Insira o próximo resultado de exame*

*Se o aluno foi aprovado*

*Adicione um à aprovações*

*Caso contrário*

*Adicione um a reprovações*

*Adicione um ao contador de alunos*

Utilizamos linhas em branco para isolar a estrutura de controle, que melhora a legibilidade.

A instrução em pseudocódigo

*Imprima um resumo dos resultados do exame e decida se a mensalidade escolar deve ser elevada*

pode ser refinada desta maneira:

```

Imprima o número de aprovações
Imprima o número de reprovações
Se mais de oito alunos forem aprovados
 Imprima 'Aumentar a mensalidade escolar'

```

O segundo refinamento completo aparece na Figura 4.15. Observe que linhas em branco também são utilizadas para destacar tura Enquanto para melhorar a legibilidade do programa. Esse pseudocódigo está agora suficientemente refinado para a conversão C++.

Conversão em análise de classe

A classe C++ que implementa o algoritmo em pseudocódigo é mostrada na Figura 4.16–Figura 4.17 e duas execuções de exemplo aparecem na Figura 4.18.

As linhas 16–18 da Figura 4.17 declaram as variáveis que a função-membro analysis utiliza para processar os resultados de teste. Observe que tiramos proveito de um recurso do C++ que permite que a inicialização da variável

incorporada ao loop (linhas 16–18) seja realizada dentro da inicialização da repetição; essa reinicialização seria normalmente realizada por instruções de atribuição em vez de em declarações, ou movendo as declarações dentro dos corpos

A instrução `do` (linhas 22–36) itera dez vezes. Durante cada iteração, o loop insere e processa um dos resultados do exame. Observe que a instrução (linhas 29–32) para processar cada resultado é aninhada na instrução `do`. A instrução `else` incrementa o resultado caso contrário, ela assume que o resultado é aprovado.

```

1 Initialize as aprovações como zero
2 Initialize as reprovações como zero
3 Initialize o contador de alunos como um
4
5 Enquanto o contador de alunos for menor ou igual a 10
6 Solicite que o usuário insira o próximo resultado de exame
7 Insira o próximo resultado de exame
8
9 Se o aluno foi aprovado
10 Adicione um a aprovações
11 Caso contrário
12 Adicione um a reprovações
13
14 Adicione um ao contador de alunos
15
16 Imprima o número de aprovações
17 Imprima o número de reprovações
18
19 Se mais de oito alunos forem aprovados
20 Imprima 'Aumentar a mensalidade escolar'

```

Figura 4.15 Pseudocódigo para o problema dos resultados do exame.

```

1 // Figura 4.16: Analysis.h
2 // Definição de análise de classe que analisa resultados de exame.

3 // A função-membro é definida em Analysis.cpp
4 // definição da classe Analysis
5 class Analysis
6 {
7 public :
8 void processExamResults(); // processa os resultados do teste de 10 alunos
9 };
10 // fim da classe Analysis

```

Figura 4.16 Problema dos resultados do exame: arquivo de cabecalho analysis.

```

1 // Figura 4.17: Analysis.cpp
2 // Definições de função-membro para a classe Analysis que
3 // analisa os resultados do teste.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 // inclui a definição de classe Analysis a partir de Analysis.h
10 #include "Analysis.h"
11
12 // processa os resultados do teste de 10 alunos
13 void Analysis::processExamResults()
14 {
15 // inicializando variáveis nas declarações
16 int passes = 0; // número de aprovações
17 int failures = 0; // número de reprovações
18 int studentCounter = 1; // contador de alunos
19 int result; // o resultado de um teste (1 = aprovado, 2 = reprovado)
20
21 // processa 10 alunos utilizando o loop controlado por contador
22 while (studentCounter <= 10)
23 {
24 // solicita ao usuário uma entrada e obtém valor fornecido pelo usuário
25 cout << "Enter result (1 = pass, 2 = fail): " ;
26 cin >> result; // lê o resultado
27
28 // if...else aninhado em while
29 if (result == 1) // se resultado for 1,
30 passes = passes +1; // incrementa aprovações;
31 else // senão resultado não for 1, então
32 failures = failures + 1; // incrementa reprovações
33
34 // incrementa studentCounter até o loop terminar
35 studentCounter = studentCounter + 1;
36 } // fim do while
37
38 // fase de terminação; exibe número de aprovados e reprovados
39 cout << "Passed " << passes << "\nFailed " << failures << endl;
40
41 // determina se mais de oito alunos passaram
42 if (passes > 8)
43 cout << "Raise tuition " << endl;
44 } // fim da função processExamResults

```

Figura 4.17 Problema dos resultados do exame: instruções de controle aninhados no arquivo de código [Analysis.cpp](#)

A linha 35 incrementa studentCounter antes de a condição de loop ser testada novamente na linha 22. Depois que dez valores foram inseridos, o loop termina e a linha 39 exibe os números finais. A instrução das linhas 42–43 determina se mais de oito alunos foram aprovados no exame e, se foram, gera a saída (alimentando a mensalidade escolar).

#### Demonstrando a classe Analysis

A Figura 4.18 cria um objeto (linha 7) e invoca a função `processExamResults()` do objeto (linha 8) para processar um conjunto de resultados de teste inserido pelo usuário. A Figura 4.18 mostra a entrada e a saída de duas execuções de exemplo. No fim da primeira execução de exemplo, a condição na linha 42 da função [Figura 4.17](#) é verdadeira — mais de oito alunos passaram no teste, então o programa realiza saída de uma mensagem que indica que a deve ser feita.

```

1 // Figura 4.18: fig04_18.cpp
2 // Programa de teste para classe Analysis.
3 #include "Analysis.h" // inclui definição de classe Analysis
4
5 int main()
6 {
7 Analysis application; // cria o objeto da classe Analysis
8 application.processExamResults(); // função de chamada para processar resultados
9 return 0; // indica terminação bem-sucedida
10 } // fim de main

```

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 2

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Passed 9

Failed 1

Raise tuition

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 2

Enter result (1 = pass, 2 = fail): 2

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 1

Enter result (1 = pass, 2 = fail): 2

Passed 6

Failed 4

Figura 4.18 Programa de teste para a classe Analysis.

#### 4.11 Operadores de atribuição

O C++ fornece vários operadores de atribuição para abreviar expressões de atribuição. Por exemplo, a instrução

c = c + 3;

pode ser abreviada como operador de atribuição de adição como

c += 3;

O operador adiciona o valor da expressão à direita do operador ao valor da variável à esquerda do operador e armazena o resultado na variável à esquerda do operador. Qualquer instrução na forma

variável = variável operador expressão;

em que a mesma variável aparece em ambos os lados da atribuição e o operador é um dos operadores binários (%(ou outros que discutiremos mais adiante no texto), pode ser escrita na forma

variável operador = expressão;

Assim a atribuição adiciona. A Figura 4.19 mostra expressões de exemplo de operadores de atribuição aritméticos que utilizam esses operadores e suas explicações.

| Operador de atribuição                       | Expressão de exemplo | Explicação   | Atribuições       |
|----------------------------------------------|----------------------|--------------|-------------------|
| Assuma: $c = 3, d = 5, e = 4, f = 6, g = 12$ |                      |              |                   |
| $+=$                                         | $+= c 7$             | $c = c + 7$  | $10 \leftarrow c$ |
| $-=$                                         | $-= d 4$             | $d = d - 4$  | $1 \leftarrow d$  |
| $*=$                                         | $*= e 5$             | $e = e * 5$  | $20 \leftarrow e$ |
| $/=$                                         | $/= f 3$             | $f = f / 3$  | $2 \leftarrow f$  |
| $%=$                                         | $%= g 9$             | $g = g \% 9$ | $3 \leftarrow g$  |

Figura 4.19 Operadores de atribuição aritméticos.

## 4.12 Operadores de incremento e decremento

Além dos operadores de atribuição aritméticos, o C++ também fornece dois operadores unários para adicionar 1 ao ou subtrair 1 de uma variável numérica. Esses são o **operador de incremento unário**, e o **operador de decremento unário**, que são resumidos na Figura 4.20. Um programa pode incrementar por 1 o valor de uma variável por meio do operador `++`, em vez da expressão `out += 1`. Um operador de incremento ou decremento que é prefixado a (colocado antes de) uma variável é referido como **operador de incremento prefixo** ou **operador de decremento prefixo**, respectivamente. Um operador de incremento ou de decremento que é colocado depois de uma variável é conhecido como **operador de pós-incremento** ou **operador de pós-decremento**, respectivamente.

Utilizar o operador de pré-incremento (ou de pré-decremento) para adicionar (ou subtrair) 1 de uma variável é conhecido como **pré-incremento** (ou **pré-decremento**) da variável. Pré-incrementar (ou pré-decrementar) faz com que a variável seja incrementada (decrementada) por 1, e então o novo valor da variável é utilizado na expressão em que ela aparece. Utilizar o operador de pós-incremento (ou pós-decremento) para adicionar (ou subtrair) 1 de uma variável é conhecido como **pós-incremento** (ou **pós-decremento**) da variável. Pós-incrementar (pós-decrementar) faz com que o valor atual da variável seja utilizado na expressão em que ela aparece, e então o valor da variável é incrementado (decrementado) por 1.



### Boa prática de programação 4.8

Diferentemente dos operadores binários, os operadores de incremento e decremento unários devem ser colocados ao lado dos seus operandos, sem espaços no meio.

A Figura 4.21 demonstra a diferença entre as versões de pré-incremento e de pós-incremento do operador de incremento e decremento. A função `main()` realiza todo o trabalho do aplicativo. Neste capítulo e no Capítulo 3, você viu exemplos consistindo em uma classe (incluindo o cabeçalho e arquivos de código-fonte para essa classe), bem como outro arquivo de código-fonte que testa a classe. Esse arquivo de código-fonte é chamado de `main()` e é o objeto da classe e chamava suas funções-membro. Neste exemplo, simplesmente queremos mostrar os mecanismos de operadores unários em um arquivo de código-fonte com a função `main()`. Ocasionalmente, quando não faz sentido tentar criar uma classe reutilizável para demonstrar um conceito simples, utilizaremos um exemplo mecânico contido inteiramente dentro de um único arquivo de código-fonte.

| Operador        | Chamado        | Expressão de exemplo | Explicação                                                                          |
|-----------------|----------------|----------------------|-------------------------------------------------------------------------------------|
| <code>++</code> | pré-incremento | <code>++a</code>     | Incrementa por 1, então utilize o novo valor de expressão em <code>main()</code> .  |
| <code>++</code> | pós-incremento | <code>a++</code>     | Utilize o valor atual da expressão em <code>main()</code> , então incremente por 1. |
| <code>--</code> | pré-decremento | <code>--b</code>     | Decrementa por 1, então utilize o novo valor de expressão em <code>main()</code> .  |
| <code>--</code> | pós-decremento | <code>b--</code>     | Utilize o valor atual da expressão em <code>main()</code> , então decremente por 1. |

Figura 4.20 Operadores de incremento e de decremento.

```

1 // Figura 4.21: fig04_21.cpp
2 // Pré-incrementando e pós-incrementando.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int c;
10
11 // demonstra pós-incremento
12 c = 5; // atribui 5 à variável c
13 cout << c << endl; // imprime 5
14 cout << c++ << endl; // imprime 5 então pós-incrementa
15 cout << c << endl; // imprime 6
16
17 cout << endl; // pula uma linha
18
19 // demonstra pré-incremento
20 c = 5; // atribui 5 à variável c
21 cout << c << endl; // imprime 5
22 cout << ++c << endl; // pré-incrementa e então imprime 6
23 cout << c << endl; // imprime 6
24 return 0; // indica terminação bem-sucedida
25 } // fim de main

```

```

5
5
6

```

Figura 4.21 Pré-incrementando e pós-incrementando.

A linha 12 inicializa a variável `c`. A linha 13 gera a saída do valor inicial 5. A linha 14 gera a saída do valor da expressão `c++`. Essa expressão pós-incrementa a variável `c`, enviando para a saída, e então incrementando o valor da variável. Portanto, a linha 14 gera a saída do valor incrementado. A linha 15 gera a saída do valor final, que é 6, porque a expressão `c++` já foi executada.

A linha 20 reinicializa o valor de `c`. A linha 21 envia o valor para a saída. A linha 22 gera a saída do valor da expressão `++c`. Essa expressão pré-incrementa a variável `c`, formando seu valor é incrementado e enviado para a saída. A linha 23 gera a saída do valor finalmente para mostrar que `c` é 6, depois que a linha 22 é executada.

Os operadores aritméticos de atribuição e os operadores de incremento e decremento podem ser utilizados para simplificar instruções de um programa. As três instruções de atribuição na Figura 4.17

```

passes = passes + 1;
failures = failures + 1;

```

podem ser escritas mais concisamente com operadores de atribuição como

```

passes += 1;
failures += 1;
studentCounter += 1;

```

com operadores de pré-incremento como

```

++passes;
++failures;
++studentCounter;

```

ou com operadores de pós-incremento como

```
passes++;
failures++;
studentCounter++;
```

Observe que, quando a **incrementação** (mentação) (uma variável ocorre em uma instrução por si mesma, as formas de pré-incremento e de pós-incremento e as formas de pré-decremento e pós-decremento têm o mesmo efeito. Sómente uma variável aparece no contexto de uma expressão maior é que pré-incrementar e pós-incrementar a variável têm efeitos diferentes para pré-decrementar e pós-decrementar).



#### Erro comum de programação 4.14

Tentar utilizar o operador de incremento ou decremento em uma expressão diferente da de um nome variável ou referência modificável, por exemplo, escrever `x + 1`, é um erro de sintaxe.

A Figura 4.22 mostra a precedência e a associatividade dos operadores introduzidos nesse ponto. Os operadores são mapeados de cima para baixo em ordem decrescente de precedência. A segunda coluna indica a associatividade dos operadores em cada nível de precedência. Note que o operador `const_cast` é considerado unário de pré-incremento (mentação) e subtração é os operadores de atribuição = e %= associam-se da direita para a esquerda. Todos os outros operadores no gráfico de precedência de operador da Figura 4.22 são associados da esquerda para a direita. A terceira coluna nomeia os vários tipos de operadores.

### 4.13 Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional)

Na Seção 3.11, começamos a primeira etapa de um projeto orientado a objetos para nosso sistema ATM — analisando o documento de requisitos e identificando as classes necessárias para implementar o sistema. Listamos os sujeitos e frases com substantivos no documento de requisitos e identificamos uma classe separada para cada sujeito que desempenha um papel significativo no sistema ATM. Em seguida, modelamos as classes e seus relacionamentos em um diagrama de classes UML 3.23). As classes contêm atributos (dados) e operações (comportamentos). Os atributos de classe são implementados em p C++ como membros de dados; e as operações de classe são implementadas como funções-membro. Nesta seção, determinaremos quais os atributos necessários no sistema ATM. No Capítulo 5, examinaremos como esses atributos representam o estado de um carro. No Capítulo 6, determinaremos as operações de classe.

#### Identificando atributos

Considere os atributos de alguns objetos do mundo real: os atributos de uma pessoa incluem a altura, peso e se a pessoa é casada ou ambidesta. Os atributos de um rádio incluem sua configuração de estações, configuração de volume e configuração de AN. Os atributos de um carro incluem as leituras do velocímetro e do odômetro, a quantidade de gasolina no tanque e a marcha em que o carro está.

| Operadores                 | Associatividade            | Tipo                 |
|----------------------------|----------------------------|----------------------|
| ()                         | da esquerda para a direita | parênteses           |
| ++ -- static_cast <tipo>() | da esquerda para a direita | unário (pós-fixo)    |
| ++ -- + -                  | da direita para a esquerda | unário (prefixo)     |
| * / %                      | da esquerda para a direita | multiplicativo       |
| + -                        | da esquerda para a direita | aditivo              |
| << >>                      | da esquerda para a direita | inserção/extracção   |
| < <= > >=                  | da esquerda para a direita | relacional           |
| == !=                      | da esquerda para a direita | igualdade            |
| ?:                         | da direita para a esquerda | ternário condicional |
| = += -= *= /= %=           | da direita para a esquerda | atribuição           |

Figura 4.22 Ordem de precedência dos operadores encontrados até agora no texto.

está. Os atributos de um computador pessoal incluem seu fabricante (por exemplo, Dell, Sun, Apple ou IBM), tipo de tela (por exemplo, LCD ou CRT), tamanho da memória principal e tamanho do disco rígido.

Podemos identificar muitos atributos das classes no nosso sistema procurando palavras e frases descritivas no documento de requisitos. Para cada palavra ou frase encontrada que desempenha um papel significativo no sistema ATM, criamos um atributo e o atribuímos a uma ou mais classes identificadas na Seção 3.11. Também criamos atributos para representar quaisquer dados adicionais de classe talvez precise, à medida que essas necessidades se tornam claras por todo o processo do projeto.

A Figura 4.23 lista as palavras ou frases no documento de requisitos que descrevem cada classe. Formamos essa lista lendo o documento de requisitos e identificando todas as palavras ou frases que se referem às características das classes no sistema. Por exemplo, a classe ATM tem a palavra 'usuário é autenticado' que descreve o usuário que faz a transação.

A Figura 4.23 nos leva a criar um atributo Booleano para a classe ATM que armazena informações sobre o estado do ATM. A frase 'usuário é autenticado' descreve um estado do ATM (introduzimos estados na Seção 5.1), portanto incluímos o atributo Booleano (isto é, um atributo comum valor). O tipo UML Boolean é equivalente ao tipo C++. Esse

atributo indicaiza se o ATM pode efetuar transações para esse usuário. Essa é a única classe que tem esse atributo.

As classes BalanceInquiry, Withdrawal e Deposit compartilham um atributo. Cada transação envolve um 'número de conta' que corresponde à conta do usuário que faz a transação. Atribuímos um atributo para a classe de transação para identificar a conta a qual um objeto da classe se aplica.

Palavras e frases descritivas no documento de requisitos também sugerem algumas diferenças nos atributos requeridos em cada classe. O documento de requisitos indica que, para sacar dinheiro ou depositar fundos, os usuários devem inserir uma 'amount' específica a ser sacada ou depositada, respectivamente. Portanto, atribuímos um atributo para as classes de transação para armazenar o valor fornecido pelo usuário. Os valores relacionados a um saque e depósito são características definidas por transações que o sistema requer para que essas transações sejam corretamente processadas. A classe Account não precisa de nenhum dado adicional para realizar sua tarefa — ela requer somente um número de conta para indicar a conta cujo saldo deve ser recuperado.

A classe Account tem vários atributos. O documento de requisitos declara que cada conta bancária deve ter um 'número de conta' e um 'PIN' que o sistema utiliza para identificar contas e autenticar usuários. Atribuímos os atributos da seguinte forma:

- accountNumber: é o número da conta.
- pin: é o PIN da conta.
- balance: é o saldo da conta.

O documento de requisitos também especifica que uma conta deve ter um 'valor disponível' ('available balance'), e que esse valor depositado pelo usuário não se torna disponível para um saque até que o banco o verifique no envelope de depósito. Todos os cheques no envelope sejam compensados. Uma conta, porém, ainda deve registrar o valor que um usuário deposita. Decidimos que uma conta deve representar um saldo que utiliza os atributos accountNumber e availableBalance.

O atributo availableBalance armazena o valor que um usuário pode sacar da conta. O atributo balance armazena o valor total

que o usuário tem 'em depósito' (isto é, o valor disponível mais o valor esperando para ser verificado ou compensado). Por exemplo, se o usuário tem \$ 1000 em depósito e \$ 500 esperando para ser verificado, o atributo availableBalance armazena \$ 1500.

| Classe         | Palavras e frases descritivas                      |
|----------------|----------------------------------------------------|
| ATM            | usuário é autenticado                              |
| BalanceInquiry | número de conta                                    |
| Withdrawal     | número de conta<br>valor                           |
| Deposit        | número de conta<br>valor                           |
| BankDatabase   | [nenhuma palavra ou frase descritiva]              |
| Account        | número de conta<br>PIN<br>saldo                    |
| Screen         | [nenhuma palavra ou frase descritiva]              |
| Keypad         | [nenhuma palavra ou frase descritiva]              |
| CashDispenser  | inicia cada dia carregado com 500 cédulas de \$ 20 |
| DepositSlot    | [nenhuma palavra ou frase descritiva]              |

Figura 4.23 Palavras e frases descritivas nos requisitos do ATM.

suponha que um usuário do ATM deposite \$ 50,00 em uma conta e, simultaneamente, tire para \$ 50,00 a fim de registrar o depósito, mas a *availableBalance* permaneceria em \$ 0. Supomos que o banco atualiza a *totalBalance* de uma *Account* logo depois que a transação ATM ocorre, em resposta à confirmação de que o valor de \$ 50,00 em dinheiro ou cheiro foi encontrado no envelope de depósito. Assumimos que essa atualização ocorre por meio de uma transação que um funcionário realiza utilizando algum software do banco diferente do ATM. Portanto, não discutimos essa transação no nosso estudo de caso.

A classe *CashDispenser* tem um atributo. O documento de requisitos afirma que o dispensador de cédulas 'começa todos os dias carregado com 500 cédulas de \$ 20'. O dispensador de cédulas deve manter um registro do número de cédulas que ele tem para determinar se há dinheiro suficiente à disposição para satisfazer as solicitações de saque. Atribuímos à classe *CashDispenser* o atributo *count*, inicialmente configurado como

Para problemas reais na indústria, não há garantias de que os documentos de requisitos serão suficientemente detalhados para que o projetista de sistemas orientado a objetos determine todos os atributos ou mesmo todas as classes. A necessidade de atributos e comportamentos adicionais pode tornar-se clara à medida que o projeto avança. A medida que progredimos por esse caso de estudo, também continuaremos a adicionar, modificar e excluir as informações sobre as classes no nosso sistema.

#### Modelando atributos

O diagrama de classes na Figura 4.24 lista alguns atributos das classes no nosso sistema — as palavras e frases descritivas de 4.23 nos ajudaram a identificar esses atributos. Por simplicidade, a Figura 4.24 não mostra as associações entre as classes — mostradas na Figura 3.23. Essa é uma prática comum entre projetistas de sistemas ao desenvolver projetos. Lembre-se, na Seção 3.2, de que na UML os atributos de uma classe são colocados no compartimento central do retângulo da classe. Listamos cada nome de atributo separado por dois-pontos, seguido, em alguns casos, por um tipo de dado.

Considere o atributo *userAuthenticated* da classe *ATM*:

*userAuthenticated* : Boolean = false

Essa declaração de atributo contém três informações sobre o atributo *userAuthenticated*. O tipo do atributo é

Boolean. Em C++, um atributo pode ser representado por um tipo fundamental ou por um tipo de classe — como discutido no Capítulo 3. Escolhemos modelar apenas atributos de tipo primitivo na Figura 4.24 — discutimos o raciocínio trás dessa decisão em breve. [Figura 4.24 lista os tipos de dados UML para os atributos. Quando implementarmos o sistema, associaremos os tipos UML com os tipos fundamentais do C++, int e double, respectivamente.]

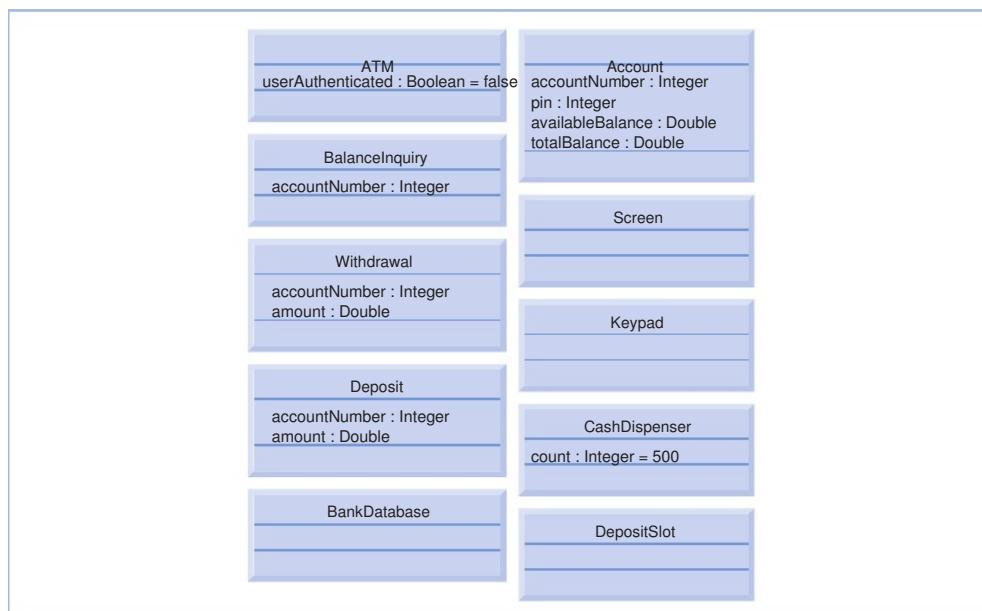


Figura 4.24 Classes com atributos.

Também podemos indicar um valor inicial para um atributo. Quando o atributo tem um valor inicial se. Isso indica que o sistema inicialmente não considera o usuário como autenticado. Se um atributo não contiver nenhum valor especificado, somente seu nome e tipo (separado por dois-pontos) são mostrados. Por exemplo, o atributo BalanceInquiry é um Integer. Aqui mostramos o valor não inicial, porque o valor desse atributo é um número que ainda não conhecemos. Esse número será determinado em tempo de execução com base no número de conta inserido pelo usuário atual do ATM.

A Figura 4.24 não inclui nenhum atributo para as classes DepositSlot. Estas são componentes importantes do nosso sistema, para as quais o processo do nosso projeto simplesmente ainda não revelou nenhum atributo. Entretanto, ainda podemos descobrir algumas nas fases restantes do projeto ou ao implementarmos essas classes em C++. Isso é perfeitamente normal para o processo iterativo de engenharia de software.



### Observação de engenharia de software 4.8

Nas etapas iniciais doprocesso deprojeto, freqüentemente faltam nas classeseatributos (e operações). Essasclasses, porém, não devem ser eliminadas, pois os atributos (e operações) podem tornar-se evidentes nas fases posteriores do projeto e implementação.

Observe que a Figura 4.24 também não inclui nenhum atributo para a classe Database. Considerando a discussão no Capítulo 3, lembre-se de que em C++ os atributos podem ser representados tanto por tipos fundamentais como por tipos de classe. Opta-se incluir somente os atributos de tipo fundamental no diagrama de classes na Figura 4.24 (e aos diagramas de classes semelhantes o estudo de caso). Um atributo de tipo de classe é modelado mais claramente como uma associação (em particular, uma compo entre a classe com o atributo e a classe do objeto do qual o atributo é uma instância. Por exemplo, o diagrama de classes na Figura 4.24 indica que a classe Database participa de um relacionamento de composição com a classe ATM. A partir desse compo, podemos determinar que, ao implementarmos o sistema ATM em C++, será necessário criar um atributo da classe Database para armazenar zero ou mais objetos de modo semelhante, atribuiremos os atributos correspondentes a seus relacionamentos de composição entre classes. Dispense o atributo Database::DepositSlot. Esses atributos baseados em composição seriam redundantes se modelados na Figura 4.24, porque as composições modeladas na Figura 3.23 já comunicam que o banco de dados contém as informações sobre zero ou mais contas e que um ATM é composto de uma tela, teclado, display de cédulas e uma abertura para depósito. Em geral, os desenvolvedores de software modelam esses relacionamentos integralmente como composições em vez de como atributos requeridos para implementar os relacionamentos.

O diagrama de classes na Figura 4.24 fornece uma base sólida para a estrutura do nosso modelo, mas o diagrama não está completo. Na Seção 5.11, identificamos os estados e atividades dos objetos no modelo e na Seção 6.22 identificamos as operações que os realizam. À medida que apresentarmos outras informações sobre a UML e o projeto orientado a objetos, continuaremos a fortalecer a estrutura do nosso modelo.

Exercícios de revisão do estudo de caso de engenharia de software

- 4.1 Em geral, identificamos os atributos das classes no nosso sistema analisando os(as) no documento de requisitos.
  - a) substantivos simples e substantivos compostos
  - b) palavras e frases descritivas
  - c) verbos e frases com verbos
  - d) Todos os acima.
- 4.2 Qual dos seguintes não é um atributo de um avião?
  - a) comprimento
  - b) envergadura da asa
  - c) vôo
  - d) número de poltronas
- 4.3 Descreva o significado da seguinte declaração de atributo no diagrama de classes na Figura 4.24:  
count : Integer = 500

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 4.1 b.
- 4.2 c. Vôo é uma operação ou comportamento de um avião, não um atributo.
- 4.3 Isso indica que é um Integer com um valor inicial. Esse atributo monitora o número de contas disponíveis na memória em determinado momento.

## 4.14 Síntese

Este capítulo apresentou técnicas básicas de solução de problemas que programadores utilizam na criação de classes e desenvolvimento de funções-membro para essas classes. Demonstramos como construir um algoritmo (isto é, uma abordagem para resolver um problema) e então como refiná-lo por meio de várias fases de desenvolvimento em pseudocódigo, resultando em código C++ que pode ser executado.

ser executado como parte de uma função-membro. Você aprendeu a utilizar o refinamento passo a passo de cima para baixo para as ações específicas que uma função deve realizar e a ordem em que ela deve realizá-las.

Você aprendeu que apenas três tipos de estruturas de controle — seqüência, seleção e repetição — são necessários para qualquer algoritmo. Demonstramos duas das instruções de seleção do C++ — ~~if~~ instrução que faz uma única seleção de seleção dupla. A instrução ~~if~~ utilizada para executar um conjunto de instruções baseadas em uma condição — se a condição for verdadeira, as instruções são executadas; se não, as instruções são ignoradas. A instrução de seleção dupla executa um conjunto de instruções se uma condição for verdadeira, e outro conjunto de instruções se a condição for falsa. Então a instrução de repetição de um conjunto de instruções é executado repetidamente se uma condição for verdadeira. Utilizamos o empilhamento de instruções de controle para totalizar e calcular a média de um conjunto de notas de alunos com a controlada por contador e por sentinela e utilizamos o aninhamento de instruções de controle para analisar e tomar decisões com um conjunto de resultados de um exame. Introduzimos operadores de atribuição, que podem ser utilizados para abreviar ins. Apresentamos os operadores de incremento e de decremento, que podem ser utilizados para adicionar ou subtrair o valor 1 variável. No Capítulo 5, “Instruções de controle: parte 2”, continuamos nossa discussão sobre as instruções de controle, introduzindo as instruções `do...while` e `switch`.

## Resumo

- Um algoritmo é um procedimento para resolver um problema em termos das ações a executar e a ordem em que executá-las.
- Especificar a ordem em que as instruções (ações) são executadas em um programa é chamado controle de programa.
- O pseudocódigo ajuda um programador a pensar sobre um programa antes de tentar escrevê-lo em uma linguagem de programação.
- Os diagramas de atividades são parte da Unified Modeling Language (UML) — um padrão da indústria para modelagem de sistemas software.
- Um diagrama de atividades modela o fluxo de trabalho (também chamado atividade) de um sistema de software.
- Os diagramas de atividades são compostos de símbolos de uso especial, como símbolos do estado da ação, losangos e pequenos círculos. Símbolos são conectados por setas de transição que representam o fluxo da atividade.
- Como ocorre com o pseudocódigo, os diagramas de atividades ajudam os programadores a desenvolver e representar algoritmos.
- O estado de uma ação é representado por um retângulo com seus lados esquerdo e direito substituídos por arcos curvados para fora. A entrada da ação aparece dentro do estado da ação.
- As setas em um diagrama de atividades representam transições, as quais indicam a ordem em que ocorrem as ações representadas pelo estado da ação.
- O círculo sólido localizado na parte superior de um diagrama de atividades representa o estado inicial — o começo do fluxo de trabalho de o programa realizar as ações modeladas.
- O círculo sólido cercado por um círculo vazio que aparece na parte inferior do diagrama de atividades representa o estado final — o fim do fluxo de trabalho depois que o programa realiza suas ações.
- Retângulos com o canto superior direito dobrado chamam-se notas na UML. As notas são observações explanatórias que descrevem o propósito dos símbolos no diagrama. Uma linha pontilhada conecta cada nota ao elemento que a nota descreve.
- Um losango ou símbolo de decisão em um diagrama de atividades indica que uma decisão deve ser tomada. O fluxo de trabalho continua longo de um caminho determinado pelas condições de guarda do símbolo associado, que podem ser verdadeiras ou falsas. Cada seta de transição que sai de um símbolo de decisão tem uma condição de guarda (especificada entre colchetes ao lado da seta de transição). Se a condição de guarda for verdadeira, o fluxo de trabalho entra no estado de ação para o qual a seta de transição aponta.
- Um losango em um diagrama de atividades também representa o símbolo de agregação, que une dois fluxos de atividade em um. Um símbolo de agregação contém duas ou mais setas de transição apontando para o losango e somente uma seta de transição apontando a partir do losango para indicar a conexão de múltiplos fluxos de atividades a fim de continuar a atividade.
- O refinamento de passo a passo de cima para baixo é um processo para refinar o pseudocódigo mantendo uma representação completa do problema.
- Há três tipos de estruturas de controle — seqüência, seleção e repetição.
- A estrutura de seqüência é parte integrante do C++ — por padrão, as instruções são executadas na ordem em que elas aparecem.
- Uma estrutura de seleção escolhe entre cursos alternativos da ação.
- A instrução de uma única ~~seleção~~ (seleciona) uma ação se uma condição for verdadeira ou ignora a ação se a condição for falsa.
- A instrução de seleção ~~dupla~~ (seleciona) uma ação se uma condição for verdadeira e realiza uma ação diferente se a condição for falsa.

- Para incluir várias instruções no corpo de uma instrução, inclua as instruções dentro de chaves ({}). Um conjunto de instruções contidas dentro de um par de chaves é chamado bloco. Um bloco pode ser colocado em qualquer lugar de um programa em que uma única instrução pode ser colocada.
- Uma instrução nula, indicando que nenhuma ação deve ser tomada, é indicada por um ponto-e-vírgula (;).
- Uma instrução repetição específica que uma ação deve ser repetida enquanto algumas condições permanecem verdadeiras.
- Um valor que contém uma parte fracionária é referido como um número de ponto flutuante e é representado aproximadamente por tipos de comfloat edouble
- A repetição controlada por contador é utilizada quando o número de repetições é conhecido antes de um loop começar a executar, isto é, quando há repetição definida.
- O operador de coerção unária pode ser utilizado para criar uma cópia de ponto flutuante temporária de seu operando.
- Os operadores unários aceitam apenas um operando; operadores binários aceitam dois.
- O manipulador de fluxo parametrizado indica o número de dígitos de precisão que deve ser exibido à direita do ponto de fração decimal.
- O manipulador de fluxo indica que a saída dos valores de ponto flutuante deve ser no chamado formato de ponto fixo, em oposição à notação científica.
- A repetição controlada por sentinela é utilizada quando o número de repetições não é conhecido antes de um loop começar a executar, isto é, quando há repetição indefinida.
- Uma instrução de controle aninhada aparece no corpo de outra instrução de controle.
- O C++ fornece os operadores de atribuição aritmética para abreviar expressões de atribuição.
- O operador de incremento operador de decremento incrementam ou decrementam uma variável por 1, respectivamente. Se o operador for prefixado à variável, a variável é primeiro incrementada ou decrementada por 1 e então seu novo valor é utilizado na expressão em que aparece. Se o operador for pós-fixado à variável, a variável é primeiro utilizada na expressão em que aparece e então o valor da variável é incrementado ou é decrementado por 1.

## Terminologia

|                                           |                                                      |                                               |
|-------------------------------------------|------------------------------------------------------|-----------------------------------------------|
| ação                                      | erro off-by-one                                      | iterar                                        |
| 'bombing'                                 | estado da ação                                       | linha pontilhada                              |
| aninhamento                               | estado final                                         | loop aninhado dentro de um loop               |
| aninhamento de instruções de controle     | execução seqüencial                                  | loop, condição de continuação                 |
| aproximação de números de ponto flutuante | expressão condicional                                | losango, símbolo                              |
| arredondando                              | expressão de ação                                    | manipulador de fluxo                          |
| associar da direita para a esquerda       | fluxo de trabalho de parte de um sistema de software | manipulador de fluxo                          |
| associar da esquerda para a direita       | formato de ponto fixo                                | manipulador de fluxo não parametrizado        |
| bloco                                     | goto, instrução                                      | manipulador de fluxo parametrizado            |
| bool                                      | incremento, operador (                               | manipulador de decisão                        |
| cálculo de média                          | instrução composta                                   | manipulador de fluxo point                    |
| constante de ponto flutuante              | instrução de controle                                | modelo de programação de ação/decisão         |
| contador                                  | instrução de controle aninhado                       | nota                                          |
| controle de programa                      | instrução de controle de entrada                     | notação científica                            |
| conversão explícita                       | única                                                | única de ponto flutuante de precisão dupla    |
| conversão implícita                       | instrução de dupla seleção                           | número de ponto flutuante de precisão simples |
| decremento, operador (                    | instrução de loop                                    | operador aritmético binário                   |
| design orientado a objetos (              | instrução de múltipla seleção                        | operador condicional (                        |
| diagrama de atividades                    | instrução de uma única seleção                       | operador de atribuição de adição (            |
| diagrama de atividades de instruções      | instrução executável                                 | operador de coerção                           |
| seqüência                                 | instrução nula                                       | operador de pós-decremento                    |
| divisão de inteiro                        | instrução vazia                                      | operador de pós-incremento                    |
| eliminação de goto                        | iterações de loop                                    | operador de pré-decremento                    |
| empilhamento de instruções de controle    | iterações de um loop                                 | operador de pré-incremento                    |
| erro fatal de lógica                      |                                                      | operador ternário                             |
|                                           |                                                      | operador unário                               |

|                                            |                                              |                               |
|--------------------------------------------|----------------------------------------------|-------------------------------|
| operador unário de adição (                | procedimento                                 | símbolo de intercalação       |
| operador unário de coerção                 | programação estruturada                      | símbolo de seta               |
| operador unário de subtração (             | promoção                                     | símbolo de seta de transição  |
| operadores de atribuição                   | promoção de inteiro                          | tipo de dados                 |
| operadores de atribuição aritméticos       | pseudocódigo                                 | tipo de dados                 |
| operando                                   | refinamento passo a passo de cima para baixo | tipo de dados                 |
| ordem em que as ações devem ser executadas | controlada por contador                      | total                         |
| palavras-chave                             | repetição controlado por sentinelas          | transferência de controle     |
| ponto flutuante, número                    | repetição definida                           | transição                     |
| pós-decremento                             | repetição indefinida                         | truncar                       |
| pós-incremento                             | repetição, instrução                         | valor "lixo"                  |
| precedência de operadores                  | segundo refinamento                          | valor de flag                 |
| precisão                                   | seleção, instrução                           | valor de sentinelas           |
| precisão-padrão                            | sequência, instrução                         | valor de sinal                |
| pré-decremento                             | símbolo da ação de estado                    | valor fictício                |
| pré-incremento                             | símbolo de círculo pequeno                   | valor indefinido              |
| primeiro refinamento                       | símbolo de círculo sólido                    | while, instrução de repetição |
| problema de oscilante                      | símbolo de decisão                           |                               |

### Exercícios de revisão

- 4.1 Responda cada uma das seguintes perguntas.
- Todos os programas podem ser escritos em termos de três tipos de estruturas de controle: , , .
  - A instrução de seleção é utilizada para executar uma ação quando a condição é diferente quando essa condição é verdadeira.
  - Repetir um conjunto de instruções por um número específico de vezes é chamado de repetição
  - Quando não se sabe antecipadamente quantas vezes um conjunto de instruções será repetido, um valor pode ser utilizado.
- 4.2 Escreva quatro instruções C++ diferentes que adicionam 1 à variável do tipo inteiro
- 4.3 Escreva instruções C++ para realizar cada uma das seguintes tarefas:
- Em uma instrução, atribua a soma deyaoz ao finalmente o valor de
  - Determine se o valor devariável é maior que 10. Se for, imprima "greater than 10".
  - Pré-decremente axvariável e então subtraia o resultado da variável
  - Calcule o resto da divisão pelo e atribua o resultado a
- 4.4 Escreva instruções C++ para realizar cada uma das seguintes tarefas.
- Declare variáveis que serão do tipo
  - Configure a variável
  - Configure a variável
  - Adicione variável e atribua o resultado à variável
  - Imprima a soma de: " seguido pelo valor das variáveis
- 4.5 Combine as instruções escritas no Exercício 4.4 em um programa em C++ que calcula e imprime a soma dos inteiros de 1 a 10. Utilize a estrutura de loop para fazer loop pelas instruções de cálculo e incremento. O loop deve terminar quando o valor de
- 4.6 Mostre os valores de cada variável depois que o cálculo é realizado. Assuma que quando cada instrução começa a executar, todas as variáveis têm o valor inteiro 5.
- product \*= x++;
  - quotient /= ++x;
- 4.7 Escreva instruções C++ únicas que realizem o seguinte:
- Insiram a variável de<> <>
  - Insiram a variável de<> <>
  - Configurem variável de<> <>
  - Configurem variável de<> <>
  - Multipliquem a variável e atribuam o resultado a
  - Pós-incrememetem variável
  - Determinem se menor que ou igual a
  - Realizem saída da variável de<> <>

- 4.8 Escreva um programa C++ que utiliza as instruções do Exercício 4.7 para calcular o potencial. O programa deve ter uma instrução de repetição.
- 4.9 Identifique e corrija os erros em cada uma das seguintes:
- `while ( c <= 5 )`  
`{`  
 `product *= c;`  
 `c++;`
  - `cin << value;`
  - `if ( gender == 1 )`  
 `cout << "Woman" << endl;`  
`else ;`  
 `cout << "Man" << endl;`
- 4.10 O que há de errado com a instrução de ação de repetição?
- ```
while ( z >= 0 )
    sum += z;
```

Respostas dos exercícios de revisão

- 4.1 a) Seqüência, seleção e repetição. b) Controlada por contador ou definida. d) Sentinel, sinal, flag ou dummy.
- 4.2 `x = x + 1;`
`x += 1;`
`++x;`
`x++;`
- 4.3 a) `z = x++ + y;`
b) `if (count > 10)`
 `cout << "Count is greater than 10" << endl;`
c) `total -= -x;`
d) `q %= divisor;`
`q = q % divisor;`
- 4.4 a) `int sum;`
b) `x = 1;`
c) `sum = 0;`
d) `sum += x;`
ou
`sum = sum + x;`
e) `cout << "The sum is: " << sum << endl;`
- 4.5 Examine o código a seguir:

```
1 // Exercício 4.5 Solução: ex04_05.cpp
2 // Calcula a soma dos inteiros de 1 a 10.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int sum; // armazena a soma de inteiros de 1 a 10
10    x = 1; // contagem de 1
11    sum = 0; // inicializa a soma
12
13    while ( x <= 10 ) // faz o loop 10 vezes
14    {
15        sum += x; // adiciona x à soma
16    }
17}
```

```

18     x++; // incrementa x
19 } // fim do while
20
21 cout << "The sum is: " << sum << endl;
22 return 0; // indica terminação bem-sucedida
23 } // fim de main

```

The sum is: 55

- 4.6 a) product = 25, x = 6;
 b) quotient = 0, x = 6;

```

1 // Exercício 4.6 Solução: ex04_06.cpp
2 // Calcula o valor de produto e quociente.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 5;
10    int product = 5;
11    int quotient = 5;
12
13    // parte a
14    product *= x++; // instrução da parte a
15    cout << "Value of product after calculation: " << product << endl;
16    cout << "Value of x after calculation: " << x << endl << endl;
17
18    // parte b
19    quotient /= ++x; // instrução da parte b
20    cout << "Value of quotient after calculation: " << quotient << endl;
21    cout << "Value of x after calculation: " << x << endl << endl;
22    return 0; // indica terminação bem-sucedida
23 } // fim de main

```

Value of product after calculation: 25

Value of x after calculation: 6

Value of quotient after calculation: 0

Value of x after calculation: 6

- 4.7 a) cin >> x;
 b) cin >> y;
 c) i = 1;
 d) power = 1;
 e) power *= x;

OU
 power = power * x;
 f) i++;
 g) if (i <= y)
 h) cout << power << endl;

- 4.8 Examine o código a seguir:

```

1 // Exercício 4.8 Solução: ex04_08.cpp
2 // Eleva x à potência de y.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10    int x; // base
11    int y; // expoente
12    int i; // conta de 1 a y
13
14    int power; // utilizado para calcular x elevado à potência de y
15    i = 1; // inicializa i para começar a contar de 1
16    power = 1; // inicializa power
17
18    cout << "Enter base as an integer: " ; // solicita a base
19    cin >> x; // insere a base
20
21    cout << "Enter exponent as an integer: " ; // solicita expoente
22    cin >> y; // insere o expoente
23
24    // conta de 1 a y e multiplica potência por x a cada vez
25    while ( i <= y )
26    {
27        power *= x;
28        i++;
29    } // fim do while
30
31    cout << power << endl; // exibe o resultado
32
33 } // fim de main

```

Enter base as an integer: 2
 Enter exponent as an integer: 3
 8

- 4.9 a) Erro: Está faltando a chave direita de fechamento do corpo do loop.
 Correção: Adicionar a chave direita de fechamento depois da instrução
 b) Erro: Utilizou inserção de fluxo em vez de extração de fluxo.
 Correção: Alterar para <>
 c) Erro: Ponto-e-vírgula depois da instrução é um erro de lógica. A segunda instrução de saída sempre será executada.
 Correção: Remover ponto-e-vírgula depois de
- 4.10 O valor da variável `h` é alterado na instrução. Portanto, se a condição de continuação do loop for sempre true, um loop infinito é criado. Para evitar o loop infinito, é necessário decrementar o valor de `h` de modo que acabe se tornando menor que 0.

Exercícios

- 4.11 Identifique e corrija o(s) erro(s) em cada um dos seguintes:

a) `if (age >= 65);`
 `cout << "Age is greater than or equal to 65" << endl;`
`else`
 `cout << "Age is less than 65 << endl" ;`

b) `if (age >= 65)`
 `cout << "Age is greater than or equal to 65" << endl;`
`else;`
 `cout << "Age is less than 65 << endl" ;`

```

c) int x = 1, total;
    while ( x <= 10)
    {
        total += x;
        x++;
    }
d) While ( x <= 100)
    total += x;
    x++;
e)while ( y > 0)
{
    cout << y << endl;
}
y++;

```

- 4.12 O que o programa a seguir imprime?

```

1 // Exercício 4.12: ex04_12.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int y; // declara y
10    int x = 1; // inicializa x
11    int total = 0; // inicializa o total
12
13    while ( x <= 10 ) // faz o loop 10 vezes
14    {
15        cout << x; // realiza os cálculos
16        cout << endl; // gera a saída dos resultados
17        total += y; // adiciona y a total
18        x++; // incrementa o contador x
19    } // fim do while
20
21    cout << "Total is " << total << endl; // exibe o resultado
22    return 0; // indica terminação bem-sucedida
23 } // fim de main

```

Para os exercícios 4.13 a 4.16, siga cada um destes passos:

- Leia a declaração do problema.
- Formule o algoritmo utilizando pseudocódigo e refinamento passo a passo de cima para baixo.
- Escreva um programa C++.
- Teste, depure e execute o programa C++.

- 4.13 Os motoristas se preocupam com o consumo de combustível dos seus automóveis. Um motorista monitorou vários tanques cheios de gasolina registrando a quilometragem dirigida e a quantidade de combustível em litros utilizados para cada tanque cheio. Desenvolva

um programa que faça o mesmo que o motorista fez. O seu programa deve solicitar ao usuário a quantidade de litros de combustível utilizados para a sua viagem e a quilometragem percorrida. Ele deve calcular a média de consumo por quilômetro percorrido e a quantidade de litros de combustível utilizados para a sua viagem.

Entre com a quilometragem (-1 para sair): 287
 Entre com os litros: 13
 km/litro deste tanque: 22.076923
 Total km/litro: 22.076923

Entre com a quilometragem (-1 para sair): 200
 Entre com os litros: 10
 km/litro deste tanque: 20.000000
 Total km/litro: 21.173913

Entre com a quilometragem (-1 para sair): 120
 Entre com os litros: 5
 km/litro deste tanque: 24.000000
 Total km/litro: 21.878571

Entre com a quilometragem: (-1 para sair): -1

- 4.14 Desenvolva um programa C++ que determinará se um cliente de uma loja de departamentos excedeu o limite de crédito em uma corrente. Para cada cliente, os seguintes fatos estão disponíveis:

- Número de conta (um inteiro)
- Balanço no início do mês
- Total de todos os itens cobrados desse cliente no mês
- Total de pagamentos feitos pelo cliente no mês
- Límite autorizado de crédito

O programa deve utilizar uma estrutura `while` para inserir cada um desses fatos, calcular o novo saldo (= saldo inicial + taxas – créditos) e determinar se o novo saldo excede o limite de crédito do cliente. Para aqueles clientes cujo limite de crédito é excedido, o programa deve exibir o número da conta do cliente, o limite de crédito, o novo saldo e a mensagem 'Límite de crédito excedido'.

Entre com o numero da conta (-1 para terminar): 100
 Entre com o saldo inicial: 5394.78
 Entre com o total de taxas: 1000.00
 Entre com o total de creditos: 500.00
 Entre com o limite de credito: 5500.00
 Novo saldo: 5894.78
 Conta: 100
 Limite de credito: 5500.00
 Saldo: 5894.78
 Limite de credito ultrapassado.

Entre com o numero da conta (ou -1 para sair): 200
 Entre com o saldo inicial: 1000.00
 Entre com o total de taxas: 123.45
 Entre com o total de creditos: 321.00
 Entre com o limite de credito: 1500.00
 Novo saldo: 802.45

Entre com o numero da conta (ou -1 sair): 300
 Entre com o saldo inicial: 500.00
 Entre com o total de taxas: 274.73
 Entre com o total de creditos: 160.00
 Entre com o limite de credito: 800.00
 Novo saldo: 674.73

Entre com o numero da conta (ou -1 sair): -1

- 4.15 Uma grande indústria química paga sua equipe de vendas por comissão. Os vendedores recebem \$200 por semana mais 9% de suas brutas por semana. Por exemplo, um vendedor que comercializa um valor de \$ 5.000 em produtos químicos por semana recebe \$ 200 reais mais 9% de \$ 5.000, ou seja, \$ 450.

9% de \$ 5.000, ou um total de \$ 650. Desenvolva um programa em C++ que lida com as vendas brutas de cada vendedor durante a última semana e calcula e exibe os rendimentos desse vendedor. Procresse os números de um vendedor vez.

Entre com as vendas em dolar (-1 para terminar): 5000.00
Salario: \$650.00

Entre com as vendas em dolar (-1 para terminar): 6000.00
Salario: \$740.00

Entre com as vendas em dolar (-1 para terminar): 7000.00
Salario: \$830.00

Entre com as vendas em dolar (-1 para terminar): -1

- 4.16 Desenvolva um programa em C++ que utilize **while** para determinar o pagamento bruto de cada um dos vários funcionários. A empresa paga 'hora normal' pelas primeiras 40 horas trabalhadas por empregado e paga 'horas extras' com 50% de gratificação por todas as horas trabalhadas além das primeiras 40 horas. Você recebe uma lista dos empregados da empresa, o número de horas trabalhadas por empregado na última semana e o salário-hora de cada empregado. Seu programa deve aceitar a entrada dessas informações para cada empregado e então determinar e exibir o salário bruto do empregado.

Entre com as horas trabalhadas (-1 para terminar): 39
Entre com o valor por hora trabalhada (\$00.00): 10.00
Salario: \$390.00

Entre com as horas trabalhadas (-1 para terminar): 40
Entre com o valor por hora trabalhada (\$00.00): 10.00
Salario: \$400.00

Entre com as horas trabalhadas (-1 para terminar): 41
Entre com o valor por hora trabalhada (\$00.00): 10.00
Salario: \$415.00

Entre com as horas trabalhadas (-1 para terminar): -1

- 4.17 O processo de localizar o maior número (isto é, o máximo de um grupo de valores) é freqüentemente utilizado em aplicativos de computador. Por exemplo, um programa que determina o vencedor de uma competição de vendas insere o número de unidades vendidas por vendedor. O vendedor que vende mais unidades ganha a competição. Escreva um programa em pseudocódigo, e então um programa em C++, que utiliza uma **instrução** para determinar e imprimir o maior número dos dez números inseridos pelo usuário. Seu programa deve utilizar três variáveis, como segue:
- counter: Um contador para contar até 10 (isto é, monitorar quantos números foram inseridos e determinar quando todos os 10 números foram processados).
 - number: A entrada numérica atual para o programa.
 - largest: O maior número encontrado até agora.

- 4.18 Escreva um programa em C++ que utilize **while** para gerar a seguinte tabela de valores:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

- 4.19 Utilizando uma abordagem semelhante àquela do Exercício 4.17, escreva um programa que imprime a seguinte tabela de valores. [O número deve ser inserido apenas uma vez.]

- 4.20 O programa de resultados de teste da Figura 4.16–Figura 4.18 supõe que qualquer valor inserido pelo usuário que não for 1 deve ser Modifique o aplicativo para validar suas entradas. Em qualquer entrada, se o valor entrado for diferente de 1 ou 2, continua o loop até usuário inserir um valor correto.

- 4.21 O que o programa a seguir imprime?

```

1 // Exercício 4.21: ex04_21.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int count = 1; // inicializa a contagem
10
11    while ( count <= 10 ) // faz o loop 10 vezes
12    {
13        // saída da linha de texto
14        cout << ( count % 2 ? "*****" : "+++++++" ) << endl;
15        count++; // incrementa a contagem
16    } // fim do while
17
18    return 0; // indica terminação bem-sucedida
19 } // fim de main

```

- 4.22 O que o programa a seguir imprime?

```

1 // Exercício 4.22: ex04_22.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4
5 using std::endl;
6
7 int main()
8 {
9     int row = 10; // inicializa a linha
10    int column; // declara a coluna
11
12    while ( row >= 1 ) // faz loop até linha < 1
13    {
14        column = 1; // configura coluna como 1 quando a iteração inicia
15
16        while ( column <= 10 ) // faz o loop 10 vezes
17        {
18            cout << ( row % 2 ? "<" : ">" ); // saída
19            column++; // incrementa coluna
20        } // fim do while interno
21
22        row--; // decrementa linha
23        cout << endl; // inicia nova linha de saída
24    } // fim do while externo
25
26    return 0; // indica terminação bem-sucedida
27 } // fim de main

```

- 4.23 (Problema de oscilação) Mostre a saída de cada um dos seguintes programas. Qual é o resultado? Observe que o compilador ignora o recuo em um programa C++. O compilador C++ sempre considera que ele seja

instruído a fazer de modo diferente pela colocaçāo de chaves, o programador pode não estar certo de qual responde a qual o que é referido como problema de chave' ('dangerous'. Eliminamos o recuo do código a seguir para tornar o problema mais desafiante [as convenções de recuo que você aprendeu.]

```
a) if ( x < 10)
    if ( y > 10)
        cout << "*****" << endl;
    else
        cout << "#####" << endl;
        cout << "$$$$$" << endl;
b) if ( x < 10)
{
    if ( y > 10)
        cout << "*****" << endl;
    else
    {
        cout << "#####" << endl;
        cout << "$$$$$" << endl;
    }
}
```

- 4.24 [Outro problema desse oscilante] Modifique o seguinte código para produzir a saída mostrada. Utilize técnicas de recuo adequadas. Você não deve fazer nenhuma alteração além de inserir chaves. O compilador ignora o recuo em um programa C++. Eliminamos o recuo do seguinte código para tornar o problema mais desafiante [que não seja necessária nenhuma modificação.]

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&&" << endl;
```

- a) Supondo que $ey = 8$, a seguinte saída é produzida.

```
#####
&&&&
```

- b) Supondo que $ey = 8$, a seguinte saída é produzida.

```
@@@@@
```

- c) Supondo que $ey = 8$, a seguinte saída é produzida.

```
#####
&&&&
```

- d) Supondo que $ey = 7$, a seguinte saída é produzida [Todas as três últimas instruções desse código pertencem a um bloco.]

```
#####
$$$$$
```

- 4.25 Escreva um programa que lê o tamanho do lado de um quadrado e, então, imprime um quadrado vazado com asteriscos e espaço branco. Seu programa deve trabalhar com quadrados de todos os tamanhos entre 1 e 20. Por exemplo, se seu programa lê um tamar 5, ele deve imprimir

```
*****
* *
* *
* *
*****
*****
```

- 4.26 Um palíndromo é um número ou uma frase de texto que é lido da mesma forma da esquerda para a direita e da direita para a esquerda. Por exemplo, cada um dos seguintes inteiros de cinco dígitos é um palíndromo: 12321, 55555, 45554 e 11611. Escreva um programa que lê em um inteiro de cinco dígitos e determine se ele é palíndromo. [Use operadores de divisão e módulo para separar o número em seus dígitos individuais.]

- 4.27 Insira um inteiro contendo somente 0s e 1s (isto é, um inteiro ‘binário’) e imprima seu equivalente decimal. Utilize os operadores de módulo.

~~é dividido por 2, o resultado é dividido por 2, e assim por diante. O resto da divisão é o dígito mais à direita. O resultado é dividido por 2, o resto é o dígito de posicional de 2, e assim por diante. O resultado é dividido por 2, o resto é o dígito de posicional de 4, e assim por diante. O resultado é dividido por 8, o resto é o dígito de posicional de 8, e assim por diante. O resultado é dividido por 16, o resto é o dígito de posicional de 16, e assim por diante. O resultado é dividido por 32, o resto é o dígito de posicional de 32, e assim por diante. O resultado é dividido por 64, o resto é o dígito de posicional de 64, e assim por diante. O resultado é dividido por 128, o resto é o dígito de posicional de 128, e assim por diante. O resultado é dividido por 256, o resto é o dígito de posicional de 256, e assim por diante. O resultado é dividido por 512, o resto é o dígito de posicional de 512, e assim por diante. O resultado é dividido por 1024, o resto é o dígito de posicional de 1024, e assim por diante. O resultado é dividido por 2048, o resto é o dígito de posicional de 2048, e assim por diante. O resultado é dividido por 4096, o resto é o dígito de posicional de 4096, e assim por diante. O resultado é dividido por 8192, o resto é o dígito de posicional de 8192, e assim por diante. O resultado é dividido por 16384, o resto é o dígito de posicional de 16384, e assim por diante. O resultado é dividido por 32768, o resto é o dígito de posicional de 32768, e assim por diante. O resultado é dividido por 65536, o resto é o dígito de posicional de 65536, e assim por diante. O resultado é dividido por 131072, o resto é o dígito de posicional de 131072, e assim por diante. O resultado é dividido por 262144, o resto é o dígito de posicional de 262144, e assim por diante. O resultado é dividido por 524288, o resto é o dígito de posicional de 524288, e assim por diante. O resultado é dividido por 1048576, o resto é o dígito de posicional de 1048576, e assim por diante. O resultado é dividido por 2097152, o resto é o dígito de posicional de 2097152, e assim por diante. O resultado é dividido por 4194304, o resto é o dígito de posicional de 4194304, e assim por diante. O resultado é dividido por 8388608, o resto é o dígito de posicional de 8388608, e assim por diante. O resultado é dividido por 16777216, o resto é o dígito de posicional de 16777216, e assim por diante. O resultado é dividido por 33554432, o resto é o dígito de posicional de 33554432, e assim por diante. O resultado é dividido por 67108864, o resto é o dígito de posicional de 67108864, e assim por diante. O resultado é dividido por 134217728, o resto é o dígito de posicional de 134217728, e assim por diante. O resultado é dividido por 268435456, o resto é o dígito de posicional de 268435456, e assim por diante. O resultado é dividido por 536870912, o resto é o dígito de posicional de 536870912, e assim por diante. O resultado é dividido por 1073741824, o resto é o dígito de posicional de 1073741824, e assim por diante. O resultado é dividido por 2147483648, o resto é o dígito de posicional de 2147483648, e assim por diante. O resultado é dividido por 4294967296, o resto é o dígito de posicional de 4294967296, e assim por diante. O resultado é dividido por 8589934592, o resto é o dígito de posicional de 8589934592, e assim por diante. O resultado é dividido por 17179869184, o resto é o dígito de posicional de 17179869184, e assim por diante. O resultado é dividido por 34359738368, o resto é o dígito de posicional de 34359738368, e assim por diante. O resultado é dividido por 68719476736, o resto é o dígito de posicional de 68719476736, e assim por diante. O resultado é dividido por 137438953472, o resto é o dígito de posicional de 137438953472, e assim por diante. O resultado é dividido por 274877906944, o resto é o dígito de posicional de 274877906944, e assim por diante. O resultado é dividido por 549755813888, o resto é o dígito de posicional de 549755813888, e assim por diante. O resultado é dividido por 1099511627776, o resto é o dígito de posicional de 1099511627776, e assim por diante. O resultado é dividido por 2199023255552, o resto é o dígito de posicional de 2199023255552, e assim por diante. O resultado é dividido por 4398046511104, o resto é o dígito de posicional de 4398046511104, e assim por diante. O resultado é dividido por 8796093022208, o resto é o dígito de posicional de 8796093022208, e assim por diante. O resultado é dividido por 17592186044416, o resto é o dígito de posicional de 17592186044416, e assim por diante. O resultado é dividido por 35184372088832, o resto é o dígito de posicional de 35184372088832, e assim por diante. O resultado é dividido por 70368744177664, o resto é o dígito de posicional de 70368744177664, e assim por diante. O resultado é dividido por 14073748835532, o resto é o dígito de posicional de 14073748835532, e assim por diante. O resultado é dividido por 28147497671064, o resto é o dígito de posicional de 28147497671064, e assim por diante. O resultado é dividido por 56294995342128, o resto é o dígito de posicional de 56294995342128, e assim por diante. O resultado é dividido por 112589990684256, o resto é o dígito de posicional de 112589990684256, e assim por diante. O resultado é dividido por 225179981368512, o resto é o dígito de posicional de 225179981368512, e assim por diante. O resultado é dividido por 450359962737024, o resto é o dígito de posicional de 450359962737024, e assim por diante. O resultado é dividido por 900719925474048, o resto é o dígito de posicional de 900719925474048, e assim por diante. O resultado é dividido por 1801439850948096, o resto é o dígito de posicional de 1801439850948096, e assim por diante. O resultado é dividido por 3602879701896192, o resto é o dígito de posicional de 3602879701896192, e assim por diante. O resultado é dividido por 7205759403792384, o resto é o dígito de posicional de 7205759403792384, e assim por diante. O resultado é dividido por 14411518807584768, o resto é o dígito de posicional de 14411518807584768, e assim por diante. O resultado é dividido por 28823037615169536, o resto é o dígito de posicional de 28823037615169536, e assim por diante. O resultado é dividido por 57646075230339072, o resto é o dígito de posicional de 57646075230339072, e assim por diante. O resultado é dividido por 115292150460678144, o resto é o dígito de posicional de 115292150460678144, e assim por diante. O resultado é dividido por 230584300921356288, o resto é o dígito de posicional de 230584300921356288, e assim por diante. O resultado é dividido por 461168601842712576, o resto é o dígito de posicional de 461168601842712576, e assim por diante. O resultado é dividido por 922337203685425152, o resto é o dígito de posicional de 922337203685425152, e assim por diante. O resultado é dividido por 1844674407370850304, o resto é o dígito de posicional de 1844674407370850304, e assim por diante. O resultado é dividido por 3689348814741700608, o resto é o dígito de posicional de 3689348814741700608, e assim por diante. O resultado é dividido por 7378697629483401216, o resto é o dígito de posicional de 7378697629483401216, e assim por diante. O resultado é dividido por 14757395258966802432, o resto é o dígito de posicional de 14757395258966802432, e assim por diante. O resultado é dividido por 29514790517933604864, o resto é o dígito de posicional de 29514790517933604864, e assim por diante. O resultado é dividido por 59029581035867209728, o resto é o dígito de posicional de 59029581035867209728, e assim por diante. O resultado é dividido por 118059162071734419456, o resto é o dígito de posicional de 118059162071734419456, e assim por diante. O resultado é dividido por 236118324143468838912, o resto é o dígito de posicional de 236118324143468838912, e assim por diante. O resultado é dividido por 472236648286937677824, o resto é o dígito de posicional de 472236648286937677824, e assim por diante. O resultado é dividido por 944473296573875355648, o resto é o dígito de posicional de 944473296573875355648, e assim por diante. O resultado é dividido por 1888946593147750711296, o resto é o dígito de posicional de 1888946593147750711296, e assim por diante. O resultado é dividido por 3777893186295501422592, o resto é o dígito de posicional de 3777893186295501422592, e assim por diante. O resultado é dividido por 7555786372591002845184, o resto é o dígito de posicional de 7555786372591002845184, e assim por diante. O resultado é dividido por 1511157274582005689036, o resto é o dígito de posicional de 1511157274582005689036, e assim por diante. O resultado é dividido por 3022314549164011378072, o resto é o dígito de posicional de 3022314549164011378072, e assim por diante. O resultado é dividido por 6044629098328022756144, o resto é o dígito de posicional de 6044629098328022756144, e assim por diante. O resultado é dividido por 12089258196656045512288, o resto é o dígito de posicional de 12089258196656045512288, e assim por diante. O resultado é dividido por 24178516393312091024576, o resto é o dígito de posicional de 24178516393312091024576, e assim por diante. O resultado é dividido por 48357032786624182049152, o resto é o dígito de posicional de 48357032786624182049152, e assim por diante. O resultado é dividido por 96714065573248364098304, o resto é o dígito de posicional de 96714065573248364098304, e assim por diante. O resultado é dividido por 193428131146496728196608, o resto é o dígito de posicional de 193428131146496728196608, e assim por diante. O resultado é dividido por 386856262292993456393216, o resto é o dígito de posicional de 386856262292993456393216, e assim por diante. O resultado é dividido por 773712524585986912786432, o resto é o dígito de posicional de 773712524585986912786432, e assim por diante. O resultado é dividido por 154742504917197382557264, o resto é o dígito de posicional de 154742504917197382557264, e assim por diante. O resultado é dividido por 309485009834394765114528, o resto é o dígito de posicional de 309485009834394765114528, e assim por diante. O resultado é dividido por 618970019668789530229056, o resto é o dígito de posicional de 618970019668789530229056, e assim por diante. O resultado é dividido por 1237940039337579060458112, o resto é o dígito de posicional de 1237940039337579060458112, e assim por diante. O resultado é dividido por 2475880078675158120916224, o resto é o dígito de posicional de 2475880078675158120916224, e assim por diante. O resultado é dividido por 4951760157350316241832448, o resto é o dígito de posicional de 4951760157350316241832448, e assim por diante. O resultado é dividido por 9903520314700632483664896, o resto é o dígito de posicional de 9903520314700632483664896, e assim por diante. O resultado é dividido por 1980704062940126496732992, o resto é o dígito de posicional de 1980704062940126496732992, e assim por diante. O resultado é dividido por 3961408125880252993465984, o resto é o dígito de posicional de 3961408125880252993465984, e assim por diante. O resultado é dividido por 7922816251760505986931968, o resto é o dígito de posicional de 7922816251760505986931968, e assim por diante. O resultado é dividido por 1584563253520101193886392, o resto é o dígito de posicional de 1584563253520101193886392, e assim por diante. O resultado é dividido por 3169126507040202387772784, o resto é o dígito de posicional de 3169126507040202387772784, e assim por diante. O resultado é dividido por 6338253014080404775545568, o resto é o dígito de posicional de 6338253014080404775545568, e assim por diante. O resultado é dividido por 1267650602816080951109136, o resto é o dígito de posicional de 1267650602816080951109136, e assim por diante. O resultado é dividido por 2535301205632161902218272, o resto é o dígito de posicional de 2535301205632161902218272, e assim por diante. O resultado é dividido por 5070602411264323804436544, o resto é o dígito de posicional de 5070602411264323804436544, e assim por diante. O resultado é dividido por 10141204822528647608873088, o resto é o dígito de posicional de 10141204822528647608873088, e assim por diante. O resultado é dividido por 20282409645057295217746176, o resto é o dígito de posicional de 20282409645057295217746176, e assim por diante. O resultado é dividido por 40564819290114590435492352, o resto é o dígito de posicional de 40564819290114590435492352, e assim por diante. O resultado é dividido por 81129638580229180870984704, o resto é o dígito de posicional de 81129638580229180870984704, e assim por diante. O resultado é dividido por 16225927716045836174196948, o resto é o dígito de posicional de 16225927716045836174196948, e assim por diante. O resultado é dividido por 32451855432091672348393896, o resto é o dígito de posicional de 32451855432091672348393896, e assim por diante. O resultado é dividido por 64903710864183344696787792, o resto é o dígito de posicional de 64903710864183344696787792, e assim por diante. O resultado é dividido por 129807421728366689393575584, o resto é o dígito de posicional de 129807421728366689393575584, e assim por diante. O resultado é dividido por 259614843456733378787151168, o resto é o dígito de posicional de 259614843456733378787151168, e assim por diante. O resultado é dividido por 519229686913466757574302336, o resto é o dígito de posicional de 519229686913466757574302336, e assim por diante. O resultado é dividido por 1038459373826933515148604672, o resto é o dígito de posicional de 1038459373826933515148604672, e assim por diante. O resultado é dividido por 2076918747653867030297209344, o resto é o dígito de posicional de 2076918747653867030297209344, e assim por diante. O resultado é dividido por 4153837495307734060594418688, o resto é o dígito de posicional de 4153837495307734060594418688, e assim por diante. O resultado é dividido por 8307674990615468121188837376, o resto é o dígito de posicional de 8307674990615468121188837376, e assim por diante. O resultado é dividido por 1661534998123093624237767472, o resto é o dígito de posicional de 1661534998123093624237767472, e assim por diante. O resultado é dividido por 3323069996246187248475534944, o resto é o dígito de posicional de 3323069996246187248475534944, e assim por diante. O resultado é dividido por 6646139992492374496951069888, o resto é o dígito de posicional de 6646139992492374496951069888, e assim por diante. O resultado é dividido por 1329227998498474893990213976, o resto é o dígito de posicional de 1329227998498474893990213976, e assim por diante. O resultado é dividido por 2658455996996949787980427952, o resto é o dígito de posicional de 2658455996996949787980427952, e assim por diante. O resultado é dividido por 5316911993993899575960855904, o resto é o dígito de posicional de 5316911993993899575960855904, e assim por diante. O resultado é dividido por 1063382397987789915192171182, o resto é o dígito de posicional de 1063382397987789915192171182, e assim por diante. O resultado é dividido por 2126764795975579830384342364, o resto é o dígito de posicional de 2126764795975579830384342364, e assim por diante. O resultado é dividido por 4253529591951159660768684728, o resto é o dígito de posicional de 4253529591951159660768684728, e assim por diante. O resultado é dividido por 8507059183902319321537369456, o resto é o dígito de posicional de 8507059183902319321537369456, e assim por diante. O resultado é dividido por 1701411836780463862307473892, o resto é o dígito de posicional de 1701411836780463862307473892, e assim por diante. O resultado é dividido por 3402823673560927724614947784, o resto é o dígito de posicional de 3402823673560927724614947784, e assim por diante. O resultado é dividido por 6805647347121855449229895568, o resto é o dígito de posicional de 6805647347121855449229895568, e assim por diante. O resultado é dividido por 13611294694423710884457791136, o resto é o dígito de posicional de 13611294694423710884457791136, e assim por diante. O resultado é dividido por 27222589388847421768915582272, o resto é o dígito de posicional de 27222589388847421768915582272, e assim por diante. O resultado é dividido por 54445178777694843537831164544, o resto é o dígito de posicional de 54445178777694843537831164544, e assim por diante. O resultado é dividido por 10889035755338968707566232908, o resto é o dígito de posicional de 10889035755338968707566232908, e assim por diante. O resultado é dividido por 21778071510677937415132465816, o resto é o dígito de posicional de 21778071510677937415132465816, e assim por diante. O resultado é dividido por 43556143021355874830264931632, o resto é o dígito de posicional de 43556143021355874830264931632, e assim por diante. O resultado é dividido por 87112286042711749660529863264, o resto é o dígito de posicional de 87112286042711749660529863264, e assim por diante. O resultado é dividido por 174224572085423493310587326328, o resto é o dígito de posicional de 174224572085423493310587326328, e assim por diante. O resultado é dividido por 348449144170846986621174652656, o resto é o dígito de posicional de 348449144170846986621174652656, e assim por diante. O resultado é dividido por 696898288341693973242349305312, o resto é o dígito de posicional de 696898288341693973242349305312, e assim por diante. O resultado é dividido por 139379657668338794648469661064, o resto é o dígito de posicional de 139379657668338794648469661064, e assim por diante. O resultado é dividido por 278759315336677589296939322128, o resto é o dígito de posicional de 278759315336677589296939322128, e assim por diante. O resultado é dividido por 557518630673355178593878644256, o resto é o dígito de posicional de 557518630673355178593878644256, e assim por diante. O resultado é dividido por 111503726134671035718757728852, o resto é o dígito de posicional de 111503726134671035718757728852, e assim por diante. O resultado é dividido por 223007452269342071437515457704, o resto é o dígito de posicional de 223007452269342071437515457704, e assim por diante. O resultado é dividido por 446014904538684142875030915408, o resto é o dígito de posicional de 446014904538684142875030915408, e assim por diante. O resultado é dividido por 892029809077368285750061830816, o resto é o dígito de posicional de 892029809077368285750061830816, e assim por diante. O resultado é dividido por 1784059618154736571500123661632, o resto é o dígito de posicional de 1784059618154736571500123661632, e assim por diante. O resultado é dividido por 3568119236309473143000247323264, o resto é o dígito de posicional de 3568119236309473143000247323264, e assim por diante. O resultado é dividido por 7136238472618946286000494646528, o resto é o dígito de posicional de 7136238472618946286000494646528, e assim por diante. O resultado é dividido por 1427247694523789257200098929056, o resto é o dígito de posicional de 1427247694523789257200098929056, e assim por diante. O resultado é dividido por 2854495389047578514400197858112, o resto é o dígito de posicional de 2854495389047578514400197858112, e assim por diante. O resultado é dividido por 5708985778095157028800395716224, o resto é o dígito de posicional de 5708985778095157028800395716224, e assim por diante. O resultado é dividido por 1141797155619035405600791543248, o resto é o dígito de posicional de 1141797155619035405600791543248, e assim por diante. O resultado é dividido por 2283594311238070811200158306496, o resto é o dígito de posicional de 2283594311238070811200158306496, e assim por diante. O resultado é dividido por 4567188622476141622400316612992, o resto é o dígito de posicional de 4567188622476141622400316612992, e assim por diante. O resultado é dividido por 9134377244952283244800633225984, o resto é o dígito de posicional de 9134377244952283244800633225984, e assim por diante. O resultado é dividido por 18268754489854566488001266451968, o resto é o dígito de posicional de 18268754489854566488001266451968, e assim por diante. O resultado é dividido por 36537508979709132976002532903936, o resto é o dígito de posicional de 36537508979709132976002532903936, e assim por diante. O resultado é dividido por 73075017959418265952005065807872, o resto é o dígito de posicional de 73075017959418265952005065807872, e assim por diante. O resultado é dividido por 14615003591883653190401013161576, o resto é o dígito de posicional de 14615003591883653190401013161576, e assim por diante. O resultado é dividido por 29230007183767306380802026323152, o resto é o dígito de pos~~

- a) Escreva um programa que lê um inteiro não negativo e calcula e imprime seu fatorial.
b) Escreva um programa que estima o valor da constante de matemática:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

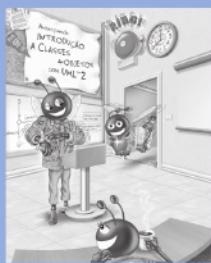
Solicite ao usuário a exatidão desejada (número de termos na adição).

- c) Escreva um programa que calcula utilizando a fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Solicite ao usuário a exatidão desejada (número de termos na adição).

- 4.36 [Nota: Este exercício corresponde à Seção 4.13, uma parte de nosso estudo de caso de engenharia de software.] Descreva em 200 palavras ou menos o que um automóvel é e o que faz. Liste os substantivos e verbos separadamente. No texto, afirmamos que cada substantivo corresponde a um objeto que precisará ser construído para implementar um sistema, nesse caso um carro. Selecione cinco dos objetos que você listou e, para cada um, liste vários atributos e comportamentos. Descreva brevemente como esses objetos interagem entre si com outros objetos na sua descrição. Você acabou de seguir vários dos passos-chave em um projeto orientado a objetos típico.



Nem tudo o que pode ser contado importa e nem tudo o que importa pode ser contado.
Albert Einstein

Quem pode controlar seu destino?
William Shakespeare

A chave utilizada é sempre brilhante.
Benjamin Franklin

Inteligência... é a faculdade de criar objetos artificiais, especialmente ferramentas para fazer ferramentas.
Henri Bergson

Toda vantagem no passado é julgada à luz do resultado final.
Demóstenes

Instruções de controle: parte 2

OBJETIVOS

Neste capítulo, você aprenderá:

Os princípios básicos da repetição controlada por contador.

Como utilizar as instruções de repetição `do...while` para executar instruções em um programa repetidamente.

A entender a seleção múltipla utilizando a instrução de seleção `switch`.

Como utilizar as instruções `break` e `continue` para alterar o fluxo de controle.

Como utilizar os operadores lógicos para formar expressões condicionais complexas em instruções de controle.

A evitar as consequências de confundir os operadores de igualdade com os operadores de atribuição.

5.1	Introdução
5.2	Princípios básicos de repetição controlada por contador
5.3	A instrução de repetição <code>for</code>
5.4	Exemplos com a estrutura <code>for</code>
5.5	Instrução de repetição <code>do...while</code>
5.6	A estrutura de seleção múltipla <code>switch</code>
5.7	Instruções <code>break</code> e <code>continue</code>
5.8	Operadores lógicos
5.9	Confundindo operadores de igualdade (<code>=</code>) com operadores de atribuição (<code>=</code>)
5.10	Resumo de programação estruturada
5.11	Estudo de caso de engenharia de software: identificando estados e atividades dos objetos no sistema ATM (opcional)
5.12	Síntese

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

5.1 Introdução

O Capítulo 4 iniciou nossa introdução aos tipos de bloco de construção que estão disponíveis para resolução de problemas. Utilizamos esses blocos de construção para empregar técnicas comprovadas de construção de programa. Neste capítulo, continuamos nossa apresentação da teoria e princípios da programação estruturada introduzindo as instruções de controle restantes do C++. As instruções de controle estudadas neste e no Capítulo 4 nos ajudarão a construir e manipular objetos. Continuamos nossa ênfase inicial em programação orientada a objetos que começou com uma discussão de conceitos básicos no Capítulo 1 e extensos exemplos e exercícios de código orientado a objetos nos capítulos 3–4.

Neste capítulo, demonstramos as [instruções de controle](#). Por uma série de breves exemplos que utilizam `while`, `for`, exploramos os princípios básicos da repetição controlada por contador. Dedicamos uma parte do capítulo à expansão da GradeBook.

~~Este capítulo apresenta mais detalhes sobre as instruções de controle, que foram introduzidas no Capítulo 4. Especificamente, examinaremos as instruções de controle de programação. Discutimos os operadores lógicos, que permitem aos programadores utilizar expressões condicionais mais poderosas em instruções de controle. Examinamos também o erro comum de confundir os operadores de igualdade com os operadores de atribuição (evitá-lo). Por fim, resumimos as instruções de controle e as comprovadas técnicas de resolução de problemas do C++ apresentadas neste capítulo e no Capítulo 4.~~

5.2 Princípios básicos de repetição controlada por contador

Esta seção utiliza a instrução de [repetição](#) introduzida no Capítulo 4 para formalizar os elementos necessários à realização da repetição controlada por contador. Repetição controlada por contador requer

1. [Nome](#) de uma variável de controle (ou contador de loop);
2. [Valor inicial](#) da variável de controle;
3. [Condição de continuação](#) que testa a [variável de controle](#) (isto é, se o loop deve continuar) e
4. [Incremento](#) (ou [decremento](#)) pelo qual a variável de controle é modificada a cada passagem pelo loop.

Considere o programa simples na Figura 5.1, que imprime os números de 1 a 10. A declaração na linha 9

~~declara a variável `counter` como um inteiro, reserva espaço para ela na memória e configura como um valor inicial de 1. As declarações que reservam memória e inicialização são feitas juntas em uma única linha. Em C, é mais preciso chamar uma declaração e uma inicialização separadamente. Por exemplo, para declarar e inicializar uma variável int, é preciso escrever:~~

~~A declaração e a inicialização (linha 9) também poderiam ter sido realizadas com as instruções~~

```
int counter; // declara a variável de controle
counter = 1; // inicializa a variável de controle como 1
```

Utilizamos ambos os métodos de inicialização de variáveis.

A linha 11 [incrementa](#) contador de loop por 1 toda vez que o corpo do loop é executado. A condição de continuação do loop (linha 11) na instrução [determina](#) se o valor da variável de controle é [menor ou igual](#) ao [valor final](#) para o qual a condição é

```

1 // Figura 5.1: fig05_01.cpp
2 // Repetição controlada por contador.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int counter = 1; // declara e inicializa a variável de controle
10
11    while ( counter <= 10 ) // condição de continuação do loop
12    {
13        cout << counter << " ";
14        counter++; // incrementa a variável de controle por 1
15    } // fim do while
16
17    cout << endl; // gera a saída de um caractere de nova linha
18    return 0; // terminação bem-sucedida
19 } // fim de main

```

1 2 3 4 5 6 7 8 9 10

Figura 5.1 Repetição controlada por contador.

true). Observe que o corpo do loop é executado mesmo quando a variável de controle é maior que o valor que é testado, quando se torna verdadeira.

A Figura 5.1 pode se tornar mais concisa iniciando o loop substituindo a instrução

```

while ( ++counter <= 10 ) // condição de continuação do loop
    cout << counter << " ";

```

Esse código salva uma instrução, porque a incrementação é feita diretamente na condição de teste. Além disso, o código elimina as chaves em torno do loop. Agora contém somente uma instrução. Codificar de modo tão condensado exige certa prática e pode resultar em programas mais difíceis de ler, depurar, modificar e manter.



Erro comum de programação 5.1

Os valores de ponto flutuante são aproximados, portanto controlar a contagem de loops com variáveis de ponto flutuante pode resultar em valores de contador imprecisos e testes imprecisos para terminação.



Dica de prevenção de erro 5.1

Controle a contagem do loop com valores de inteiro.



Boa prática de programação 5.1

Coloque uma linha em branco antes e depois de cada instrução de controle para destacá-la no programa.



Boa prática de programação 5.2

Muitos níveis de aninhamento podem tornar um programa difícil de entender. De modo geral, tente evitar o uso de mais de três níveis de recuo.



Boa prática de programação 5.3

O espaçamento vertical acima e abaixo de instruções de controle e o recuo do corpo das instruções de controle dentro dos cabeçalhos de instrução de controle fornecem aos programas uma aparência bidimensional que melhora significativamente a legibilidade.

5.3 A instrução de repetição `for`

A Seção 5.2 apresentou os princípios básicos da repetição controlada por contador. A instrução `for` complementa a instrução `while`, que especifica os detalhes da repetição controlada por contador em uma única linha de código. Para ilustrar a especificidade da instrução `for`, o resultado é mostrado na Figura 5.2.

Quando a instrução (linhas 11–12) começa a executar, a variável de controle é inicializada como 1. Então, a condição de continuação do loop é verificada. O valor inicial de 1, então a condição é satisfeita e a instrução do corpo (linha 12) imprime o valor de 1. Em seguida, a expressão incrementa a variável de controle counter e o loop inicia novamente com o teste de continuação do loop. A variável de controle é agora igual a 2; portanto, o valor não é excedido e o programa realiza a instrução de corpo novamente. Esse processo continua até o corpo do loop ter executado 10 vezes. Se a variável de controle ser incrementada para 11 — isso faz com que o teste de continuação do loop (linha 11 entre os ponto-e-vírgulas) falle e com que a repetição termine. O programa continua executando a primeira instrução depois da instrução caso, a instrução de saída na linha 14).

Componentes do cabeçalho da instrução

A Figura 5.3 oferece um exame mais minucioso do cabeçalho da instrução. Observe que o cabeçalho da instrução ‘faz tudo’ — ele especifica cada um dos itens necessários para repetição controlada por contador com uma variável de controle. Se houver mais de uma instrução no corpo, o programador esquecerá necessárias para incluir o corpo do loop.

Note que a Figura 5.2 utiliza a condição de continuação do loop corretamente. Se o programador esquecesse incorretamente, então o loop executaria apenas 9 vezes. Esse é um erro.

```

1 // Figura 5.2: fig05_02.cpp
2 // Repetição controlada por contador com a instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     // cabeçalho da instrução for inclui inicialização,
10    // condição de continuação do loop e incremento.
11    for ( int counter = 1; counter <= 10; counter++ )
12        cout << counter << " ";
13
14    cout << endl; // gera a saída de um caractere de nova linha
15    return 0; // indica terminação bem-sucedida
16 } // fim de main

```

1 2 3 4 5 6 7 8 9 10

Figura 5.2 Repetição controlada por contador com a instrução

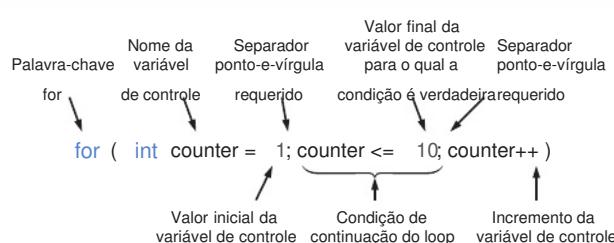


Figura 5.3 Componentes de cabeçalho de instrução



Erro comum de programação 5.2

Utilizar um operador relacional incorreto ou um valor final incorrecto de um contador de loop na condição de loop pode causar erro **off-by-one**.



Boa prática de programação 5.4

Utilizar o valor final na condição de uma instrução **for** e utilizar o operador relacional ajudará a evitar **off-by-one**. Para um loop utilizado para imprimir os valores de 1 a 10, por exemplo, a condição de continuação do loop deve ser `<= 10` em vez de `< 10` (que é um erro **off-by-one**). `counter < 11` (que, porém, é correto). Muitos programadores preferem a chamada **baseada em zero** que para contar 10 vezes pelo loop, seria inicializado como zero e o teste de continuação do loop seria `< 10`.

A forma geral da instrução

```
for ( inicialização ; condiçãoDeContinuaçãoDoLoop ; incremento )
    instrução
```

onde a expressão **inicialização** inicializa a variável de controle do loop. A expressão **condiçãoDeContinuaçãoDoLoop** determina se o loop deve continuar executando (essa condição, em geral, contém o valor final da variável de controle **para que a condição é verdadeira**) e a expressão **incremento** incrementa a variável de controle. Na maioria dos casos, a estrutura é apresentada por uma instrução, como segue:

```
inicialização ;
while ( condiçãoDeContinuaçãoDoLoop )
{
    instrução
    incremento;
}
```

Há uma exceção a essa regra, que discutiremos na Seção 5.7.

Se a expressão **inicialização** no cabeçalho de instrução **para a variável de controle** (isto é, o tipo da variável de controle é especificado antes do nome variável), a variável de controle só pode ser utilizada **variável de controle** não será conhecida fora da instrução. Essa utilização restrita do nome da variável de controle é da variável como

Escopo é a área de especificação onde ele pode ser utilizado em um programa. O escopo é discutido em detalhes no Capítulo 10.



Erro comum de programação 5.3

Quando a variável de controle de uma instrução declarada na seção de inicialização do cabeçalho de instrução a variável de controle depois do corpo da instrução é um erro de compilação.



Dica de portabilidade 5.1

No padrão C++, o escopo da variável de controle declarada na seção de inicialização **é limitado ao bloco** em compiladores C++ mais antigos. Em compiladores pré-padrão, o escopo da variável de controle não termina no fim do bloco que define o corpo da instrução em vez disso, o escopo termina no fim do bloco que inclui a instrução C++ criado com compiladores C++ pré-padrão pode quebrar quando compilado em compiladores compatíveis com o padrão. Se você estiver trabalhando com compiladores pré-padrão e quiser certificar-se de que seu código funcionará com compiladores compatíveis com o padrão, há duas estratégias de programação defensivas que você pode utilizar: declarar as variáveis de controle com nomes diferentes em cada instrução ou, se preferir utilizar o mesmo nome para a variável de controle em várias instruções, declarar a variável de controle antes da primeira instrução.

O operador vírgula tem a precedência mais baixa de todos os operadores C++. O valor e o tipo da expressão mais à direita na lista de expressões separadas por vírgulas é frequentemente utilizado em instruções de loop. A aplicação principal é permitir ao programador utilizar múltiplas expressões de inicialização e/ou múltiplas expressões de incremento. Por exemplo, pode haver diversas variáveis de controle em que devem ser inicializadas e incrementadas.



Boa prática de programação 5.5

Coloque apenas expressões que envolvem as variáveis de controle nas seções de inicialização e incremento de uma instrução. As manipulações de outras variáveis devem aparecer antes do loop (se tiverem de executar apenas uma vez, como as instruções de inicialização) ou no corpo do loop (se tiverem de executar uma vez por repetição, como as instruções de incremento ou decremento).

As três expressões no cabeçalho das instruções (mas os dois separadores ponto-e-vírgula são necessários). Se a condição de continuação do loop é omitida, o C++ pressupõe que a condição é verdadeira, criando assim um loop infinito. Pode-se omitir a expressão de inicialização se a variável de controle for inicializada anteriormente no programa. Pode-se omitir a expressão de incremento se o incremento for calculado por instruções usadas no loop. O incremento não é necessário. A expressão de incremento na instrução atua como uma instrução independente no final do loop.

```
counter = counter + 1
counter += 1
```

```
++counter
```

São todas equivalentes na parte de incremento (no final da instrução, outro código aparece aí). Muitos programadores preferem a forma `counter++`, porque os loops avaliam a expressão de incremento depois que o corpo do loop executa. A forma pós-incremento portanto parece mais natural. A variável sendo incrementada aqui não aparece em uma expressão maior, assim tanto a pré-incremendo como a pós-incremendo realmente têm o mesmo efeito.



Erro comum de programação 5.4

Utilizar vírgulas em vez dos dois ponto-e-vírgulas obrigatórios em um cabeçalho de sintaxe.



Erro comum de programação 5.5

Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito de um cabeçalho de loop dessa instrução é uma instrução vazia. Isso normalmente é um erro de lógica.



Observação de engenharia de software 5.1

Colocar ponto-e-vírgula logo depois de um loop é uma vez utilizada para criar um loop com retardos. Esse loop com um corpo vazio ainda realiza o loop pelo número indicado de vezes, não fazendo nada além de contar. Por exemplo, você poderia utilizar um loop de retardos para tornar lento um programa que está produzindo saídas na tela rápido demais para serem lidas. Mas seja cuidadoso, porque um retardos assim irá variar entre sistemas com diferentes velocidades de processador.

As expressões de inicialização, condição de continuação do loop e incremento de loop são aritméticas. Por exemplo, se `x = 10`, `x` e `y` não são modificados no corpo do loop, o cabeçalho

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

é equivalente a

```
for ( int j = 2; j <= 80; j += 5 )
```

O ‘incremento’ de uma instrução pode ser negativo, caso em que é realmente um decremento e o loop realmente conta para baixo (como mostrado na Seção 5.4).

Se a condição de continuação de loop é inicialmente falsa, o loop não é executado. No entanto, a execução prossegue com a instrução seguinte ao loop.

Frequentemente, a variável de controle é impressa ou utilizada em cálculos mas não é mencionada na instrução. É comum utilizar a variável de controle para controlar repetição sem nunca mencioná-la no corpo da instrução.



Dica de prevenção de erro 5.2

Embora o valor da variável de controle possa ser alterado no corpo de uma instrução, isso porque essa prática pode levar a erros de lógica.

Diagrama de atividades UML da instrução for

O diagrama de atividades UML da instrução for é semelhante ao da instrução (Figura 4.6). A Figura 5.4 mostra o diagrama de atividades da instrução (Figura 5.2). O diagrama torna claro que a inicialização ocorre uma vez antes de o teste de continuação do loop ser avaliado pela primeira vez, e que o incremento ocorre depois da avaliação da condição de continuação.

Observe que (além de um estado inicial, setas de transição, uma agregação, um estado final e várias notas) o diagrama contém

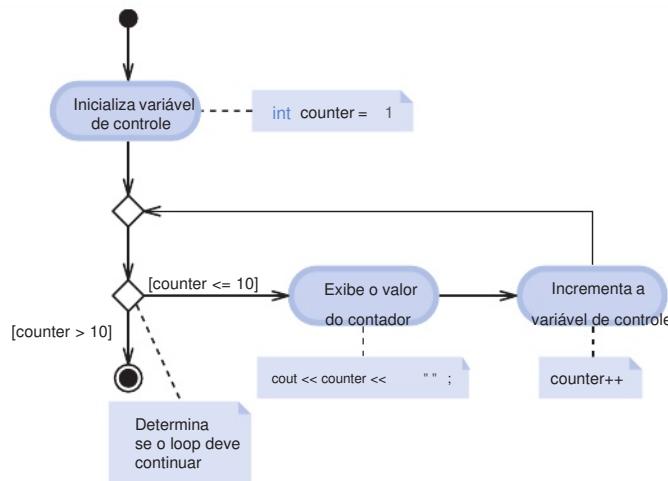


Figura 5.4 Diagrama de atividades UML para a instrução for.

estados de ação e uma decisão. Imagine, novamente, que o programador tem um contêiner de diagramas de atividades UML de cães vazias — quantas o programador possa precisar para empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos da decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.

5.4 Exemplos com a estrutura for

Os seguintes exemplos mostram métodos de variação de variável de controle para loops. Em cada caso, inserimos o cabeçalho da instrução for apropriado. Observe a alteração no operador relacional para loops que decrementam a variável de controle.

- a) Altere a variável de controle de incrementos de

`for (int i = 1; i <= 100; i++)`

- b) Altere a variável de controle para incrementos de

`for (int i = 100; i >= 1; i--)`

- c) Altere a variável de controle para passos de

`for (int i = 7; i <= 77; i += 7)`

- d) Altere a variável de controle para passos de

`for (int i = 20; i >= 2; i -= 2)`

- e) Altere a variável de controle sobre a seguinte sequência de valores:

`for (int i = 2; i <= 20; i += 3)`

- f) Altere a variável de controle sobre a seguinte sequência de valores: 11, 0.

`for (int i = 99; i >= 0; i -= 11)`



Erro comum de programação 5.6

Não utilizar o operador relacional adequado na condição de continuação do loop de um loop que conta para baixo (como utilizar incorretamente `= 1` em vez de `-= 1` em um loop que conta para baixo até 1) é normalmente um erro de lógica que leva a resultados incorretos quando o programa executa.

Aplicativo: somando os inteiros pares de 2 a 20

Os dois próximos exemplos fornecem aplicações simples de loops. Figura 5.5 utiliza uma estrutura for para somar os inteiros de 2 a 20. Cada iteração do loop (linhas 12–13) adiciona o valor atual da variável de controle

Observe que o corpo da instrução Figura 5.5 não poderia ser realmente fundido na parte de incremento do cabeçalho utilizando o operador vírgula como mostrado a seguir:

```
for ( int number = 2; // inicialização
      number <= 20; // condição de continuação do loop
      total += number, number += 2 ) // total e incremento
; // corpo vazio
```



Boa prática de programação 5.6

Embora as instruções que precedem as instruções no corpo `for` possam ser freqüentemente fundidas no cabeçalho `for`, fazer isso pode tornar o programa mais difícil de ler, manter, modificar e depurar.



Boa prática de programação 5.7

Limite o tamanho dos cabeçalhos da instrução de controle a uma única linha, se possível.

Aplicativo: cálculos de juros compostos

O próximo exemplo calcula os juros compostos utilizando a seguinte declaração do problema:

Uma pessoa investe \$ 1.000,00 em uma conta-poupança que rende 5% de juros. Supondo que todos os juros sejam deixados na conta, calcule e imprima o valor em dinheiro na conta ao fim de cada ano durante 10 anos. Utilize a seguinte fórmula para determinar essas quantidades:

$$a = p(1 + r)^n$$

onde

- p é a quantidade srccinal investida (isto é, o principal)
- r é a taxa de juros anual
- n é o número de anos e
- a é a quantidade em depósito

Esse problema envolve um loop que realiza o cálculo indicado para cada um dos 10 anos que o dinheiro permanece em depósito. A solução é mostrada na Figura 5.6.

A instrução (linhas 28–35) executa o seu corpo 10 vezes, alterando uma variável de controle de 1 a 10 em incrementos de 1.

A função inclui um operador de multiplicação ponto final (`pow`) que calcula a potência de um número. Nesse exemplo, a expressão `pow(1.0 + rate, year)`, onde a variável `rate` representa a taxa de juros e a variável `year` representa o ano, calcula a potência de 1.0 + `rate` para o número `year`. A função `pow` aceita dois argumentos do tipo `double` e retorna um `double`.

```
1 // Figura 5.5: fig05_05.cpp
2 // Somando inteiros com a instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int total = 0; // inicializa o total
10
11    // total de inteiros pares de 2 a 20
12    for ( int number = 2; number <= 20; number += 2 )
13        total += number;
14
15    cout << "Sum is " << total << endl; // exibe resultados
16    return 0; // terminação bem-sucedida
17 } // fim de main
```

Sum is 110

Figura 5.5 Somando inteiros com a instrução `for`.

```

1 // Figura 5.6: fig05_06.cpp
2 // Cálculos de juros compostos com for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setw; // permite que o programa configure a largura de um campo
10 using std::setprecision;
11
12 #include <cmath> // biblioteca de matemática C++ padrão
13 using std::pow; // permite ao programa utilizar a função pow
14
15 int main()
16 {
17     double amount; // quantia em depósito ao fim de cada ano
18     double principal = 1000.0; // quantia inicial antes dos juros
19     double rate = .05; // taxa de juros
20
21     // exibe cabeçalhos
22     cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
23
24     // configura o formato de número de ponto flutuante
25     cout << fixed << setprecision( 2 );
26
27     // calcula quantia de depósito para cada um dos dez anos
28     for ( int year = 1; year <= 10; year++ )
29     {
30         // calcula nova quantia durante ano especificado
31         amount = principal * pow( 1.0 + rate, year );
32
33         // exibe o ano e a quantia
34         cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
35     } // fim do for
36
37     return 0; // indica terminação bem-sucedida
38 } // fim de main

```

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

Figura 5.6 Cálculos de juros compostos código.

Esse programa não compilará sem incluir arquivos de cabecalho. A função `pow` quer dois argumentos. Observe que é um inteiro. O cabeçalho inclui informações que instruem o compilador a converter o valor de representação temporária antes de chamar a função. Essas informações estão copiadas. O Capítulo 6 fornece um resumo de outras funções de biblioteca de matemática.



Erro comum de programação 5.7

Em geral, esquecer de incluir o arquivo de cabeçalho apropriado ao utilizar funções de biblioteca-padrão (por exemplo, em um programa que utiliza funções de biblioteca de matemática) é um erro de compilação.

Uma nota de atenção sobre o uso do tipo `double` para valores monetários

Note que as linhas 17–19 declaram as variáveis `rate` como tipo `double`. Fizemos isso para simplificar porque estamos lidando com partes fracionárias de valores monetários e precisamos de um tipo que permita pontos decimais em seus dígitos. Infelizmente, isso pode causar problema. Eis uma explicação simples do que pode dar errado: se tentarmos representar quantias monetárias (^{supondo que} é utilizado para especificar dois dígitos de precisão na impressão): duas quantias monetárias armazenadas na máquina poderiam ser 14,234 (que é impressa como 14,23) e 18,673 (que é impressa com 18,67). Quando são adicionadas, essas quantidades produzem a soma interna 32,907, que é impressa como 32,91. Assim, sua impressão pode parecer com

$$\begin{array}{r} +14.23 \\ +18.67 \\ \hline 32.91 \end{array}$$

mas uma pessoa que adiciona os números individuais como impressos esperaria a soma 32,90! Você foi avisado!



Boa prática de programação 5.8

Não utilize variáveis definidas como `double` para realizar cálculos monetários. A imprecisão de números de ponto flutuante pode causar erros que resultam em valores monetários incorretos. Nos Exercícios, exploramos o uso de inteiros para realizar cálculos monetários. [Alguns fornecedores independentes vendem bibliotecas de classes C++ que realizam cálculos monetários precisos. Incluímos vários URLs no Apêndice I.]

Utilizando manipuladores de fluxo para formatar a saída numérica

A instrução de saída na linha 25 ~~antes desse loop~~ de saída na linha 34 combinam para imprimir os valores das variáveis `amount` com a formatação especificada pelos manipuladores ~~de fluxo parametrizada~~. O manipulador de fluxo não parametrizado `dec4` especifica que a próxima saída de valor deve aparecer em uma largura de campo 4 — isto é, imprime o valor com pelo menos 4 posições de caractere. Se o valor a ser enviado para a saída for menor do que a largura de 4 posições de campo, ele é preenchido com zeros à direita. Pode notar que a saída do loop é

~~simplesmente gerada a saída do manipulador de fluxo~~ — não é alinhada à direita. Isso ocorre porque o manipulador de fluxo não parametrizado `dec4` é aplicado à saída do loop, que é gerada por um manipulador de fluxo parametrizado `<iostream>`. O alinhamento à direita pode ser restaurado gerando a saída do manipulador de fluxo não parametrizado

A outra formatação nas instruções de saída indica que o valor deve ser impresso como um valor de ponto fixo com um ponto de fração decimal (especificado na linha 25 com o manipulador de fluxo `setprecision(2)`) e com 21 posições de caractere (especificadas na linha 34 com dois dígitos de precisão à direita do ponto de fração decimal (especificado na linha 25 com o manipulador `setprecision(2)`). Aplicamos os manipuladores `setprecision` ao fluxo de saída (isto é, `cout`) antes do loop, porque essas configurações de formato permanecem em vigor até serem alteradas — tais configurações são chamadas de ~~adaptações de fluxo~~. Portanto, elas não precisam ser aplicadas durante cada iteração do loop. Entretanto, a largura de campo especificada é simplificada somente à próxima saída de valor. Discutimos as poderosas capacidades de formatação de entrada/saída do C++ em detalhes no Capítulo 15.

Observe que o cálculo `rate * amount`, que aparece como um argumento ~~para esta iteração~~ no corpo da `for`-loop, esse cálculo produz o mesmo resultado durante cada iteração do loop, então repeti-lo é um desperdício — ele deve ser refeito uma vez antes do loop.



Dica de desempenho 5.1

Evite colocar expressões cujos valores não mudam dentro de loops — mas, mesmo se você colocá-las, muitos dos compiladores de otimização sofisticados de hoje irão, automaticamente, colocar essas expressões fora dos loops no código de linguagem de máquina gerado.



Dica de desempenho 5.2

Muitos compiladores contêm recursos de otimização que aprimoram o desempenho do código que você escreve, mas ainda é melhor escrever direito o código desde o início.

Por diversão, não deixe de experimentar o problema de Peter Minuit no Exercício 5.29. Esse problema demonstra as maravilhas dos juros compostos.

5.5 Instrução de repetição do..while

A instrução de repetição é semelhante à instrução de repetição, o teste de condição de continuação do loop ocorre no começo do loop antes de o corpo do loop executar. A instrução de continuação do loop depois de o corpo do loop executar; portanto, o corpo do loop executa sempre pelo menos uma vez. Execução continua com a instrução depois da clausula que não é necessário utilizar chaves na estrutura somente uma instrução no corpo; no entanto, a maioria dos programadores inclui as chaves para evitar confusão entre as instruções while e do..while. Por exemplo,

```
while ( condição )
```

normalmente é considerado o cabeçalho de uma instrução sem chaves em torno do único corpo de instrução aparece como:

```
do
  instrução
```

```
while ( condição );
```

que pode ser confuso. A última linha de condição — poderia ser interpretada erroneamente pelo leitor como uma instrução while contendo como o seu corpo uma instrução vazia. Contudo, única instrução costuma ser escrita como a seguir para evitar confusão:

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



Boa prática de programação 5.9

Incluir sempre as chaves em uma instrução ajuda a eliminar a ambigüidade entre a instrução a instrução... while contendo uma instrução.

A Figura 5.7 utiliza uma instrução para imprimir os números 1–10. No início da instrução 13 gera saída do valor counter, e a linha 14 incrementa. Então o programa avalia o teste de continuação do loop na parte inferior do loop (linha 15). Se a condição for verdadeira, o loop continua a partir da primeira linha de código. Se a condição for falsa, o loop termina e o programa continua com a próxima instrução depois do loop (linha 17).

```
1 // Figura 5.7: fig05_07.cpp
2 // instrução de repetição do...while.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int counter = 1; // inicializa o contador
10
11    do
12    {
13        cout << counter << " "; // exibe o contador
14        counter++; // incrementa o contador
15    } while ( counter <= 10); // fim da instrução do...while
16
17    cout << endl; // gera a saída de um caractere de nova linha
18    return 0; // indica terminação bem-sucedida
19 } // fim de main
```

```
1 2 3 4 5 6 7 8 9 10
```

Figura 5.7 Instrução de repetição do..while .

Diagrama de atividades UML da instrução while

A Figura 5.8 contém o diagrama de atividades UML para a instrução de repetição do loop while. Esse diagrama torna claro que a condição de continuação do loop não é avaliada enquanto o loop não executar os estados de ação do corpo do loop pelo menos uma vez. Compare esse diagrama de atividades com aquele da Figura 4.6. Novamente, observe que (além de um estado inicial, setas de transição, um agregado, um estado final e várias notas) o diagrama contém apenas estados de ação e uma decisão. Imagine, novamente, que o programador tem acesso a um contêiner de diagramas de atividades. Utilizadas instruções, o programador possa precisar para empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.

5.6 A estrutura de seleção múltipla switch

Discutimos a instrução de uma única seleção de switch no Capítulo 4. O C++ fornece a instrução de

switch com múltiplas ações (isto é, em que cada alternativa pode ter mais de uma ação a ser executada). Cada ação que são avaliadas como um valor constante inteiro (que a variável de ação que processa pode assumir).

ClasseGradeBook com a instrução switch para contar as notas A, B, C, D e F

No próximo exemplo apresentamos uma versão anterior da classe GradeBook definida no Capítulo 3 e desenvolvida no Capítulo 4.

A nova versão da classe solicita que o usuário insira um conjunto de notas baseadas em letras e, então, exibe um resumo do número de alunos que recebeu cada nota. A classe utilizada para terminar se cada nota inserida é um A, B, C, D ou F e para incrementar o contador de notas apropriado. A classe é definida na Figura 5.9 e suas definições de função-membro aparecem na Figura 5.10. A Figura 5.11 mostra entradas e saídas de exemplo para processar um conjunto de notas.

Como nas versões anteriores da definição de classe GradeBook (Figura 5.8), contém protótipos de função para as funções-membro CourseName (linha 13), CourseName (linha 14), displayMessage (linha 15), bem como para o construtor da classe (linha 12). A definição de classe também declara o membro de dados counter.

A classe GradeBook (Figura 5.9) agora contém cinco membros de dados (linhas 20–24) — variáveis de contador para cada categoria de nota (isto é, A, B, C, D e F). A classe também contém duas funções-membro displayGradeReport. A função-membro grades (declarada na linha 16) lê um número arbitrário de notas de letra fornecidas pelo usuário utilizando a repetição controlada por sentinela e atualiza o contador de notas apropriado para cada nota inserida. A função-membro displayGradeReport (declarada na linha 17) gera saída de um relatório contendo o número de alunos que recebeu cada nota de letra.

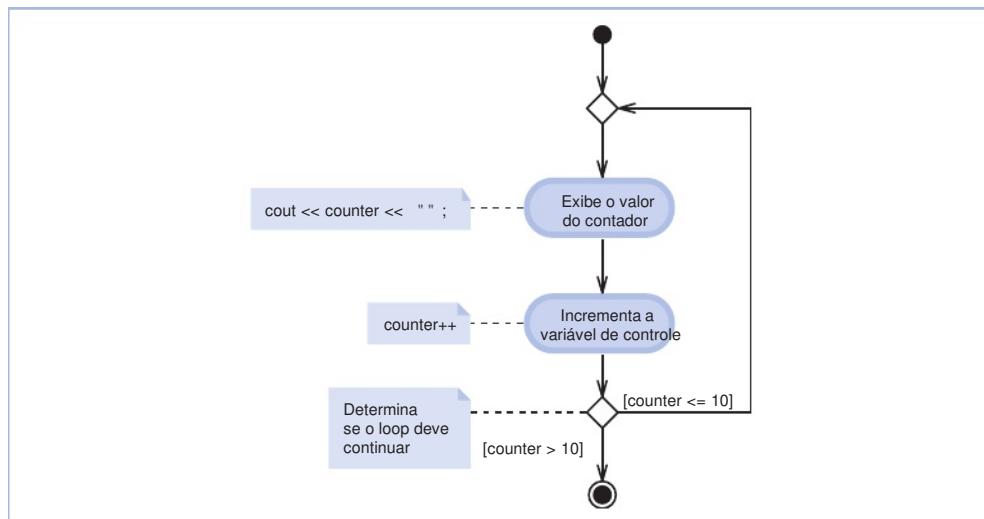


Figura 5.8 O diagrama de atividades UML para a instrução de repetição while da Figura 5.7.

```

1 // Figura 5.9: GradeBook.h
2 // Definição da classe GradeBook que conta as notas A, B, C, D e F.
3 // As funções-membro são definidas no GradeBook.cpp
4
5 #include <string> // o programa utiliza classe de string padrão C++
6 using std::string;
7
8 // definição da classe GradeBook
9 class GradeBook
10 {
11 public :
12     GradeBook( string ); // o construtor inicializa o nome do curso
13     void setCourseName( string ); // função para configurar o nome do curso
14     string getCourseName(); // função para recuperar o nome do curso
15     void displayMessage(); // exibe uma mensagem de boas-vindas
16     void inputGrades(); // insere número arbitrário de notas do usuário
17     void displayGradeReport(); // exibe um relatório baseado nas notas
18 private :
19     string courseName; // nome do curso para esse GradeBook
20     int aCount; // contagem de notas A
21     int bCount; // contagem de notas B
22     int cCount; // contagem de notas C
23     int dCount; // contagem de notas D
24     int fCount; // contagem de notas F
25 }; // fim da classe GradeBook

```

Figura 5.9 A definição da classe GradeBook

O arquivo de código fonte GradeBook.cpp (Figura 5.10) contém as definições de função-membro para a classe GradeBook. Note que quando é criado um novo objeto GradeBook, os valores iniciais para as contagens de notas são automaticamente definidos na função-membro constructor. As definições de função-membro setCourseName e displayMessage são idênticas às encontradas nas versões anteriores da classe. Vamos considerar as novas funções-membro em detalhes.

```

1 // Figura 5.10: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // utiliza uma instrução switch para contar as notas A, B, C, D e F.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // inclui a definição de classe GradeBook
10
11 // construtor inicializa courseName com string fornecido como argumento;
12 // e inicializa contadores de notas para zero
13 GradeBook::GradeBook( string name )
14 {
15     setCourseName( name ); // valida e armazena courseName
16     aCount = 0; // inicializa a contagem de notas A como 0
17     bCount = 0; // inicializa a contagem de notas B como 0
18     cCount = 0; // inicializa a contagem de notas C como 0
19     dCount = 0; // inicializa a contagem de notas D como 0

```

Figura 5.10 A classe GradeBook utiliza a instrução switch para contar as notas de letras A, B, C, D e F.

(continua)

```

20     fCount = 0; // inicializa a contagem de notas F como 0
21 } // fim do construtor GradeBook
22
23 // função para configurar o nome do curso; limita o nome a 25 ou menos caracteres
24 void GradeBook::setCourseName( string name )
25 {
26     if ( name.length() <= 25 ) // se o nome tiver 25 ou menos caracteres
27         courseName = name; // armazena o nome do curso no objeto
28     else // se o nome tiver mais que 25 caracteres
29     {
30         // configura courseName como os primeiros 25 caracteres do nome de parâmetro
31         courseName = name.substr( 0, 25 ); // seleciona os primeiros 25 caracteres
32         cout << "Name \"<< name << "\" exceeds maximum length (25).\n"
33         << "Limiting courseName to first 25 characters.\n" << endl;
34     } // fim do if...else
35 } // fim da função setCourseName
36
37 // função para recuperar o nome do curso
38 string GradeBook::getCourseName()
39 {
40     return courseName;
41 } // fim da função getCourseName
42 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
43 void GradeBook::displayMessage()
44 {
45     // essa instrução chama getCourseName para obter o
46     // nome do curso que esse GradeBook representa
47     cout << "Welcome to the grade book for\n" << getCourseName() << "\n"
48     << endl;
49 } // fim da função displayMessage
50
51 // insere número arbitrário de notas fornecidas pelo usuário; atualiza o contador de notas
52 void GradeBook::inputGrades()
53 {
54     int grade; // nota inserida pelo usuário
55
56     cout << "Enter the letter grades." << endl
57     << "Enter the EOF character to end input." << endl;
58
59     // faz loop até usuário digitar a seqüência de teclas de fim do arquivo
60     while ( ( grade = cin.get() ) != EOF )
61     {
62         // determina que nota foi inserida
63         switch ( grade ) // instrução switch aninhada em while
64         {
65             case 'A' : // a nota era letra A maiúscula
66             case 'a' : // ou a minúscula
67                 aCount++; // incrementa aCount
68                 break; // necessário para fechar switch
69
70             case 'B' : // a nota era B maiúscula
71             case 'b' : // ou b minúscula
72                 bCount++; // incrementa bCount
73                 break; // fecha o switch
74
75             case 'C' : // a nota era C maiúscula

```

Figura 5.10 A classe GradeBook utiliza a instrução `switch` para contar as notas de letras A, B, C, D e F.

(continua)

```

76     case 'c' : // ou c minúscula
77         cCount++; // incrementa cCount
78         break; // fecha o switch
79
80     case 'D' : // a nota era D maiúscula
81     case 'd' : // ou d minúscula
82         dCount++; // incrementa dCount
83         break; // fecha o switch
84
85     case 'F' : // a nota era F maiúscula
86     case 'f' : // ou f minúscula
87         fCount++; // incrementa fCount
88         break; // fecha o switch
89
90     case '\n' : // ignora nova linha,
91     case '\t' : // tabulações
92     case ' ' : // e espaços em entrada
93         break; // fecha o switch
94
95     default : // captura todos os outros caracteres
96         cout << "Incorrect letter grade entered."
97             << " Enter a new grade." << endl;
98         break; // opcional; sairá de switch de qualquer jeito
99     } // fim do switch
100 } // fim do while
101 } // fim da função inputGrades
102
103 // exibe um relatório baseado nas notas inseridas pelo usuário
104 void GradeBook::displayGradeReport()
105 {
106     // gera a saída de resumo de resultados
107     cout << "\n\nNumber of students who received each letter grade:"
108     << "\nA: " << aCount // exibe número de notas A
109     << "\nB: " << bCount // exibe número de notas B
110     << "\nC: " << cCount // exibe número de notas C
111     << "\nD: " << dCount // exibe número de notas D
112     << "\nF: " << fCount // exibe número de notas F
113     << endl;
114 } // fim da função displayGradeReport

```

Figura 5.10 A classe GradeBook utiliza a instrução `switch` para contar as notas de letras A, B, C, D e F.

(continuação)

Lendo a entrada de caracteres

O usuário insere notas baseadas em letras para um cursor na função `inputGrades()`. Dentro do cabeçalho na linha 60, a atribuição entre `parênteses` executa primeiro. A função lê um caractere do teclado e armazena esse caractere na variável `grade` declarada na linha 54). Os caracteres são normalmente armazenados em variáveis de tipo entretanto, os caracteres podem ser armazenados em qualquer tipo de dados inteiro, porque são representados:

sem a utilização de tipos no computador. Portanto, você pode tratar um caractere como um inteiro ou como um caractere, depend

```

cout << "The character (" << 'a' << ") has the value "
<< static_cast < int > ( 'a' ) << endl;

```

imprime o caractere seu valor inteiro como mostrado a seguir:

```
The character (a) has the value 97
```

O inteiro 97 é a representação numérica do caractere no computador. A maioria dos computadores hoje utiliza o **res ASCII (American Standard Code for Information),** em que 97 representa a letra 'm' (índice 128) da tabela dos caracteres ASCII e seus equivalentes decimais é apresentada no Apêndice B.

As instruções de atribuição têm como um todo, o valor que é atribuído à variável, no valor da expressão de atribuição. `le = cin.get()` tem o mesmo valor que o retornado por atribuído à variável.

O fato de que as instruções de atribuição têm valores pode ser útil para atribuir o mesmo valor a diversas variáveis. Por exem

primeiro avalia a atribuição (porque o operador `=` associa da direita para a esquerda) e levará o valor da atribuição = 0 (que é 0). Em seguida, a variável `valor` (que também é 0). No programa, o valor da atribuição `valor = cin.get()` é comparado com o símbolo cujo acrônimo `siginifica` fim do arquivo).

Utilizamos (que normalmente tem o valor -1) como o **valor de sentinela**, que digita o valor -1, nem as letras EOF como o valor de sentinela vez disso, você digita uma combinação de pressionamentos de teclas dependente do sistema que significa end-of-file para indicar que não tem mais dados armazenados constante inteira simbólica definida no arquivo de cabeçalho <iostream>. Se o valor atribuído for igual a EOF o loop (linhas 60–100) termina. Escolhemos representar os caracteres inseridos nesse programa porque ele tem um valor inteiro.

Em sistemas UNIX/Linux e muitos outros, o fim do arquivo é inserido digitando

<ctrl> d

em uma linha isolada. Essa notação significa pressionar a tecla `Enter`, pressionando a tecla `End` e depois a tecla `Shift`. Em sistemas, como o Microsoft Windows, o fim do arquivo pode ser inserido digitando

Ctrl+V

[Nota: Em alguns casos, você deve pressionar a tecla espaço para encerrar a sequência de teclas precedentes. Além disso, algumas vezes, os caracteres aparecem na tela para representar o fim do arquivo, como é mostrado na Figura 5.11.]



Dica de portabilidade 5.2

As combinações de teclas pressionadas para inserir o fim do arquivo são dependentes de sistema.



Dica de portabilidade 53

Testar a constante simbólica EOF em vez de -1 torna os programas mais portáveis. O padrão ANSI/ISO C, a partir do qual o C++ adota a definição de EOF, declara EOF com um valor inteiro negativo (mas não necessariamente EOF pode ter diferentes valores em sistemas diferentes).

Nesse programa, o usuário insere notas no teclado. Quando **`entre = input('Digite uma letra:')`** são lidos pela função `get()`, um caractere por vez. Se o caractere inserido não for fim do arquivo, o fluxo de controle entra na instrução (linhas 63-99), que incrementa o contador de notas de letras apropriado com base na letra inserida.

Detalhes da instrução switch

A instrução `for` consiste em uma série de `do` e um `as default` opcional. Essas sequências são utilizadas nesse exemplo para determinar qual contador incrementar, com base na nota. Quando o fluxo programático chega à expressão entre parênteses (neste caso se segue à palavra-chave `else`), essa expressão é `abandonada`.

entre parenteses) que se segue a palavra `while` (linha 63). Essa expressão é chamada de **controle**. A instrução `switch` compara o valor da expressão de controle com `Supadahártuke` ou o usuário `insiranadatra` nota. O programa `compara` ade no `switch`. Se ocorrer uma correspondência (linha 75), o programa executa as instruções para `se a letra` (linhas 76–77) incrementar tempor. A instrução `break` (linha 78) faz com que o controle de programa prossiga com a primeira instrução depois do loop. No final do programa, o controle é transferido para a linha 100. Essa linha marca o fim do corpo do loop. Depois de inserir notas (linhas 60–100), assim o controle `faz` (linea 100) para determinar se o loop deve continuar executando.

Os casos nas linhas 65–66 que testam os valores possíveis para `note` funcionariam corretamente dessa maneira, sem instruções entre elas, porque todos os comandos de controle — quando a expressão de controle é avaliada corretamente, as instruções nas linhas 67–68 serão executadas. Outra verificação: A instrução de seleção difere de outras instruções de controle porque não exige que as múltiplas instruções em cada

Sem instruções, toda vez que uma correspondência é encontrada, as executações para esse comando são encerradas e o processo continua com o próximo comando. Isso costuma ser referido como é o processo em que a instrução percorre sucessivas estruturas. (Esse recurso é perfeito para escrever um programa conciso que exibe a canção iterativa 'The Twelve Days of Christmas' no Exercício 5.28.)



• Erro comum de programação 5.8

Esquecer uma instrução quando esta for necessária em uma instituição é um erro de lógica.



Erro comum de programação 5.9

Omitir o espaço entre a palavra e o valor inteiro sendo testado em uma instrução pode causar um erro de lógica. Por exemplo, escrever `case 3:` em vez de `case 3:` simplesmente cria um rótulo não utilizado. Falaremos mais sobre isso no Apêndice E, "Tópicos sobre o código C legado". Nessa situação, a linguagem realizará as ações apropriadas quando a expressão de controle `switch` tiver um valor de 3.

Fornecendo um `default`

Se não ocorrer nenhuma correspondência entre o valor da expressão de controle e a constante (linhas 96–98) é executado. Utilizamos esse exemplo para processar todos os valores de expressão de controle que não forem notas válidas nem caracteres de nova linha, tabulações ou espaços (em breve discutiremos como o programa trata esses caracteres de em branco). Se não ocorrer correspondência, as linhas 96–97 imprimem uma mensagem de erro que indica que uma nota de letra incorreta foi inserida. Se não ocorrer nenhuma correspondência em uma instrução

`default`, o controle de programa simplesmente continua com a primeira instrução depois de



Boa prática de programação 5.10

Forneça um `default` em instruções `switch`. Os casos não explicitamente testados em uma instrução em um caso `default` são ignorados. Incluir um `default` concentra o programador na necessidade de processar condições excepcionais. Há situações em que nenhum processamento é necessário. Embora as cláusulas `default` em uma instrução `switch` possam ocorrer em qualquer ordem, é prática comum colocá-la após o último.



Boa prática de programação 5.11

Em uma instrução `switch` que lista a cláusula `default` por último, a cláusula `default` não requer uma instrução. Alguns programadores incluem `break` para clareza e simetria com outros casos.

Ignorando caracteres de nova linha, tabulações e em branco na entrada

Observe que as linhas 90–93 na Figura 5.10 fazem com que o programa pule os caracteres de nova linha, tabulação e espaço. Ler um caractere por vez pode causar alguns problemas. Para fazer o programa ler os caracteres, devemos enviá-los ao computador pressionando `Enter`. Isso coloca um caractere de nova linha na entrada depois do caractere que desejamos

processar. Esse é um exemplo de problema de saída de terminal que deve ser considerado para fazer o programa funcionar com uma nova linha, tabulação ou espaço encontrado na entrada.



Erro comum de programação 5.10

Não processar a nova linha e outros caracteres de espaço em branco na entrada ao ler caracteres individualmente (um caractere por vez) pode causar erros de lógica.

Testando a classe `GradeBook`

A Figura 5.11 cria um objeto (`GradeBook`) (linha 9). A linha 11 invoca a função `display` do objeto para realizar saída de uma mensagem de boas-vindas ao usuário. A linha 12 invoca a função `processReport` para gerar um conjunto de notas fornecidas pelo usuário e monitora o número de alunos que recebeu cada nota. Observe que a janela de entrada/saída na Figura 5.11 mostra uma mensagem de erro exibida em resposta à inserção de `Aluno 0`. O invocada a função-membro `GradeBook::displayGradeReport` (definida nas linhas 104–114 da Figura 5.10), que gera saída de um relatório baseado nas notas inseridas (como na saída da Figura 5.11).

Diagrama de atividades UML da instrução `switch`

A Figura 5.12 mostra o diagrama de atividades UML para a instrução `switch` com múltiplas instruções.

Utiliza uma instrução `case` para terminar a instrução depois de processar. A Figura 5.12 enfatiza isso, incluindo instruções no diagrama de atividades. Semelhantemente, não seria transferido para a próxima instrução após a instrução, depois que a mesma fosse processado. Em vez disso, o controle seria transferido para as ações do próximo `case`.

O diagrama torna claro que a ação de `break` faz com que o controle saia imediatamente da instrução. Novamente, observe que (além de um estado inicial, setas de transição, um estado final e várias notas) o diagrama contém estações e decisões. Além disso, observe que o diagrama utiliza símbolos de agregação para unir as transições das instruções ao estado final.

```

1 // Figura 5.11: fig05_11.cpp
2 // Cria o objeto GradeBook, insere notas e exibe relatório de notas.
3
4 #include "GradeBook.h" // inclui a definição de classe GradeBook
5
6 int main()
7 {
8     // cria objeto GradeBook
9     GradeBook myGradeBook( "CS101 C++ Programming");
10
11     myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
12     myGradeBook.inputGrades(); // lê as notas fornecidas pelo usuário
13     myGradeBook.displayGradeReport(); // exibe relatório baseado em notas
14
15 } // fim de main

```

Welcome to the grade book for
CS101 C++ Programming!

Enter the letter grades.

Enter the EOF character to end input.

a
B
c
C
A
d
f
C
E

Incorrect letter grade entered. Enter a new grade.

D
A
b
^Z

Number of students who received each letter grade:

A: 3
B: 2
C: 3
D: 2
F: 1

Figura 5.11 Criando um objeto GradeBook chamando suas funções-membro.

Imagine, novamente, que o programador tem um contêiner de diagramas de atividades UML para instruções

implementação de práticas de projeto para algoritmos e programação dos diagramas de atividades de fluxo de trabalho das decisões dentro de parafusos ação e condições de guarda apropriadas ao algoritmo. Observe que, embora as instruções de controle aninhadas sejam comum localizar instruções aninhadas em um programa.

Ao utilizar a instrução, lembre-se de que ela só pode ser utilizada para testar uma expressão inteira combinação de constantes de caractere e constantes inteiras que são avaliadas como um valor constante inteiro. Uma constante é representada como o caractere específico entre aspas simples. Um inteiro é simplesmente um valor de inteiro. Além disso, cada pode especificar apenas uma expressão inteira constante.

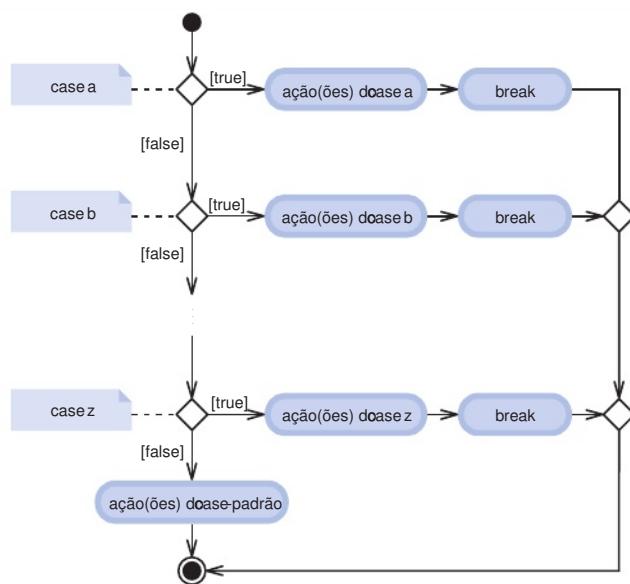


Figura 5.12 Diagrama de atividades UML de instrução de seleção múltipla com instruções break.

[Erro comum de programação 5.11](#)

Especificando uma expressão incluindo variáveis (por exemplo rótulo case de uma instrução switch) é um erro de sintaxe.

[Erro comum de programação 5.12](#)

Fornecer rótulos idênticos em uma instrução switch é um erro de compilação. Fornecer rótulos contendo expressões diferentes que são avaliadas como o mesmo valor também é um erro de compilação. Por exemplo, caso 2: na mesma instrução switch é um erro de compilação, porque ambos são equivalentes ao

No Capítulo 13 apresentamos uma maneira mais elegante de implementar a técnica chamada polimorfismo para criar programas que são freqüentemente mais claros, mais concisos, mais fáceis de manter e mais fáceis de entender que programas que utilizam lógica

Notas sobre tipos de dados

O C++ tem tamanhos flexíveis de tipo de dados (ver Apêndice C, “Tipos fundamentais”). Aplicativos diferentes, por exemplo, talvez precisem de inteiros de tamanhos diferentes. O C++ fornece vários tipos de dados para representar inteiros. O intervalo de valores para cada tipo depende do hardware do computador particular. Além dos tipos int e long, existem outras abreviações: short int (uma abreviação de short) e long (uma abreviação de long). O intervalo mínimo de valores para int é -32.768 a 32.767. Para a

variável dechar, o intervalo é de 0 a 255. O tipo char é equivalente ao tipo int.

Na maioria das arquiteturas de computador, o intervalo mínimo de valores para long é -2.147.483.648 o mesmo que o de int. O tipo de dado long pode ser utilizado para representar qualquer dos caracteres no conjunto de caracteres do computador. Também pode ser utilizado para representar inteiros pequenos.

[Dica de portabilidade 5.4](#)

Comint s podem variar de tamanho entre sistemas, utilizando long se você espera processar inteiros fora do intervalo de -32.768 a 32.767 e quiser executar o programa em diversos sistemas de computador diferentes.



Dica de desempenho 5.3

Se a memória for muito cara, talvez seja desejável utilizar tamanhos de inteiro menores.



Dica de desempenho 5.4

Utilizar tamanhos menores de inteiro pode resultar em um programa mais lento se as instruções de máquina para manipulá-las não forem tão eficientes quanto aquelas para os inteiros de tamanho natural, isto é, inteiros cujo tamanho é igual ao tamanho de palavra da máquina (por exemplo, de 32 bits em uma máquina de 32 bits, de 64 bits em uma máquina de 64 bits). Sempre teste ‘atualizações’ de eficiência propostas para certificar-se de que elas realmente melhoraram o desempenho.

5.7 Instruções break e continue

Além das instruções de seleção, o C++ também possui instruções de loop. Estas são divididas em loop while, loop for e loop pre-break em uma instrução de repetição.

Instrução break

A instrução `break`, quando executada em uma instrução `while` ou `switch`, ocasiona sua saída imediata dessa instrução. A execução de programa continua com a próxima instrução. Utilizaremos essa mesma instrução para sair de um loop ou pular o restante de uma instrução (veja a Figura 5.10). A Figura 5.13 demonstra a instrução `break` saindo de uma instrução de repetição.

Quando a instrução `break` é executada dentro de uma instrução de loop, o programa prossegue para a linha 19 (imediatamente depois da instrução) exibindo uma mensagem indicando o valor da variável de controle que terminou o loop. A instrução `break` executa completamente o seu corpo por apenas quatro vezes em vez de 10. Observe que a variável de controle `count` é definida fora do cabeçalho de `main()`. Assim, podemos utilizar a variável de controle no corpo do loop e depois que o loop completar sua execução.

```

1 // Figura 5.13: fig05_13.cpp
2 // a instrução break sai de uma instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int count; // variável de controle também utilizada depois que loop termina
10
11    for ( count = 1; count <= 10; count++ ) // itera 10 vezes
12    {
13        if ( count == 5 )
14            break; // quebra o loop somente se x for 5
15
16        cout << count << " ";
17    } // fim do for
18
19    cout << "\nBroke out of loop at count = " << count << endl;
20    return 0; // indica terminação bem-sucedida
21 } // fim de main

```

1 2 3 4
Broke out of loop at count = 5

Figura 5.13 Instrução `break` saindo de uma instrução.

Instrução continue

A instrução `continue`, quando executada em uma instrução `while`, ignora as instruções restantes no corpo dessa instrução e prossegue com a próxima iteração do loop. Em outras palavras, o teste de continuação é imediatamente avaliado depois de a instrução executar. Na instrução `for`, a expressão de incremento executa e, então, o teste de continuação do loop é avaliado.

A Figura 5.14 utiliza a instrução (linha 12) em uma instrução `for` para pular a instrução de saída (linha 14) quando o aninhado (linhas 11–12) determina que o loop deve ser encerrado. Quando a instrução `continue` é executada, o controle de programa continua com o incremento da variável de controle (`count++`) e faz o loop mais cinco vezes.

Na Seção 5.3, declaramos que a instrução `break` é preferível ser utilizada na maioria dos casos para representar a instrução `break`. No entanto, isso não ocorre quando a expressão de incremento `segunda instrução` é utilizada. Nesse caso, o incremento não executa antes de o programa testar a condição de continuação, e não é poupado da mesma maneira que o

**Boa prática de programação 5.12**

Alguns programadores consideram que a instrução `continue` viola a programação estruturada. Os efeitos dessas instruções podem ser alcançados por técnicas de programação estruturada que logo aprenderemos, portanto esses programadores não utilizam `continue`. A maioria dos programadores considera o uso da instrução `break` aceitável.

**Dica de desempenho 5.5**

As instruções `break` e `continue`, quando utilizadas adequadamente, desempenham mais rapidamente que as técnicas estruturadas correspondentes.

**Observação de engenharia de software 5.2**

Há uma tensão entre alcançar engenharia de software de qualidade e alcançar o software de melhor desempenho. Freqüentemente, um desses objetivos é alcançado à custa do outro. Para todas as situações, exceto as de desempenho muito alto, aplique a seguinte regra geral: primeiro, faça seu código simples e correto; então, torne-o rápido e pequeno, mas apenas se necessário.

```

2 // Figura 5.14: fig05_14.cpp
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     for ( int count = 1; count <= 10; count++ ) // itera 10 vezes
10    {
11        if ( count == 5 ) // se count for 5,
12            continue; // pula o código restante no loop
13
14        cout << count << " ";
15    } // fim do for
16
17    cout << "\nUsed continue to skip printing 5" << endl;
18    return 0; // indica terminação bem-sucedida
19 } // fim de main

```

1 2 3 4 6 7 8 9 10
Used continue to skip printing 5

Figura 5.14 A instrução `continue` que termina uma única iteração de uma instrução

5.8 Operadores lógicos

Até agora estudamos **condições simples** como `counter <= 10 , total > 1000`. Expressamos essas condições em termos dos operadores **relacionais** e operadores de **igualdade**. Cada decisão testou precisamente uma condição. Para testar múltiplas condições ao tomar uma decisão, realizamos esses testes em instruções separadas ou `else` aninhadas.

O C++ fornece **operadores lógicos** que são utilizados para formar condições mais complexas combinando condições simples. Os operadores lógicos (**E lógico**) (**OU lógico**) (**NÃO lógico**, também chamado de negação lógica).

Operador E lógico `&&`

Suponha que quiséssemos assegurar que duas condições sejam satisfeitas para escolhermos certo caminho de execução. Nesse caso, podemos utilizar o **operador**, como segue:

```
if ( gender == 1 && age >= 65 )
    seniorFemales++;
```

Esta instrução contém duas condições simples. A condição utilizada aqui para determinar se uma pessoa é do sexo feminino. A condição `65` determina se uma pessoa é um(a) cidadão(à) idoso(a). A condição simples à esquerda do operador é avaliada primeiro, porque a precedência é menor. Se for necessário, a condição simples à direita do operador `&&` é avaliada em seguida, porque a precedência é maior. A precedência é discutida em breve, o lado direito de uma expressão E lógica é avaliado somente se o lado esquerdo for considerado a condição combinada

```
gender == 1 && age >= 65
```

Essa condição é só e unicamente ambas as condições simples. Portanto, essa condição combinada, for de fato a instrução no corpo da `if` é executada a contagem de `seniorFemales`. Se qualquer uma das condições simples for ambas), então o programa ignora a incrementação e prossegue com a `if` seguinte. A condição precedente pode tornar-se mais legível adicionando-se parênteses redundantes:

```
( gender == 1 ) && ( age >= 65 )
```



Erro comum de programação 5.13

Embora `x < 7` seja uma condição matematicamente correta, ela não é avaliada como você talvez imagine em C++. Utilize `(3 < x && x < 7)` para obter a avaliação adequada em C++.

A Figura 5.15 resume o **operador**. A tabela mostra todas as quatro possíveis combinações entre expressões de operadores relacionais. Essas tabelas costumam ser **tabeladas de C++**, avaliando todas as expressões que incluem operadores relacionais, operadores de igualdade e/ou operadores lógicos.

Operador OU lógico `||`

Agora vamos considerar o **operador**. Suponha que quiséssemos assegurar em algum ponto em um programa que qualquer uma das condições fosse de escolhermos certo caminho de execução. Nesse caso, utilizamos o operador no segmento de programa seguinte:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
    cout << "Student grade is A" << endl;
```

Essa condição precedente também contém duas condições simples. A condição simples é avaliada para determinar se o aluno merece um 'A' no curso devido a um sólido desempenho ao longo de todo o semestre. A condição simples `>= 90` é avaliada para determinar se o aluno merece um 'A' no curso devido a um desempenho destacado no exame final. A instrução `if` então considera a condição combinada

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

e premia o aluno com um 'A' se qualquer uma ou ambas as condições simples forem verdadeiras. O resultado é que a condição simples é impressa a menos que ambas as condições simples falso. A Figura 5.16 é uma tabela-verdade para o operador lógico (

O operador tem uma precedência mais alta que os operadores associados da esquerda para a direita. Uma expressão contendo os operadores é avaliada somente até que a condição de verdade ou falsidade da expressão seja conhecida. Portanto, a avaliação da expressão

```
( gender == 1 ) && ( age >= 65 )
```

pára imediatamente se a condição for igual a falso. Isso é, a expressão interrompe a avaliação se for igual (isto é, a expressão inteira poderia ainda ser a condição `= 65` fosse verdade). Esse recurso de desempenho para a avaliação de expressões com E lógico e OU lógico é chamado de **curto-circuito**.

expressão1	expressão2	expressão1 && expressão2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 5.15 Tabela-verdade do operador &&(E lógico).

expressão1	expressão2	expressão1 expressão2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 5.16 Tabela-verdade do operador || (OU lógico).

**Dica de desempenho 5.6**

Em expressões que utilizam o operador `&&`, se as condições separadas forem independentes uma da outra, torne a condição que é mais provavelmente falsa a condição mais à esquerda. Em expressões utilizando o operador `||`, coloque a condição que é mais provavelmente verdadeira a condição mais à esquerda. Essa utilização de avaliação em curto-circuito pode reduzir o tempo de execução de um programa.

Operador de negação lógica `!`

O C++ fornece o operador `!<operador lógico>` também chamado de `!operador lógico` para permitir ao programador ‘inverter’ o significado de uma condição. Diferentemente dos operadores `&&` e `||` que avaliam duas condições, o operador unário de negação lógica tem apenas uma única condição como um operando. O operador unário de negação lógica é colocado antes de uma condição que estamos interessados em escolher um caminho de execução se a condição for falsa (sentido oposto ao operador de negação lógica) no seguinte segmento de programa:

```
if ( !( grade == sentinelValue ) )
    cout << "The next grade is " << grade << endl;
```

Os parênteses em torno da condição são necessários uma vez que o operador de negação lógica tem uma precedência mais alta que o operador de igualdade.

Na maioria dos casos, o programador pode evitar a utilização da negação lógica expressando a condição com um operador relacional ou de igualdade apropriado. Por exemplo, a condição anterior também pode ser escrita da seguinte maneira:

```
if ( grade != sentinelValue )
    cout << "The next grade is " << grade << endl;
```

Essa flexibilidade pode freqüentemente ajudar um programador a expressar uma condição de maneira mais ‘natural’ ou conveniente. Figura 5.17 é uma tabela-verdade para o operador `!<operador lógico>`.

expressão	! expressão
false	true
true	false

Figura 5.17 Tabela-verdade do operador `!<operador lógico>`.

Exemplo de operadores lógicos

A Figura 5.18 demonstra os operadores lógicos produzindo suas tabelas-verdade. A saída mostra cada expressão que é avaliada resultando em `true` ou `false`. Por padrão, valores `true` e `false` são exibidos juntos pelo operador de inserção de fluxo `<<`, entretanto, utilizando o operador `boolalpha` na linha 11 para especificar que o valor de cada expressão deve ser exibido como a palavra 'true' ou 'false'. Por exemplo, resultado da expressão `!true`, então a segunda linha de saída inclui a palavra 'false'. As linhas 11–15 produzem a tabela-verdade para `Logical AND (&&)`. As linhas 18–22 produzem a tabela-verdade para `Logical OR (||)`. As linhas 25–27 produzem a tabela-verdade para `Logical NOT (!)`.

```

1 // Figura 5.18: fig05_18.cpp
2 // Operadores lógicos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha; // faz com que valores bool sejam impressos como "true" ou "false"
7
8 int main()
9 {
10     // cria a tabela-verdade para o operador && (E lógico)
11     cout << boolalpha << "Logical AND (&&)"
12     << "\nfalse && false: " << ( false && false )
13     << "\nfalse && true: " << ( false && true )
14     << "\ntrue && false: " << ( true && false )
15     << "\ntrue && true: " << ( true && true ) << "\n\n" ;
16
17     // cria a tabela-verdade para o operador || (OU lógico)
18     cout << "Logical OR (||)"
19     << "\nfalse || false: " << ( false || false )
20     << "\nfalse || true: " << ( false || true )
21     << "\ntrue || false: " << ( true || false )
22     << "\ntrue || true: " << ( true || true ) << "\n\n" ;
23
24     // cria a tabela-verdade para o operador ! (negação lógica)
25     cout << "Logical NOT (!)"
26     << "\nfalse: " << ( ! false )
27     << "\ntrue: " << ( ! true ) << endl;
28
29 } // fim de main

```

Logical AND (&&
`false && false: false`
`false && true: false`
`true && false: false`
`true && true: true`

Logical OR (||)
`false || false: false`
`false || true: true`
`true || false: true`
`true || true: true`

Logical NOT (!)
`!false: true`
`!true: false`

Figura 5.18 Operadores lógicos.

Resumo de precedência e associatividade de operadores

A Figura 5.19 adiciona os operadores lógicos ao gráfico de precedência e associatividade de operadores. Os operadores são mordenha para baixo em ordem decrescente de precedência.

5.9 Confundindo operadores de igualdade (`==`) com operadores de atribuição (`=`)

Há um tipo de erro que programadores em C++, independentemente de sua experiência, tendem a cometer com tanta freqüênciata que sentimos que ele requer uma seção separada. Esse erro faz acidentalmente confundir operadores de igualdade com operadores de atribuição. O que faz com que essas trocas sejam tão prejudiciais é o fato de que elas normalmente não causam erro de sintaxe. Em vez de instruções com esses erros tendem a compilar corretamente e os programas executam até a conclusão, gerando muitas vezes resultados incorretos por meio de erros de lógica em tempo de execução. [Alguns compiladores publicam um aviso quando um contexto em que normalmente esperado.]

Há dois aspectos de C++ que contribuem para esses problemas. Um é que qualquer expressão que produz um valor pode ser utilizada na parte de decisão de qualquer instrução de controle. Se o valor da expressão, sendo zero, é interpretado como tratado como zero. O segundo aspecto é que as atribuições produzem um valor — a saber, o valor atribuído à variável no lado esquerdo do operador de atribuição. Por exemplo, suponha que pretendêssemos escrever

```
if ( payCode == 4 )
    cout << "You get a bonus!" << endl;
mas acidentalmente escrevemos
if ( payCode = 4 )
    cout << "You get a bonus!" << endl;
```

A primeira instrução adequadamente um bônus à pessoa que tem código 4. A segunda instrução com o erro — avalia a expressão de atribuição como constante 4. Qualquer valor diferente de zero é interpretado como a condição nessa instrução, e a pessoa sempre recebe um bônus independentemente de qual seja o código de pagamento real! Pior ainda, o código de pagamento foi modificado quando se esperava que ele fosse apenas examinado!



Erro comum de programação 5.14

Usar o operador `=` para atribuição e usar o operador `==` para igualdade são erros de lógica.

Operadores	Associatividade	Tipo
<code>()</code>	da esquerda para a direita	parênteses
<code>++ -- static_cast < tipo >()</code>	da esquerda para a direita	unário (pós-fixo)
<code>++ -- + - !</code>	da direita para a esquerda	unário (prefixo)
<code>*</code> / %	da esquerda para a direita	multiplicativo
<code>+</code> -	da esquerda para a direita	aditivo
<code><< >></code>	da esquerda para a direita	inserção/extracção
<code>< <= > >=</code>	da esquerda para a direita	relacional
<code>== !=</code>	da esquerda para a direita	igualdade
<code>&&</code>	da esquerda para a direita	E lógico
<code> </code>	da esquerda para a direita	Ou lógico
<code>?:</code>	da direita para a esquerda	ternário condicional
<code>= += -= *= /= %=</code>	da direita para a esquerda	atribuição
<code>,</code>	da esquerda para a direita	vírgula

Figura 5.19 Precedência e associatividade de operadores.



Dica de prevenção de erro 5.3

Normalmente, programadores escrevem condições como o nome da variável à esquerda e a constante à direita. Invertendo para que a constante esteja à esquerda e o nome da variável à direita, o programador que accidentalmente substituir o operador por = será protegido pelo compilador. O compilador trata isso como um erro de compilação, porque você não pode alterar o valor de uma constante. Isso evitará a potencial devastação de um erro de lógica em tempo de execução.

Diz-se que os nomes de variáveis são valores, porque podem ser utilizados à esquerda do operador de atribuição. Diz-se que as constantes são valores, porque podem ser utilizadas somente à direita do operador de atribuição. Observe que valores também podem ser utilizados somente à direita.

Há outra situação igualmente desagradável. Suponha que o programador queira atribuir um valor a uma variável com uma instância simples como

```
x = 1;
```

mas, em vez disso, escreve:

```
x == 1;
```

Aqui também esse não é um erro de sintaxe. Em vez disso, o compilador simplesmente ignora a expressão condicional. Se a condição é verdadeira, a expressão é avaliada como se a condição fosse falsa e a expressão é avaliada como o valor falso. Independentemente do valor da expressão, não há nenhum operador de atribuição, portanto o valor permanece permanecendo inalterado, causando provavelmente um erro de lógica em tempo de execução. Infelizmente, não temos nenhuma truque útil disponível para ajudá-lo com esse problema!



Dica de prevenção de erro 5.4

Utilize seu editor de textos para procurar todas as ocorrências de == em seu programa e verifique se você tem o operador de atribuição correto ou operador lógico em cada lugar.

5.10 Resumo de programação estruturada

Da mesma forma que os arquitetos projetam edifícios empregando o conhecimento coletivo de sua profissão, assim também os programadores devem projetar programas. Nossa campo é mais jovem que a arquitetura e nossa sabedoria coletiva é consideravelmente mais esparsa. Aprendemos que a programação estruturada produz programas mais fáceis de entender, testar, depurar, modificar e demonstrar como corretos em um sentido matemático do que os programas não estruturados.

A Figura 5.20 usa diagramas de atividades para resumir as instruções de controle do C++. Os estados inicial e final indicam o ponto de entrada e o único ponto de saída de cada instrução de controle. Conectar símbolos individuais arbitrariamente em um diagrama de atividades pode levar a programas não estruturados. Portanto, a profissão do programador utiliza somente um conjunto limitado de instruções de controle que podem ser combinadas apenas de duas maneiras simples para construir programas estruturados.

Por simplicidade, são utilizadas apenas instruções de controle de entrada única/saída única — há somente uma maneira de entrar e somente uma maneira de sair de cada instrução de controle. É simples conectar instruções de controle sequencialmente para programas estruturados — o estado final de uma instrução de controle é conectado ao estado inicial da próxima instrução de controle — isto é, as instruções de controle são colocadas uma depois da outra em um programa. Chamamos isso de ‘empilhamento de instruções de controle’. As regras para formar programas estruturados também permitem que instruções de controle sejam aninhadas.

A Figura 5.21 mostra as regras para formar programas estruturados. As regras assumem que os estados de ação podem ser usados para indicar qualquer ação. As regras também assumem que iniciamos com o chamado diagrama de atividades mais simples (Figura 5.22) consistindo em somente um estado inicial, um estado de ação, um estado final e setas de transição.

Aplicar as regras da Figura 5.21 resulta sempre em um diagrama de atividades com uma aparência organizada dos blocos de construção. Por exemplo, aplicar a Regra 2 repetidamente ao diagrama de atividades mais simples resulta em um diagrama de atividades que contém muitos estados de ação em seqüência (Figura 5.23). A Regra 2 gera uma pilha de instruções de controle, então vamos aplicar a Regra 3 [Nota: As linhas tracejadas verticais na Figura 5.23 não fazem parte da UML. Utilizamos essas linhas para separar os quatro diagramas de atividades que demonstram a Regra 2 da Figura 5.21 sendo aplicada.]

A Regra 3 é chamada de aninhamento. Aplicar a Regra 3 repetidamente ao diagrama de atividades mais simples resulta

em um diagrama de atividades com instruções de controle organizadamente aninhadas. Por exemplo, na Figura 5.24, o estado de ação é substituído por uma instrução de seleção dupla. A aplicação da Regra 4 substitui os estados de ação na instrução de seleção dupla, substituindo cada um desses estados de ação por uma instrução de seleção dupla. Os símbolos de estado de ação tracejados em torno de cada uma das instruções de seleção dupla representam um estado de ação substituído no diagrama de atividades. As linhas tracejadas e os símbolos de estado de ação tracejados mostrados na Figura 5.24 não fazem parte da UML. São utilizados aqui como recursos didáticos para ilustrar que qualquer estado de ação pode ser substituído por uma instrução de controle.]

A Regra 4 gera estruturas maiores, mais complexas e mais profundamente aninhadas. Os diagramas que emergem da aplicação das regras na Figura 5.21 constituem o conjunto de todos os possíveis diagramas de atividades e, portanto, o conjunto de todos os pro-

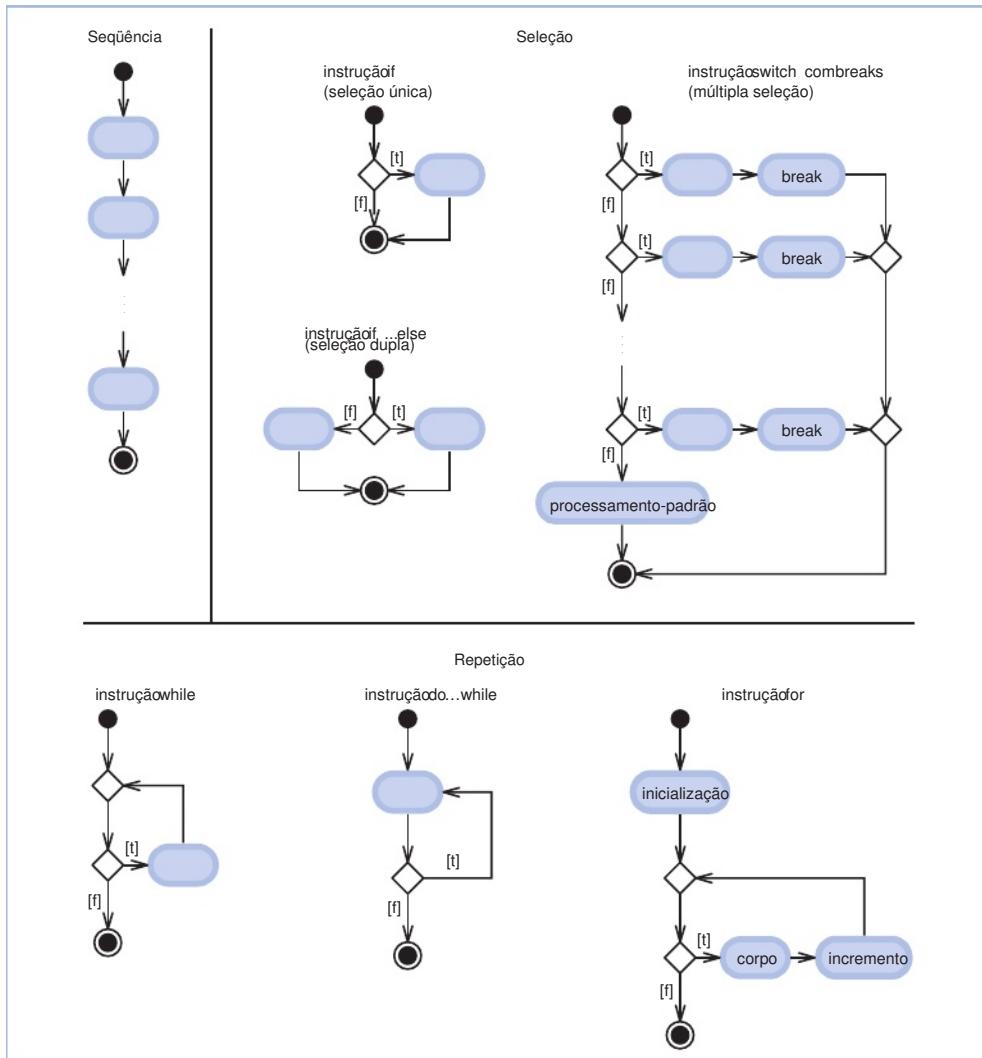


Figura 5.20 As instruções de seqüência de entrada única/saída única, seleção e repetição do C++.

Regras para formar programas estruturados

- 1) Comece com o “diagrama de atividades mais simples” (Figura 5.22).
- 2) Qualquer estado de ação pode ser substituído por dois estados de ação em seqüência.
- 3) Qualquer ação pode ser substituída por qualquer instrução de controle (seqüência, if...while ou for).
- 4) As Regras 2 e 3 podem ser aplicadas com a freqüência que você quiser em qualquer ordem.

Figura 5.21 As regras para formar programas estruturados.

estruturados possíveis. A beleza da abordagem estruturada é que utilizamos apenas sete instruções simples de entrada/saída e os montamos de apenas duas maneiras simples.

Se as regras da Figura 5.21 forem seguidas, um diagrama de atividades com sintaxe ilegal (como aquele da Figura 5.25) não será criado. Se você não tiver certeza se um diagrama particular é válido, aplique as regras da Figura 5.21 na ordem inversa para o diagrama ao diagrama de atividades mais simples. Se for reduzível ao mais simples diagrama de atividades, o diagrama será estruturado; caso contrário, não é.

A programação estruturada promove a simplicidade. Böhm e Jacopini nos deram o resultado de que apenas três formas de construção são necessárias:

- Seqüência
- Seleção
- Repetição

A estrutura de seqüência é trivial. Liste simplesmente as instruções para executar na ordem em que elas devem executar.

A seleção é implementada de uma destas três maneiras:

- instrução `seleção única`
- instrução `if...else` (seleção dupla)
- instrução `switch` (seleção múltipla)

É simples e direto provar que a instrução `seleção única` é suficiente para fornecer qualquer forma de seleção — tudo o que pode ser feito com a instrução `if...else` e a instrução `switch` pode ser implementado (embora talvez não de modo tão claro e eficiente) combinando-se instruções.



Figura 5.22 Diagrama de atividades mais simples.

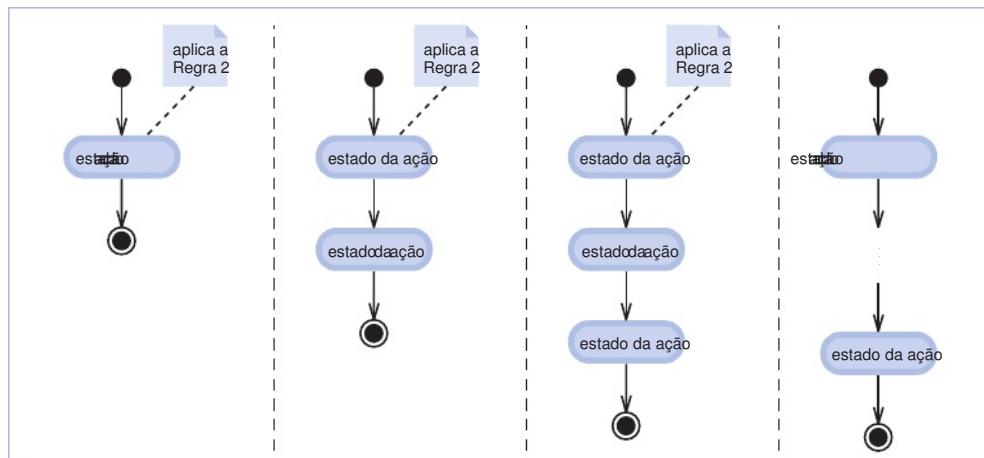


Figura 5.23 Aplicando repetidamente a Regra 2 da Figura 5.21 ao mais simples diagrama de atividades.

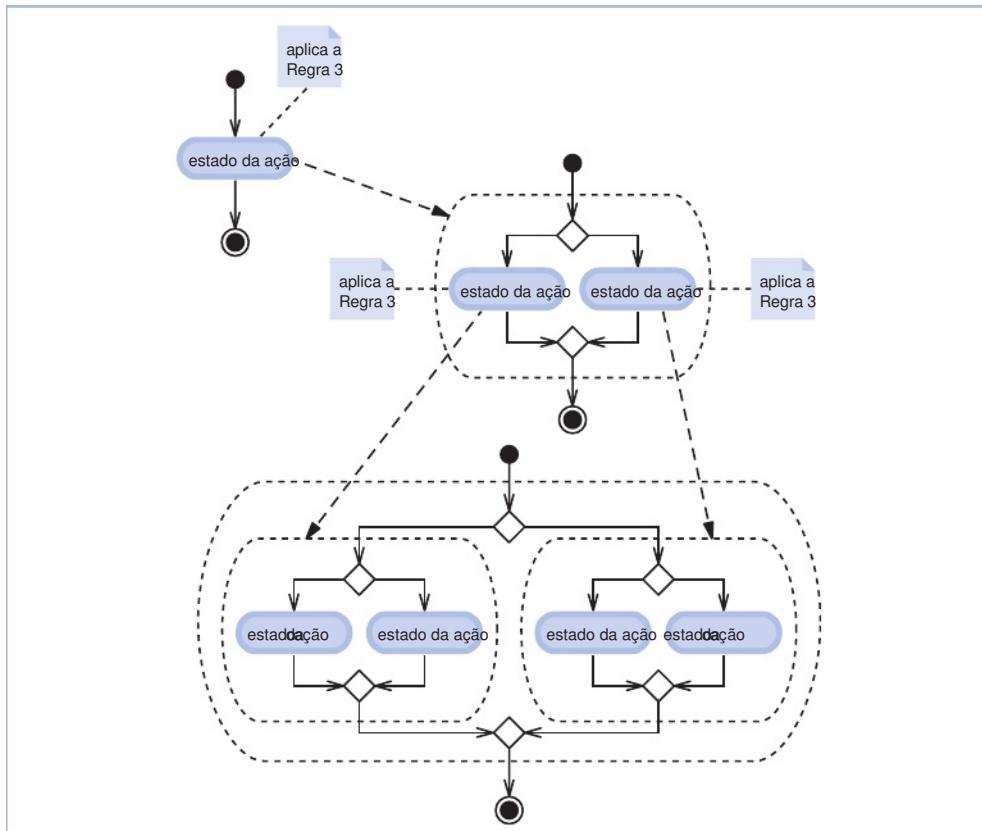


Figura 5.24 Aplicando várias vezes a Regra 3 da Figura 5.21 ao mais simples diagrama de atividades.

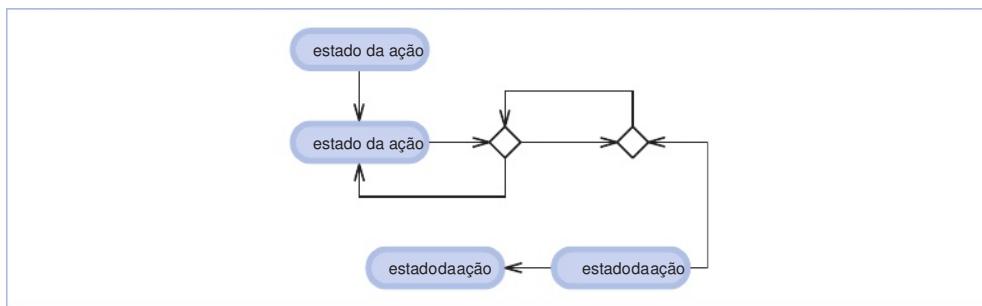


Figura 5.25 O diagrama de atividades com sintaxe inválida.

A repetição é implementada de uma destas três maneiras:

- Instrução `do`
- Instrução `do..while`
- Instrução `for`

É simples provar que a instrução `do` é suficiente para fornecer qualquer forma de repetição. Tudo o que pode ser feito com a instrução `do..while` e a instrução `for` pode ser feito (embora talvez não tão suavemente) com a instrução `do`.

Combinar esses resultados ilustra que qualquer forma de controle alguma vez necessária em um programa C++ pode ser expressa nos termos a seguir:

- seqüência
- instrução (seleção)
 - while

e quando essas instruções (seleção) de controle podem ser combinadas somente de duas maneiras — empilhamento e aninhamento. De fato, a programação estruturada promove simplicidade.

5.11 Estudo de caso de engenharia de software: identificando estados e atividades dos objetos no sistema ATM (opcional)

Na Seção 4.13, identificamos muitos dos atributos de classe necessários para implementar o sistema ATM e os adicionamos ao diagrama de classes na Figura 4.24. Nesta seção, mostramos como esses atributos representam o estado de um objeto. Identificamos algumas das chaves que nossos objetos podem ocupar e discutimos como os objetos mudam de estado em resposta a vários eventos que ocorrem no sistema. Também discutimos o fluxo de trabalho, ou seja, como os objetos realizam suas atividades-chave no sistema ATM.

Diagramas de máquina de estado

Todo objeto em um sistema passa por uma série de estados discretos. O estado atual de um objeto é indicado pelos valores dos atributos de classe que modelam os estados de (comummente chamados de estados-chave) de um objeto e mostram sob que circunstâncias o objeto muda de estado. Ao contrário dos diagramas de classes

que focalizaram principalmente a estrutura do sistema, os diagramas de estados abordam aspectos particulares de cada caso, que focalizaram principalmente a estrutura do sistema, os diagramas de estados simples que modelam alguns estados de um projeto de classe. A Figura 5.26 é um diagrama de estados simples que modela alguns estados de um projeto de classe. Um estado em um diagrama de estados é nomeado com o nome do estado posicionado dentro dele. Um sólido com uma seta designada inicialmente. Lembre-se de que modelamos essas informações de estado como o atributo `userAuthenticated` no diagrama de classes da Figura 4.24. Esse atributo é inicializado no estado ‘Usuário não autenticado’, de acordo com o diagrama de estados.

As setas indicam a transição entre estados. Um objeto pode transitar de um estado para outro em resposta a vários eventos que ocorrem no sistema. O nome ou descrição do evento que causa uma transição é escrito perto da linha que corresponde à transição. Por exemplo, se o usuário muda o estado ‘Usuário não autenticado’ para o estado ‘Usuário autenticado’ depois que o banco de dados autentica o usuário. A partir do documento de requisitos, lembre-se de que o banco de dados autentica um usuário comparando o número de conta inserido pelo usuário com os da conta correspondente no banco de dados. Se o banco de dados indicar que o usuário inseriu um número de conta válido e o PIN correto, muda o estado ‘Usuário não autenticado’ para o estado ‘Usuário autenticado’ e suaude seu atributo `userAuthenticated` para o valor `true`. Quando o usuário sair do sistema escolhendo a opção ‘saída’ a partir do menu principal do objeto ‘Usuário não autenticado’ no estado de preparação para o próximo usuário ATM.



Observação de engenharia de software 5.3

Em geral, os engenheiros de software não criam diagramas de estados que mostram cada estado possível e transição de estado para todos os atributos — há simplesmente muitos deles. Os diagramas de estados em geral mostram apenas os estados e transições de estado mais importantes ou complexos.

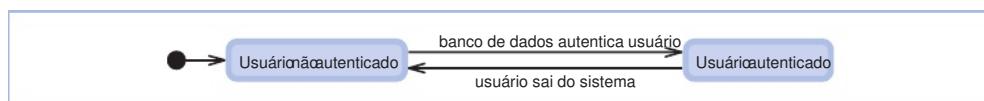


Figura 5.26 Diagrama de estados do objeto ATM

Diagramas de atividades

Como um diagrama de estados, um diagrama de atividades modela aspectos do comportamento de sistema. Ao contrário de um diagrama de estados, um diagrama de atividades modela o fluxo de trabalho de um objeto (seqüência de eventos) durante a execução de uma tarefa. Um diagrama de atividades modela as ações que o objeto realizará e em que ordem. Lembre-se de que utilizamos os diagramas de atividades UML para ilustrar o fluxo de controle para as instruções de controle apresentadas nos capítulos 4 e 5.

O diagrama de atividades na Figura 5.27 modela as ações envolvidas na execução de uma transação de saque. O diagrama inclui as ações que ocorrem depois de o usuário selecionar uma consulta de saldo do menu principal e antes de o ATM retornar o usuário para o menu principal. Afinal, o usuário nem inicia essas ações, então não as modelamos aqui. O diagrama inicia com a recuperação do saldo disponível da conta do usuário a partir do banco de dados, seguida de um recuperação do saldo total da conta. Por fim, a transação exibe o saldo na tela. Essa ação completa a execução da transação.

A UML representa uma ação em um diagrama de atividades como um estado de ação modelado por um retângulo com seus lados esquerdo e direito substituídos por arcos que se curvam para fora. Todo estado de ação contém uma expressão de ação, ou seja, disponibiliza saldo da conta do usuário a partir do banco de dados. Isso especifica uma ação a ser realizada. Uma seta conecta os estados de ação, indicando a ordem em que ocorrem as ações representadas pelos estados de ação. O círculo sólido (na parte inferior da Figura 5.27) representa o estado inicial da atividade — o começo do fluxo de trabalho antes de o objeto realizar as ações modeladas. Nesse caso, a transação primeiro executa a expressão de ação ‘obter saldo disponível da conta do usuário a partir do banco de dados’, em seguida, a transação recupera o saldo total. Por fim, a transação exibe ambos os saldos na tela. O círculo sólido dentro de um círculo vazado (na parte inferior da Figura 5.27) representa o estado final — o fim do fluxo de trabalho depois de o objeto realizar as ações modeladas.

A Figura 5.28 mostra um diagrama de atividades de uma **Associação** que um número de conta válida foi atribuído ao objeto `aval`. Não modelamos o usuário que seleciona a opção de saque do menu principal ou a ATM que retorna o usuário para o menu principal porque essas não são ações realizadas **transação**. O menu de dinheiro exibe um menu de valores-padrão de retirada (Figura 2.17) e uma opção de cancelamento da transação. A transação então realiza a entrada de uma condição de guarda associada. Se o usuário cancelar a transação, o sistema exibe uma mensagem apropriada. Em seguida, o cancelamento alcança um símbolo de agregação, em que esse fluxo de atividade se funde com outros possíveis fluxos de atividade da transação (que discutiremos em breve). Observe que uma agregação pode ter qualquer número de setas que entram na trama apenas uma seta que sai da transação. A decisão na parte inferior do diagrama determina se a transação deve repetir desde Quando o usuário tiver cancelado a transação, a condição de guarda 'dinheiro fornecido ou usuário cancelou a transação' é verdadeira.

então o controle salta para a etapa final de decisão de menu, que usa a guarda da classe `Salvo` para controlar o fluxo de atividades. O resultado é que o saldo disponível é automaticamente modelado na Figura 4.24) como o valor escolhido pelo usuário. A próxima transação obtém o saldo disponível da conta do usuário (isto é, o atributo `Balance` do objeto `Account` do usuário) a partir do banco de dados. O fluxo de atividade então chega à outra decisão. Se o valor da retirada solicitada exceder o saldo disponível do usuário, o sistema exibe uma mensagem apropriada informando o usuário do problema. O controle então se funde com o outro fluxo de atividades antes de alcançar a decisão final da parte inferior do diagrama. A decisão de guarda ‘dinheiro não fornecido e usuário não cancelou a transação’ é verdadeira, então o fluxo de atividades retorna à parte superior do diagrama e a transação solicita ao usuário a entrada de uma nova quantia.

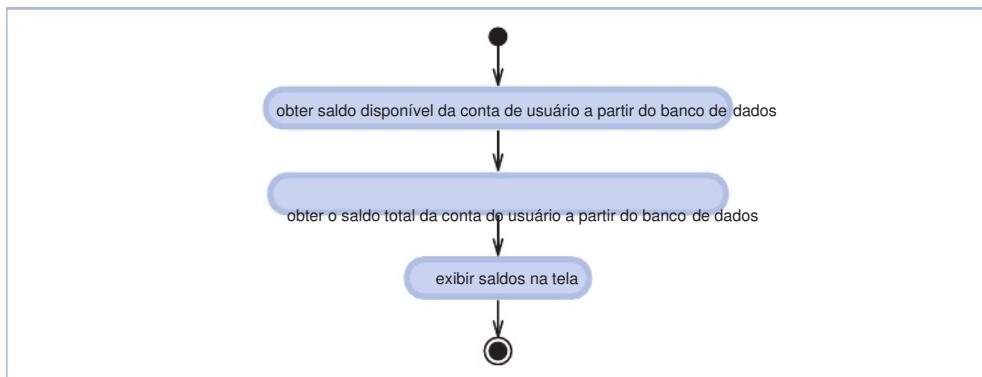


Figura 5.27 Diagrama de atividades de uma transação `BalancInquiry`.

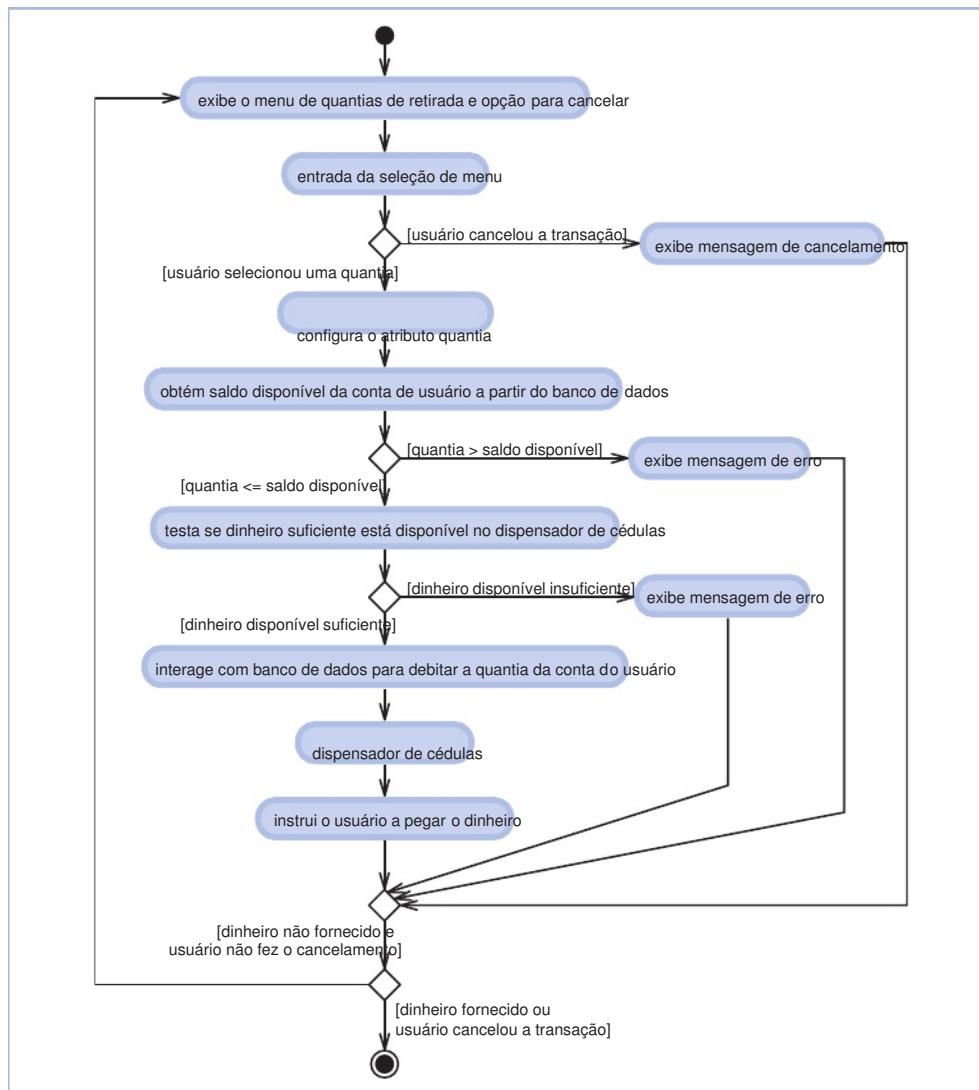


Figura 5.28 Diagrama de atividades de uma transação de retirada (Withdrawal).

Se a quantia de retirada solicitada for menor que ou igual ao saldo disponível do usuário, a transação testa se o dispensador de dinheiro tem o suficiente para satisfazer a solicitação de retirada. Se não tiver, a transação exibe uma mensagem de erro apropriada e retorna ao diagrama de atividades e a transação solicita ao usuário para escolher uma nova quantia. Se dinheiro suficiente estiver disponível, a transação interage com o banco de dados para debituar o valor de retirada da conta de usuário (isto é, subtrair a quantia tanto do availableBalance como do balance do objeto account do usuário). A transação então libera a quantia de dinheiro desejada e instrui o usuário a pegar o dinheiro que é liberado. O principal fluxo de atividade segue funde-se com os dois fluxos de erro e o de cancelamento. Nesse caso, o dinheiro foi fornecido, portanto o fluxo de atividades alcança o estado final.

Demos os primeiros passos na modelagem do comportamento do sistema ATM e mostramos como os atributos de um objeto participam da execução das atividades do objeto. Na Seção 6.22, investigamos as operações de nossas classes para criar um modelo completo do comportamento do sistema.

Exercícios de revisão do estudo de caso de engenharia de software

- 5.1 Determine se a seguinte sentença é verdadeira ou falsa, e, se for falsa, explique por quê: Diagramas de estados modelam aspectos estruturais de um sistema.
- 5.2 Um diagrama de atividades modela os(as) que um objeto realiza e a ordem em que ele os(as) realiza.
- ações
 - atributos
 - estados
 - transições de estado
- 5.3 Com base no documento de requisitos, crie um diagrama de atividades para uma transação de depósito.
Respostas aos exercícios de revisão do estudo de caso de engenharia de software
- 5.1 Falsa. Diagramas de estados modelam algum comportamento de um sistema.
- 5.2 a.
- 5.3 A Figura 5.29 apresenta um diagrama de atividades para uma transação de depósito. O diagrama modela as ações que ocorrem depois que o usuário escolher a opção de depósito no menu principal e antes de o ATM retornar o usuário para o menu principal. Lembre-se de que pa

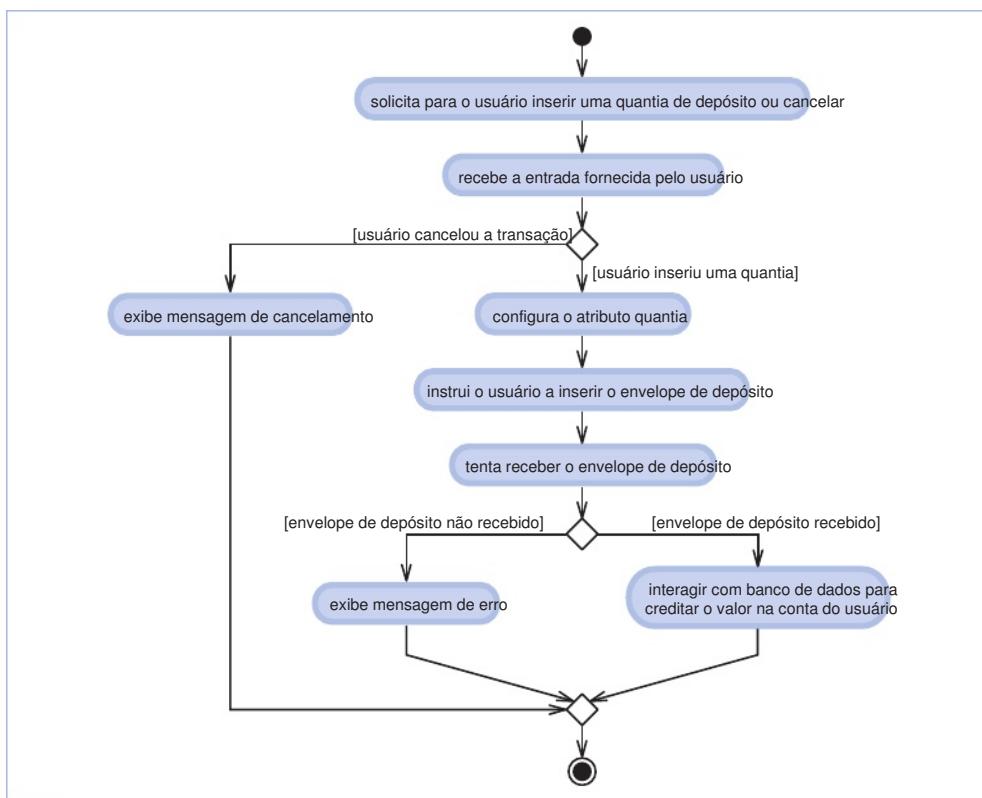


Figura 5.29 Diagrama de atividades de uma transação de depósito.

do recebimento de um depósito do usuário envolve converter um número inteiro de centavos em uma quantia em dólar. Lembre-se também que fazer um depósito em uma conta envolve aumentar o atributo `availableBalance` do objeto `Account` do usuário. O banco só atualiza o atributo `availableBalance` do objeto `Account` do usuário depois de confirmar a quantia em dinheiro no envelope de depósito e, no caso dos cheques, depois da compensação — isso ocorre independentemente do sistema ATM.

5.12 Síntese

Neste capítulo, completamos nossa introdução às instruções de controle do C++, que permitem aos programadores controlar o fluxo de execução em funções. O Capítulo 4 discutiu as instruções `if ..else`, `switch`. Este capítulo demonstrou as demais instruções de controle do C++: `for`, `do..while`, `break`, `continue`. Mostramos que qualquer algoritmo pode ser desenvolvido utilizando combinações da estrutura de seqüência (isto é, instruções listadas na ordem em que devem executar), os três tipos de instruções de seleção `if ..else`, `switch` — e os três tipos de instruções de repetição `for`, `do..while`, `break`. Neste capítulo e no Capítulo 4, discutimos como os programadores podem combinar esses blocos de construção para utilizar as comprovadas técnicas de construção de solução de problemas. Este capítulo também introduziu operadores lógicos do C++, que permitem aos programadores utilizar expressões condicionais mais complexas em instruções de controle. Por fim, examinamos os erros comuns de confundir o operador de igualdade com o de atribuição e fornecemos sugestões para evitar esses erros.

No Capítulo 3, introduzimos a programação C++ com os conceitos básicos de objetos, classes e funções-membro. O Capítulo 4 e este capítulo forneceram uma introdução completa aos tipos de instruções de controle que os programadores geralmente utilizam para especificar lógica do programa em funções. No Capítulo 6, examinamos as funções em maior profundidade.

Resumo

- Em C++, é preciso chamar uma declaração que também reserva memória de uma definição.
- A instrução de repetição `for` fornece todos os detalhes da repetição controlada por contador. O formato geral da instrução


```
for ( inicialização ; condiçãoDeContinuaçãoDoLoop ; incremento )
    instrução
```

 onde `inicialização` inicializa a variável de controle, `condiçãoDeContinuaçãoDoLoop` é a condição que determina se o loop deve continuar executando e `incremento` incrementa a variável de controle.
- Em geral, as instruções utilizadas para repetição controlada por contador são utilizadas para repetição controlada por sentinela.
- O escopo de uma variável define onde ela pode ser utilizada em um programa. Por exemplo, uma variável de controle declarada no cabeçalho de uma instrução pode ser utilizada no corpo da instrução, mas a variável de controle não será conhecida fora da instrução.
- As expressões de inicialização e incremento no cabeçalho podem incluir listas de expressões separadas por vírgulas. As vírgulas, como utilizadas nessas expressões, são operadores vírgula, que garantem que as listas de expressões sejam avaliadas da esquerda para a direita. O operador vírgula tem a precedência mais baixa de todos os operadores C++. O valor e o tipo de uma lista de expressões separadas por vírgulas são o valor e o tipo da expressão mais à direita na lista.
- As expressões de inicialização, condição de continuação do loop e incremento contêm expressões aritméticas. Além disso, o incremento de uma variável pode ser negativo, caso em que é realmente um decremento e o loop conta para baixo.
- Se a condição de continuação do loop em `for` é verdadeira, o corpo da instrução é executado. Em vez disso, a execução prossegue com a instrução seguinte ao loop.
- A função de biblioteca padrão `pow` calcula o valor elevado à potência. A função `pow` recebe dois argumentos de tipo `double` e retorna um `double`.
- Manipulador de fluxo `setw` especifica a largura de campo em que a próxima saída de valor deve aparecer. Por padrão, o valor é alinhado à direita no campo. Se o valor a ser enviado para a saída for maior que a largura de campo, a largura de campo é estendida para acomodar o valor inteiro. O manipulador de fluxo `setprecision` (`setprecision(n)`) pode ser utilizado para fazer com que
- `cout` imprima suas saídas com larguras de campo especificadas, mesmo quando o manipulador `setw` não estiver em vigor até serem alteradas.
- A instrução de repetição `do..while` testa a condição de continuação do loop no final do loop, então o corpo do loop será executado pelo menos uma vez. O formato da instrução é:

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

- A instrução de seleção **múltipla** executa diferentes ações com base nos possíveis valores de uma variável ou expressão. Cada ação é associada com o valor de uma expressão inteira constante (isto é, qualquer combinação de constantes de caractere e constantes inteiros que avaliadas como um valor constante inteiro) que a variável **switch** pode assumir.
- A instrução **switch** consiste em uma série de rótulos **case** opcionais.
- A função **get()** lê um caractere do teclado. Os caracteres são normalmente armazenados na memória sob o tipo **char**. Outros tipos podem ser armazenados em qualquer tipo de dados inteiro, porque são representados como inteiros de 1 byte no computador. Portanto caractere pode ser tratado como um inteiro ou como um caractere, dependendo de seu uso.
- O indicador de fim do arquivo é uma combinação de pressionamento de teclas dependentes de sistema que especifica que não há mais dados inseridos no arquivo. É uma constante simbólica inteira definida no arquivo de cabeçalha ‘fim de arquivo’.
- A expressão entre os parênteses que se segue à palavra-chave **switch** é a expressão controlada da instrução **switch**, que compara o valor da expressão de controle a cada rótulo.
- Listas de instruções consecutivamente sem instruções entre elas formam o mesmo conjunto de instruções.
- Cada rótulo pode ter múltiplas instruções. A instrução **break** sai de outras instruções de controle porque não exige que as múltiplas instruções estejam entre chaves.
- A instrução **switch** pode ser utilizada somente para testar uma expressão inteira constante. Uma constante de caractere é representada como o caractere específico entre aspas simples. Um constante de inteiro é simplesmente um valor de inteiro. Além disso, cada rótulo pode especificar apenas uma expressão inteira constante.
- O C++ fornece vários tipos de dados para representar inteiros long. O intervalo de valores inteiros para cada tipo depende do hardware do computador particular.
- A instrução **break**, quando executada em uma das instruções de repetição (**for**, **while**), ocasiona saída imediata da instrução.
- A instrução **continue**, quando executada em uma das instruções de repetição (**for**, **while**) ignora todas as instruções restantes no corpo da instrução de repetição e prossegue com a próxima iteração do loop. Esta instrução continua com a próxima avaliação da condição. Em forma de estrutura continua com a expressão de incremento no cabeçalho de instrução.
- Os operadores lógicos permitem aos programadores formar condições complexas combinando condições simples. Os operadores lógicos são (E lógico), (OU lógica) (NÃO lógico, também chamado de negação lógica).
- O operador (E lógico) certifica-se de que duas condições sejam verdadeiras para escolher certo caminho de execução.
- O operador (OU lógico) certifica-se de que qualquer uma das condições seja verdadeira para escolher certo caminho de execução.
- Uma expressão contendo os operadores é avaliada somente até que a condição de verdade ou falsidade da expressão seja conhecida. Esse recurso de desempenho para avaliação de expressões com E lógico e OU lógico é chamado avaliação de curto-circuito.
- O operador (NÃO lógico, também chamado de negação lógica) permite ao programador ‘inverter’ o significado de uma condição. O operador unário de negação lógica é colocado antes de uma condição escolher um caminho de execução se a condição for falsa. Na maioria dos casos, o programador pode evitar a utilização da negação lógica expressando a condição com um operador relacional ou de igualdade apropriado.
- Quando utilizado como uma condição, qualquer valor não-zero é convertido implicitamente para true.
- Por padrão, os valores false são exibidos por **cout**, respectivamente. O manipulador de fluxo **setf** especifica que o valor de cada expressão deve ser exibido como a palavra ‘true’ ou ‘false’.
- Qualquer forma de controle que possa ser necessária um dia em um programa C++ pode ser expressa em termos de instruções de seqüência, seleção e repetição e essas podem ser combinadas somente de duas maneiras — empilhamento e aninhamento.

Terminologia

!, operador NÃO lógico	condição simples	erro off-by-one
&& operador lógico	configuração aderente	escopo de uma variável
operador lógico	conjunto de caracteres ASCII	expressão controladora
alinhamento à direita	continua, instrução	loop de retardo
alinhamento à esquerda	curto-circuito, avaliação	loop while
boolalpha, manipulador de fluxo	decrementar uma variável de controle	loop while
break, instrução	default, caso, enquanto	incrementar uma variável de controle
case, rótulo	definição	largura de campo
char, tipo fundamental	E lógico	left, manipulador de fluxo
condição de continuação do loop	empilhamento, regra	

<code>lvalue(left value)</code>	OU lógico)	tabela-verdade
manipulador de fluxo	pow função da biblioteca-padrão	valor final de uma variável de controle
NÃO lógico (regra de aninhamento	valor inicial de uma variável de controle
negação lógica (rvalue(right value)	vírgula, operador
nome de uma variável de controle	setw, manipulador de fluxo	zero, contagem baseada em
operador lógico	switch , instrução de múltipla seleção	

Exercícios de revisão

- 5.1 Determine se as seguintes afirmações são verdadeiras ou falsas. Se a resposta for falsa, explique por quê.
- O caso `default` é requerido na instrução `de seleção`.
 - A instrução `case` é requerida no caso-padrão de uma instrução `para`.
 - A expressão `if (expresão)` é considerada uma expressão lógica baseada na seguinte regra: os termos forem
- 5.2 Escreva uma instrução C++ ou um conjunto de instruções C++ para realizar cada uma das seguintes tarefas:
- Somar os inteiros ímpares entre 1 e 99 utilizando `Assumindo que as variáveis int foram declaradas.`
 - Imprimir o valor `12345672em` uma largura de `campo` caracteres com precisões `decimais`. Imprimir cada número na mesma linha. Alinear à esquerda cada número em seu campo. O que os três valores imprimem?
 - Calcular o valor `elevado` à potência `base`, utilizando a função `pow` para imprimir o resultado com uma `precisão` de `largura` de campo `de posições`. O que é impresso?
 - Imprimir os inteiros de 1 a 20 utilizando `uma variável de controle`. `Assumir que a variável declarada mas não foi inicializada. Imprimir somente 5 inteiros de tabulação o cálculo 5. Quando o valor disso for 0, imprimir um caractere de nova linha; caso contrário, imprimir um caractere de tabulação.]`
 - Repetir o Exercício 5.2 (d) utilizando `uma instrução`
- 5.3 Localize o(s) erro(s) em cada um dos seguintes segmentos de código e explique como corrigi-lo(s).
- ```
x = 1;
while (x <= 10);
 x++;
}
```
  - ```
for (y = 1; y != 1.0; y += .1)
{
    cout << y << endl;
```
 - ```
switch (n)
{
 case 1:
 cout << "The number is 1" << endl;
 case 2:
 cout << "The number is 2" << endl;
 break;
 default :
 cout << "The number is not 1 or 2" << endl;
 break;
}
```
  - O seguinte código deve imprimir os valores 1 a 10.
- ```
n = 1;
while (n < 10)
    cout << n++ << endl;
```

Respostas dos exercícios de revisão

- 5.1
- Falsa. O `default` é opcional. Se nenhuma ação-padrão é necessária, então não é necessário.
 - Falsa. Se a condição de saída é considerada boa engenharia de software sempre fornecer um caso.
 - Falsa. A instrução `break` é utilizada para sair da instrução `switch`. `break` não é requerida quando a ação é o último caso. Nem a instrução `break` será necessária se fizer sentido ter o controle prosseguindo com o próximo caso.
 - Falsa. Ao utilizar o operador `as` as expressões relacionais `devem ser` expressão inteira seja

5.2

a) sum = 0;
`for (count = 1; count <= 99; count += 2)
 sum += count;`

b) cout << fixed << left
`<< setprecision(1) << setw(15) << 333.546372
 << setprecision(2) << setw(15) << 333.546372
 << setprecision(3) << setw(15) << 333.546372
 << endl;`

A saída é:
`333.5 333.55 333.546`

c) cout << fixed << setprecision(2)
`<< setw(10) << pow(2.5, 3)
 << endl;`

A saída é:
`15.63`

d) x = 1;

`while (x <= 20)
{
 cout << x;

 if (x % 5 == 0)
 cout << endl;
 else
 cout << '\t' ;

 x++;
}
e) for (x = 1; x <= 20; x++)
{
 cout << x;

 if (x % 5 == 0)
 cout << endl;
 else
 cout << '\t' ;
}`

ou

`for (x = 1; x <= 20; x++)
{
 if (x % 5 == 0)
 cout << x << endl;
 else
 cout << x << '\t' ;
}`

5.3

a) Erro: O ponto-e-vírgula depois do aberto de um loop infinito.
Correção: Substitua o ponto-e-vírgula por `\n`.

b) Erro: Utilizar um número de ponto flutuante para controlar um loop de repetição.
Correção: Utilize um inteiro e realize o cálculo adequado para obter os valores que você deseja.

`for (y = 1; y != 10; y++)
 cout << (static_cast < double>(y) / 10) << endl;`

c) Erro: Instrução ausente no prêmio

Correção: Adicione uma instrução das instruções para o prêmio. Observe que esse não é um erro se o programador quiser que a instrução seja executada toda vez que a instrução executar.

d) Erro: Operador relacional impróprio usado na condição de repetição de continuação do loop.
Correção: Utilize `==` vez de `=` altere para 1.

Exercícios

5.4 Localize o(s) erro(s) em cada um dos seguintes:

- `For(x = 100, x >= 1, x++)`
`cout << x << endl;`
- O seguinte código deve imprimir ~~sempre~~ se for par ou ímpar:

```
switch ( value % 2 )
{
    case0:
        cout << "Even integer" << endl;
    case1:
        cout << "Odd integer" << endl;
}
```
- O código ~~deve seguir~~, deve dar saída dos inteiros ímpares de 19 a 1:
`for (x = 19, x >= 1, x--)`
`cout << x << endl ;`
- O código seguinte deve dar saída dos inteiros pares de 2 a 100:
`counter = 2;`
`do`
`{`
 `cout << counter << endl;`
 `counter += 2;`
`} While(counter < 100);`

5.5 Escreva um programa que utiliza ~~uma pausa no teclado~~ uma seqüência de inteiros. Assuma que o primeiro inteiro lido especifica o número de valores que restam a ser inseridos. Seu programa deve ler somente um valor por instrução de entrada. Uma típica seqüência de entrada talvez seja

5 100 200 300 400 500

onde ~~5~~ indica que os ~~valores~~ seguintes devem ser somados.

5.6 Escreva um programa que utiliza ~~uma pausa no teclado~~ e imprimir a média de vários inteiros. Assuma que o último valor lido

é o ~~símbolo de final de arquivo~~. Uma típica seqüência de entrada talvez seja

10 8 11 7 9 9999 que indica que o programa deve calcular a média de todos ~~os~~ valores que precedem

5.7 O que o seguinte programa faz?

```

1 // Exercício 5.7: ex05_07.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10     int x; // declara x
11     int y; // declara y
12
13     cout solicita Entrada de duas inteiros no range 1-20: "      ";
14     cin >> x >> y; // lê valores para x e y
15
16
17     for ( int i = 1; i <= y; i++ ) // conta de 1 a y
18     {
19         for ( int j = 1; j <= x; j++ ) // conta de 1 a x
20             cout <<      '@'; // envia @ para saída
21

```

```

22     cout << endl; // inicia nova linha
23 } // fim do for externo
24
25     return 0; // indica terminação bem-sucedida
26 } // fim de main

```

- 5.8 Escreva um programa que utiliza ~~uma impressora local~~ para imprimir o menor de vários inteiros. Assuma que o primeiro valor lido especifica o número de valores restantes e que o primeiro número não é um dos inteiros a ser comparado.
- 5.9 Escreva um programa que utiliza ~~uma impressora local~~ e imprimir o produto dos inteiros ímpares de 1 a 15.
- 5.10 A função factorial é freqüentemente utilizada em problemas de probabilidade. Utilizando a definição de factorial no Exercício 4.35, escreva um programa que utiliza ~~uma impressora local~~ para avaliar o factorial dos inteiros de 1 a 5. Imprima os resultados no formato de tabela. Que dificuldade poderia impedir você de calcular o factorial de 20?
- 5.11 Modifique o programa de juros compostos da Seção 5.4 para repetir seus passos para as taxas de juros de 5%, 6%, 7%, 8%, 9% e 10%. Utilize uma instrução para variar a taxa de juros.
- 5.12 Escreva um programa que utiliza ~~uma impressora local~~ para imprimir os seguintes padrões separadamente, um embaixo do outro. Utilize loops for para gerar os padrões. Todos os desenhos impressos por uma única instrução na fórmula faz com que os asteriscos sejam impressos lado a lado. Os dois padrões requerem que cada linha inicie com um número apropriado de espaços em branco extra. Combine seu código dos quatro problemas separados em um único programa que imprime todos os quatro padrões lado a lado usando inteligência de endereços de saída.
- (a) (b) (c) (d)
- | | | | |
|-------|-------|-------|-------|
| ***** | ***** | ***** | * |
| *** | ***** | ***** | ** |
| *** | ***** | ***** | *** |
| **** | ***** | ***** | *** |
| ***** | ***** | ***** | **** |
| ***** | ***** | ***** | ***** |
| ***** | *** | *** | ***** |
| ***** | ** | ** | ***** |
| ***** | * | * | ***** |
- 5.13 Uma aplicação interessante dos computadores são os desenhos de gráficos e gráficos de barras. Escreva um programa que lê cinco números (cada um entre 1 e 30). Suponha que o usuário insira apenas valores válidos. Para cada número lido, seu programa deve imprimir uma linha contendo esse número de asteriscos adjacentes. Por exemplo, se seu programa lê o número 7, ele deve imprimir:
- 5.14 Uma empresa depedidos pelo correio vende cinco produtos diferentes cujos preços de varejo são: produto 1 — \$ 2,98, produto 2 — \$ 4,50, produto 3 — \$ 9,98, produto 4 — \$ 4,49 e produto 5 — \$ 6,87. Escreva um programa que lê uma série de pares de números como mostrado a seguir:
- número de produto
 - quantidade vendida
- Seu programa deve utilizar uma ~~instrução~~ para determinar o preço de varejo de cada produto. Seu programa deve calcular e exibir o valor de varejo total de todos os produtos vendidos. Utilize um loop controlado por sentinela para determinar quando o programa deve parar o loop e exibir os resultados finais.
- 5.15 Modifique o programa ~~Booldas~~ da Figura 5.9—Figura 5.11 para que ele calcule a média de notas baseada em pontos para o conjunto de notas. Uma nota A vale 4 pontos, B vale 3 pontos etc.
- 5.16 Modifique o programa na Figura 5.6 e, então, utilize somente inteiros para ~~cada uma das questões~~ [monetárias como números inteiros em centavos. Então 'divida' o resultado em suas partes dólar e centavos utilizando as operações de divisão e módulo, respectivamente. Qual uma das seguintes instruções imprime? Os parênteses são necessários em cada caso?
- cout << (i == 1) << endl;
 - cout << (j == 3) << endl;
 - cout << (i >= 1 && j < 4) << endl;
 - cout << (m <= 99 && k < m) << endl;
 - cout << (j >= i || k == m) << endl;
 - cout << (k + m < j || 3 - j >= k) << endl;

- g) cout << (!m) << endl;
- h) cout << (!(j - m)) << endl;
- i) cout << (!(k > m)) << endl;

- 5.18 Escreva um programa que imprime uma tabela dos equivalentes binários, octais e hexadecimais dos números decimais no intervalo 1 a 256. Se não estiver familiarizado com esses sistemas de números, leia primeiro o Apêndice D, "Sistemas de numeração".

- 5.19 Calcule o valor das séries infinitas

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Imprima uma tabela que mostra o valor a_n para cada um dos primeiros 1.000 termos dessa série.

- 5.20 (Triplos de Pitágoras) Um triângulo retângulo pode ter lados que são todos inteiros. Um conjunto de três valores inteiros para os lados de um triângulo retângulo é chamado de triplo de Pitágoras. Esses três lados devem satisfazer o relacionamento de que a soma dos quadrados dos dois lados seja igual ao quadrado da hipotenusa. Localize todos os lados inteiros que formam triplos de Pitágoras para todos os valores menores que 1000. Força bruta: amarrado que temos todas as possibilidades. Esse é um exemplo de computação baseada na força bruta. Você aprenderá em cursos mais avançados de ciência da computação que há muitos problemas interessantes para os quais não há nenhuma abordagem algorítmica conhecida diferente da pura força bruta.

- 5.21 Uma empresa paga seus empregados como gerentes (que recebem um salário fixo por semana), horistas (que recebem um salário-hora fixo pelas primeiras 40 horas trabalhadas e mais hora extra com 50% de acréscimo, isto é, 1,5 vez seu salário-hora, para as horas extra trabalhadas), comissionados (que recebem \$ 250 mais 5,7% bruto das vendas semanais) ou trabalhadores por produção (que recebem uma quantia fixa de dinheiro para cada item que produzem — cada trabalhador por produção trabalha apenas em um tipo de item nessa empresa). Escreva um programa para computar o pagamento semanal de cada empregado. Você não sabe antecipadamente o número de empregados. Cada tipo de empregado tem seu próprio código de salário: os gerentes têm código 1, os horistas têm código 2, os comissionados têm código 3 e os trabalhadores por produção têm código 4. Calcule o salário de cada empregado de acordo com o código de pagamento desse empregado. Dentre o usuário (isto é, o caixa que faz a folha de pagamento) insira os fatos apropriados de que o programa precisa para calcular o salário de cada empregado de acordo com o código de pagamento desse empregado.

- 5.22 (Leis de De Morgan) Neste capítulo, discutimos os operadores lógicos de De Morgan às vezes podem tornar mais conveniente para nós expressarmos uma expressão lógica. Essas leis também podem nos ajudar a expressar convenientemente equivalente à $\neg(\neg p \wedge \neg q)$. Além disso, a expressão $\neg(p \wedge \neg q)$ é logicamente equivalente à expressão $\neg p \vee q$. Utilize as leis de De Morgan para escrever expressões equivalentes

para cada uma das seguintes, e então escreva um programa para mostrar que a expressão srinal e a nova expressão em cada caso são iguais.

- a) $!(x < 5) \&\& !(y > 7)$
 - b) $!(a == b) \parallel !(g != 5)$
 - c) $!((x <= 8) \&\& (y > 4))$
 - d) $!((i > 4) \parallel (j <= 6))$

- 5.23 Escreva um programa que imprime a seguinte forma de losango. Você pode utilizar instruções de saída que imprimem um único asterisco (*) ou um único espaço em branco. Maximize sua utilização de repetição (adas) estruturas e o número de instruções de saída.

★

★

- 5.24** Modifique o programa que você escreveu no Exercício 5.23 para ler um número ímpar no intervalo de 1 a 19 a fim de especificar o número de linhas no losango e, então, exiba um losango do tamanho apropriado.

- 5.25 Uma crítica às ~~instruções~~ é que elas não são instruções estruturadas. De fato, elas podem ser sempre substituídas por instruções estruturadas, embora possa ser inconveniente fazer isso. Descreva de maneira geral como você removeria qualquer instrução break de um loop em um programa e a substituiria por alguma estrutura de controle [um loop a partir de dentro do corpo do loop. Outra maneira de deixar é fazendo falhar o teste de continuação do loop. Considere a possibilidade de utilizar

no teste de continuação do loop um segundo teste que indica 'saída prévia por causa de uma condição 'break''.] Utilize a técnica que você desenvolveu aqui para remover ~~break~~ o programa da Figura 5.13.

5.26 O que o seguinte segmento de programa faz?

```

1   for ( int i = 1; i <= 5; i++ )
2   {
3       for ( int j = 1; j <= 3; j++ )
4       {
5           for ( int k = 1; k <= 4; k++ )
6               cout << *;
7
8           cout << endl;
9       } // fim do for interno
10
11      cout << endl;
12  } // fim do for externo

```

5.27 Descreva de maneira geral como você removeria ~~break~~ de instrução em um programa e a substituiria por alguma equivalente estruturada. Utilize a técnica que você desenvolveu aqui para remover ~~break~~ o programa da Figura 5.14.

5.28 (A canção 'The Twelve Days of Christmas') Escreva um programa que utiliza instruções de repetição para imprimir a canção Twelve Days of Christmas. Uma instrução `cout` deve ser utilizada para imprimir o dia (isto é, 'First', 'Second' etc.) Uma instrução `switch` separada deve ser utilizada para imprimir o restante de cada verso. www2.cs.uic.edu/~cs100/StoryWorlds/12daysofmas.htm para obter a letra completa da canção.

5.29 (Problema de Peter Minuit) ~~Diz~~ a lenda que, em 1626, Peter Minuit comprou a Ilha de Manhattan por \$ 24,00 na base da troca. Será que ele fez um bom investimento? Para responder a essa pergunta, modifique o programa de juros compostos da Figura 5.6 para iniciar com um capital de \$ 24,00 e calcular o valor dos juros em depósito se esse dinheiro continuasse depositado até este ano (por exemplo, 37 anos até 2005). Coloque ~~cout~~ para realizar o cálculo de juros compostos ~~externamente~~ varia a taxa de juros de 5% a 10% para observar as maravilhas dos juros compostos.



A forma nunca segue a função.
Louis Henri Sullivan

E pluribus unum.

(Virgílio composto de muitos.)

Chama o dia de ontem, faze que o tempo atrás retorne.
William Shakespeare

Chamem-me Ismael.
Herman Melville

Quando você me chamar assim sorria!
Owen Wister

Responda-me em uma palavra.
William Shakespeare

Há um ponto em que os métodos se autodevoram.
Franz Fanon

A vida só pode ser compreendida olhando-se para trás; mas só pode ser vivida olhando-se para a frente.
Soren Kierkegaard

Funções e uma introdução à recursão

OBJETIVOS

Neste capítulo, você aprenderá:

A construir programas modularmente a partir de funções.

A utilizar funções de matemática comuns disponíveis na C++ Standard Library.

A criar funções com múltiplos parâmetros.

Os mecanismos para passar informações entre funções e retornar resultados.

Como o mecanismo de chamada/retorno de função é suportado pela pilha de chamadas de função e os registros de ativação.

A utilizar a geração de números aleatórios para implementar aplicativos de jogos.

Como a visibilidade de identificadores é limitada a regiões específicas de programas.

A escrever e utilizar funções recursivas, isto é, funções que chamam a si mesmas.

6.1	Introdução
6.2	Componentes de um programa em C++
6.3	Funções da biblioteca de matemática
6.4	Definições de funções com múltiplos parâmetros
6.5	Protótipos de funções e coerção de argumentos
6.6	Arquivos de cabeçalho da biblioteca-padrão C++
6.7	Estudo de caso: geração de números aleatórios
6.8	Estudo de caso: jogo de azar e introdução <code>enum</code>
6.9	Classes de armazenamento
6.10	Regras de escopo
6.11	Pilha de chamadas de função e registros de ativação
6.12	Funções com listas de parâmetro vazias
6.13	Funções inline
6.14	Referências e parâmetros de referência
6.15	Argumentos-padrão
6.16	Operador de solução de escopo unário
6.17	Sobrecarga de funções
6.18	Templates de funções
6.19	Recursão
6.20	Exemplo que utiliza recursão: série de Fibonacci
6.21	Recursão <i>versus</i> iteração
6.22	Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)
6.23	Síntese

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

6.1 Introdução

A maioria dos programas de computador que resolvem problemas do mundo real é bem maior que os programas apresentados nesse capítulo. A experiência tem mostrado que a melhor maneira de desenvolver e manter um programa grande é com a partir de pequenas e simples partes, ou componentes. Essas técnicas se chamarão funções (como partes de programa) no Capítulo 3. Neste capítulo, estudamos funções em mais profundidade. Enfatizamos como declarar e funções para facilitar o projeto, a implementação, a operação e a manutenção de grandes programas.

Apresentaremos a visão geral de uma parte das funções de matemática da C++ Standard Library, mostrando as diversas funções que requerem mais de um parâmetro. Em seguida, você aprenderá a declarar uma função com mais de um parâmetro. Também apresentaremos informações adicionais sobre protótipos de função e como o compilador os utiliza para converter o tipo de um argumento para a chamada de função para o tipo especificado em uma lista de parâmetros da função, se necessário.

Em seguida, faremos uma breve digressão sobre as técnicas de simulação com a geração de números aleatórios (ou randomicos). Desenvolveremos uma versão do jogo de dados. Depois, utilizaremos as técnicas de programação aprendidas até agora no livro.

A seguir, apresentaremos as classes de armazenamento e as regras de escopo do C++. Elas determinam o período em que existe na memória e onde seu identificador pode ser referenciado em um programa. Você também aprenderá como o C++ é usado para monitorar as funções que estão atualmente em execução, como os parâmetros e outras variáveis de funções locais são manipulados.

Por fim, discutiremos templates de função, que permitem que uma única função possa ser usada para diferentes tipos de dados. Discutiremos como essas técnicas podem ser utilizadas para passar grandes itens de dados para funções eficientemente.

Muitos dos aplicativos que você desenvolve terão mais de uma função com o mesmo nome. Essa técnica, chamada sobrecarga, é utilizada pelos programadores para implementar as funções que realizam tarefas semelhantes para argumentos de tipos diferentes ou possivelmente para diferentes números de argumentos. Consideraremos templates de função — um mecanismo para definir um conjunto de funções sobrecarregadas. O capítulo encerra com uma discussão de funções que chamam a si próprias, direta ou indiretamente — um tópico chamado recursão que é discutido extensamente em cursos de nível superior de ciência da computação.

6.2 Componentes de um programa em C++

Em geral, os programas C++ são escritos combinando novas funções e classes que o programador escreve com funções ‘pré-das’ e classes disponíveis na C++ Standard Library. Neste capítulo, vamos nos concentrar nas funções.

A C++ Standard Library fornece uma rica coleção de funções para realizar cálculos matemáticos comuns, manipulações de manipulações de caractere, entrada/saída, verificação de erros e muitas outras operações úteis. Isso torna o trabalho do programador mais fácil porque essas funções fornecem muitas das capacidades de que o programador precisa. As funções da C++ Standard Library são fornecidas como parte do ambiente de programação C++.



Observação de engenharia de software 6.1

Leia a documentação do seu compilador para se familiarizar com as funções e classes da C++ Standard Library.

As funções (chamadas [procedimento](#) ou ainda [procedure](#)) em outras linguagens de programação permitem que o programador modularize um programa separando suas tarefas em unidades autocontidas. Você utilizou funções em todos os programas que escreveu. Essas funções são às vezes referidas como [funções definidas pelo usuário](#) ou [funções definidas pelo programador](#). As instruções no corpo das funções são escritas apenas uma vez, talvez reutilizadas a partir de diversas localizações em um programa e ocultadas de outras funções.

Há várias motivações para modularizar um programa com funções. Uma delas é a abordagem de dividir e conquistar, que o desenvolvimento de programas mais gerenciável, possibilitando que eles sejam construídos a partir de fragmentos simples. A reusabilidade de software — utilizar funções existentes como blocos de construção para criar novos programas. Por exemplo, em primeiros programas, não tínhamos de definir como ler uma linha de texto a partir do teclado — o C++ fornece essa capacidade por meio da função `getline` do arquivo de cabeçalho `<iostream.h>`. Um terceiro motivo é evitar a repetição de código. Além disso, dividir um programa em funções significativas torna o programa mais fácil de depurar e manter.



Observação de engenharia de software 6.2

Para promover a capacidade de reutilização de software, todas as funções devem ser limitadas à realização de uma única tarefa bem definida e o nome da função deve expressar essa tarefa efetivamente. Essas funções tornam os programas mais fáceis de escrever, testar, depurar e manter.



Dica de prevenção de erro 6.1

Uma pequena função que realiza uma tarefa é mais fácil de testar e depurar do que uma função maior que realiza muitas tarefas.



Observação de engenharia de software 6.3

Se você não puder escolher um nome conciso que expresse a tarefa de uma função, sua função pode estar tentando realizar um número excessivo de tarefas. Em geral, é melhor dividir essa função em várias funções menores.

Como você sabe, uma função é invocada por uma chamada de função e, quando a função chamada completa sua tarefa, ela retorna um resultado ou simplesmente retorna o controle para o chamador. Uma analogia a essa estrutura de programa é a forma hierárquica de gerenciamento (Figura 6.1). Um chefe (semelhante à função chamadora) solicita que um trabalhador (semelhante à função-chamada) realize uma tarefa e informe (isto é, retorne) os resultados depois de completar a tarefa. A função-chefe não sabe como a função-trabalhador realiza as tarefas designadas. O trabalhador também pode chamar outras funções-trabalhador, sem que o chefe saiba. Esse nível de detalhes da implementação promove a boa engenharia de software. [A Figura 6.1 ilustra esta estrutura.](#)

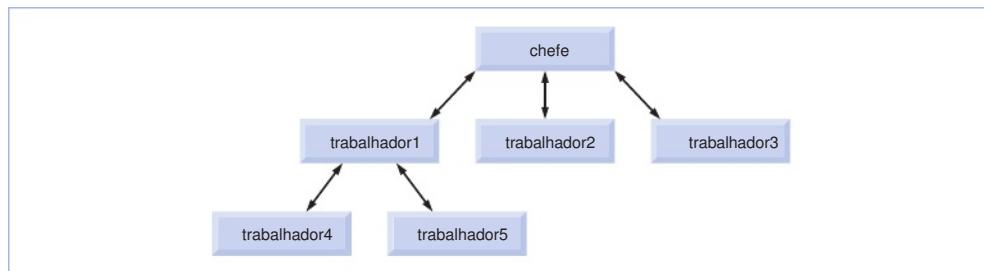


Figura 6.1 Relacionamento hierárquico entre a função-chefe e as funções-trabalhador.

funções-trabalhador de maneira hierárquica. Ademais, a responsabilidade entre as várias funções-trabalhador. Observe que a função-trabalhador1 atua como uma ‘função-chefe’ para as funções-trabalhador2, 3, 4 e 5.

6.3 Funções da biblioteca de matemática

Como você sabe, uma classe pode fornecer funções-membro que realizam os serviços da classe. Por exemplo, nos capítulos 4 e 5 chamou as funções-membro de várias versões de um objeto a mensagem de base `getGrade()` para obter seu nome do curso, obter um conjunto de notas e calcular a média dessas notas.

As vezes as funções não são membros de uma classe. Essas funções, denominadas de funções-membro de uma classe, os protótipos de função para funções globais são colocados em arquivos de cabeçalho, para que as funções possam ser reutilizadas em qualquer programa que inclua o arquivo de cabeçalho e possa ser linkado ao código-objeto da função. Por exemplo, lembre-se de que utilizamos a função `pow()` de cabeçalho para elevar um valor a uma potência na Figura 5.6. Introduzimos aqui várias funções a partir do arquivo `cmath.h` para apresentar o conceito de funções globais que não

pertencem a uma classe particular. Nestes capítulos subsequentes, veremos uma combinação de funções globais (como `sqrt()`) e funções-membro de classes particulares. Nestes capítulos subsequentes, veremos uma combinação de funções globais (como `sqrt()`) e funções-membro de classes particulares.

O arquivo de cabeçalho `cmath.h` fornece uma coleção de funções que permite realizar cálculos matemáticos comuns. Por exemplo, você pode calcular a raiz quadrada de um número com a seguinte chamada de função:

```
sqrt( 900.0 )
```

A expressão anterior é avaliada. A função `sqrt()` aceita um argumento do tipo `double` e retorna um resultado. Observe que não há necessidade de criar qualquer objeto antes de chamar a função. As funções no arquivo de cabeçalho `cmath.h` são funções globais — assim, cada uma é chamada simplesmente especificando o nome da função seguido por parênteses contendo os argumentos da função.

Os argumentos de função podem ser constantes, variáveis ou expressões mais complexas. Se a instrução

```
cout << sqrt( c + d * f ) << endl;
```

calcula e imprime a raiz quadrada de $c + d \cdot f = 25.0$ — a saber, $\sqrt{25.0} = 5.0$. Algumas funções de biblioteca de matemática são resumidas na Figura 6.2. Na figura, `base` é substituída por `e`.

Função	Descrição	Exemplo
<code>ceil(x)</code>	arredonda para o menor inteiro não menor que	<code>ceil(9.2)</code> é 10.0 <code>ceil(-9.8)</code> é -9.0
<code>cos(x)</code>	co-seno trigonométrico (radianos)	<code>cos(0.0)</code> é 1.0
<code>exp(x)</code>	função exponencial	<code>exp(1.0)</code> é 2.71828 <code>exp(2.0)</code> é 7.38906
<code>fabs(x)</code>	valor absoluto de	<code>fabs(5.1)</code> é 5.1 <code>fabs(0.0)</code> é 0.0 <code>fabs(-8.76)</code> é 8.76
<code>floor(x)</code>	arredonda para o maior inteiro não maior que	<code>floor(9.2)</code> é 9.0 <code>floor(-9.8)</code> é -10.0
<code>fmod(x, y)</code>	resto de <code>x</code> como um número de ponto flutuante	<code>fmod(2.6, 1.2)</code> é 0.2
<code>log(x)</code>	logaritmo natural (base <code>e</code>)	<code>log(2.71828)</code> é 1.0 <code>log(7.38906)</code> é 2.0
<code>log10(x)</code>	logaritmo (base 10)	<code>log10(10.0)</code> é 1.0
<code>pow(x, y)</code>	<code>x</code> elevado à potência <code>y</code>	<code>log10(100.0)</code> é 2.0 <code>pow(2, 7)</code> é 128 <code>pow(9, .5)</code> é 3
<code>sin(x)</code>	seno trigonométrico (radianos)	<code>sin(0.0)</code> é 0
<code>sqrt(x)</code>	raiz quadrada de <code>x</code> (<code>x</code> é um valor não negativo)	<code>sqrt(9.0)</code> é 3.0
<code>tan(x)</code>	tangente trigonométrica (radianos)	<code>tan(0.0)</code> é 0

Figura 6.2 Funções da biblioteca de matemática.

6.4 Definições de funções com múltiplos parâmetros

Os capítulos 3 a 5 apresentaram as classes com funções simples que tinham no máximo um parâmetro. As funções costumam mais de uma informação para realizar suas tarefas. Consideramos agora funções com múltiplos parâmetros.

O programa nas figuras 6.3–6.5 modifica o **GradeBook** para que determine e retorne o maior de três notas. Quando o aplicativo inicia a execução (Figura 6.4 da Figura 6.5) cria um objeto da classe GradeBook (linha 8) e chama a função-membro inputGrades() (linha 11) para ler três notas do tipo inteiro fornecidas pelo usuário. No arquivo de implementação (Figura 6.6), as linhas 54–55 da função-membro maximum() solicitam para o usuário inserir três valores do tipo inteiro e, se é a partir do usuário. A linha 58 chama a função-membro maximum() definida nas linhas 62–75. A função determina o maior valor e, em seguida, a (linha 70) retorna esse valor para o ponto em que a função foi chamada (Figura 6.6). A função-membro maximum() então armazena valor de retorno de um membro de dados mGradeReport. Esse valor é então enviado para a saída obtendo a função GradeReport() (linha 12 da Figura 6.5). [Chamamos essa função GradeReport porque as versões subsequentes da classe GradeBook utilizarão essa função para exibir um relatório de notas completo, incluindo notas máximas e mínimas.] No Capítulo 7, “Arrays e vetores”, aprimoraremos GradeBook para processar um número arbitrário de notas.

```

1 // Figura 6.3: GradeBook.h
2 // Definição da classe GradeBook que localiza a máxima de três notas.
3 // As funções-membro são definidas no GradeBook.cpp
4 #include <iostream> // o programa utiliza classe de string padrão C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public :
11     GradeBook( string ); // o construtor inicializa o nome do curso
12     void setCourseName( string ); // função para configurar o nome do curso
13     string getCourseName(); // função para recuperar o nome do curso
14     void displayMessage(); // exibe uma mensagem de boas-vindas
15     void inputGrades(); // insere três notas fornecidas pelo usuário
16     void displayGradeReport(); // exibe um relatório baseado nas notas
17     int maximum(int , int , int ); // determina o máximo de 3 valores
18 private :
19     string courseName; // nome do curso para esse GradeBook
20     int studentMaximum;/ máxima de três notas
21 }; // fim da classe GradeBook

```

Figura 6.3 Arquivo de cabeçalho GradeBook

```

1 // Figura 6.4: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // determina a máxima de três notas.
4 #include <iostream>
5 using std::cout;
6
7 using std::endl;
8
9 #include "GradeBook.h"// inclui a definição de classe GradeBook
10
11 // construtor inicializa courseName com string fornecida como argumento;
12 // inicializa studentMaximum como 0
13 GradeBook::GradeBook( string name )

```

Figura 6.4 Classe GradeBook define a função maximum

(continua)

```

14  {
15      setCourseName( name ); // valida e armazena courseName
16      studentMaximum = 0; // esse valor será substituído pela nota máxima
17  } // fim do construtor GradeBook
18
19 // função para configurar o nome do curso; limita o nome a 25 ou menos caracteres
20 void GradeBook::setCourseName( string name )
21 {
22     if ( name.length() <= 25 ) // se o nome tiver 25 ou menos caracteres
23         courseName = name; // armazena o nome do curso no objeto
24     else // se o nome tiver mais que 25 caracteres
25     {
26         // configura courseName como os primeiros 25 caracteres do nome de parâmetro
27         courseName = name.substr( 0, 25 ); // seleciona os primeiros 25 caracteres
28         cout << "Name '"<< name << "' exceeds maximum length (25).\\n"
29             << "Limiting courseName to first 25 characters.\\n" << endl;
30     } // fim do if...else
31 } // fim da função setCourseName
32
33 string GradeBook::getCourseName()
34 {
35     return courseName;
36 } // fim da função getCourseName
37
38 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
39 void GradeBook::displayMessage()
40 {
41     // essa instrução chama getCourseName para obter o
42     // nome do curso que esse GradeBook representa
43     cout << "Welcome to the grade book for\\n" << getCourseName() << "\\n"
44     << endl;
45 } // fim da função displayMessage
46
47 // insere três notas a partir do usuário; determina a máxima
48 void GradeBook::inputGrades()
49 {
50     int grade1; // primeira nota inserida pelo usuário
51     int grade2; // segunda nota inserida pelo usuário
52     int grade3; // terceira nota inserida pelo usuário
53
54     cout << "Enter three integer grades: " ;
55     cin >> grade1 >> grade2 >> grade3;
56
57     // armazena máxima no membro studentMaximum
58     studentMaximum = maximum( grade1, grade2, grade3 );
59 } // fim da função inputGrades
60
61 // retorna o máximo dos seus três parâmetros inteiros
62 int GradeBook::maximum( int x, int y, int z )
63 {
64     int maximumValue = x; // supõe que x é o maior valor inicial
65
66     // determina se y é maior que maximumValue
67     if ( y > maximumValue )
68         maximumValue = y; // torna y o novo maximumValue
69
70     // determina se z é maior que maximumValue

```

Figura 6.4 ClasseGradeBook define a função maximum

(continua)

```

71 if ( z > maximumValue )
72     maximumValue = z // torna z o novo maximumValue
73
74     return maximumValue;
75 } // fim da função maximum
76
77 // exibe um relatório baseado nas notas inseridas pelo usuário
78 void GradeBook::displayGradeReport()
79 {
80     // gera a saída da nota máxima entre as notas inseridas
81     cout << "Maximum of grades entered: " << studentMaximum << endl;
82 } // fim da função displayGradeReport

```

Figura 6.4 Classe GradeBook define a função maximum

(continuação)

```

1 // Figura 6.5: fig06_05.cpp
2 // Cria o objeto GradeBook, insere notas e exibe relatório de notas.
3 #include "GradeBook.h" // inclui a definição de classe GradeBook
4
5 int main()
6 {
7     // cria objeto GradeBook
8     GradeBook myGradeBook("CS101 C++ Programming")
9
10    myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
11    myGradeBook.inputGrades(); // lê as notas fornecidas pelo usuário
12    myGradeBook.displayGradeReport(); // exibe relatório baseado em notas
13    return 0; // indica terminação bem-sucedida
14 } // fim de main

```

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 86 67 75
Maximum of grades entered: 86

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 86 75
Maximum of grades entered: 86

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 75 86
Maximum of grades entered: 86

Figura 6.5 Demonstrando a função maximum



Observação de engenharia de software 6.4

As vírgulas utilizadas na linha 58 da Figura 6.4 para separar os argumentos são operadores vírgula como discutido na Seção 5.3. O operador vírgula garante que seus operandos sejam avaliados da esquerda para a direita. Entretar a ordem de avaliação dos argumentos de uma função não é especificada pelo padrão C++. Portanto, os diferentes compiladore

podem avaliar argumentos de função em ordens diferentes. O padrão C++ garante que todos os argumentos em uma chamada de função sejam avaliados antes de a função chamada executar.



Dica de portabilidade 6.1

Às vezes, quando os argumentos de uma função são expressões mais complexas, como aquelas com chamadas para outras funções, a ordem em que o compilador avalia os argumentos poderia afetar os valores de um ou mais dos argumentos. Se a ordem de avaliação muda entre compiladores, os valores de argumento passados para a função poderiam variar, causando erros de lógica sutis.



Dica de prevenção de erro 6.2

Se você tiver dúvidas quanto à ordem de avaliação dos argumentos de uma função e se a ordem poderia afetar ou não os valores passados à função, avalie as argumentos em instruções de atribuição separadas antes da chamada de função, atribua o resultado a uma variável e imprima o resultado.

de cada expressão a uma variável local e, então, passe essas variáveis como argumentos à função. O protótipo de função `mamonto` (Figura 6.3, linha 17) indica que a função retorna um valor do tipo inteiro, que o nome da função `mamonto` indica. A linha 18 indica que ela requer três parâmetros do tipo inteiro para realizar sua tarefa. O protótipo de função `calculo` (Figura 6.3, linha 62) corresponde ao protótipo de função e indica que os três primeiros parâmetros são inteiros (Figura 6.3, linha 58), o parâmetro `metro` é inicializado com o valor do argumento `arg1`, o parâmetro `centimetro` é inicializado com o valor do argumento `arg2` e deve haver um argumento na chamada de função para cada metro (também chamado `parâmetro formal`) na definição de função.

Note que múltiplos parâmetros são especificados tanto no protótipo de função como no cabeçalho de função como uma lista separada por vírgulas. O compilador referencia o protótipo de função para verificar se as ~~tipos~~ ~~informações~~ tipos dos argumentos corretos e se esses tipos estão na ordem correta. Além disso, o compilador utiliza o protótipo para assegurar que o valor retornado pela função pode ser utilizado corretamente na expressão que chamou a função (por exemplo, uma chamada de função que retorna não pode ser utilizada como o lado direito de uma instrução de atribuição). Cada argumento deve ser consistente com o tipo do parâmetro correspondente. Por exemplo, o compilador aceita valores como 7,35, 22 ou -0,03456, mas não uma string como. Se os argumentos passados para uma função não corresponderem aos tipos especificados no protótipo da função, o compilador tenta converter os argumentos nesses tipos. A Seção 6.5 discute essa conversão.



Erro comum de programação 6.1



Erro comum de programação 6.2

Os erros de compilação ocorrem se o protótipo de função, o cabeçalho de função e as chamadas de função realmente não corresponderem em número, tipo e ordem de argumentos e parâmetros e no tipo de retorno.



Observação de engenharia de software 6.5

Uma função com muitos parâmetros pode realizar tarefas demais. Considere dividir a função em funções menores que realizam a tarefas separadas. Se possível, limite o cabeçalho de função a uma linha.

Para determinar o valor máximo (linhas 62–75 da Figura 6.4), iniciamos com `maximumValue`, parâmetro então a linha 64 da função declara a variável `local`. A inicializa com o valor do parâmetro. Nesse ponto, é possível que o parâmetro tenham o maior valor real, portanto devemos comparar cada um desses valores com `maximumValue`. As linhas 67–68 determinam se o `for`, atribui `maximumValue`. As instruções linhas 71–72 determinam se `maximumValue` se for, atribui `maximumValue`. Nesse ponto, o maior dos três valores está em `maximumValue`. Então a linha 74 retorna esse valor à chamada na linha 58. Quando o controle do programa retornar ao ponto no código em `maximum`, foi chamado, os parâmetros `maximum` não serão mais acessíveis ao programa. Veremos por que na

Há três maneiras de retornar o controle para o ponto em que uma função foi invocada. Se a função não retornar um resultado, a função tem um tipo de retorno que retorna quando o programa alcançar a chave de fechamento direita da função ou pela execução da instrução

```
return ;
```

Se a função retorna um resultado, a instrução

return expressão

avalia expressão retorna o valor desse **só chamador**.

6.5 Protótipos de funções e coerção de argumentos

Um protótipo de função (também ~~chamado de~~^{informa ao compilador o nome de uma função, o tipo de dados retornado pela função, o número de parâmetros que a função espera receber, os tipos desses parâmetros e a ordem em que eles}



Observação de engenharia de software 6.6

Os protótipos de função são requeridos em C++. Utilize as diretrizes de pré-processador para os protótipos de função das funções da C++ Standard Library a partir dos arquivos de cabeçalho para as bibliotecas apropriadas (por exemplo, o protótipo para a função de matemática `sqrt` no arquivo de cabeçalho `<cmath>` lista parcial de arquivos de cabeçalho C++ Standard Library aparece na Seção 6.6). Além disso, ~~para~~^{obter} arquivos de cabeçalho que contêm protótipos de função escritos por você ou membros do seu grupo.



Erros comuns de programação 6.2

Sé uma função for definida antes de ser invocada, então a definição da função também serve como o protótipo da função, portanto um protótipo separado é desnecessário. Se uma função é invocada antes de ser definida e essa função não tiver um protótipo de função, ocorre um erro de compilação.



Observação de engenharia de software 6.7

Forneça sempre protótipos de função, mesmo que seja possível omiti-los quando as funções são definidas antes de serem utilizadas (caso em que o cabeçalho de função também atua como o protótipo de função). Fornecer os protótipos evita associar o código à ordem em que as funções são definidas (o que pode mudar facilmente à medida que o programa cresce).

Assinaturas de função

A parte de um protótipo de função que inclui o nome da função e os tipos de ~~seus argumentos~~^{argumentos} é chamada de assinatura. A assinatura de função não especifica o tipo de retorno da função. A função no mesmo escopo deve ter assinaturas únicas. O escopo de uma função é a região de um programa em que a função é conhecida e acessível. Falaremos sobre escopo na Seção 6.10.



Erros comuns de programação 6.4

É um erro de compilação se duas funções do mesmo escopo tiverem a mesma assinatura, mas diferentes tipos de retorno.

Na Figura 6.3, se o protótipo de função na linha 17 fosse escrito

```
void maximum(int , int , int );
```

o compilador informaria um erro, porque o tipo do protótipo de função iria diferir do tipo do cabeçalho de função. De maneira semelhante, esse protótipo faria com que a instrução

```
cout << maximum( 6, 9, 0 );
```

gerasse um erro de compilação, porque essa instrução depende de um valor a ser exibido.

Coerção de argumento

Um recurso importante de protótipos de função é a coerção de argumento, isto é, forçar argumentos aos tipos apropriados especificados pelas declarações de parâmetro. Por exemplo, um programa pode chamar uma função com um argumento do tipo inteiro, mas que o protótipo de função especifica um argumento de float; a função ainda funcionará corretamente.

Regras de promoção de argumento

Às vezes, os valores de argumento que não correspondem precisamente aos tipos de parâmetro no protótipo de função podem ser convertidos pelo compilador no tipo adequado antes que a função seja chamada. Essas conversões ocorrem de acordo com as regras de promoção.

As regras de promoção indicam como converter entre tipos de dados diferentes. Um exemplo é a conversão de inteiros para floats. Um float é um tipo de ponto flutuante que pode armazenar números de maior magnitude. Entretanto, a conversão de dados pode ser considerável. Os valores também podem ser modificados ao converter tipos inteiro grandes em tipos inteiro pequenos (por exemplo, long para short), com sinal em sem sinal, ou sem sinal em com sinal.

As regras de promoção se aplicam a expressões que contêm valores de dois ou mais tipos de dados; essas expressões são referidas como **expressões de tipo misto**. O tipo de cada valor em uma expressão de tipo misto é promovido para 'o mais alto' tipo na expressão (na realidade, uma versão temporária de cada valor é criada e utilizada para a expressão — os valores originais permanecem inalterados). A promoção também ocorre quando o tipo de um argumento de função não corresponde ao tipo de parâmetro especificado na definição ou protótipo de função. A Figura 6.6 lista os tipos de dados fundamentais na ordem do 'tipo mais alto' ao 'tipo mais baixo'.



Figura 6.6 Hierarquia de promoção para tipos de dados fundamentais.

Converter valores em tipos fundamentais mais baixos pode resultar em valores incorretos. Portanto, um valor pode ser convertido em um tipo fundamental mais baixo apenas atribuindo explicitamente o valor a uma variável de tipo mais baixo (alguns compiladores emitirão um aviso nesse caso) ou utilizando um operador de coerção (ver Seção 4.9). Os valores de argumento de função são convertidos nos tipos de parâmetro em um protótipo de função como se estivessem sendo atribuídos diretamente às variáveis desses tipos de função. Por exemplo, se uma função que utiliza um parâmetro de inteiro é chamada com um argumento de ponto flutuante, o argumento é convertido em um tipo mais baixo e poderia retornar um valor incorreto. Por exemplo, na Figura 6.5.



Erro comum de programação 6.5

Converter de um tipo de dados mais alto, na hierarquia de promoção, em um tipo mais baixo, ou entre com sinal e sem sinal, pode corromper o valor dos dados, causando perda de informações.



Erro comum de programação 6.6

É um erro de compilação se os argumentos em uma chamada de função não correspondem ao número e tipos dos parâmetros declarados no protótipo de função correspondente. Também é um erro se o número de argumentos na chamada for correspondido, mas os argumentos não puderem ser implicitamente convertidos nos tipos esperados.

6.6 Arquivos de cabeçalho da biblioteca-padrão C++

A C++ Standard Library é dividida em muitas partes, cada qual com seu próprio arquivo de cabeçalho. Os arquivos de cabeçalho contêm os protótipos de função para as funções relacionadas que formam cada parte da biblioteca. Os arquivos de cabeçalho também contêm definições de vários tipos de classe e funções, bem como as constantes de que essas funções precisam. Um arquivo de cabeçalho serve para interfacear com a biblioteca e os componentes escritos pelo usuário.

A Figura 6.7 lista alguns arquivos de cabeçalho comuns da C++ Standard Library, a maioria dos quais é discutida mais adiante. O termo ‘macro’ utilizado várias vezes na Figura 6.7 é discutido no Apêndice F, ‘Pré-processador’. Os nomes dos arquivos de cabeçalho que mencionamos na Figura 6.7 só foram mencionados pelos pré-processadores de cabeçalho da C++ Standard Library. Utilizamos neste livro apenas as versões da C++ Standard Library de cada arquivo de cabeçalho que asseguram que nossos exemplos funcionarão na maioria dos compiladores C++ padrão.

6.7 Estudo de caso: geração de números aleatórios

Agora vamos para uma breve e, esperamos, interessante diversão em um aplicativo de programação popular, a saber: a simulação da execução de um jogo. Nesta e na próxima seção, desenvolvemos um programa de jogo que inclui múltiplas funções. O programa usa muitas das instruções de controle e conceitos discutidos até agora.

Arquivo de cabeçalho C++ Standard Library	Explicação
<iostream>	Contém protótipos de função para as funções de entrada e saída padrão do C++, introduzidas no Capítulo 2 e discutidas em detalhes no Capítulo 15, “Entrada/saída de fluxo”. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>ios.h</code> .
<iomanip>	Contém protótipos de função para manipuladores de fluxo que formatam fluxos de dados. Esse arquivo de cabeçalho é inicialmente utilizado na Seção 4.9 e discutido em mais detalhes no Capítulo 15. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>iomanip.h</code> .
<cmath>	Contém protótipos de função para funções da biblioteca de matemática (discutidas na Seção 6.3). Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>math.h</code> .
<cstdlib>	Contém protótipos de função para conversões de números em texto, de texto em números, alocação de memória, números aleatórios e várias outras funções utilitárias. Partes do arquivo de cabeçalho são abrangidas na Seção 6.7; no Capítulo 11, “Sobrecarga de operadores; objetos string e array”; no Capítulo 16, “Tratamento de exceções”; no Capítulo 19, “Programação Web”; no Capítulo 22, “Bits, caracteres, strings e Ápêndice E, ‘Tópicos sobre o código C legado’. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>stdlib.h</code> .
<ctime>	Contém protótipos de função e tipos para manipular a data e a hora. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>time.h</code> . Esse arquivo de cabeçalho é utilizado na Seção 6.7.
<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>	Esses arquivos de cabeçalho contêm classes que implementam os contêineres da C++ Standard Library. Os contêineres armazenam dados durante a execução de um programa em C++ . Discutimos todos esses arquivos de cabeçalho no Capítulo 23, “Standard Template Library (STL)”.
<cctype>	Contém protótipos de função para funções que testam caracteres quanto a certas propriedades (como o fato de o caractere ser um dígito ou uma pontuação) e protótipos de função para funções que podem ser utilizadas para converter letras minúsculas em maiúsculas e vice-versa. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>cctype.h</code> . Esses tópicos são discutidos no Capítulo 8, “Ponteiros e strings baseadas em ponteiro” e no Capítulo 22, “Bits, caracteres, strings C”.
<cstring>	Contém protótipos de função para funções de processamento de string no estilo C. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>cstring.h</code> . Esse arquivo de cabeçalho é utilizado no Capítulo 11, “Sobrecarga de operadores; objetos string e array”.
<typeinfo>	Contém classes para identificação de tipo em tempo de execução (determinando tipos de dados em tempo de execução). Esse arquivo de cabeçalho é discutido na Seção 13.8.
<exception>, <stdexcept>	Esses arquivos de cabeçalho contêm classes que são utilizadas para tratamento de exceções (discutidos no Capítulo 16).
<memory>	Contém classes e funções utilizadas pela C++ Standard Library para alocar memória aos contêineres da C++ Standard Library. Esse cabeçalho é utilizado no Capítulo 16, “Tratamento de exceções”.
<fstream>	Contém protótipos de função para funções que realizam entrada de arquivos em disco e saída para arquivos em disco (discutido no Capítulo 17, “Processamento de arquivo”). Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>fstream.h</code> .
<string>	Contém a definição de classe <code>string</code> C++ Standard Library (discutida no Capítulo 18).
<sstream>	Contém protótipos de função para funções que realizam entrada de strings e saída para strings na memória (discutidos no Capítulo 18, “strings e processamento de fluxo de string”).
<functional>	Contém classes e funções utilizadas por algoritmos da C++ Standard Library. Esse arquivo de cabeçalho é utilizado no Capítulo 23, “Standard Template Library (STL)”.
<iterator>	Contém classes para acessar dados nos contêineres da C++ Standard Library. Esse arquivo de cabeçalho é utilizado no Capítulo 23, “Standard Template Library (STL)”.
<algorithm>	Contém funções para manipular dados em contêineres da C++ Standard Library. Esse arquivo de cabeçalho é utilizado no Capítulo 23.

Figura Arquivos de cabeçalho da C++ Standard Library.

(continua)

Arquivo de cabeçalho C++ Standard Library	Explicação
<cassert>	Contém macros para adicionar diagnósticos que auxiliam na depuração de programa. Isso substitui o arquivo de cabeçalho <code>assert.h</code> do C++ pré-padrão. Esse arquivo de cabeçalho é utilizado no Apêndice F, “Pré-processador”.
<cfloat>	Contém os limites de tamanho de ponto flutuante do sistema. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>float.h</code> .
<climits>	Contém os limites de tamanho inteiros do sistema. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>limits.h</code> .
<cstdio>	Contém protótipos de função para as funções de biblioteca de entrada/saída padrão no estilo C e informações utilizadas por elas. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>stdio.h</code> .
<locale>	Contém classes e funções normalmente utilizadas pelo processamento de fluxo para processar dados na forma natural para diferentes idiomas (por exemplo, formatos monetários, classificação de strings, apresentação de caractere etc.).
<limits>	Contém classes para definir os limites de tipos de dados numéricos em cada plataforma de computador.
<utility>	Contém classes e funções utilizadas por muitos arquivos de cabeçalho da C++ Standard Library.

Figura Arquivos de cabeçalho da C++ Standard Library.

(continuação)

O elemento chance pode ser introduzido em aplicativos de computador Utilizando a Standard Library. Considere a seguinte instrução:

```
i = rand();
```

A função `rand()` gera um inteiro sem sinal entre 0 e 32.767. Uma constante simbólica definida no arquivo de cabeçalho `<cstdlib>`. O valor `RAND_MAX` deve ser pelo menos 32.767 — o valor positivo máximo de um inteiro de dois bytes (16 bits). Para o GNU C++, o valor de `RAND_MAX` é 214.748.647; para o Visual Studio é 214.748.647. Reproduz verdadeiramente inteiros de modo aleatório, todo número entre 0 e `RAND_MAX` tem uma igual probabilidade de ser escolhido todos juntos.

O intervalo de valores produzido diretamente pela função `rand()` é diferente daquele que um aplicativo específico requer. Por exemplo, um programa que simula lançamento de moeda talvez requeria somente 0 para ‘caras’ e 1 para ‘coroas’. Um programa que simula a rolagem de um dado de seis lados requer inteiros aleatórios no intervalo de 1 a 6. Um programa que aleatoriamente prevê o próximo tipo de nave espacial (uma entre quatro possibilidades) que voará no horizonte de um videogame poderia exigir inteiros aleatórios no intervalo de 1 a 4.

Lançando um dado de seis faces

Para demonstrar, vamos desenvolver um programa (Figura 6.8) para simular 20 lançamentos de um dado de seis lados e imprimir o valor de cada lançamento. O protótipo de função é mostrado a seguir. Para produzir inteiros no intervalo de 0 a 5, utilizamos o operador `% módulo` como mostrado a seguir:

```
rand() % 6
```

Isso é chamado de **escalonamento**. O número 6 é chamado de **máximo de escalonamento**. Em seguida, podemos gerar intervalo de números produzidos adicionando 1 ao nosso resultado anterior. A Figura 6.8 confirma que os resultados estão no intervalo 1 a 6.

Lançando um dado de seis faces 6.000.000 vezes

Para mostrar que os números produzidos podem aparecer aproximadamente com igual probabilidade, a Figura 6.9 simula 6.000.000 lançamentos de um dado. Cada inteiro no intervalo de 1 a 6 deve aparecer cerca de 1.000.000 vezes. Isso é confirmado na Figura 6.9.

Compreendendo o uso da Standard Library, você pode simular o lançamento de um dado de seis lados escalonando e deslocando os resultados produzidos por `rand()`. Observe que o programa nunca deve gerar resultados negativos (-51) fornecido na estrutura porque a expressão de controle só tem sempre valores no intervalo 1–6; entretanto, é uma exceção ao caso de questão de boa prática. Depois de estudarmos os arrays no Capítulo 7, mostraremos na Figura 6.9 elegantemente com uma instrução de uma única linha.



Dica de prevenção de erro 6.3

Forneça um `default` em `switch` para capturar erros mesmo se você estiver absolutamente certo de que não tem nenhum bug!

```

1 // Figura 6.8: fig06_08.cpp
2 // Inteiros aleatórios deslocados e escalonados.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contém o protótipo de função para rand
11 using std::rand;
12
13 int main()
14 {
15     // itera 20 vezes
16     for ( int counter = 1; counter <= 20; counter++ )
17     {
18         // escolhe um número aleatório de 1 a 6 e o envia para saída
19         cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21         // se o contador for divisível por 5, inicia uma nova linha de saída
22         if ( counter % 5 == 0 )
23             cout << endl;
24     } // fim do for
25
26     return 0; // indica terminação bem-sucedida
27 } // fim de main

```

```

6      6      5      5      6
5      1      1      5      3
6      6      2      4      2
6      2      3      4      1

```

Figura 6.8 Inteiros deslocados e escalonados produzidos por `rand() % 6`.

```

1 // Figura 6.9: fig06_09.cpp
2 // Lança um dado de seis lados 6.000.000 vezes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contém o protótipo de função para rand
11 using std::rand;
12
13 int main()
14 {
15     int frequency1 = 0; // contagem de 1s lançados
16     int frequency2 = 0; // contagem de 2s lançados
17     int frequency3 = 0; // contagem de 3s lançados
18     int frequency4 = 0; // contagem de 4s lançados
19     int frequency5 = 0; // contagem de 5s lançados

```

Figura 6.9 Lançando um dado de seis lados 6.000.000 vezes.

(continua)

```

20 int frequency6 = 0; // contagem de 6s lançados
21
22 int face; // armazena o valor lançado mais recentemente
23
24 // resume os resultados de 6.000.000 lançamentos de um dado
25 for ( int roll = 1; roll <= 6000000; roll++ )
26 {
27     face = 1 + rand() % 6; // número aleatório de 1 a 6
28
29     // determina valor de lançamento de 1 a 6 e incrementa o contador apropriado
30     switch ( face )
31     {
32         case 1:
33             ++frequency1; // incrementa o contador de 1s
34             break;
35         case 2:
36             ++frequency2; // incrementa o contador de 2s
37             break;
38         case 3:
39             ++frequency3; // incrementa o contador de 3s
40             break;
41         case 4:
42             ++frequency4; // incrementa o contador de 4s
43             break;
44         case 5:
45             ++frequency5; // incrementa o contador de 5s
46             break;
47         case 6:
48             ++frequency6; // incrementa o contador de 6s
49             break;
50         default : // valor inválido
51     }
52     cout << "Program should never get here!";
53 } // fim do for
54
55 cout << "Face" << setw( 13 ) << "Frequency" << endl; // cabeçalhos de saída
56 cout << " 1" << setw( 13 ) << frequency1
57 << "\n 2" << setw( 13 ) << frequency2
58 << "\n 3" << setw( 13 ) << frequency3
59 << "\n 4" << setw( 13 ) << frequency4
60 << "\n 5" << setw( 13 ) << frequency5
61 << "\n 6" << setw( 13 ) << frequency6 << endl;
62 return 0; // indica terminação bem-sucedida
63 } // fim de main

```

Face	Frequency
1	999702
2	1000823
3	999378
4	998898
5	1000777
6	1000422

Figura 6 Lançando um dado de seis lados 6.000.000 vezes.

(continuação)

Aleatorizando o gerador de número aleatório

Executar o programa da Figura 6.8 novamente produz

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

Note que o programa imprime exatamente a mesma seqüência de valores mostrada na Figura 6.8. Como esses números são aleatórios? Ironicamente, essa repetitividade é uma característica desejada. Independentemente do programa de simulação, essa repetitividade é essencial para comprovar que as correções no programa funcionam adequadamente.

A função `rand` na realidade gera os pseudo-aleatórios ou pseudo-randômicos. Chamando repetidamente produz uma seqüência de números que parece ser aleatória. Entretanto, a seqüência repete-se toda vez que o programa executa. Uma vez que o programa foi completamente depurado, ele pode ser condicionado a produzir uma seqüência diferente de números aleatórios | execução. Isso chama-se **semente** ou **randomização** e é realizado com a função `rand` da Standard Library. A função `rand` aceita um argumento do tipo `int` para fornecer a função para produzir uma seqüência diferente de números aleatórios para cada execução do programa.

A Figura 6.10 demonstra a função `rand`. O programa utiliza o tipo de dado `int`, que é a abreviação de `int`. Um `int` é armazenado pelo menos em dois bytes de memória (em geral, quatro bytes de memória nos sistemas de 32 bits populares) e pode ter valores positivo e negativo. Uma variável `time_t` também é armazenada em pelo menos dois bytes de memória. Um `unsigned int` de dois bytes pode ter apenas valores não negativos no intervalo 0–65 535. Um `int` de dois bytes pode ter somente valores não negativos no intervalo 0–42 949 672 95. A função `rand` usa `time_t` como um argumento. O protótipo de função para `rand` está no arquivo de cabeçalho `<time.h>`.

Vamos executar o programa várias vezes e observar os resultados. Note que diferentes sementes produzem uma seqüência aleatória toda vez que ele executa, desde que o usuário insira uma semente diferente. Utilizamos a mesma semente nas três saídas de exemplo, então a mesma série de 10 números é exibida em cada uma dessas saídas.

Para aleatorizar, ou randomizar, sem inserir uma semente toda vez, podemos utilizar uma instrução como

```
 srand( time( 0 ) );
```

Isso faz com que o computador leia o relógio dele para obter o valor `time_t` para a semente. A semente é escrita na instrução precedente) retorna a hora atual como o `time_t` (que é o segundo de 1970 à noite do Greenwich

O protótipo de `rand` é:

 **Erro comum de programação 6.7**

Chamar a função `rand` mais de uma vez em um programa reinicia a seqüência de números pseudo-aleatórios e pode afetar a aleatoriedade dos números produzidos por

Escalonamento e deslocamento generalizados de números aleatórios

Anteriormente, demonstramos como escrever uma única instrução para simular a rolagem de um dado de seis lados com a instrução:

```
 face = 1 + rand() % 6;
```

que sempre atribui um inteiro (aleatoriamente) entre 1 e 6. Observe que a largura desse intervalo (isto é, o número de inteiros consecutivos no intervalo) é 6, e o número inicial no intervalo é 1. Examinando a instrução precedente, veja que a largura do intervalo é determinada pelo número utilizado para o módulo (isto é, 6), e o número inicial do intervalo é igual ao número (isto é, 1) que é adicionado. Pode-se generalizar esse resultado como

```
 número = valorDeDeslocamento + rand() % fatorDeEscala;
```

onde `valorDeDeslocamento` é igual ao primeiro número no intervalo desejado de inteiros consecutivos e `fatorDeEscala` é a largura

de intervalos desejados de inteiros no intervalo desejado. Os inteiros consecutivos que é possível escolher aleatoriamente a partir

 **Erro comum de programação 6.8**

Utilizar `rand` no lugar de `rand` para tentar gerar números aleatórios é um erro de compilação ou recompilação devido ao valor.

```

1 // Figura 6.10: fig06_10.cpp
2 // Aleatorizando o programa de lançamento de dados.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstdlib> // contém protótipos para funções srand e rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17     unsigned seed; // armazena a semente inserida pelo usuário
18
19     cout << "Enter seed: ";
20     cin >> seed;
21     srand( seed ); // semeia o gerador de número aleatório
22
23     // itera 10 vezes
24     for ( int counter = 1; counter <= 10; counter++ )
25     {
26         // escolhe um número aleatório de 1 a 6 e o envia para saída
27         cout << setw( 10 ) << ( 1 + rand() % 6 );
28
29         // se o contador for divisível por 5, inicia uma nova linha de saída
30         if ( counter % 5 == 0 )
31             cout << endl;
32     } // fim do for
33
34     return 0; // indica terminação bem-sucedida
35 } // fim de main

```

Enter seed: 67
6 1 4 6 2
1 6 1 6 4

Enter seed: 432
4 6 3 1 6
3 1 5 4 2

Enter seed: 67
6 1 4 6 2
1 6 1 6 4

Figura 6.10 Aleatorizando o programa de lançamento de dados.

6.8 Estudo de caso: jogo de azar e introdução a enum

Um dos jogos de azar mais populares é um jogo de dados com jogadores em cassinos e becos por todo o mundo. As regras do jogo são simples e diretas:

Um jogador rola dois dados. Cada dado tem seis faces. Essas faces contêm 1, 2, 3, 4, 5 e 6 pontos. Depois que os dados param rolar, a soma dos pontos nas faces viradas para cima é calculada. Se a soma é 7 ou 11 na primeira rolagem dos dados, o jogador ganha. Se a soma é 2, 3 ou 12 na primeira rolagem o jogador perde (isto é, a 'casa' ganha). Se a soma

for 4, 5, 6, 8, 9 ou 10 na primeira rolagem dos dados, essa soma torna-se a ‘pontuação’ do jogador. Para ganhar, você deve continuar a lançar o dado até ‘fazer sua pontuação’. O jogador perde se obtiver um 7 antes de fazer sua pontuação.

O programa na Figura 6.11 simula ojogo de dados

Nas regras do jogo, note que o jogador deve lançar dois dados na primeira e em todas as rolagens subsequentes. Definimos rollDice (linhas 71–83) para lançar o dado e calcular e imprimir a soma dos dados. No entanto, a função só é chamada de dois lugares (linhas 27 e 51) no programa. Quando somos fornecidos com argumentos, então indicamos uma lista de parâmetros vazia no protótipo (linha 14) e no cabeçalho de função (linha 19). A função retorna a soma dos dois dados, então o tipo de retorno é indicado no protótipo de função e no cabeçalho de função.

O jogo é razoavelmente complexo. O jogador pode ganhar ou perder no primeiro ou em qualquer lançamento subsequente. O programa utiliza a variável status para monitorar isso.

A variável gameStatus é declarada como do tipo int (linha 19) declara um tipo definido pelo usuário chamado enumStatus. Uma enumeração, introduzida pela palavra-chave enum de tipo (nesse caso), é um conjunto de constantes do tipo inteiro representadas por identificadores. Os valores inteiros devem ser menores que

especificado de outro modo, e incrementam por 1.

Na enumeração anterior, existem 3 constantes. A constante LOST tem o valor 0, a constante WON tem o valor 2. Os identificadores em uma enumeração devem ser únicos, mas as constantes enumeradas separadas podem ter o mesmo valor inteiro (em breve mostraremos como isso).

```

1 // Figura 6.11: fig06_11.cpp
2 // Simulação do jogo de dados craps.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // contém protótipos para funções srand e rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // contém protótipo para a função time
12 using std::time;
13
14 int rollDice(); // lança o dado, calcula e exibe a soma
15
16 int main()
17 {
18     // enumeração com constantes que representam o status do jogo
19     enumStatus { CONTINUE, WON, LOST }; // todas as maiúsculas em constantes
20
21     int myPoint; // pontos se não ganhar ou perder na primeira rolagem
22     Status gameStatus; // pode conter CONTINUE, WON ou LOST
23
24     // torna aleatório o gerador de número aleatório utilizando a hora atual
25     srand( time( 0 ) );
26
27     int sumOfDice = rollDice(); // primeira rolagem dos dados
28
29     // determina status e pontuação do jogo (se necessário) com base no primeiro lançamento de
30     // dados
31     switch ( sumOfDice )
32     {
33         case 7: // ganha com 7 no primeiro lançamento
34         case 11: // ganha com 11 no primeiro lançamento
35             gameStatus = WON;
36             break;
37     }
38
39     if ( gameStatus == CONTINUE )
40     {
41         cout << "Sua pontuação é " << sumOfDice << endl;
42
43         myPoint = sumOfDice;
44
45         while ( gameStatus == CONTINUE )
46         {
47             cout << "Lançando o dado..." << endl;
48             cout << endl;
49
50             int roll = rollDice();
51             cout << "O resultado é " << roll << endl;
52
53             if ( roll == 7 )
54                 gameStatus = LOST;
55             else if ( roll == myPoint )
56                 gameStatus = WON;
57             else
58                 gameStatus = CONTINUE;
59
60             cout << endl;
61         }
62
63         cout << "O resultado final é " << gameStatus << endl;
64     }
65     else
66     {
67         cout << "O resultado final é " << gameStatus << endl;
68     }
69 }
```

Figura 6.11 Simulação do jogo de dados

(continua)

```

36     case 2: // perde com 2 no primeiro lançamento
37     case 3: // perde com 3 no primeiro lançamento
38     case 12 // perde com 12 no primeiro lançamento
39     gameStatus = LOST
40         break;
41     default : // não ganhou nem perdeu, portanto registra a pontuação
42     gameStatus = CONTINUE// jogo não terminou
43     myPoint = sumOfDice; // informa a pontuação
44     cout << "Point is " << myPoint << endl;
45         break; // opcional no final do switch
46 } // fim de switch
47
48 // enquanto o jogo não estiver completo
49 while ( gameStatus == CONTINUE// nem WON nem LOST
{
50
51     sumOfDice = rollDice(); // lança os dados novamente
52
53     // determina o status do jogo
54     if ( sumOfDice == myPoint ) // vitória por pontuação
55     gameStatus = WON
56     else
57         if ( sumOfDice == 7 ) // perde obtendo 7 antes de atingir a pontuação
58     gameStatus = LOST
59 } // fim do while
60
61 // exibe uma mensagem ganhou ou perdeu
62 if ( gameStatus == WON
63     cout << "Player wins" << endl;
64 else
65     cout << "Player loses" << endl;
66
67 return 0; // indica terminação bem-sucedida
68 } // fim de main
69
70 // lança os dados, calcula a soma e exibe os resultados
71 int rollDice()
72 {
73     // seleciona valores aleatórios do dado
74     int die1 = 1 + rand() % 6; // primeiro lançamento do dado
75     int die2 = 1 + rand() % 6; // segundo lançamento do dado
76
77     int sum = die1 + die2; // calcula a soma de valores do dado
78
79     // exibe os resultados desse lançamento
80     cout << "Player rolled " << die1 << " + " << die2
81     << " = " << sum << endl;
82     return sum; // fim da função rollDice
83 } // fim da função rollDice

```

Player rolled 2 + 5 = 7
Player wins

Player rolled 6 + 6 = 12
Player loses

Figura 6.11 Simulação do jogo de dados.

(continua)

```

Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6

Player rolled 2 + 3 = 50
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses

```

Figura 6.11 Simulação do jogo de dados

(continuação)



Boa prática de programação 6.1

Torne maiúscula a primeira letra de um identificador utilizado como um nome de tipo definido pelo usuário.



Boa prática de programação 6.2

Utilize somente letras maiúsculas nos nomes das constantes enumeradas. Isso faz com que essas constantes sejam destacadas em programa e lembra o programador de que constantes enumeradas não são variáveis.

As variáveis do tipo definido `pelos sujeitos` receber somente um dos três valores declarados na enumeração. Quando o `jogo é ganho`, o programa configura `gameStatus` para `vivo` (linhas 34 e 55). Quando o `jogo é perdido`, o programa configura `a variável gameStatus` para `morto` (linhas 39 e 58). Caso contrário, o programa configura `gameStatus` para `vivo` (linha 42) para indicar que os dados devem ser rolados novamente.

Outra enumeração popular é

```
enum Months { JAN= 1, FEB= 2, MAR= 3, APR= 4, MAY= 5, JUN= 6, JUL= 7, AUG= 8, SEP= 9, OCT= 10, NOV= 11, DEC= 12 }
```

que cria o tipo definido `pelos sujeitos` constantes enumeradas que representam os meses do ano. O primeiro valor na enumeração anterior é explicitamente configurado como os valores restantes incrementais para os valores a12. Qualquer enumeração constante pode receber um valor inteiro na definição de enumeração, e cada uma das constantes subsequentes tem um valor 1 mais alto que a constante anterior na lista até a próxima configuração explícita.

Depois do primeiro lançamento, se o `jogo é ganho ou perdido`, o programa `pula as linhas 43 a 59` para a instrução `if` (linhas 62–65), o que imprime “

```
Se gameStatus for igual VIVO Player wins” Se gameStatus for igual MORTO
```

Depois do primeiro lançamento, se o `jogo não acabar`, o programa `pula as linhas 43 a 59` para a instrução `while`, porque `gameStatus é igual VIVO`. Durante cada iteração da `instrução d` o programa chama `rollDice` para produzir uma nova `Se` correspondente a `point` o programa configura `gameStatus` como `O` (linha 55), o teste de falha, a

`instrução if` else imprime “Player wins” e a execução termina. Se `gameStatus` como `O` (linha 58), o teste de falha, a `instrução if` imprime “Player loses” e a execução termina.

Observe a interessante utilização dos vários mecanismos de controle de programa que já discutimos. O programa do jogo `crap` utiliza duas funções `rollDice` — e as instruções `while`, `if ..else, if ..else` aninhadas aninhadas. Nos exercícios, investigamos várias características interessantes da execução do jogo de dados.



Boa prática de programação 6.3

Utilizar enumerações em vez de constantes do tipo inteiro pode tornar os programas mais claros e mais sustentáveis. Você pode configurar o valor de uma constante enumerada uma vez na declaração da enumeração.



Erro comum de programação 6.9

Atribuir o equivalente inteiro de uma constante enumerada a uma variável do tipo enumerado é um erro de compilação.



Erro comum de programação 6.10

Depois que uma constante enumerada tiver sido definida, tentar atribuir outro valor à constante enumerada é um erro de compilação.

6.9 Classes de armazenamento

Os programas que você viu até agora utilizam identificadores para nomes de variáveis. Os atributos de variáveis incluem nome, tamanho e valor. Este capítulo também usa identificadores como nomes para funções definidas pelo usuário. De fato, todo identificador em um programa tem outros atributos, incluindo nome, escopo e linkagem.

O C++ fornece especificadores de classe de armazenamento `register`, `extern`, `mutable` e `static`. Esta seção discute os especificadores de classe de armazenamento `static`. O especificador de classe de armazenamento (discutido em detalhes no Capítulo 24) é utilizado exclusivamente com classes.

Classe de armazenamento, escopo e linkagem

A classe de armazenamento de um identificador determina o período em que esse identificador existe na memória. Alguns identificadores existem brevemente, algum são criados repetidamente e destruídos, e outros existem por toda a execução de um programa. E discute duas classes de armazenamento automática.

O escopo de um identificador é onde o identificador pode ser referenciado em um programa. Alguns identificadores podem ser referenciados por todo um programa; outros podem ser referenciados somente a partir de áreas limitadas de um programa. A Se discute o escopo dos identificadores.

A linkagem de um identificador determina se um identificador só é conhecido no arquivo de fonte onde ele é declarado ou em múltiplos arquivos que são compilados e, então, linkados. O especificador de classe de armazenamento de um identificador é determinar sua classe de armazenamento e linkagem.

Categorias de classe de armazenamento

Os especificadores de classe de armazenamento podem ser divididos em duas classes de armazenamento: classe de armazenamento automática e classe de armazenamento estático. As palavras-chave `auto` e `static` são usadas para declarar as variáveis da classe de armazenamento automática. As variáveis são criadas quando o bloco em que elas são declaradas é executado.

Variáveis locais

Somente variáveis locais de uma função podem ser de classe de armazenamento automática. As variáveis locais e parâmetros de função normalmente são da classe de armazenamento automática. O especificador de classe de armazenamento `auto` indica que variáveis locais da classe de armazenamento automática — elas só existem no par de chaves de fechamento mais próximo do corpo da função em que a definição aparece:

```
auto double x, y;
```

As variáveis locais são da classe de armazenamento automática por padrão, menos quando é utilizada. Para o restante do texto, iremos nos referir as variáveis de classe de armazenamento automática simplesmente como variáveis `auto`.



Dica de desempenho 6.1

O armazenamento automático é um meio de economizar memória, porque as variáveis de classe de armazenamento automática existem na memória quando o bloco em que são definidas estiver executando.



Observação de engenharia do software 6.8

O armazenamento automático é um exemplo de privilégio que é fundamental para a boa engenharia de software.

No contexto de um aplicativo, o princípio declara que deve ser concedida ao código somente a quantidade de privilégio e acesso de que ele precisa para realizar sua tarefa designada, não mais que isso. Por que devemos ter variáveis armazenadas na memória e acessíveis quando não são necessárias?

Variáveis de registro

Os dados na versão de linguagem de máquina de um programa normalmente são carregados em registros para cálculos e operações.



Dica de desempenho 6.2

O especificador de classe de armazenamento ser colocado antes de uma declaração de variável automática para sugerir que o compilador mantém a variável em um dos registros de hardware de alta velocidade do computador em vez de na memória. Se variáveis intensamente utilizadas como contadores ou totais são mantidas em registros de hardware, elimina-se o overhead de carregar repetidamente as variáveis de memória nos registros e armazenar os resultados de volta na memória.



Erro comum de programação 6.11

Utilizar múltiplos especificadores de classe de armazenamento para um identificador é um erro de sintaxe. Somente um especificador de classe de armazenamento pode ser aplicado a um identificador. `register, int, não é possível combinar auto.`

O compilador talvez ignore declarações de exemplo, talvez não haja um número suficiente de registros disponíveis para o compilador ou talvez a seguinte definição de variável não seja colocada em um dos registros do computador;

```
register int counter = 1;
```

A palavra-chave `register` só pode ser utilizada com variáveis locais e parâmetros de função.



Dica de desempenho 6.3

Freqüentemente, a palavra-chave `register` é desnecessária. Os atuais compiladores de otimização são capazes de reconhecer freqüentemente variáveis utilizadas e podem decidir colocá-las em registros `sem precisar de uma declaração` mador.

Classe de armazenamento estática

As palavras-chave `static` declaram identificadores para variáveis da classe de armazenamento estática e para funções. As variáveis de classe de armazenamento estática existem a partir do ponto em que o programa começa a execução e duram todo o programa. O armazenamento de uma variável de uma classe de armazenamento estático é alocado quando o programa é executado. Essa variável é inicializada uma vez, quando sua declaração é encontrada. Para funções, o nome da função existe desde o momento em que o programa começa a executar, assim como ocorre para todas as outras funções. Entretanto, mesmo que as funções existam desde o início da execução do programa, isso não significa que esses identificadores podem ser portados para o programa. A classe de armazenamento e escopo (onde um nome pode ser utilizado) são questões separadas, como visto na Seção 6.10.

Identificadores com classe de armazenamento estática

Há dois tipos de identificadores com classe de armazenamento estática — os identificadores externos (como os nomes de função globais) e as variáveis locais declaradas com o especificador de classe de armazenamento `static`. As variáveis globais são criadas colocando-se as declarações de variável fora de qualquer classe ou definição de função. As variáveis globais recebem valores por toda a execução do programa. Variáveis globais e funções globais podem ser referenciadas por qualquer função que suas declarações ou definições no arquivo-fonte.



Observação de engenharia de software 6.9

Declarar uma variável como global em vez de declará-la como local permite que ocorram efeitos colaterais indesejáveis quando uma função que não precisa de acesso à variável a modifica acidental ou maliciosamente. Esse é outro exemplo do princípio do menor privilégio. Em geral, exceto por recursos verdadeiramente globais, o uso de variáveis globais deve ser evitado a não ser em certas situações com requisitos de desempenho únicos.



Observação de engenharia de software 6.10

As variáveis utilizadas apenas em uma função particular devem ser declaradas como variáveis locais nessa função, em vez de declaradas como variáveis globais.

As variáveis locais declaradas com a palavra-chave `static` são conhecidas apenas na função em que são declaradas, mas, ao contrário das variáveis automáticas, as variáveis locais recebem valores quando a função retorna para seu chamador. A próxima vez que a função é chamada, as variáveis locais recebem valores que tinham quando a função completou pela última vez a execução. A instrução seguinte declara a variável local `count` inicializada como 1:

```
static int count = 1;
```

Todas as variáveis numéricas da classe de armazenamento estática são inicializadas como zero se não forem explicitamente declaradas pelo programador, mas, de qualquer maneira, é uma boa prática inicializar explicitamente todas as variáveis.

Os especificadores de classe de ~~armazenamento~~ significado especial quando aplicados explicitamente a identificadores externos como variáveis globais e nomes de função globais. No Apêndice E, “Tópicos sobre o código C legado”, discute-se como utilizar `extern static` com identificadores externos e programas de múltiplos arquivos-fonte.

6.10 Regras de escopo

A parte do programa em que um identificador pode ser utilizado é conhecida como escopo. Por exemplo, quando declaramos uma variável local em um bloco, ela pode ser referenciada apenas nesse bloco e nos blocos aninhados dentro desse bloco. Esta seção aborda quatro escopos para um identificador: escopo de função, escopo de bloco, escopo de protótipo de função e escopo de namespace (Capítulo 9).

Mais adiante examinaremos dois outros escopos (Capítulo 9) e o escopo de namespaces (Capítulo 24).

Um identificador declarado fora de qualquer função ou classe tem escopo de arquivo. Tal identificador é ‘conhecido’ em toda a função de definição de ponta à função até o final do arquivo. Todas as variáveis globais, definições de função e protótipos de função declarados dentro de um bloco têm escopo de bloco.

~~Oscópios~~ (identificadores seguidos por dois pontos) são os únicos identificadores com escopo de função. Os rótulos podem ser utilizados em qualquer lugar na função em que aparecem, mas não podem ser referenciados fora do corpo da função. Rótulos são utilizados em `goto` (Apêndice E). Os rótulos são detalhes de implementação que as funções ocultamumas das outras.

Os identificadores declarados dentro de um bloco têm escopo de bloco. O escopo de bloco começa na declaração do identificador e termina na chave de fechamento `} do bloco` em que o identificador é declarado. As variáveis locais têm escopo de bloco, assim como os parâmetros de função, que também são variáveis locais da função. Todo bloco pode conter declarações de variável. Quando blocos são aninhados e um identificador em um bloco externo tem o mesmo nome de um identificador em um bloco interno, o identificador no bloco externo fica ‘oculto’ até que o bloco interno termine. Durante a execução no bloco interno, este vê o valor de seu identificador local e não o valor do identificador chamado identicamente no bloco que engloba. As variáveis locais declaradas ainda têm escopo de bloco, mesmo que existam a partir do momento em que o programa comece a executar. A duração do aninhamento não afeta o escopo de um identificador.

Os únicos identificadores com escopo de protótipo de função são aqueles utilizados na lista de parâmetros de um protótipo de função. Como mencionado anteriormente, os protótipos de função não exigem nomes na lista de parâmetros — apenas os tipos são necessários. Os nomes que aparecem na lista de parâmetros de um protótipo de função são ignorados pelo compilador. Os identificadores usados em um protótipo de função podem ser reutilizados em outra parte no programa sem ambigüidade. Em um único protótipo, um identificador particular pode ser utilizado apenas uma vez.



Erro comum de programação 6.12

Normalmente é um erro de lógica utilizar accidentalmente o mesmo nome de um identificador em um bloco interno que é utilizado para um identificador em um bloco externo, quando de fato o programador quer que o identificador no bloco externo esteja ativo até o fim do bloco interno.



Boa prática de programação 6.4

Evite nomes de variáveis que ocultam nomes em escopos externos. Isso pode ser realizado evitando-se o uso de identificadores duplicados em um programa.

O programa da Figura 6.12 demonstra questões de escopo com variáveis globais, variáveis locais automáticas e variáveis `static`.

A linha 11 declara e inicializa a variável global `glo`. Esta variável global é ocultada em todo bloco (ou função) que declara uma variável chamada `glo`. A linha 15 declara uma variável local `glo` e inicializa como 5. A linha 17 gera saída dessa variável para mostrar que a variável global está oculta. Em seguida, as linhas 19–23 definem um novo bloco e uma variável local `glo`, que é inicializada como 7 (linha 20). A linha 22 gera saída dessa variável para mostrar que a variável global foi destruída. Em seguida, a linha 25 gera saída da variável local no bloco externo para mostrar que ela não está mais oculta.

Para demonstrar outros escopos, o programa define três funções, que não aceitam argumentos e não retornam nada. A função `Local` (linhas 39–46) declara a variável `local` e inicializa como 25. Quando o programa é executado, a função `Local` imprime a variável, incrementa essa variável e a imprime novamente antes de a função retornar o controle de programa ao seu chamar. Toda vez que o programa chamar essa função, ela recriará a variável automaticamente como 25.

A função `StaticLocal` (linhas 51–60) declara a variável `local` e inicializa como 50. As variáveis locais declaradas como `static` retêm seus valores mesmo quando estão fora de escopo (isto é, a função em que são declaradas não está em execução) e o programa chama `StaticLocal`, a função imprime imediatamente o valor da variável. A função `StaticLocal` imprime novamente o valor da variável antes de a função devolver o controle do programa ao seu chamar. Na próxima chamada a essa função, a variável `local` é inicializada automaticamente como 51. A inicialização na linha 53 só ocorre uma vez — na primeira vez em que a função é chamada.

```

1 // Figura 6.12: fig06_12.cpp
2 // Um exemplo de escopo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void useLocal( void ); // protótipo de função
8 void useStaticLocal( void ); // protótipo de função
9 void useGlobal( void ); // protótipo de função
10
11 int x = 1; // variável global
12
13 int main()
14 {
15     int x = 5; // variável local para main
16
17     cout << "local x in main's outer scope is " << x << endl;
18
19     { // inicia novo escopo
20         int x = 7; // oculta x no escopo externo
21
22         cout << "local x in main's inner scope is " << x << endl;
23     } // fim do novo escopo
24
25     cout << "local x in main's outer scope is " << x << endl;
26
27     useLocal(); // useLocal tem uma variável local x
28     useStaticLocal(); // useStaticLocal tem x estático local
29     useGlobal(); // useGlobal utiliza x global
30     useLocal(); // useLocal reinicializa seu x local
31     useStaticLocal(); // x estático local retém seu valor anterior
32     useGlobal(); // x global também retém seu valor
33
34     cout << "\nlocal x in main is " << x << endl;
35     return 0; // indica terminação bem-sucedida
36 } // fim de main
37
38 // useLocal reinicializa a variável local x durante cada chamada
39 void useLocal( void )
40 {
41     int x = 25; // inicializada toda vez que useLocal é chamada
42
43     cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44     x++;
45     cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // fim da função useLocal
47
48 // useStaticLocal inicializa a variável estática local x somente
49 // na primeira vez em que a função é chamada; o valor de x é salvo
50 // entre as chamadas a essa função
51
52 {
53     static int x = 50; // inicializada na primeira vez em que useStaticLocal é chamada
54
55     cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56     << endl;
57     x++;

```

Exemplo de escopo.

(continua)

```

58     cout << "local static x is "    << x << " on exiting useStaticLocal"
59         << endl;
60 } // fim da função useStaticLocal
61
62 // useGlobal modifica a variável global x durante cada chamada
63 void useGlobal( void )
64 {
65     cout << "\nglobal x is "   << x << " on entering useGlobal" << endl;
66     x *= 10;
67     cout << "global x is "   << x << " on exiting useGlobal" << endl;
68 } // fim da função useGlobal

```

local x in main's outer scope is 5
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5

Exemplo de escopo.

(continuação)

A função `useGlobal` (linhas 63–68) não declara nenhuma variável. Portanto, ~~após o uso da variável global é feita uma nova chamada da função que imprime a variável global para-a por 10 e a imprime novamente antes de a função devolver o controle do programa ao seu chamador. Na próxima vez que o programa chama a função `useGlobal`, a variável global tem seu valor modificado, 10. Depois de executar as duas chamadas de função, o programa imprime a variável local novamente para mostrar que nenhuma das chamadas de função modificou o valor de `x` em `main` porque todas as funções referenciavam as variáveis em outros escopos.~~

6.11 Pilha de chamadas de função e registros de ativação

~~Parte de dentro da pilha de chamadas de função é a pilha de registros de ativação que está ligada ao escopo da função. A pilha de registros de ativação é, assim, a pilha de escopos. Quando um escopo é criado, ele é colocado na parte inferior da pilha de escopos. Quando um escopo é destruído, ele é removido da parte inferior da pilha de escopos. Quando um escopo é criado, ele é colocado na parte superior da pilha de escopos. Quando um escopo é destruído, ele é removido da parte superior da pilha de escopos. As pilhas são conhecidas como estruturas de dados do tipo LIFO — o último item inserido na pilha é o primeiro item que é removido da pilha.~~

Um dos mecanismos mais importantes para os alunos de ciência da computação entenderem é o da pilha de chamadas de função. Essa estrutura de dados — que funciona ‘nos bastidores’ — suporta o mecanismo de chamada/retorno de função. Ela suporta também a criação, manutenção e destruição de variáveis automáticas de uma função chamada. Explicamos o comportamento de pilhas últimamente, no tópico [Pilhas](#). Vamos usar esse exemplo de

empilhamento de pratos. Como veremos nas figuras 6.14–6.16, esse comportamento LIFO é exatamente o que uma função faz à função que a chamou.

A medida que cada função é chamada, ela pode, por sua vez, chamar outras funções que, por sua vez, podem chamar outra — tudo antes mesmo de qualquer uma das funções retornar. Por fim, cada função deve devolver o controle à função que ela chamou. Portanto, de certo modo, devemos monitorar os endereços de retorno de que cada função precisa para retornar o controle à função que a chamou. A pilha de chamadas de função é a estrutura de dados perfeita para o tratamento dessas informações. Toda vez que uma função chama outra função, uma entrada é empurrada sobre a pilha. Essa entrada, chamada de quadro de pilha, contém o endereço de retorno de que a função chamada precisa para retornar à função chamadora. Ela também contém algumas informações adicionais que logo discutiremos. Se a função chamada retorna, em vez de chamar outra função, antes de retornar, o quadro de pilha para a chamada de função é removido e o controle é transferido para o endereço de retorno no quadro de pilha removido.

O apelo da pilha de chamadas é que cada função chamada sempre localiza as informações de que precisa para retornar ao mador na parte superior da pilha de chamadas. E, se uma função faz uma chamada para outra função, um quadro de pilha para a chamada de função é simplesmente inserido na pilha de chamadas. Portanto, o endereço de retorno requerido pela função recei-

paras retidas desse pilha de chamadas permanece na parte superior da pilha. As funções tem variáveis automáticas — parâmetros quer variáveis locais que a função declarar. As variáveis automáticas precisam existir enquanto uma função estiver em execução. Precisam permanecer ativas se a função fizer chamadas para outras funções. Mas quando uma função chamada retorna, suas variáveis automáticas da função chamada precisam 'desaparecer'. O quadro de pilha da função chamada é um lugar perfeito para a memória para as variáveis automáticas da função chamada. Esse quadro de pilha existe enquanto a função chamada estiver. Quando a função chamada retornar — e não precisar mais de suas variáveis automáticas locais —, seu quadro de pilha é removido e suas variáveis automáticas locais não são mais conhecidas pelo programa.

Naturalmente, a quantidade de memória em um computador é finita, portanto somente certa quantidade de memória pode ser usada para armazenar os registros de ativação na pilha de chamadas de função. Se houver mais chamadas de função do que as que os seus registros de ativação armazenados na pilha de chamadas de função, ocorre um erro conhecido como

Pilha de chamadas de função em ação

Portanto, como vimos, a pilha de chamadas e registros de ativação suportam o mecanismo de chamada/retorno de função e a destruição de variáveis automáticas. Agora consideremos a maneira como a pilha de chamadas suporta a operação de uma função chamada por (linhas 11–17 da Figura 6.13). Primeiro o sistema operaional abre um registro de ativação na

```

1 // Figura 6.13: fig06_13.cpp
2 // Função square utilizada para demonstrar a pilha
3 // de chamadas de função e os registros de ativação.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 int square( int ); // protótipo para a função square
10
11 int main()
12 {
13     int a = 10; // valor para square (variável automática local em main)
14
15     cout << a << " squared: " << square( a ) << endl; // exibe o quadrado de um int
16     return 0; // indica terminação bem-sucedida
17 } // fim de main
18
19 // retorna o quadrado de um inteiro
20 int square( int x ) // x é uma variável local
21 {
22     return x * x; // calcula square e retorna o resultado
23 } // fim da função square

```

10 squared: 100

Figura 6.13 A função square utilizada para demonstrar a pilha de chamadas de função e os registros de ativação.

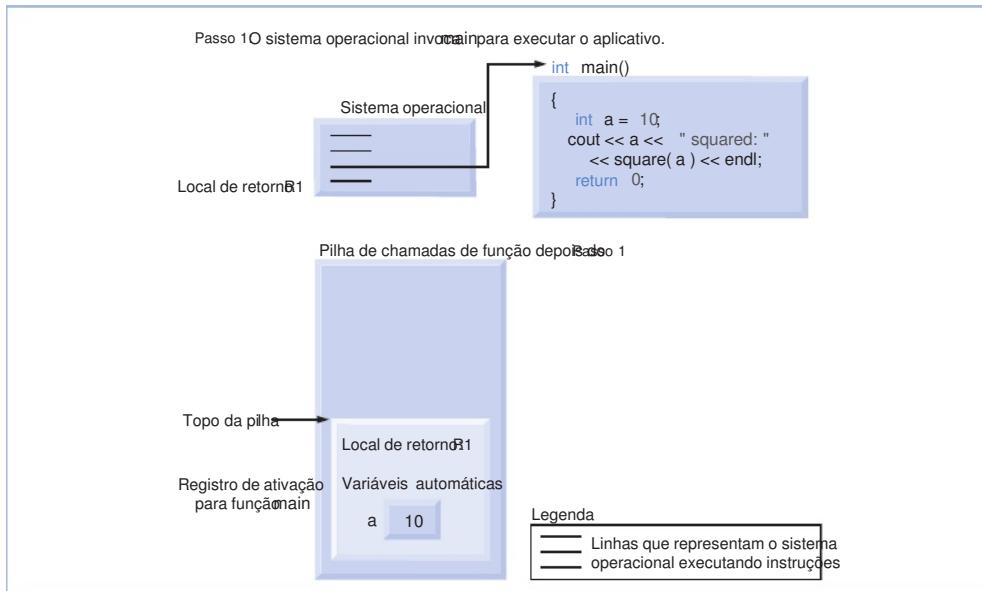


Figura 6.14 A pilha de chamadas de função depois que o sistema operacional invoca main() para executar o aplicativo.

pilha (mostrado na Figura 6.14). O registro de ativação para retornar ao sistema operacional (isto é, transferir para retornar o endereço de retorno) contém o espaço para variáveis locais estáticas que é inicializada como

A função `main` — antes de retornar ao sistema operacional — ~~apaga a pilha~~ (Figura 6.13). Isso faz com que um quadro de pilha (linhas 20–23) seja inserido na pilha de chamadas de função (Figura 6.15). Esse quadro de pilha contém o endereço de retorno preciso para retornar (isto é), e a memória para a variável automática de `square` (isto é).

Depois de calcular o quadrado de ~~sete variáveis~~ (isto é, torna a e não precisa mais da memória para sua variável automática). Portanto, o quadro de pilha da função é ~~removido~~ (isto é, realização de ~~retorno~~) e perde sua variável automática. A Figura 6.16 mostra a pilha de chamadas de função depois de o registro de ativação de ~~sete variáveis~~ (isto é) ter sido removido.

A função `main` agora exibe o resultado da chamada (Figura 15) e então executa a instrução `return` (Figura 16). Isso faz com que o registro de ativação ~~sete variáveis~~ (isto é, removido da pilha). Isso fornece o endereço ~~retorno~~ para o sistema operacional (isto é, na Figura 6.14) e faz com que a memória da variável ~~sete variáveis~~ (isto é, torna-se indisponível).

Você viu agora o quanto é valiosa a noção da estrutura de dados de pilha ao implementar um mecanismo-chave que suporta a do programa. As estruturas de dados têm muitas aplicações importantes na ciência da computação. Discutimos as pilhas, filas árvores e outras estruturas de dados no Capítulo 21, “Estruturas de dados”, e no Capítulo 23, “Standard Template Library (STL)

6.12 Funções com listas de parâmetro vazias

Em C++, uma lista de parâmetros vazia é especificada ~~simplesmente~~ entre parênteses. O protótipo

`void print();` especifica que a função aceita argumentos e não retorna um valor. A Figura 6.17 demonstra as maneiras de declarar e utilizar funções com listas de parâmetros vazias.



Dica de portabilidade 6.2

O significado de uma lista de parâmetros de função vazia em C++ é significativamente diferente daquele em C. Em C, significa que toda verificação de argumento está desativada (isto é, a chamada de função pode passar todos os argumentos que ela quiser). Em C++, significa que a função não aceita argumentos explicitamente. Portanto, os programas C que utilizam esse recurso podem causar erros de compilação quando compilados em C++.

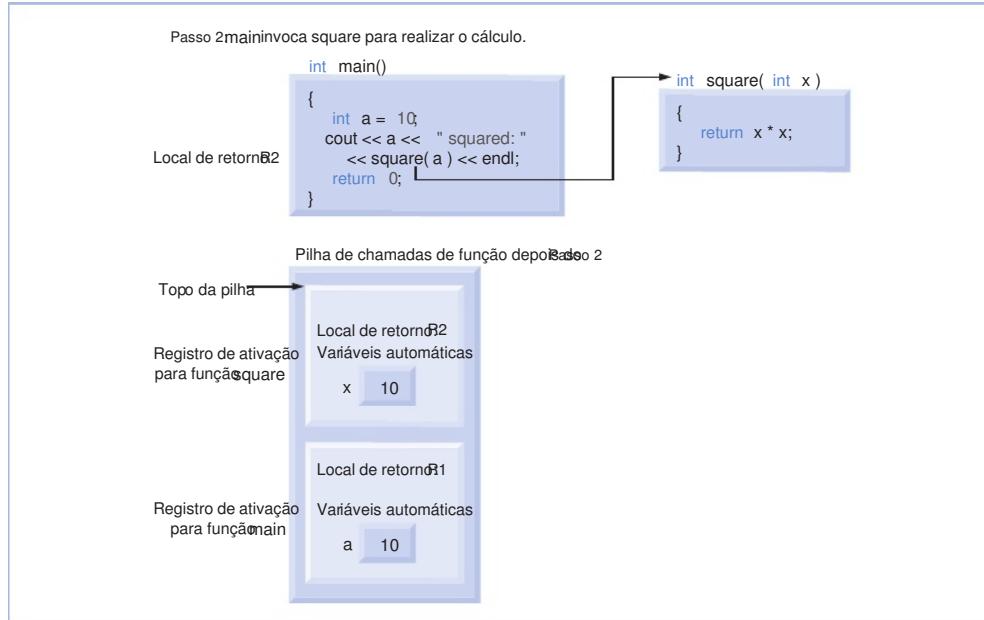


Figura 6.15 Pilha de chamadas de função depois de main invocar a função square para realizar o cálculo.

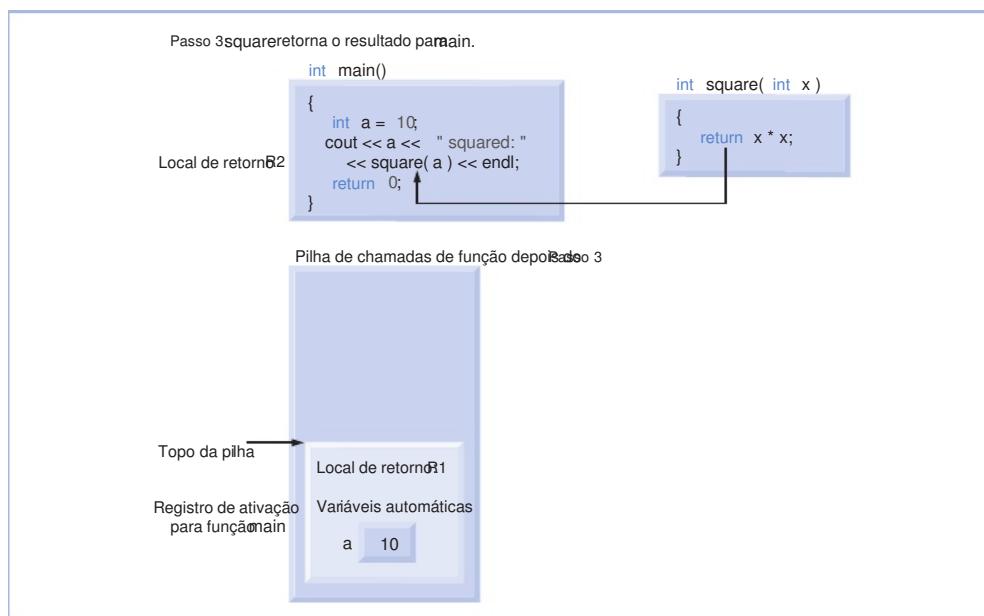


Figura 6.16 Pilha de chamadas de função depois de a função square retornar para main

```

1 // Figura 6.17: fig06_17.cpp
2 // Funções que não aceitam argumentos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void function1(); // função que não aceita argumentos
8 void function2( void ); // função que não aceita argumentos
9
10 int main()
11 {
12     function1(); // chama function1 sem argumentos
13     function2(); // chama function2 sem argumentos
14     return 0; // indica terminação bem-sucedida
15 } // fim de main
16
17 // function1 utiliza uma lista de parâmetros vazia para especificar que
18 // a função não recebe argumentos
19 void function1()
20 {
21     cout << "function1 takes no arguments" << endl;
22 } // fim de function1
23
24 // function2 utiliza uma lista de parâmetros void para especificar que
25 // a função não recebe argumentos
26 void function2( void )
27 {
28     cout << "function2 also takes no arguments" << endl;
29 } // fim de function2

```

function1 takes no arguments
function2 also takes no arguments

Figura 6.17 Funções que não aceitam argumentos.



Erro comum de programação 6.13

Os programas C++ não compilam a menos que protótipos de funções sejam oferecidos a cada função ou que cada função seja definida antes de ser chamada.

6.13 Funções inline

Implementar um programa como um conjunto de funções é bom do ponto de vista da engenharia de software, mas as chaves de função envolvem overhead de tempo de execução. O compilador pode reduzir o overhead de chamada de função — especialmente para funções pequenas. Colocando o identificador de retorno de uma função na definição de função ‘aconselha’ o compilador a gerar uma cópia do código da função no seu lugar (quando apropriado) para evitar uma chamada de função. Em troca, múltiplas cópias do código de função são inseridas no programa (tornando freqüentemente o programa maior).

O uso de funções inline é útil quando a função é chamada muitas vezes, porque o código é copiado e executado a cada vez que a função é chamada. O compilador pode



Observação de engenharia de software 6.11

Qualquer alteração em uma função poderia exigir que todos os clientes da função fossem recompilados. Isso pode ser significativo em algumas situações de desenvolvimento e manutenção de programas.



Boa prática de programação 6.5

O qualificador `inline` deve ser utilizado somente com funções pequenas, freqüentemente utilizadas.



Dica de desempenho 6.4

Utilizar funções pode reduzir o tempo de execução, mas pode aumentar o tamanho do programa.

A Figura 6.18 utiliza a função (linhas 11–14) para calcular o volume de um cubo. A definição da função (linhas 11) informa ao compilador que a função não modifica as variáveis que o valor de `side` não é alterado pela função quando o cálculo é realizado. (Detalhes mais detalhes nos capítulos 7, 8 e 10.) Note que a definição completa da função de ela ser utilizada no programa. Isso é necessário para que o compilador saiba expandir uma chamada de função dentro de seu código inline. Por essa razão, as funções inline reutilizáveis são normalmente colocadas em arquivos de cabeçalho, para que suas definições possam ser incluídas em cada arquivo-fonte que as utiliza.



Observação de engenharia de software 6.12

O qualificador `const` deve ser utilizado para impor o princípio do menor privilégio. Utilizar o princípio do menor privilégio para projetar o software de modo adequado pode reduzir significativamente o tempo de depuração e efeitos colaterais, e pode tornar o programa mais fácil de modificar e manter.

6.14 Referências e parâmetros de referência

Duas maneiras de passar argumentos para funções em muitas linguagens de programação são

referência Quando um argumento é passado por referência, a alteração feita no argumento é feita e passada (na pilha de chamadas de função) para a função chamada. As alterações na cópia não afetam o valor da variável original no chamador. Isso previne os

```

1 // Figura 6.18: fig06_18.cpp
2 // Utilizando uma função inline para calcular o volume de um cubo.
3 #include <iostream>
4
5 using std::cin;
6 using std::endl;
7
8 // Definição da função inline cube. A definição de função aparece antes
9 // de a função ser chamada, então um protótipo de função não é necessário.
10 // A primeira linha da definição de função atua como o protótipo.
11 inline double cube( const double side )
12 {
13     return side * side * side;    // calcula o cubo
14 } // fim da função cube
15
16 int main()
17 {
18     double sideValue; // armazena o valor inserido pelo usuário
19     cout << "Enter the side length of your cube: ";
20     cin >> sideValue; // lê o valor fornecido pelo usuário
21
22     // calcula o cubo de sideValue e exibe o resultado
23     cout << "Volume of cube with side "
24         << sideValue << " is " << cube( sideValue ) << endl;
25     return 0; // indica terminação bem-sucedida
26 } // fim de main

```

Enter the side length of your cube: 3.5
 Volume of cube with side 3.5 is 42.875

Figura 6.18 Função inline que calcula o volume de um cubo.

colaterais accidentais que tanto impedem o desenvolvimento de sistemas de software confiáveis e corretos. Até agora, cada arç passado nos programas deste capítulo foi passado por valor.



Dica de desempenho 6.5

Uma desvantagem de passar por valor é que, se um item de dados grande estiver sendo passado, copiar esses dados pode exigir grande quantidade considerável de tempo de execução e espaço de memória.

Parâmetros de referência

Esta seção introduz [parâmetros de referência](#), a primeira de duas maneiras como o C++ permite passagem por referência. Com a passagem por referência, o chamador fornece à função chamada a capacidade de acessar os dados do chamador diretamente; esses dados se a função chamada escolher fazer isso.



Dica de desempenho 6.6

A passagem por referência é boa por razões de desempenho, porque pode eliminar o overhead da passagem por valor de cópias grandes quantidades de dados.



Observação de engenharia de software 6.13

A passagem por referência pode enfraquecer a segurança, porque a função chamada pode corromper os dados do chamador.

Mais adiante, mostraremos como alcançar a vantagem de desempenho da passagem por referência enquanto alcançamos, normalmente, a vantagem de engenharia de software de proteger os dados do chamador contra corrupção.

Um parâmetro de referência ([é uma pelícola](#), pronuncia-se ‘álias’) para seu argumento correspondente em uma chamada de função. Para indicar que um parâmetro de função é passado por referência, coloque um ‘`&`’ antes do nome do parâmetro no protótipo de função; use a mesma convenção ao listar o tipo do parâmetro no cabeçalho de função. Por exemplo:

```
int &count
```

quando lida da direita para a esquerda, é pronunciada ‘referência para’. Na chamada de função, simplesmente mencione a variável por nome para passá-la por referência. Então, mencionar a variável por seu nome de parâmetro no corpo da função chamada na realidade referencia a variável `srcinal` na função chamadora; e a variável `srcinal` pode ser modificada diretamente pela função chamada. Como sempre, o protótipo de função e o cabeçalho devem ser correspondentes.

Passando argumentos por valor e por referência

A Figura 6.19 compara a passagem por valor e por referência com os parâmetros de referência. Os ‘estilos’ dos argumentos nomes diferentes — Value e `ValueByReference` — são idênticos — ambas as variáveis são simplesmente mencionadas por nome nas chamadas de função. Sem verificar os protótipos de função ou definições de função, não é possível informar, considerando as chamadas isoladamente, se alguma função pode modificar seus argumentos. Mas como os protótipos de função são obrigatórios, o compilador não tem problemas para resolver a ambigüidade.



Ero comum de programação 6.14

Como os parâmetros de referência são mencionados apenas pelo nome no corpo da função chamada, o programador poderia inadvertidamente tratar os parâmetros de referência como parâmetros passados por valor. Isso pode causar efeitos colaterais inesperados, se as cópias `srcinal` das variáveis forem alteradas pela função.

O Capítulo 8 discute ponteiros; estes permitem uma forma alternativa de passagem por referência na qual o estilo da chamada claramente a passagem por referência (e o potencial para modificar os argumentos do chamador).



Dica de desempenho 6.7

Para passar objetos grandes, utilize um parâmetro de referência constante a fim de simular a aparência e a segurança da passagem por valor e evitar o overhead de passar uma cópia do objeto grande.



Observação de engenharia de software 6.14

Muitos programadores não se incomodam em declarar parâmetros passados por valor como `const`. Se a função chamada não devia modificar o argumento passado, a palavra-chave `const` protegeria apenas uma cópia do argumento `srcinal`, não o próprio argumento `srcinal`, que, quando passado por valor, é protegido contra modificação pela função chamada.

```

1 // Figura 6.19: fig06_19.cpp
2 // Comparando a passagem por valor e a passagem por referência com as referências.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int squareByValue(int ); // protótipo de função (passagem por valor)
8 void squareByReference(int &); // protótipo de função (passagem por referência)
9
10 int main()
11 {
12     int x = 2; // valor para square utilizando squareByValue
13     int z = 4; // valor para square utilizando squareByReference
14
15     // demonstra squareByValue
16     cout << "x = " << x << " before squareByValue\n";
17     cout << "Value returned by squareByValue: "
18         << squareByValue( x ) << endl;
19     cout << "x = " << x << " after squareByValue\n" << endl;
20
21     // demonstra squareByReference
22     cout << "z = " << z << " before squareByReference" << endl;
23     squareByReference( z );
24     cout << "z = " << z << " after squareByReference" << endl;
25     return 0; // indica terminação bem-sucedida
26 } // fim de main
27
28 // squareByValue multiplica um número por ele próprio, armazena o
29 // resultado em number e retorna o novo valor de number
30 int squareByValue(int number )
31 {
32     return number *= number;// argumento do chamador não modificado
33 } // fim da função squareByValue
34
35 // squareByReference multiplica numberRef por si mesmo e armazena o resultado
36 // na variável à qual numberRef se refere na função main
37 void squareByReference(int &numberRef )
38 {
39     numberRef *= numberRef; // argumento do chamador modificado
40 } // fim da função squareByReference

```

x = 2 before squareByValue
 Value returned by squareByValue: 4
 x = 2 after squareByValue

z = 4 before squareByReference
 z = 16 after squareByReference

Figura 6.19 Passando argumentos por valor e por referência.

Para especificar uma referência a uma constante, coloque `const` antes do especificador de tipo na declaração de parâmetro.

Observe na linha 37 da Figura 6.19 que listado como parâmetros de função. Alguns programadores em C++ preferem escrever `numberRef`



Observação de engenharia de software 6.15

Pelas razões combinadas de clareza e desempenho, muitos programadores em C++ preferem que argumentos modificáveis seja passados para as funções utilizando ponteiros (que estudamos no Capítulo 8), pequenos argumentos não modificáveis sejam passados por valor e argumentos grandes não modificáveis sejam passados para funções utilizando referências a constantes.

Referências como aliases dentro de uma função

As referências também podem ser utilizadas como aliases para outras variáveis dentro de uma função (embora, em geral, sejam com funções, como mostrado na Figura 6.19). Por exemplo, o código

```
int count = 1; // declara a variável count do tipo inteiro
int &cRef = count; // cria cRef como um alias para count
cRef++; // incrementa count (utilizando seu alias cRef)
```

incrementa a variável utilizando seu alias. As variáveis de referência devem ser inicializadas em suas declarações (ver figuras

6.20 e 6.21) e não podem ser reatribuídas como aliases a outras variáveis. Uma vez que uma referência é declarada como um alias para outra variável, todas as operações supostamente realizadas no alias (isto é, a referência) são realmente realizadas na variável original. Aceitar o endereço de uma referência e comparar referências não produz erros de sintaxe; em vez disso, cada operação ocorre na variável para a qual a referência é um alias. A menos que seja referência a constante, um argumento de referência (por exemplo, um nome variável), não uma constante ou expressão que retorna um valor (por exemplo, o resultado de um cálculo). Veja a Seção 5.9 para uma definição das termos

Retornando uma referência de uma função

As funções podem retornar referências, mas isso pode ser perigoso. Ao retornar referência a uma variável declarada na função, a variável deve ser declarada dentro dessa função. Caso contrário, a referência referencia uma variável automática que é descartada quando a função termina; diz-se que essa variável é ‘indefinida’ e o comportamento do programa é imprevisível. As referências indefinidas são chamadas oscilantes



Erro comum de programação 6.15

Não inicializar uma variável de referência quando ela é declarada é um erro de compilação, a menos que a declaração faça parte da lista de parâmetros de uma função. Os parâmetros de referência são inicializados quando a função em que são declarados é chamada.

```
1 // Figura 6.20: fig06_20.cpp
2 // As referências devem ser inicializadas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;
10    int &y = x; // y referencia (é um alias para) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // realmente modifica x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indica terminação bem-sucedida
16 } // fim de main
```

```
x = 3
y = 3
x = 7
y = 7
```

Figura 6.20 Inicializando e utilizando uma referência.

```

1 // Figura 6.21: fig06_21.cpp
2 // As referências devem ser inicializadas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;
10    int &y; // Erro: y deve ser inicializado
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indica terminação bem-sucedida
16 } // fim de main

```

Mensagem de erro do compilador de linha de comando da Borland C++:

Error E2304 C:\cpphtp5_examples\ch06\Fig06_21\fig06_21.cpp 10:
Reference variable 'y' must be initialized in function main()

Mensagem de erro do compilador Microsoft Visual C++:

C:\cpphtp5_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' :
references must be initialized

Mensagem de erro do compilador GNU C++:

fig06_21.cpp:10: error: 'y' declared as a reference but not initialized

Figura 6.21 Referência não inicializada produz um erro de sintaxe.



Erro comum de programação 6.16

Tentar reatribuir uma referência anteriormente declarada como um alias de outra variável é um erro de lógica. O valor da outra variável é simplesmente atribuído à variável para a qual a referência já é um alias.



Erro comum de programação 6.17

Retornar uma referência a uma variável automática em uma função chamada é um erro de lógica. Alguns compiladores emitem um aviso quando isso ocorre.

Mensagens de erro para referências não inicializadas

Observe que o padrão C++ não especifica as mensagens de erro que os compiladores utilizam para indicar erros particulares razão, a Figura 6.21 mostra as mensagens de erro produzidas pelo compilador de linha de comando Borland C++ 5.5, pelo Microsoft Visual C++ .NET e pelo compilador GNU C++ quando uma referência não é inicializada.

6.15 Argumentos-padrão

Não é incomum que um programa invoque uma função repetidamente com o mesmo valor de argumento para um parâmetro particular. Nessas casos, o programador pode especificar que tal parâmetro tem um valor-padrão a ser passado a esse parâmetro. Quando um programa omite um argumento para um parâmetro com um argumento-padrão em uma chamada, o compilador reescreve a chamada de função e insere o valor-padrão desse argumento a ser passado como um argumento para a função.

Argumentos-padrão devem ser os argumentos mais à direita (finais) em uma lista de parâmetros da função. Quando chama a função com dois argumentos-padrão, se o argumento omitido não for o argumento mais à direita na lista de argumentos, então

os argumentos à direita desse argumento devem também ser omitidos. Os argumentos-padrão devem ser especificados com a ocorrência do nome de função — em geral, no protótipo de função. Se o protótipo de função é omitido porque a definição de função também serve como o protótipo, então os argumentos-padrão devem ser especificados no cabeçalho de função. Os valores-padrão podem ser qualquer expressão, inclusive constantes, variáveis globais ou chamadas de função. Os argumentos-padrão também podem ser utilizados com funções.

A Figura 6.22 demonstra como utilizar argumentos-padrão no cálculo do volume de uma caixa. O protótipo de função para a função `boxVolume` (linha 8) especifica que todos os três parâmetros recebem argumentos-padrão. Os nomes de variável no protótipo de função são irrelevantes para uma questão de legibilidade. Como sempre, os nomes de variável não são necessários nos protótipos de função.

```

1 // Figura 6.22: fig06_22.cpp
2 // Utilizando argumentos-padrão.

3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // protótipo de função que especifica argumentos-padrão
8 int boxVolume(int length = 1, int width = 1, int height = 1);
9
10 int main()
11 {
12     // nenhum argumento — utilize valores-padrão para todas as dimensões
13     cout << "The default box volume is: " << boxVolume();
14
15     // especifica o comprimento; largura e altura-padrão
16     cout << "\n\nThe volume of a box with length 10,\n"
17     << "width 1 and height 1 is: " << boxVolume(10);
18
19     // especifica comprimento e largura; altura-padrão
20     cout << "\n\nThe volume of a box with length 10,\n"
21     << "width 5 and height 1 is: " << boxVolume(10, 5);
22     // especifica todos os argumentos
23
24     cout << "\n\nThe volume of a box with length 10,\n"
25     << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
26     << endl;
27
28 } // fim de main
29
30 // função boxVolume calcula o volume de uma caixa
31 int boxVolume(int length, int width, int height)
32 {
33     return length * width * height;
34 } // fim da função boxVolume

```

The default box volume is: 1

The volume of a box with length 10,

width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100

Figura 6.22 Argumentos-padrão para uma função.



Erro comum de programação 6.18

É um erro de compilação especificar argumentos-padrão no protótipo e no cabeçalho da função.

A primeira chamada (linha 13) não especifica nenhum argumento, utilizando assim os três valores-padrão de 1 para os argumentos. A segunda chamada (linha 17) passa argumentos utilizando assim valores-padrão de 1 para os argumentos. A terceira chamada (linha 21) passa argumentos utilizando assim um valor-padrão de 1 para o argumento ght. A última chamada (linha 25) passa argumentos para ht, utilizando assim valores não-padrão. Observe que quaisquer argumentos passados à função explicitamente são atribuídos aos parâmetros da função da esquerda para a direita quando o valor é recebido por argumento. A função atribui o valor desse argumento ao seu parâmetro mais à esquerda na lista de parâmetros. Quando se receberem dois argumentos, a função atribui os valores desses argumentos a seus parâmetros width e height nessa ordem. Por fim, quando se receberem os três argumentos, a função atribui o valor desse argumento a seus parâmetros width, height e length, respectivamente.



Boa prática de programação 6.6

Utilizar argumentos-padrão pode simplificar a escrita de chamadas de função. Entretanto, alguns programadores sentem que é mais claro especificar todos os argumentos explicitamente.



Observação de engenharia de software 6.16

Se os valores-padrão de uma função mudam, todo o código-cliente utilizando a função deve ser recompilado.



Erro comum de programação 6.19

Especificar e tentar utilizar um argumento-padrão que não é um argumento mais à direita (final) (enquanto não simultaneamente assumindo o padrão para todos os argumentos mais à direita) é um erro de sintaxe.

6.16 Operador de solução de escopo unário

É possível declarar variáveis locais e globais do mesmo nome. O C# fornece o operador unário de escopo para acessar uma variável global quando uma variável local do mesmo nome estiver no escopo. O operador unário de resolução de escopo é sempre precedido pelo operador de negação (`!`) e sempre que a variável global padronizada é usada.

A Figura 6.23 demonstra o operador unário de resolução de escopo com variáveis locais e globais do mesmo nome (linhas 7 e 8). A Figura 6.23 enfatiza que versões locais e globais de número variável são distintas, o programa adota a maioria de uma variável de tipo.

Utilizar o operador unário de resolução de escopo para uma variável dado é opcional quando a única variável com esse nome é uma variável global.



Erro comum de programação 6.20

É um erro tentar utilizar o operador unário de resolução de escopo para uma variável não global em um bloco externo.

Se não existir nenhuma variável global com esse nome, ocorre um erro de compilação. Se existir uma variável global com esse nome, é um erro de lógica, porque o programa referenciará a variável global quando pretendia acessar a variável não global no bloco externo.



Boa prática de programação 6.7

Sempre utilizar o operador unário de resolução de escopo para referenciar as variáveis globais torna os programas mais fáceis de ler e entender, porque torna claro que você está pretendendo acessar uma variável global em vez de uma variável não global.



Observação de engenharia de software 6.17

Utilizar sempre o operador unário de resolução de escopo para referenciar a variáveis globais torna os programas mais fáceis de modificar, reduzindo o risco de colisões de nome com as variáveis não globais.



Dica de prevenção de erro 6.4

Sempre utilizar o operador unário de resolução de escopo para referenciar uma variável global elimina possíveis erros de lógica que podem ocorrer se uma variável não global ocultar a variável global.

```

1 // Figura 6.23: fig06_23.cpp
2 // Utilizando o operador unário de resolução de escopo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int number = 7; // variável global chamada number
8
9 int main()
10 {
11     double number = 10.5; // variável local chamada number
12
13     // exibe valores de variáveis locais e globais
14     cout << "Local double value of number = " << number
15     << endl;
16     return 0; // indica terminação bem-sucedida
17 } // fim de main

```

Local double value of number = 10.5
Global int value of number = 7

Figura 6.23 Operador unário de resolução de escopo.

**Dica de prevenção de erro 6.5**

Evite utilizar variáveis do mesmo nome para propósitos diferentes em um programa. Embora isso seja permitido em várias circunstâncias, pode levar a erros.

 C++ permite que **várias funções** do mesmo nome sejam definidas, contanto que essas funções tenham conjuntos diferentes de parâmetros (pelo menos no que diz respeito aos tipos de parâmetro ou o número de parâmetros, ou à ordem dos tipos de parâmetros). A capacidade é chamada **sobrecarga de funções**. Quando uma função sobrecarregada é chamada, o compilador C++ seleciona a função adequada examinando o número, os tipos e a ordem dos argumentos na chamada. A sobrecarga de funções é comumente utilizada para criar várias funções do mesmo nome que realizam tarefas semelhantes, mas em tipos de dados diferentes. Por exemplo, muitas funções na biblioteca de matemática são sobrecarregadas para tipos de dados diferentes.

**Boa prática de programação 6.8**

Sobrekarregar funções que realizam tarefas intimamente relacionadas pode tornar os programas mais legíveis e compreensíveis.

Funções square sobrekarregadas

A Figura 6.24 utiliza funções sobrekarregadas para calcular o quadrado de um número inteiro (linhas 15–19). A linha 23 invoca a versão square passando o valor 10. O C++ trata o valor de número literal inteiro como tipo por padrão. De maneira semelhante, a linha 25 invoca a função square passando o valor 10.0, que o C++ trata como tipo float por padrão. Em cada caso o compilador escolhe a chamada de função adequada, com base no tipo do argumento. As duas últimas linhas da janela de saída confirmam que a função adequada foi chamada em cada caso.

Como o compilador diferencia as funções sobrekarregadas?

As funções sobrekarregadas são distinguidas por suas assinaturas. Uma assinatura é uma combinação de um nome da função e os tipos de parâmetro (em ordem). O compilador codifica cada identificador de função com o número e os tipos de seus parâmetros referidos. A decoração de nome para permitir a ligação segura para tipos (safe linkage) é uma linkagem segura para o tipo que garante que a função sobrekarregada adequada seja chamada e que os tipos dos argumentos correspondam aos tipos dos parâmetros.

¹ O padrão C++ exige as versões sobrekarregadas das funções matemáticas da biblioteca de matemática discutidas na Seção 6.3.

```

1 // Figura 6.24: fig06_24.cpp
2 // Funções sobrecarregadas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // função square para valores int
8 int square( int x )
9 {
10     cout << "square of integer " << x << " is " ;
11     return x * x;
12 } // fim da função square com argumento int
13
14 // função square para valores double
15 double square( double y )
16 {
17     cout << "square of double " << y << " is " ;
18     return y * y;
19 } // fim da função square com argumento double
20
21 int main()
22 {
23     cout << square( 7 ); // chama versão int
24     cout << endl;
25     cout << square( 7.5 ); // chama versão double
26     cout << endl;
27     return 0; // indica terminação bem-sucedida
28 } // fim de main

```

```

square of integer 7 is 49
square of double 7.5 is 56.25

```

Figura 6.24 Funções sobrecarregadas.

A Figura 6.25 foi compilada com o compilador de linha de comando Borland C++ 5.6.4. Em vez de mostrar a saída de execução do programa (como normalmente faríamos), mostramos os nomes de função desfigurados produzidos na linguagem assinada por Borland C++. Cada nome desfigurado segue o nome de função. O nome de função então é separado da lista de parâmetros desfigurados. Na lista de parâmetros da função 25; veja a quarta linha de saída. Aí, `a` representa `int`, `b` representa `double`, `c` representa `char` (isto é, uma referência para `char`) e `d` representa `double` (isto é, uma referência para `double`). Na lista de parâmetros da função 26; veja a quinta linha de saída. Aí, `a` representa `int`, `b` representa `char`, `c` representa `char` e `d` representa `double`. As duas funções são distinguidas por suas listas de parâmetros diferentes. Novamente, pode ter duas funções com a mesma assinatura e diferentes tipos de retorno. Observe que a desfiguração de nome de função é feita pelo compilador. Observe também que não é possível sobrecarregar a função `main`, porque não pode ser sobrecarregada.



Erro comum de programação 6.21

Criar funções sobrecarregadas com listas de parâmetros idênticos e tipos de retorno diferentes é um erro de compilação.

O compilador utiliza apenas as listas de parâmetros para distinguir entre funções do mesmo nome. As funções sobrecarregadas precisam ter o mesmo número de parâmetros. Os programadores devem ter atenção ao sobrecarregar funções com parâmetros porque isso pode causar ambigüidade.

Operadores sobrecarregados

No Capítulo 11, discutimos como sobrecarregar operadores para definir a maneira como eles devem operar em objetos de tipos definidos pelo usuário. (De fato, utilizamos muitos operadores sobrecarregados até agora, incluindo o operador de inserção de `<<` e de extração de `>>`.) Um dos quais é sobrecarregado para ser capaz de exibir dados de todos os tipos fundamentais. Falarei

```

1 // Figura 6.25: fig06_25.cpp
2 // Desfiguração de nomes.
3
4 // função square para valores int
5 int square( int x )
6 {
7     return x * x;
8 } // fim da função square
9
10 // função square para valores double
11 double square( double y )
12 {
13     return y * y;
14 } // fim da função square
15
16 // função que recebe argumentos dos tipos
17 // int, float, char e int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20     // esvazia o corpo da função
21 } // fim da função nothing1
22
23 // função que recebe argumentos dos tipos
24 // char, int, float & e double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27     return 0;
28 } // fim da função nothing2
29
30 int main()
31 {
32 } // fim de main

```

```

@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main

```

Figura 6.25 Desfiguração de nome para permitir linkagem segura para tipos.

mais sobre como sobrecarregar para ser capaz de tratar objetos de tipos definidos pelo usuário no Capítulo 11.) A Seção 6.18 introduz templates de função para gerar automaticamente as funções sobrecarregadas que realizam tarefas idênticas em tipos diferentes.



Erro comum de programação 6.22

Uma função com argumentos-padrão omitidos poderia ser chamada de modo idêntico a outra função sobrecarregada; isso é um erro de compilação. Por exemplo, ter em um programa uma função que não aceita explicitamente nenhum argumento e uma função do mesmo nome que contém todos os argumentos-padrão resulta em um erro de compilação quando se tenta utilizar esse nome função em uma chamada sem passar argumentos. O compilador não sabe que versão da função escolher.

6.18 Templates de funções

Funções sobrecarregadas são normalmente utilizadas para realizar operações semelhantes que envolvem lógica de programa de diferentes tipos de dados. Se a lógica e as operações do programa forem idênticas para cada tipo de dados, a sobrecarga pode ser

```

1 // Figura 6.26: maximum.h
2 // Definição do template de função maximum.
3
4 template < class T > // ou template< typename T >
5 T maximum( T value1, T value2, T value3 )
6 {
7     T maximumValue = value1// pressupõe que value1 é máximo
8
9     // determina se value2 é maior que maximumValue
10    if ( value2 > maximumValue )
11        maximumValue = value2;
12
13    // determina se value3 é maior que maximumValue
14    if ( value3 > maximumValue )
15        maximumValue = value3;
16
17    return maximumValue;
18 } // fim do template de função maximum

```

Figura 6.26 Arquivo de cabeçalho do template de função maximum

de forma mais compacta e conveniente utilizando o template de função. Considerando os tipos de argumento fornecidos em chamadas para essa função, gera automaticamente separadas para tratar cada tipo de chamada de maneira adequada. Portanto, definir um único template de função define essencialmente uma família inteira de funções sobrecarregadas.

A Figura 6.26 contém a definição de um template de função (linhas 4-18) para determinar o maior de três valores. Todas as definições de template de função iniciam com o nome da função entre colchetes. Cada parâmetro na lista de parâmetros do template (frequentemente referido como **um tipo formal**) é precedido pela palavra-chave **parâmetro** e seguido por uma chave **chave**.

Essas são as regras de uso para templates: o nome da função deve ser o mesmo que o definido pelo template (linha 5) e declarar variáveis dentro do corpo da definição de função (linha 7). Um template de função é definido como qualquer função, mas utiliza os parâmetros formais de tipo como marcadores de lugar para tipos de dados reais.

O template de função na Figura 6.26 declara um único parâmetro formal **um marcador de lugar para o tipo dos dados a ser testado**. O nome de um parâmetro de tipo deve ser único na lista de parâmetros de template para uma definição de template particular. Quando o compilador detecta uma definição de template no programa, o tipo dos dados passados para substituído por toda a definição de template, e o C++ cria uma função completa para determinar o máximo de três valores do tipo de dados especificado. Então a função recém-criada é compilada. Portanto, os templates são de gerar código.



Erro comum de programação 6.23

Não colocar a palavra-chave **typename** antes de cada parâmetro de tipo formal de um template de função (por exemplo, escrever `class S, T >` em vez de `class S, class T >`) é um erro de sintaxe.

A Figura 6.27 utiliza o template de função (linhas 20, 30 e 40) para determinar o maior de três valores double e três valores

```

1 // Figura 6.27: fig06_27.cpp
2 // Programa de teste do template de função maximum.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;

```

Figura 6.27 Demonstrando o template de função maximum

(continua)

```

7
8 #include "maximum.h" // inclui a definição do template de função maximum
9
10 int main()
11 {
12     // demonstra maximum com valores int
13     int int1, int2, int3;
14
15     cout << "Input three integer values: " ;
16     cin >> int1 >> int2 >> int3;
17
18     // invoca a versão int de maximum
19     cout << "The maximum integer value is: "
20         << maximum( int1, int2, int3 );
21
22     // demonstra maximum com valores double
23     double double1, double2, double3;
24
25     cout << "\n\nInput three double values: " ;
26     cin >> double1 >> double2 >> double3;
27
28     // invoca a versão double de maximum
29     cout << "The maximum double value is: "
30         << maximum( double1, double2, double3 );
31
32     // demonstra maximum com valores char
33     char char1, char2, char3;
34
35     cout << "\n\nInput three characters: " ;
36     cin >> char1 >> char2 >> char3;
37
38     // invoca versão char de maximum
39     cout << "The maximum character value is: "
40         << maximum( char1, char2, char3 ) << endl;
41     return 0; // indica terminação bem-sucedida
42 } // fim de main

```

Input three integer values: 1 2 3

The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1

The maximum double value is: 3.3

Input three characters: A C B

The maximum character value is: C

Figura 6.27 Demonstrando o template de função maximum

(continuação)

Na Figura 6.27, três funções são criadas como resultado das chamadas nas linhas 20, 30 e 40 — esperando três valores ~~valores~~ ~~double~~ e três ~~valores~~ ~~char~~ respectivamente. A especialização de template de função obsteia a capacidade de reunião de ~~point~~ como mostrado a seguir:

```

int maximum(int value1, int value2, int value3)
{
    int maximumValue = value1;

    // determina se value2 é maior que maximumValue

```

```

if ( value2 > maximumValue )
    maximumValue = value2;

// determina se value3 é maior que maximumValue
if ( value3 > maximumValue )
    maximumValue = value3;

return maximumValue;
} // fim do template de função maximum

```

6.19 Recursão

Os programas que discutimos geralmente são estruturados como funções que chamam umas às outras de uma maneira hierárquica. Para alguns problemas, é útil ter as funções chamando ~~umas~~^{uma} a mesma função.

si mesma, direta ou indiretamente (pô). A recursão é um tópico importante, discutido detalhadamente em cursos de nível superior de ciência da computação. Esta seção é a próxima apresentar exemplos simples de recursão. Este livro contém um tratamento da recursão. A Figura 6.33 (no final da Seção 6.21) resume os exemplos e exercícios de recursão no livro.

Primeiro consideramos a recursão conceitualmente e, em seguida, examinamos dois programas contendo funções recursivas. Abordagens de solução de problemas de recursão têm um número de elementos em comum. Uma função recursiva é chamada para resolver um problema. A função realmente sabe como resolver somente o(s) caso(s) mais simples, ou os chamado(s). Se a função é chamada com um caso básico, ela simplesmente retorna um resultado. Se a função é chamada com um problema complexo, em geral, ela divide o problema em duas partes conceituais — uma parte que a função sabe fazer e outra que não: tornar a recursão realizável, a última parte deve parecer-se com o problema original, mas ser uma versão ligeiramente mais simples. Esse novo problema é parecido com o problema original; portanto, a função carrega (chama) uma cópia na própria para trabalhar no problema menor — isso é referido como ~~um~~^o também é chamado de recursão. O passo de recursão freqüentemente inclui a palavra-chave `return`, seu resultado será combinado com a parte do problema que a função soube resolver para formar um resultado que será passado de volta ao chamador original, possivelmente

O passo de recursão executa enquanto a chamada original à função ainda está aberta, isto é, não terminou de executar. O recursão pode resultar em muito mais dessas chamadas recursivas, uma vez que a função continua dividindo cada novo sub-com o qual a função é chamada em duas partes conceituais. Para que a recursão, por fim, termine, toda vez que a função chama com uma versão ligeiramente mais simples do problema original, essa seqüência de problemas cada vez menor deve, finalmente, convergir para o caso básico. Nesse ponto, a função reconhece o caso básico e retorna um resultado à cópia anterior da função.

Comparado ao tipo de resolução de problemas convencional que utilizamos até agora, é um exemplo desses como em operação, vamos escrever um programa recursivo para realizar um cálculo matemático popular.

O fatorial de um inteiro negativo, escrito pronunciado ‘fatorial’, é o produto

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

com $1!$ igual a 1, e $0!$ definido como 1. Por exemplo, $5!$ é o produto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que é igual a 120.

O fatorial de um inteiro maior que ou igual a 0, pode ser calculado (não recursivamente) utilizando uma instrução como mostrado a seguir:

```

factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
    factorial *= counter;

```

Chega-se a uma definição recursiva da função factorial observando o seguinte relacionamento:

$$n! = n \cdot (n-1)!$$

Por exemplo, $5!$ é claramente igual a $5 \cdot 4!$ Como mostrado a seguir:

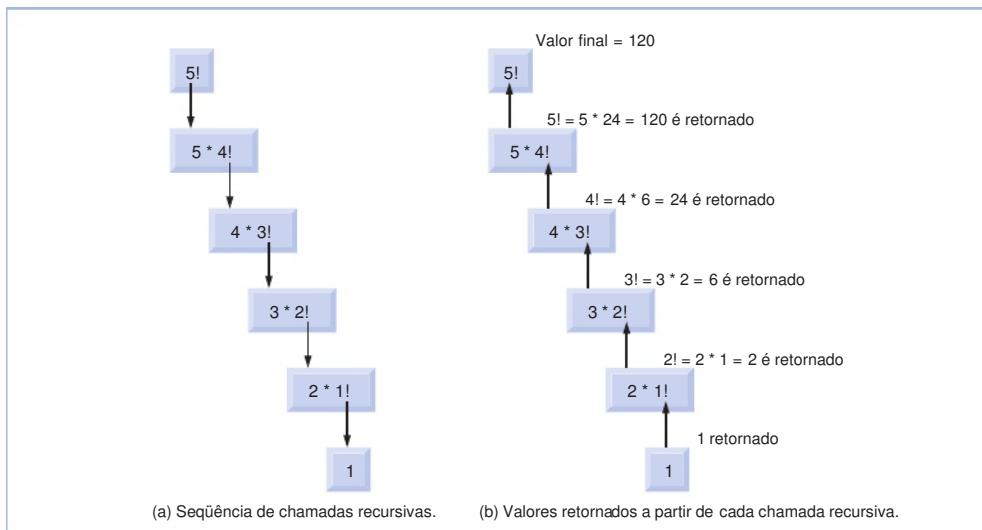
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4!) \cdot 3 \cdot 2 \cdot 1$$

A avaliação de $5!$ prosseguiria como mostrado na Figura 6.28. A Figura 6.28(a) mostra como a sucessão de chamadas recursivas prossegue até $1!$, que é avaliado como 1, o que termina a recursão. A Figura 6.28(b) mostra os valores retornados de cada chamada recursiva para seu chamador até que o valor final seja calculado e retornado.

O programa da Figura 6.29 utiliza recursão para calcular e imprimir o fatorial dos inteiros de 0–10. (A escolha do tipo de dados `signed long` é explicada daqui a pouco.) A função `main` (linhas 23–29) primeiro determina se a condição de terminação

² Embora muitos compiladores permitam a definição de funções dentro de outras funções, a Seção 3.6.1, parágrafo 3, da documentação do padrão C++ indica que não deve ser chamado a partir de dentro de um programa. Seu único propósito é ser o ponto inicial para a execução de programas.

Figura 6.28 Avaliação recursiva de $5!$.

number ≤ 1 (linha 25) é verdadeira. Se for de fato menor que ou igual a 1, a função retornará 1 (linha 26), nenhuma recursão adicional será necessária e a função terminará. Se 1, a linha 28 expressa o problema como o produto de $number - 1$ e uma chamada recursiva para avaliar o factorial de $number - 1$. Observe que factorial($number - 1$) é um problema ligeiramente mais simples que o cálculo original.

A Escolha da notação de recursão para resolver o problema do factorial é particularmente conveniente porque a variável de tipo `long int` seja armazenada em pelo menos quatro bytes (32 bits); portanto, ela pode armazenar um valor no intervalo de 0 a pelo menos 4.294.967.295. (O tipo `double` também é armazenado em pelo menos quatro bytes e pode armazenar um valor pelo menos no intervalo de -2.147.483.648 a 2.147.483.647.) Como pode ser visto na Figura 6.29, valores fatoriais tornam-se rapidamente. Escolhemos o tipo `double` para que o programa possa calcular fatoriais maiores que 7! em computadores com inteiros pequenos (como dois bytes). Infelizmente, produz valores grandes com tanta rapidez que até long não nos ajuda a calcular muitos valores fatoriais antes mesmo de o gerador de exceções inviável.

Os exercícios exploram o uso de variáveis de tipo `double` para calcular fatoriais de números maiores. Isso aponta para uma fraqueza na maioria das linguagens de programação, a saber, que as linguagens não são facilmente estendidas para tratar os únicos de vários aplicativos. Como veremos ao discutir a programação orientada a objetos em maior profundidade, o C++ é uma linguagem extensível que permite criar classes que podem representar inteiros arbitrariamente grandes se quisermos. Tais classes disponíveis em bibliotecas de classes³, em classes semelhantes nos exercícios 9.14 e 11.5.



Erro comum de programação 6.24

Omitir o caso básico ou escrever o passo de recursão incorretamente de modo que ele não converja para o caso básico causa recursão ‘infinita’, esgotando por fim a memória. Isso é análogo ao problema de um loop infinito em uma solução iterativa (não recursiva).

6.20 Exemplo que utiliza recursão: série de Fibonacci

A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subsequente é a soma dos dois números de Fibonacci anteriores.

³ Essas classes podem ser encontradas em www.cse.lehigh.edu/~hetrick/c-sources.html e www.trumphurst.com/cplibs/datapage.shtml?category='intro'.

```

1 // Figura 6.29: fig06_29.cpp
2 // Testando a função factorial recursiva.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial( unsigned long ); // protótipo de função
11
12 int main()
13 {
14     // calcula o factorial de 0 a 10
15     for ( int counter = 0; counter <= 10; counter++ )
16         cout << setw( 2 ) << counter << "!" = " " << factorial( counter )
17         << endl;
18
19     return 0; // indica terminação bem-sucedida
20 } // fim de main
21
22 // definição recursiva da função factorial
23 unsigned long factorial( unsigned long number )
24 {
25     if ( number <= 1 ) // testa caso básico
26         return 1; // casos básicos: 0! = 1 e 1! = 1
27     else // passo de recursão
28         return number * factorial( number - 1 );
29 } // fim da função factorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

Figura 6.29 Demonstrando a função factorial .

A série ocorre na natureza e, em particular, descreve a forma de uma espiral. A relação de números de Fibonacci sucessivos para um valor constante de 1,618.... Esse número, também, ocorre freqüentemente na natureza e é chamado de áurea. Humanos tendem a achar a média áurea esteticamente agradável. Os arquitetos freqüentemente projetam janelas, salas e cujo comprimento e largura estão na relação da média áurea. Os cartões-postais freqüentemente são projetados com uma razão comprimento/largura da média áurea.

A série de Fibonacci pode ser definida recursivamente como segue:

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(i) &= \text{fibonacci}(i-1) + \text{fibonacci}(i-2) \end{aligned}$$

O programa da Figura 6.30 calcula o número de Fibonacci recursivamente utilizando apenas números inteiros. Os números de Fibonacci também tendem a se tornar rapidamente grandes, embora mais lentamente do que os fatoriais. Portanto, escolheu-se o tipo de dados unsigned long para o tipo de parâmetro e o tipo de retorno. A Figura 6.30 mostra a execução do programa, que exibe os valores de Fibonacci para vários números.

```

1 // Figura 6.30: fig06_30.cpp
2 // Testando a função fibonacci recursiva.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 unsigned long fibonacci( unsigned long ); // protótipo de função
9
10 int main()
11 {
12     // calcula os valores de fibonacci de 0 a 10
13     for ( int counter = 0; counter <= 10; counter++ )
14         cout << "fibonacci( " << counter << " ) = "
15             << fibonacci( counter ) << endl;
16
17     // exibe valores fibonacci mais altos
18     cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
19     cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
20     cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
21     return 0; // indica terminação bem-sucedida
22 } // fim de main
23
24 // função fibonacci recursiva
25 unsigned long fibonacci( unsigned long number )
26 {
27     if ( ( number == 0 ) || ( number == 1 ) ) // casos básicos
28         return number;
29     else // passo de recursão
30         return fibonacci( number - 1 ) + fibonacci( number - 2 );
31 } // fim da função fibonacci

```

```

fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465

```

Figura 6.30 Demonstrando a função fibonacci.

O aplicativo inicia com uma instrução que calcula e exibe os valores de Fibonacci para os inteiros 0–10 e é seguida por três chamadas para calcular os valores de Fibonacci dos inteiros 20, 30 e 35 (linhas 18–20). As chamadas a 20 e 20 provenientes da linha 18 são recursivas, mas as chamadas a 30 são recursivas. Toda vez que o programa invoca fibonacci (linhas 25–31), a função testa imediatamente o caso básico (linhas 27–28). Se isso for verdadeiro, a linha 28 retorna imediatamente. Se for maior que 1, o passo de recursão (linha 30) gera duas chamadas recursivas, cada uma para um problema ligeiramente menor do que a figura da lista mostra como a função fibonacci avalia fibonacci(3).

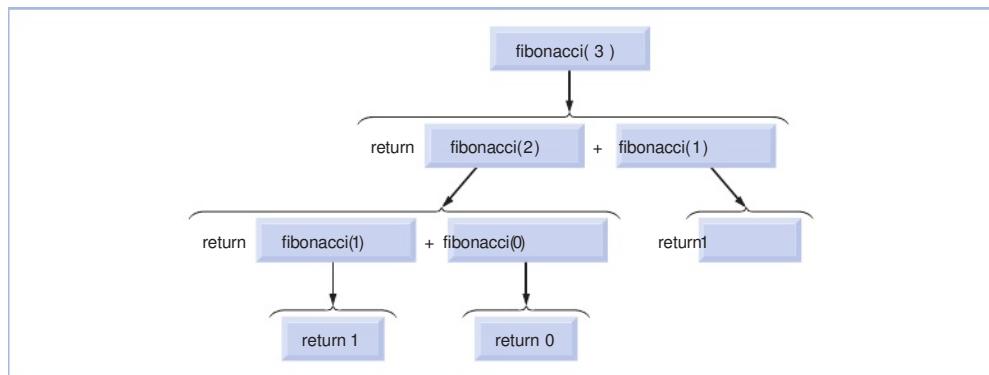


Figura 6.31 Conjunto de chamadas recursivas à função fibonacci.

Essa figura levanta algumas questões interessantes sobre a ordem em que compiladores C++ avaliarão os operandos dos operadores. Essa é uma questão separada da ordem em que os operadores são aplicados aos seus operandos, a saber, a ordem ditada pela precedência e associatividade de operadores. A Figura 6.1 mostra ~~prova~~ chamadas recursivas, a saber, `fibonacci(2)` e `fibonacci(1)`. Mas em que ordem essas chamadas são feitas? A maioria dos programadores simplesmente pressupõe que os operandos são avaliados da esquerda para a direita. A linguagem C++ não especifica a ordem em que os operandos são avaliados. Portanto, o programador não deve fazer nenhuma suposição sobre a ordem em que essas chamadas executam. As chamadas poderiam ser executadas da seguinte maneira: `fibonacci(1)`, ou poderiam executar na ordem inversa: em seguida, `fibonacci(2)`. Nesse programa e na maioria dos outros, revela-se que o resultado final seria o mesmo. Entretanto, em alguns programas a avaliação de diferentes partes pode ter nos valores dos dados) que poderiam afetar o resultado final da expressão.

A linguagem C++ especifica a ordem de avaliação dos operandos de apenas quatro operadores — a saber,

O operando mais à esquerda é avaliado em sequência e o resultado é usado para dividir o terceiro operando (verdadeiro), o operando do meio é avaliado em seguida e o último operando é ignorado; se o operando mais à esquerda for como zero (falso), o terceiro operando é avaliado em sequida e o operando do meio é ignorado.



• Erro comum de programação 6.25

Escrever programas que dependem da ordem de avaliação dos operandos de operadores diferentes dos operadores vírgula (pode levar a erros de lógica.



Dica de portabilidade 63

Os programas que dependem da ordem de avaliação dos operandos de operadores diferentes (`)` podem funcionar diferentemente em sistemas com compiladores diferentes.

Uma palavra de cautela está em ordem sobre programas recursivos como o que utilizamos aqui para gerar números de Fibonacci. Cada nível de recursão ~~tem~~^{realiza} o efeito de duplicar o número de chamadas de função; isto é, o número de chamadas recursivas que é requerido para calcular o ~~ultimo~~^{vigésimo} número de Fibonacci está na ~~base~~^{teoria} amplamente foge do controle. Calcular somente o vigésimo número de Fibonacci exigiria um número de chamadas de ~~um~~^{mais de 1000} chamadas, cal-

poladiano, o ministro da Fazenda, exige um mundo de que o Brasil é um dos países mais avançados no mundo, até os computadores mais poderosos do mundo! Questões de complexidade em geral, e de complexidade exponencial em particular, são discutidas em detalhes em cursos de nível superior de ciência da computação geralmente chamados de 'Algoritmos'.



Dica de desempenho 6.8

Evite programas recursivos no estilo de Fibonacci que resultam em uma 'explosão' exponencial de chamadas.

6.21 Recursão versus iteração

Nas duas seções anteriores, estudamos duas funções que podem ser facilmente implementadas recursiva ou iterativamente. E compara as duas abordagens e discute por que o programador poderia escolher uma abordagem à outra em uma situação particular.

Tanto iteração como recursão se baseiam em uma estrutura de controle: a iteração utiliza uma estrutura de repetição; a recursão utiliza uma estrutura de seleção. Ambas envolvem repetição: a iteração utiliza explicitamente uma estrutura de repetição; a recursão faz a repetição por chamadas de função repetidas. Iteração e recursão envolvem um teste de terminação: a iteração termina quando a condição de continuação do loop falha; a recursão termina quando um caso básico é reconhecido. A iteração com repetição pode por contador e a recursão gradualmente se aproximam do término: a iteração modifica um contador até que o contador assuma que faz a condição de continuação do loop falhar; a recursão produz versões mais simples do problema secundário até que o caso seja alcançado. Tanto uma como outra podem ocorrer infinitamente: um loop infinito ocorre com a iteração se o teste de continuação do loop nunca se tornar falso; a recursão infinita ocorre se o passo de recursão não reduz o problema durante cada chamada de uma maneira que convirja para o caso básico.

Para ilustrar as diferenças entre iteração e recursão, examinemos uma solução iterativa do problema factorial (Figura 6.32). O que uma instrução de repetição é utilizada (linhas 28–29 da Figura 6.32) em vez da instrução de seleção da solução recursiva (linhas 25–28 da Figura 6.29). Observe que as duas soluções utilizam um teste de terminação. Na solução recursiva, a linha 25 testa se o caso básico é alcançado. Na solução iterativa, a linha 28 testa a condição de continuação do loop — se o teste falhar, o loop termina. Pode-se observar que em vez de produzir a versão mais simples do problema secundário, a solução iterativa utiliza um contador que é modificado a cada iteração.

A recursão tem muitos pontos negativos. Ela invoca repetidamente o mecanismo, e consequentemente o overhead das chamadas de função. Isso pode ter um alto preço tanto em tempo de processador como em espaço de memória. Cada chamada recursiva faz uma cópia da função (na realidade, somente as variáveis da função) seja criada; isso pode consumir memória considerável. A recursão normalmente ocorre dentro de uma função, então o overhead das chamadas de função repetidas e a atribuição extra de memória são omitidos. Então, por que escolher recursão?



Observação de engenharia de software 6.18

Qualquer problema que pode ser resolvido recursivamente também pode ser resolvido iterativamente (não recursivamente). Uma abordagem recursiva normalmente é escolhida preferencialmente a uma abordagem iterativa quando a abordagem recursiva espelha mais naturalmente o problema e resulta em um programa que é mais fácil de entender e depurar. Outra razão de escolher uma solução recursiva é que uma solução iterativa não é evidente.



Dica de desempenho 6.9

Evite utilizar recursão em situações de desempenho. Chamadas recursivas levam tempo e consomem memória adicional.



Erro comum de programação 6.26

Ter accidentalmente uma função não recursiva chamando a si própria, direta ou indiretamente (por outra função), é um erro de lógica.

A maioria dos manuais de programação introduz recursão muito mais tarde do que fizemos aqui. Mas nós acreditamos que a recursão é um tópico complexo e suficientemente rico e é melhor introduzi-lo mais cedo e espalhar os exemplos pelo restante do texto. A Figura 6.33 resume os exemplos e exercícios de recursão no texto.

```

1 // Figura 6.32: fig06_32.cpp
2 // Testando a função factorial iterativa.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setw;
8
9
10 unsigned long factorial( unsigned long ); // protótipo de função
11

```

```

12 int main()
13 {
14     // calcula o factorial de 0 a 10
15     for ( int counter = 0; counter <= 10; counter++ )
16         cout << setw( 2 ) << counter << "!" << factorial( counter )
17         << endl;
18
19     return 0;
20 } // fim de main
21
22 // função factorial iterativa
23 unsigned long factorial( unsigned long number )
24 {
25     unsigned long result = 1;
26
27     // declaração iterativa da função factorial
28     for ( unsigned long i = number; i >= 1; i-- )
29         result *= i;
30
31     return result;
32 } // fim da função factorial

```

0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040

8! = 40320
 10! = 3628800

Solução factorial iterativa.

(continuação)

Localização no texto	Exemplos e exercícios de recursão
Capítulo 6	Seção 6.19, Figura 6.29
	Seção 6.19, Figura 6.30
	Exercício 6.7
	Exercício 6.40
	Exercício 6.42
	Exercício 6.44
	Exercício 6.45
	Exercícios 6.50 e 6.51
Capítulo 7	Exercício 7.18
	Exercício 7.21
	Exercício 7.31

Figura 6.3 Resumo de exemplos e exercícios de recursão no texto.

(continua)

Localização no texto	Exemplos e exercícios de recursão
Capítulo 7	Exercício 7.32 Determineseumastringéumpalíndromo
	Exercício 7.33 Pesquisadilinear
	Exercício 7.34 Pesquisabinária
	Exercício 7.35 OitRainhas
	Exercício 7.36 Imprimarray
	Exercício 7.37 Imprimumastringdetrásparaafrente
	Exercício 7.38 Valomínimoemumarray
Capítulo 8	Exercício 8.24 Quicksort
	Exercício 8.25 Percorrendoumlabirinto
	Exercício 8.26 Geradordelabirintosaleatório
	Exercício 8.27 Labirintosdequalquertamanho
Capítulo 20	Seção 20.3.3, figuras 20.5–20.7 Classificação por intercalação
	Exercício 20.8 Pesquisadilinear
	Exercício 20.9 Pesquisabinária
	Exercício 20.10 Quicksort
Capítulo 21	Seção 21.7, figuras 21.20–21.22 Inserção de árvore binária
	Seção 21.7, figuras 21.20–21.22 Percurso na pré-ordem de uma árvore binária
	Seção 21.7, figuras 21.20–21.22 Percurso na ordem de uma árvore binária
	Seção 21.7, figuras 21.20–21.22 Percurso na pós-ordem de uma árvore binária
	Exercício 21.20 Imprimauamalistavinculadetrásparaafrente
	Exercício 21.21 Pesquiseumalistavinculada
	Exercício 21.22 Exclusãodeárvorebinária
Capítulo 22	Exercício 21.25 Impressãodeárvore

Figura 6.3 Resumo de exemplos e exercícios de recursão no texto.

(continuação)

6.22 Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)

Nas seções de “Estudo de caso de engenharia de software” no final dos capítulos 3, 4 e 5, seguimos os primeiros passos do projeto a objetos do nosso sistema ATM. No Capítulo 3, identificamos as classes que precisaremos implementar e criamos nosso diagrama de classes. No Capítulo 4, descrevemos alguns atributos das nossas classes. No Capítulo 5, examinamos estados do transições de estado e atividades dos objetos modelados. Nesta seção, determinamos algumas operações de classe (ou comportamentos) necessárias para implementar o sistema ATM.

Identificando operações

Uma operação é um serviço que os objetos de uma classe fornecem aos clientes da classe. Pense nas operações de alguns objetos do real. As operações de um rádio incluem configurar sua estação e volume (em geral invocadas por uma pessoa que ajusta os botões do rádio). As operações de um carro incluem acelerar (invocadas pelo motorista ao pressionar pedal do acelerador), desacelerar (invocadas pelo motorista que pressiona o pedal do freio ou solta o pedal do acelerador), mudar de direção e trocar de marchas. Os objetos de software também podem oferecer operações — por exemplo, um objeto de um software gráfico poderia oferecer operações para desenhar um círculo, uma linha, um quadrado etc. Um objeto de um software de planilha poderia oferecer operações como imprimir a planilha, somar os elementos em uma linha ou coluna e diagramar informações na planilha, como um gráfico de barras ou gráfico de torta.

Podemos derivar várias operações de cada classe examinando os verbos e frases com verbos-chave no documento de requisitos. Então relacionamos cada um desses aspectos a classes particulares no nosso sistema (Figura 6.34). As frases com verbos na ajuda a determinar as operações de cada classe.

Modelando operações

Para identificar operações, examinamos as frases com verbos listadas para cada classe na Figura 6.34. A frase ‘executa transações financeiras’ associada à classe ATM descreve que a classe executa as transações a serem executadas. Portanto, as classes Withdrawal e Deposit precisam de uma operação para fornecer esse serviço ao ATM. Colocamos essa operação (que identificamos como execute) no terceiro compartimento das três classes de transação no diagrama de classes atualizado da Figura 6.35. Durante a sessão no ATM, o usuário poderá a operação de cada objeto de transação para instruí-lo a executar.

A UML representa operações (implementadas como funções-membro em C++) listando o nome da operação, seguido por um parâmetro separado por vírgulas de parâmetros entre parênteses, dois-pontos e o tipo de retorno:

```
nomeDaOperação ( parâmetro1, parâmetro2, ..., parâmetroN ) : tipo de retorno
```

Cada parâmetro na lista separada por vírgulas de parâmetros consiste em um nome de parâmetro, seguido por dois-pontos e o tipo de retorno.

```
nomeDoParâmetro : tipoDoParâmetro
```

Agora, não listamos os parâmetros das nossas operações — identificaremos e modelaremos os parâmetros de algumas operações mais adiante. Para algumas operações, ainda não conhecemos os tipos de retorno, portanto também iremos omiti-los do diagrama. Omissões são perfeitamente normais nesse ponto. A medida que o nosso projeto e implementação avançarem, adicionaremos os tipos de retorno remanescentes.

Operações da classe BankDatabase e da classe Account

A Figura 6.34 lista a frase ‘autentica um usuário’ no lado da classe BankDatabase. O lado da classe é o objeto que contém informações sobre uma conta necessárias para determinar se o número da conta e o PIN inserido por um usuário correspondem àquele conta mantida pelo banco. Portanto, a classe BankDatabase precisa de uma operação que forneça um serviço de autenticação ao ATM. Colocamos a operação authenticateUser no terceiro compartimento da classe BankDatabase (Figura 6.35). Entretanto, um objeto da classe Account não da classe BankDatabase armazena o número da conta e o PIN que devem ser acessados para autenticar um usuário; dessa forma, a classe Account deve fornecer um serviço para validar um PIN, obtido por meio da entrada do usuário, contra um PIN armazenado em um objeto da classe Account. Portanto, adicionamos uma operação à classe Account. Observe que especificamos um tipo de retorno para as operações authenticateUser e validatePIN. Cada operação retorna um valor que indica que a operação foi bem-sucedida na realização de sua tarefa (isto é, um valor booleano) ou um valor de retorno.

A Figura 6.34 lista várias frases com verbos adicionais, como ‘recupera um saldo em conta’, ‘credita uma quantia depositada em uma conta’ e ‘debita uma quantia retirada de uma conta’. Como ocorre com ‘autentica um usuário’, essas frases se referem aos serviços que o banco de dados deve fornecer ao ATM, porque o banco de dados armazena todos os dados de

Classe	Verbos e frases com verbos
ATM	executa transações financeiras
BalanceInquiry	[nenhuma no documento de requisitos]
Withdrawal	[nenhuma no documento de requisitos]
Deposit	[nenhuma no documento de requisitos]
BankDatabase	autentica um usuário, recupera um saldo em conta, credita uma quantia depositada em uma conta, debita uma quantia retirada de uma conta
Account	recupera um saldo em conta, credita uma quantia depositada em uma conta, debita uma quantia retirada de uma conta
Screen	exibe uma mensagem para o usuário
Keypad	recebe entrada numérica do usuário
CashDispenser	fornecendo dinheiro, indica se contém dinheiro suficiente para satisfazer uma solicitação de saque
DepositSlot	recebe um envelope de depósito

Figura 6.34 Os verbos e frases com verbo para cada classe no sistema ATM.

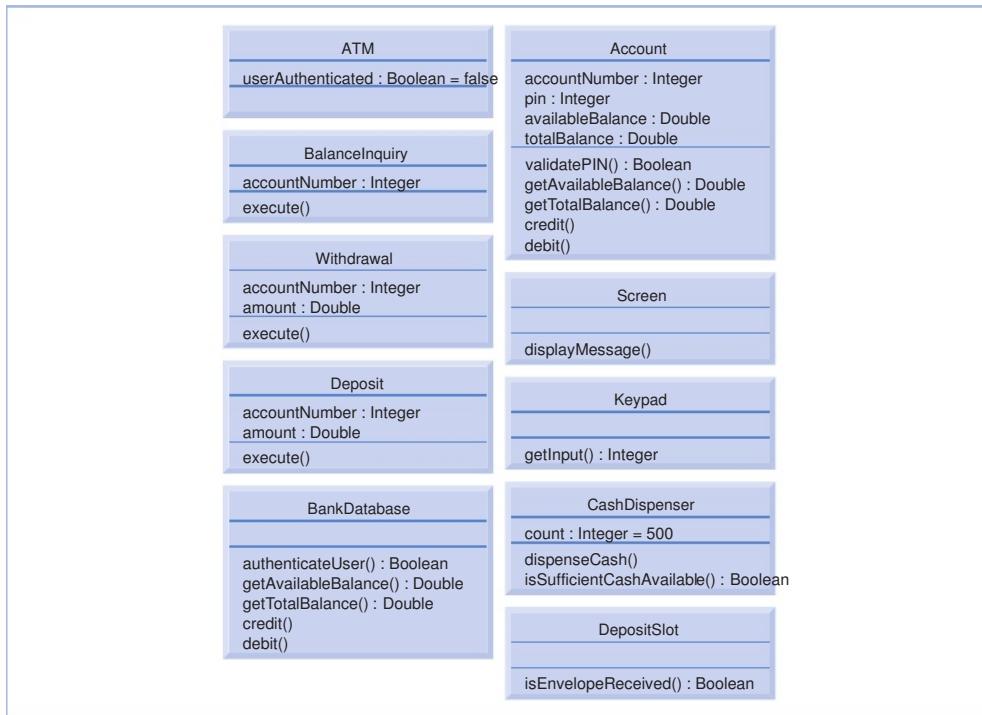


Figura 6.35 Classes no sistema ATM com atributos e operações.

utilizada para autenticar um usuário e realizar as transações no ATM. **Entretanto, devemos utilizar classes** operações às quais essas frases se referem. Portanto, atributos **uma classe** que elas correspondam a cada uma dessas frases. Lembre-se, a partir do que foi discutido na Seção 3.11, de que, como conta bancária informações sigilosas, não permitimos que o ATM acesse contas diretamente. O banco de dados atua como um intermediário ATM e os dados da conta, evitando assim acesso não autorizado. Como vimos na Seção 7.12, a classe **BankDatabase** é separada, cada uma das quais por sua vez invoca a operação com o mesmo nome na classe

A frase ‘recupera um saldo em conta’ sugere que essas classes precisam de uma operação. Entretanto, lembre-se de que criamos dois **atributos finados** para representar um saldo: `availableBalance` e `totalBalance`. Uma consulta de saldo requer acesso aos dois atributos de saldo para poder exibi-los ao usuário, mas um saque precisa verificar o valor de `availableBalance`. Para permitir que objetos no sistema obtenham cada atributo de saldo individualmente, adicionamos operações `getAvailableBalance` e `getTotalBalance` ao terceiro compartimento das classes. Especificamos um tipo de retorno para cada uma dessas operações porque os atributos de saldo que eles recuperam são do tipo `double`.

As frases ‘credita uma quantia depositada em uma conta’ e ‘debita uma quantia retirada de uma conta’ indicam que as classes **BankDatabase** e **Account** devem realizar operações para atualizar uma conta durante um depósito e um saque, respectivamente. Portanto, adicionamos as operações `credit()` e `debit()` às classes **BankDatabase** e **Account**. Lembre-se de que creditar em uma conta (como em um depósito) só adiciona uma quantia monetária ao atributo `availableBalance` de uma conta (‘como em um saque’), por outro lado, subtrai a quantia dos dois atributos de saldo. Ocultamos esses detalhes de implementação. Esta é a função da classe **BankDatabase**: fornecer um exemplo do encapsulamento e ocultamento de informações.

Se isso fosse um sistema ATM real, as classes **Account** também forneceria um conjunto de operações para permitir que outro sistema de operações bancárias atualizasse um saldo na conta do usuário depois de uma confirmação ou rejeição de t de um depósito. A operação `depositAmount`, por exemplo, adicionaria uma quantia monetária ao atributo `availableBalance`, tornando assim os fundos depositados disponíveis para saque. A operação `withdrawAmount` subtrairia uma quantia monetária do atributo `availableBalance` para indicar que uma quantia específica, que foi recentemente depositada por meio do ATM e adicionada ac

`totalBalance`, não foi encontrada no envelope de depósito. O banco invocaria essa operação depois de determinar que o usuário incluiu a quantia monetária correta ou que um cheque não foi compensado (isto é, ele ‘voltou’). Adicionar essas operações torri sistema mais completo, porém não as incluímos nos nossos diagramas de classes nem na nossa implementação porque elas do escopo desse estudo de caso.

Operações da classe `Screen`

A classe `Screen` ‘exibe uma mensagem para o usuário’ em vários momentos em uma sessão no ATM. Toda a saída visual ocorre p da tela do ATM. O documento de requisitos descreve muitos tipos de mensagens (por exemplo, uma mensagem de boas-vindas, uma mensagem de erro, uma mensagem de agradecimento) que a tela exibe para o usuário. O documento de requisitos também i a tela exibe prompts e menus para o usuário. Entretanto, um prompt na verdade é apenas uma mensagem que descreve o qu deve inserir em seguida e um menu é essencialmente um tipo de prompt que consiste em uma série de mensagens (isto é, opção exibida consecutivamente). Portanto, em vez de utilizar operações individual a fim de exibir cada tipo de mensagem, prompt e menu, simplesmente criamos uma operação que possa exibir qualquer mensagem especificada por um parâmetro. C

[esse parágrafo (Meio) só serve para o parâmetro de mensagem da operação de classe `Screen` (Figura 6.34). Onde meicutar de nesta seção]

Operações da classe `Keypad`

Na frase ‘recebe entrada numérica do usuário’ dada [Figura 6.34](#), concluímos que podemos realizar uma operação `getInput`. Como o teclado do ATM, diferentemente de um teclado de computador, contém somente os números de 0 a 9, especificamos que essa operação retorna um valor inteiro. Lembre-se de que no documento de requisitos, em diferentes s talvez seja necessário que o usuário insira um tipo diferente de número (por exemplo, um número de conta, um PIN, o número opção de menu, uma quantia de depósito como um número decimal). A operação `getInput` aceita um valor numérico para um cliente da classe — ela não determina se o valor atende quaisquer critérios específicos. Toda classe que utiliza essa operação deve verificar se o usuário insere números apropriados e, se não inserir, deve exibir mensagens de erro por meio da classe `message`. [Nota: Quando implementarmos o sistema, simulamos o teclado do ATM com um teclado de computador e, por simplicidade, supomos que o usuário não irá inserir uma entrada não-numérica utilizando as teclas no teclado de computador que não aparecem no teclado. Mais adiante no livro, você aprenderá a examinar entradas para determinar se elas são de tipos particulares.]

Operações da classe `CashDispenser` e classe `DepositSlot`

A Figura 6.34 lista ‘fornecer dinheiro para saque’. Portanto, criamos a operação `provide` e listamos sob a classe `CashDispenser` na [Figura 6.35](#). A classe `DepositSlot` também ‘indica se contém dinheiro suficiente para satisfazer uma solicitação de saque’. Portanto, incluimos `CashAvailable`, uma operação que retorna um valor booleano UML, na classe `DepositSlot`. [Nota: A Figura 6.34 também lista ‘recebe um envelope de depósito’ para abertura para depósito deve indicar se recebeu um envelope, portanto colocamos para dentro do terceiro compartimento da classe `slot`.] Um hardware real da abertura para depósito provavelmente enviará um sinal para o ATM para indicar que um envelope foi recebido. Entretanto, simulamos esse comportamento para a operação na classe `slot` que a classe `slot` possa ser invocada a fim de descobrir se a abertura para depósito recebeu um envelope.]

Operações da classe `ATM`

Não listamos nenhuma operação [para essa classe](#). Ainda não estamos cientes de nenhum serviço que a classe fornece para outras classes no sistema. Entretanto, ao implementarmos o sistema com código C++, operações dessa classe, e adicionais das outras classes no sistema, podem surgir.

Identificando e modelando parâmetros de operação

Até esse momento, não nos preocupamos com os parâmetros das nossas operações — tentamos apenas obter um entendimento sobre as operações de cada classe. Agora, vamos examinar mais detalhadamente alguns parâmetros de operação. Identificaremos os râmetros de uma operação examinando quais dados a operação requer para realizar sua tarefa atribuída.

Considere a operação `authenticateUser` da classe `BankDatabase`. Para autenticar um usuário, essa operação deve conhecer o número de conta e o PIN fornecido pelo usuário. Assim, especificamos que a operação requer dois parâmetros inteiros `userAccountNumber` e `userPIN` os quais a operação deve comparar com o número de conta e PIN de um objeto

[descrição: Profíxos nos encadeamentos de parâmetros com vírgulas para evitar conflito entre Figuras 6.34 e 6.35 de parâmetros da operação a classe `BankDatabase`.] É perfeitamente normal modelar somente uma classe em um diagrama de classes. Nesse caso, estamo mais preocupados em examinar os parâmetros dessa única classe em particular, portanto omitimos as outras classes. Nos dia classes posteriores neste estudo de caso, em que parâmetros não são mais o foco, omitimos esses parâmetros para economizar. Mas não se esqueça de que as operações listadas nesses diagramas ainda contêm parâmetros.]

Lembre-se de que a UML modela cada parâmetro em uma lista de parâmetros separada por vírgulas da operação e listando o parâmetro seguido por um caractere de dois-pontos e pelo tipo de parâmetro (na notação da UML). Assim, a [Figura 6.36](#) é que a operação `authenticateUser` recebe dois parâmetros `userAccountNumber` e `userPIN` ambos do tipo `integer`. Quando implementarmos o sistema em C++, representaremos esses parâmetros com os valores

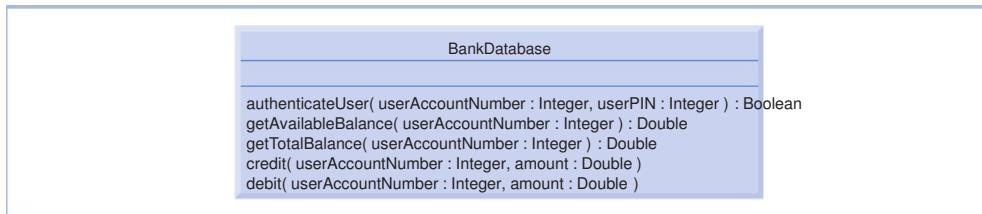


Figura 6.36 A classe BankDatabase com parâmetros de operação.

As classes BankDatabase e Account possuem operações que devem receber parâmetros. Por exemplo, as operações authenticateUser e getTotalBalance possuem parâmetros de tipo Integer, que representam o número da conta e o PIN respectivamente. As operações getAvailableBalance e debit possuem um parâmetro de tipo Double, que indica a quantia monetária a ser debitada. As operações getTotalBalance e credit possuem dois parâmetros de tipo Double, que representam o número da conta e a quantia monetária a ser debitada ou creditada, respectivamente.

O diagrama de classes na Figura 6.37 modela os parâmetros das operações da classe Account. Note que a classe Account não requerem um parâmetro que contém o PIN especificado pelo usuário a ser comparado com o PIN associado com a conta. Como ocorre com suas contrapartes nas classes BankDatabase e User, as operações credit e debit na classe Account requerem um parâmetro amount que indica a quantia monetária envolvida na operação. As operações getTotalBalance e validatePIN na classe Account não requerem nenhum dado adicional para realizar suas tarefas. Observe que as operações da classe Account não requerem um parâmetro de número de conta — cada uma dessas operações pode ser invocada especificando o objeto desse tipo.

A Figura 6.38 modela a classe Screen com um parâmetro especificado para a operação displayMessage. Essa operação requer somente um parâmetro message que indica o texto a ser exibido. Lembre-se de que os tipos de parâmetros listados nos nossos diagramas de classes estão em notação UML. No entanto, na Figura 6.38 refere-se ao tipo UML. Quando implementarmos o sistema em C++, iremos, de fato, utilizar um parâmetro para representar esse parâmetro.

O diagrama de classes na Figura 6.39 especifica que a operação withdrawDispense recebe um parâmetro Double amount para indicar a quantia monetária (em dólares) a ser entregue. A operação deposit também recebe um parâmetro Double amount para indicar a quantia monetária em questão.

Observe que não discutimos os parâmetros para as operações cancelInquiry, Withdrawal e Deposit, a operação setInput da classe Keypad e a operação EnvelopeReceived da classe DepositSlot. Nesta fase do nosso processo de projeto, não podemos determinar se essas operações exigem dados adicionais para realizar suas tarefas, assim deixamos suas listas de vazias. A medida que avançamos pelo estudo de caso, podemos decidir adicionar parâmetros a essas operações.

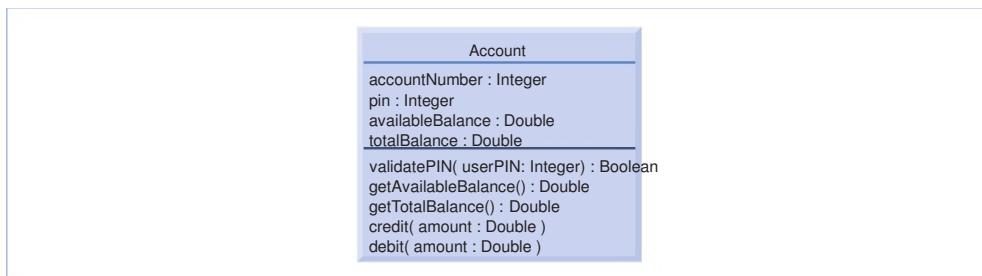


Figura 6.37 A classe Account com parâmetros de operação.



Figura 6.38 A classe Screen com parâmetros de operação.

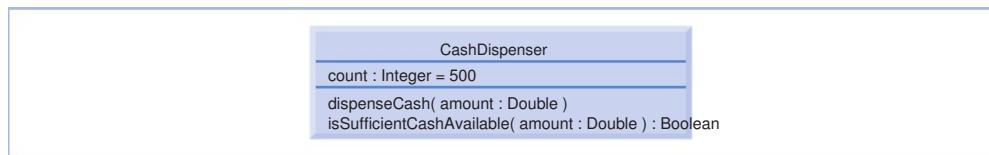


Figura 6.39 A classe CashDispenser com parâmetros de operação.

Nesta seção, determinamos várias operações realizadas pelas classes no sistema ATM. Identificamos os parâmetros e os retorno de algumas operações. A medida que prosseguimos pelo nosso processo de projeto, o número de operações que pertence à classe talvez varie — poderíamos descobrir que novas operações são necessárias ou que algumas operações atuais são desejadas. Poderíamos determinar que algumas das nossas operações de classe precisam de parâmetros adicionais e tipos de retorno diferentes. Exercícios de revisão do estudo de caso de engenharia de software

- 6.1 Qual das seguintes alternativas não é um comportamento?
 - a) ler dados a partir de um arquivo
 - b) imprimir a saída
 - c) gerar saída de texto
 - d) obter a entrada do usuário
- 6.2 Se você fosse adicionar ao sistema ATM uma operação que retorna dinheiro, como e onde você especificaria essa operação no diagrama de classes da Figura 6.35?
- 6.3 Descreva o significado da listagem da operação a seguir que poderia aparecer em um diagrama de classes em um projeto orientado a uma calculadora:
 $\text{add}(x : \text{Integer}, y : \text{Integer}) : \text{Integer}$

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 6.1 C.
- 6.2 Para especificar uma operação que retorna dinheiro, a operação a seguir seria colocada no compartimento de operações (note, no entanto, que a classe CashDispenser não tem operações).
- 6.3 Essa é uma operação que aceita inteiros como parâmetros e retorna um valor inteiro.

6.23 Síntese

Neste capítulo, você aprendeu mais sobre os detalhes das declarações de função. As funções têm partes diferentes, como a assinatura, o cabeçalho e o corpo de função. Você aprendeu sobre a coerção de argumento, isto é, forçar argumentos para serem especificados pelas declarações de parâmetro de uma função. Demonstramos como utilizar funções para gerar conjuntos de números aleatórios que podem ser utilizados para simulações. Você também aprendeu sobre o escopo de uma função e a parte de um programa em que um identificador pode ser utilizado. Duas maneiras diferentes de passar argumentos para uma função foram discutidas — passagem por valor e por referência. Na passagem por referência, as referências são utilizadas como um tipo de variável. Você aprendeu que múltiplas funções em uma classe podem ser sobreescritas fornecendo funções com o mesmo nome e assinaturas diferentes. Essas funções podem ser utilizadas para realizar as mesmas tarefas, ou tarefas semelhantes, utilizando diferentes ou números diferentes de parâmetros. Depois demonstramos uma maneira mais simples de sobreescalar funções — utilizar templates de função, onde uma função é definida uma vez, mas pode ser utilizada para vários tipos diferentes. Entendemos o conceito de recursão, onde uma função chama a si mesma para resolver um problema.

Na próxima aplicação de projeto de banco de dados e suas versões alternativas, você verá uma implementação baseada em arrays. Neste projeto, você aprenderá a manter listas e tabelas de dados em arrays. Você verá uma implementação baseada em arrays para armazenar as notas inseridas.

Resumo

- A experiência mostra que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de partes pequenas e separadas, ou módulos. Essa técnica se chama dividir para conquistar.

- Em geral, os programas C++ são escritos combinando novas funções e classes que o programador escreve com funções ‘pré-empacota classes disponíveis na C++ Standard Library.
- As funções permitem ao programador modularizar um programa separando suas tarefas em unidades autocontidas.
- As instruções no corpo das funções são escritas apenas uma vez, talvez reutilizadas a partir de diversas localizações em um programa, ocultadas de outras funções.
- O compilador se refere ao protótipo de função para verificar se as chamadas para uma função contêm o número e os tipos de argumentos corretos, se os tipos dos argumentos estão na ordem correta e se o valor retornado pela função pode ser utilizado corretamente na expressão que a função.
- Há três maneiras de retornar o controle para o ponto em que uma função foi invocada. Se a função não retornar um resultado, o controle retorna quando o programa alcançar a chave direita de fechamento da função, ou pela execução da instrução

```
return ;
```

Se a função retorna um resultado, a instrução
`return` expressa

avalisação de expressão retorna o valor dessa expressão para o chamador.

- Um protótipo de função informa ao compilador o nome de uma função, os tipos de dados retornados por essa função, o número de parâmetros que a função espera receber, os tipos desses parâmetros e a ordem em que os parâmetros desses tipos são esperados.
- A parte de um protótipo de função que inclui o nome da função e os tipos de seus argumentos é chamada assinatura de função ou simples assinatura.
- Um recurso importante dos protótipos de função é a coerção de argumento — isto é, forçar argumentos para os tipos apropriados especificados pelas declarações de parâmetro.
- Os valores de argumento que não correspondem precisamente aos tipos de parâmetro no protótipo de função podem ser convertidos pelo compilador no tipo adequado como especificado pelas regras de promoção do C++. As regras de promoção indicam como converter entre tipos diferentes de dados.
- O elemento chance pode ser introduzido em aplicativos de computador utilizando a Standard Library.
- A função `rand()` realmente gera números pseudo-aleatórios, ou pseudo-randomizados. Ela produz uma seqüência de números que parece aleatória. Entretanto, a seqüência repete-se toda vez que o programa executa.
- Uma vez que um programa foi completamente depurado, ele pode ser condicionado a produzir uma seqüência diferente de números aleatórios para cada execução. Isso é chamado aleatorização ou randomização.
- A função `rand()` aceita um argumento do tipo `int` que semeia a função para produzir uma seqüência diferente de números aleatórios para cada execução do programa.
- Os números aleatórios em um intervalo podem ser gerados com

```
número = valorDeDeslocamento + rand() % fatorDeEscala;
```

onde `valorDeDeslocamento` é igual ao primeiro número no intervalo desejado de inteiros e `fatorDeEscala` é a largura do intervalo desejado de inteiros consecutivos.

- Uma enumeração, introduzida pela palavra-chave `enum` por um nome de tipo, é um conjunto de constantes do tipo inteiro representadas por identificadores. Os valores dessas constantes em uma enumeração especificado de outro modo e incrementa por 1.
- A classe de armazenamento de um identificador determina o período em que esse identificador existe na memória.
- O escopo de um identificador é onde o identificador pode ser referenciado em um programa.
- A linkagem de um identificador determina se um identificador só é conhecido no arquivo de fonte onde ele é declarado ou por múltiplos arquivos que são compilados e, então, linkados.
- As palavras-chave `register` são utilizadas para declarar as variáveis da classe de armazenamento automática. Essas variáveis são criadas quando a execução do programa insere o bloco em que elas são definidas, existem enquanto o bloco está ativo e são destruídas quando o programa sai do bloco.
- Somente variáveis locais de uma função podem ser de classe de armazenamento automática.
- O especificador de classe de armazenamento explicitamente variáveis de classe de armazenamento automática. As variáveis locais são da classe de armazenamento automática por padrão.
- As palavras-chave `static` declaram identificadores para variáveis da classe de armazenamento estática e para funções. As variáveis de classe de armazenamento estática existem a partir do ponto em que o programa começa a execução e duram até o fim do programa.
- O armazenamento de uma variável de uma classe de armazenamento estático é alocado quando o programa começa a execução. Essa variável é inicializada uma vez quando sua declaração é encontrada. Para funções, o nome da função existe a partir do momento em que o programa começa a executar, assim como ocorre para todas as outras funções.

- Há dois tipos de identificadores com classe de armazenamento estática — os identificadores externos (como as variáveis globais e os no função globais) e as variáveis locais declaradas com o especificador da classe de armazenamento
- As variáveis globais são criadas colocando-se as declarações de variável fora de qualquer classe ou definição de função. As variáveis g retêm seus valores por toda a execução do programa. Variáveis globais e funções globais podem ser referenciadas por qualquer função q suas declarações ou definições no arquivo-fonte.
- As variáveis locais declaradas com a palavra-chave conhecidas apenas na função em que são declaradas, mas, ao contrário das variáveis automáticas, as variáveis locais ~~retêm~~ seus valores quando a função retorna para seu chamador. A próxima vez em que a função é chamada, as variáveis locais ~~lootem~~ os valores que tinham quando a função completou pela última vez a execução.
- Um identificador declarado fora de qualquer função ou classe tem escopo de arquivo.
- Os rótulos são os únicos identificadores com escopo de função. Os rótulos podem ser utilizados em qualquer lugar na função em que apas não podem ser referenciados fora do corpo da função.
- Os identificadores declarados dentro de um bloco têm escopo de bloco. O escopo de bloco começa na declaração do identificador e term chave de fechamento) do bloco em que o identificador é declarado.
- Os únicos identificadores com escopo de protótipo de função são aqueles utilizados na lista de parâmetros de um protótipo de função.
- As pilhas são conhecidas como estruturas de dados do tipo útil para armazenar o resultado de uma operação. A pilha é a estrutura inserida na pilha é o primeiro item que é removido da pilha.
- Um dos mecanismos mais importantes para ser entendido pelos alunos de ciência da computação é a pilha de chamadas de função (à referida como pilha de execução do programa). Essa estrutura de dados suporta o mecanismo chamada/retorno de função.
- A pilha de chamadas de função também suporta a criação, manutenção e destruição de variáveis automáticas de cada função chamada.
- Toda vez que uma função chama outra função, uma entrada é empurrada sobre a pilha. Esta estrutura, chamada de quadro de pilha (ou registro de ativação, contém o endereço de retorno de que a função chamada precisa para retornar à função chamadora, bem como as variáveis automáticas para a chamada de função).
- O quadro de pilha existe enquanto a função chamada estiver ativa. Quando a função chamada retornar — e não precisar mais de suas variáveis automáticas locais —, seu quadro de pilha é removido da pilha e essas variáveis automáticas locais não são mais conhecidas pelo programa.
- No C++, uma lista vazia de parâmetros é especificada inserindo nada entre parênteses.
- O C++ fornece funções inline para ajudar a reduzir o overhead da chamada de função — especialmente para funções pequenas. Colocando o关键字 inline antes do tipo de retorno de uma função na definição de função 'aconselha' o compilador a gerar uma cópia do código da função no lugar para evitar uma chamada de função.
- Duas maneiras de passar argumentos para funções em muitas linguagens de programação são a passagem por valor e a passagem por referência.
- Quando um argumento é passado por valor, uma cópia do valor do argumento é feita e passada (na pilha de chamadas de função) para a chamada. As alterações na cópia não afetam o valor da variável original no chamador.
- Com a passagem por referência, o chamador fornece à função chamada a capacidade de acessar os dados do chamador diretamente e isso se a função chamada escolher fazer isso.
- Um parâmetro de referência (*alias*, ou 'apelido', pronuncia-se 'álias') para seu argumento correspondente em uma chamada de função.
- Para indicar que um parâmetro de função é passado por referência, simplesmente coloque o parâmetro no protótipo de função; use a mesma convenção ao listar o tipo do parâmetro no cabeçalho de função.
- Uma vez que uma referência é declarada como um alias para outra variável, todas as operações supostamente realizadas no alias (isto é, referência) são realmente realizadas na variável original. O alias tem simplesmente outro nome para a variável original.
- Não é incomum que um programa invoque repetidamente uma função com o mesmo valor de argumento para um parâmetro particular. Nesses casos, o programador pode especificar que tal parâmetro tem um argumento-padrão, isto é, um valor-padrão a ser passado a esse parâmetro.
- Quando um programa omite um argumento para um parâmetro com um argumento-padrão, o compilador reescreve a chamada de função com o valor-padrão desse argumento a ser passado como um argumento para a chamada de função.
- Argumentos-padrão devem ser os argumentos mais à direita (finais) em uma lista de parâmetros da função.
- Os argumentos-padrão devem ser especificados com a primeira ocorrência do nome de função — em geral, no protótipo de função.
- O C++ fornece o operador unário de resolução de escopo `operator&` para uma variável global quando uma variável local do mesmo nome está no escopo.
- O C++ permite que várias funções do mesmo nome sejam definidas, contanto que essas funções tenham diferentes conjuntos de parâmetros. Essa capacidade é chamada de sobrecarga de funções.
- Quando uma função sobrecarregada é chamada, o compilador C++ seleciona a função adequada examinando o número, os tipos e a ordem dos argumentos na chamada.
- As funções sobrecarregadas são distinguidas por suas assinaturas.

- O compilador codifica cada identificador de função com o número e os tipos de seus parâmetros para permitir a linkagem segura para o tipo. A linkagem segura para o tipo garante que a função sobrecarregada adequada seja chamada e que os tipos dos argumentos correspondam aos dos parâmetros.
- Funções sobrecarregadas são normalmente utilizadas para realizar operações semelhantes que envolvem lógica de programa diferente e diferentes tipos de dados. Se a lógica do programa e as operações forem idênticas para cada tipo de dados, a sobrecarga pode ser realizada compacta e convenientemente utilizando templates de função.
- O programador escreve uma única definição de template de função. Dados os tipos de argumentos fornecidos em chamadas para essa função, o C++ gera automaticamente especializações separadas de template de função para tratar cada tipo de chamada apropriadamente. Portanto, um único template de função define essencialmente uma família de funções sobrecarregadas.
- Todas as definições de template de função iniciam com a palavra-chave `template`, seguida por uma lista de parâmetros de template para o template de função entre colchetes `[]` e angulares `()`.
- Os parâmetros de tipo formais são marcadores de lugar para tipos fundamentais ou tipos definidos pelo usuário. Esses marcadores de lugar são utilizados para especificar os tipos dos parâmetros da função, especificar o tipo de retorno da função e declarar variáveis dentro do corpo da definição de função.
- Uma função recursiva é uma função que chama a si mesma, direta ou indiretamente.
- Uma função recursiva sabe resolver somente o(s) caso(s) simples(s) ou o(s) chamado(s) caso(s) básico(s). Se a função é chamada com um caso básico, a função simplesmente retorna um resultado.
- Se a função for chamada com um problema mais complexo, em geral, a função divide o problema em duas partes conceituais — uma parte que ela sabe como fazer e uma parte que ela não sabe. Para tornar a recursão realizável, a última parte deve assemelhar-se ao problema original em uma versão mais simples ou menor dele.
- Para que a recursão, por fim, termine, toda vez que a função chamar a si mesma com uma versão ligeiramente mais simples do problema, se essa seqüência de problemas cada vez menor deve, finalmente, convergir para o caso básico.
- A relação de números de Fibonacci sucessivos converge para um valor constante de 1,618.... Esse número ocorre freqüentemente na natureza e foi chamado de relação áurea ou média áurea.
- A iteração e a recursão têm muitas semelhanças: ambas são baseadas em uma instrução de controle, envolvem repetição, envolvem um teste de terminação, aproximam-se gradativamente do término e podem ocorrer infinitamente.
- A recursão tem muitas desvantagens. Ela invoca repetidamente o mecanismo, e consequentemente o overhead, das chamadas de função. Isso pode representar um alto custo em tempo de processador como em espaço de memória. Cada chamada recursiva faz com que outra cópia da função (na realidade, somente as variáveis da função) seja criada; isso pode consumir memória considerável.

Terminologia

& para declarar referência	declaração de função	especificadores de classe de armazenamento
abordagem de dividir para conquistar	coerção de nome	estouro de pilha
alias	definição de função	excluir uma pilha
argumento-padrão	definição de template	expressão de tipo misto
argumentos mais à direita (finais)	desconfiguração de nome	fator de escalonamento
assinatura	deslocar um intervalo de números	fatorial
assinatura de função	efeito colateral de uma expressão	Fibonacci, série de
avaliação recursiva	enumpalavra-chave	fora de escopo
bloco externo	enumeração	função de semântico
bloco interno	escalonamento	função de template
blocos aninhados	escopo de arquivo	função definida pelo programador
caso(s) básico(s)	escopo de bloco	função definida pelo usuário
chamada recursiva	escopo de classe	função global
classe de armazenamento	escopo de função	função inline
classe de armazenamento automática	escopo de namespaces	função recursiva
classe de armazenamento estática	escopo de protótipo de função	funções 'pré-empacotadas'
coerção de argumento	escopo de um identificador	inicializar uma referência
colocar em uma pilha	especialização de template de função	line , palavra-chave
complexidade exponencial	especificador de classe de armazenamento	inteiros deslocados, escalonados
condição de terminação	auto	invocar uma função
constante enumerada	especificador de classe de armazenamento	largura
convergir para um caso básico	extern	largura de intervalo de número aleatório

LIFO	last-in, first-out	passar por referência	rótulo
limites de tamanho de inteiros	passar por valor	semente	sequência de números aleatórios
limites de tipo de dados numéricos	passo de recursão	sobrecarga de função	sobrecarregando
linkagem	pilha	solução iterativa	solução recursiva
linkagem segura para tipos	pilha de chamadas de função	srand função	static , especificador de classe de armazenamento
lista de parâmetros de template	pilha de execução do programa	static , palavra-chave	static , variável local
loop infinito	princípio do menor privilégio	template de função	template, palavra-chave
média áurea	procedimento	terminar chave direta	terminar chave direta em bloco
métodos	protótipo de função	teste de terminação	teste de terminação
modularizar um programa com funções	tipos de função obrigatórios	tipo 'mais alto'	tipo mais baixo
mutable especificador de classe de quadro de pilha	rand, função	tipo de uma variável	tipo de uma variável
armazenamento	RAND_MAX	truncar parte fracionária	truncar parte fracionária
nome de função	constante simbólica	validar uma chamada de função	validar uma chamada de função
nome de função desfigurado	randomizar	valor de deslocamento	valor de deslocamento
nome de uma variável	recursão	variável global	variável global
nome do tipo (enumerações)	recursão infinita	variável local automática	variável local automática
número aleatório	referência a uma constante	reutilização de software	reutilização de software
números pseudo-aleatórios	referência a uma variável automática		
operador unário de resolução de escopo	oscilante		
otimizar o compilador	register , especificador de classe de tipo definido pelo usuário		
overhead de chamada de função	armazenamento		
overhead de recursão	registro de ativação		
parâmetro	regras de promoção		
parâmetro de referência	relação áurea		
parâmetro de tipo	repetitividade da função		
parâmetro de tipo formal	retornar uma referência a partir de uma função		
parâmetro formal	reutilização de software		

Exercícios de revisão

6.1 Complete cada uma das sentenças:

- a) Os componentes de programa em C++ são chamados _____ e _____.
- b) Uma função é invocada com um(a) _____.
- c) Uma variável que só é conhecida dentro da função em que ela é definida é chamada de _____.
- d) A instrução _____ em uma função chamada passa o valor de uma expressão de volta à função chamadora.
- e) A palavra-chave _____ é utilizada em um cabeçalho de função para indicar que uma função não retorna um valor ou que uma função não contém parâmetros.
- f) O(A) _____ de um identificador é a parte do programa em que o identificador pode ser utilizado.
- g) As três maneiras de retornar o controle de uma função chamada a um chamar _____, _____ e _____.
- h) Um(a) _____ permite ao compilador verificar o número, tipos e ordem dos argumentos passados para uma função.
- i) A função _____ é utilizada para produzir números aleatórios.
- j) A função _____ é utilizada para configurar a semente de número aleatório para aleatorizar um programa.
- k) Os especificadores de classe de armazenamento são _____, _____ e _____.
- l) Pressupõe-se que as variáveis declaradas em um bloco ou na lista de parâmetros de uma função são da classe de armazenamento _____ a menos que especificado de outro modo.
- m) O especificador de classe de armazenamento _____ é uma recomendação para o compilador armazenar uma variável em um de registros do computador.
- n) Para uma variável local em uma função referir seu valor entre chamadas para a função, ela deve ser declarada com o especificador de classe de armazenamento _____.
- o) Os seis possíveis escopos de um identificador são _____, _____, _____, _____, _____ e _____.
- p) Uma função que chama a si própria diretamente (isto é, por outra função) é uma função _____.
- q) Uma função recursiva em geral tem dois componentes: um componente que fornece um meio para a recursão terminar testando um _____ e um que expressa o problema como uma chamada recursiva para um problema ligeiramente mais simples que a chamada original.

- s) Em C++, é possível ter várias funções com o mesmo nome que operam em diferentes tipos ou diferentes números de argumentos. I
é chamado _____ de função.
- t) O(A)_____ permite acesso a uma variável global com o mesmo nome de uma variável no escopo atual.
- u) O qualificador _____ é utilizado para declarar variáveis de leitura.
- v) Um _____ de função permite que uma única função seja definida para realizar uma tarefa em muitos tipos de dados diferentes.
- 6.2 Para o programa na Figura 6.40, declare o escopo (escopo de função, de arquivo, de bloco ou de protótipo de função) de cada um dos seguintes elementos:
- A variável `main`
 - A variável `cube`
 - A função `cube`
 - A função `main`
 - O protótipo de função para
- 6.3 Escreva um programa que testa se os exemplos da chamada de função da biblioteca de matemática mostrados na Figura 6.2 realmente produzem os resultados indicados.
- 6.4 Forneça o cabeçalho de função para cada uma das seguintes funções:
- A função `hypotenuse`, que aceita dois argumentos de ponto flutuante com dupla precisão. [Retorna um resultado de ponto flutuante com dupla precisão.]
 - A função `ceil`, que aceita três inteiros e retorna um inteiro.
 - A função `abs`, que não recebe argumentos e não retorna valores. [Estas funções são comumente utilizadas para exibir instruções ao usuário.]
 - A função `ToDouble`, que aceita um argumento de inteiro e retorna um resultado de ponto flutuante com dupla precisão.
- 6.5 Forneça o protótipo de função para cada uma das seguintes:
- A função descrita no Exercício 6.4(a). c) A função descrita no Exercício 6.4(c).
 - A função descrita no Exercício 6.4(b). d) A função descrita no Exercício 6.4(d).
- 6.6 Escreva uma declaração para cada uma das sentenças:
- O tipo `count` que deve ser mantido em um registro. [Indique]
 - A variável `val` de ponto flutuante com dupla precisão que deve reter seu valor entre chamadas para a função em que ela é definida.

```

1 // Exercício 6.2: Ex06_02.cpp
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5
6 int cube( int y ); // protótipo de função
7
8 int main()
9 {
10     int x;
11
12     for ( x = 1; x <= 10; x++ ) // itera 10 vezes
13         cout << cube( x ) << endl; // calcula cubo de x e gera a saída dos resultados
14
15     return 0; // indica terminação bem-sucedida
16 } // fim de main
17
18 // definição de função cube
19 int cube( int y )
20 {
21     return y * y * y;
22 } // fim da função cube

```

Figura 6.40 Programa para o Exercício 6.2.

- 6.7 Localize o erro em cada um dos seguintes segmentos de programa e explique como o erro pode ser corrigido (consulte também Exe 6.53):
- `int g(void)`
`{`
 `cout << "Inside function g" << endl;`
 `int h(void)`
`{`
 `cout << "Inside function h" << endl;`
`}`
`}`
 - `int sum(int x,int y)`
`{`
 `int result;`

 `result = x + y;`
`}`
 - `int sum(int n)`
`{`
 `if (n == 0)`
 `return 0;`
 `else`
 `n + sum(n - 1);`
`}`
 - `void f(double a);`
`{`
 `float a;`
 `cout << a << endl;`
`}`
 - `void product(void)`
`{`
 `int a;`
 `int b;`
 `int result;`
 `cout << "Enter three integers: " ;`
 `cin >> a >> b >> c;`
 `result = a * b * c;`
 `cout << "Result is " << result;`
 `return result;`
`}`
- 6.8 Por que um protótipo de função conteria uma declaração de tipo de parâmetro como `int`?
- 6.9 (Verdadeiro/Falso) Todos os argumentos para as chamadas de função em C++ são passados por valor.
- 6.10 Escreva um programa completo que solicita ao usuário o raio de uma esfera e calcula e imprime o volume dessa esfera. Utilize uma função `sphereVolume` que retorna o resultado da seguinte expressão: $3.14159 \times \text{pow}(\text{radius}, 3)$.

Respostas dos exercícios de revisão

- 6.1 a) funções, classes. b) chamada de função. c) variável local. d) escopo de retorno; return; encontrar a chave direita de fechamento de uma função. h) protótipo de função, register, extern, static, l) auto, m) register. n) global, static. o) escopo de função, escopo de arquivo, escopo de bloco, escopo de protótipo de função, escopo de classe, escopo de namespaces. q) recursiva. r) básico. s) sobre carga. t) operador un de resolução de escopo. v) template.
- 6.2 a) escopo de bloco. b) escopo de bloco. c) escopo de arquivo. d) escopo de arquivo. e) escopo de arquivo. f) escopo de protótipo de função.
- 6.3 Veja o seguinte programa:

```

1 // Exercício 6.3: Ex06_03.cpp
2 // Testando a biblioteca de funções matemáticas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using namespace std;
13
14 int main()
15 {
16     cout << fixed << setprecision( 1 );
17
18     cout << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
19     << endl;
20     cout << "nsqrt(" << 9.0 << ") = " << sqrt( 9.0 );
21     cout << "nexp(" << 1.0 << ") = " << setprecision( 6 )
22     << exp( 1.0 ) << endl;
23     cout << "nexp(" << setprecision( 6 ) << exp( 2.0 );
24     cout << "nlog(" << 2.718282 << ") = " << setprecision( 1 )
25     << log( 2.718282 );
26     cout << "nlog(" << setprecision( 6 ) << 7.389056 << ") = "
27     << setprecision( 1 ) << log( 7.389056 );
28     cout << "nlog10(" << 1.0 << ") = " << log10( 1.0 )
29     cout << "nlog10(" << 10.0 << ") = " << log10( 10.0 )
30     cout << "nlog10(" << 100.0 << ") = " << log10( 100.0 );
31
32     cout << "nfabs(" << 0.5 << ") = " << fabs( 0.5 );
33     cout << "nceil(" << 9.2 << ") = " << ceil( 9.2 )
34     << endl;
35     cout << "nceil(" << -9.8 << ") = " << ceil( -9.8 );
36     cout << "nfloor(" << 9.2 << ") = " << floor( 9.2 )
37     << endl;
38     cout << "nfloor(" << -9.8 << ") = " << floor( -9.8 );
39     cout << "npow(" << 2.0 << ", " << 7.0 << ") = "
40     << pow( 2.0, 7.0 ) << endl;
41     cout << "npow(" << 9.0 << ", "
42     << 0.5 << ") = " << pow( 9.0, 0.5 );
43     cout << setprecision( 3 ) << "nmod("
44     << 13.675 << ", " << 2.333 << ") = "
45     << fmod( 13.675, 2.333 ) << setprecision( 1 );
46     cout << "nsin(" << 0.0 << ") = " << sin( 0.0 );
47     cout << "ncos(" << 0.0 << ") = " << cos( 0.0 );
48     cout << "ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
49
50     return 0; // indica terminação bem-sucedida
51 } // fim de main

```

sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0

```

fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0

```

- 6.4 a) `double hypotenuse(double side1, double side2)`
 b) `int smallest(int x, int y, int z)`
 c) `void instructions(void)` // em C++ (`void`) pode ser escrito ()
 d) `double intToDouble(int number)`
- 6.5 a) `double hypotenuse(double, double);`
 b) `int smallest(int, int, int);`
 c) `void instructions(void);` // em C++ (`void`) pode ser escrito ()
 d) `double intToDouble(int);`
- 6.6 a) `register int count = 0;`
 b) `static double lastVal;`
- 6.7 a) Erro: A função é definida na função
 Correção: Mova a definição para fora da definição de
 b) Erro: A função supostamente deve retornar um inteiro, mas não o faz.
 Correção: Exclua a ~~return~~ e coloque a seguinte instrução na função:
`return x + y;`
 c) Erro: O resultado de `sum(n - 1)` não é retornado. Isso retorna um resultado inadequado.
 Correção: Reescreva a instrução ~~return sum(n - 1)~~
`return n + sum(n - 1);`
 d) Erros: O ponto-e-vírgula depois do parêntese direito que inclui a lista de parâmetros define o parâmetro função.
 Correções: Exclua o ponto-e-vírgula após o parêntese direito da lista de parâmetros e exclua a declaração
 e) Erro: A função retorna um valor quando supostamente não deveria.
 Correção: Elimine a instrução
- 6.8 Isso cria um parâmetro de referência do tipo ~~const~~ para achar que muda à função modificar a variável `srcinal` na função chama-dora.
- 6.9 Falso. O C++ permite passar por referência utilizando parâmetros de referência (e ponteiros, como discutimos no Capítulo 8).
- 6.10 Veja o seguinte programa:

```

1 // Solução do Exercício 6.10: Ex06_10.cpp
2 // Função inline que calcula o volume de uma esfera.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
7
8 #include <cmath>
9 using std::pow;
10
11 const double PI = 3.14159 // define a constante global PI
12
13 // calcula o volume de uma esfera

```

```

14 inline double sphereVolume(const double radius )
15 {
16     return 4.0 / 3.0 * PI * pow( radius, 3 );
17 } // fim da função inline sphereVolume
18
19 int main()
20 {
21     double radiusValue;
22
23     // solicita o raio ao usuário
24     cout << "Enter the length of the radius of your sphere: " ;
25     cin >> radiusValue; // insere o raio
26
27     // utiliza radiusValue para calcular volume da esfera e exibe o resultado
28     cout << "Volume of sphere with radius " << radiusValue
29     << " is " << sphereVolume( radiusValue ) << endl;
30
31 } // fim de main

```

Exercícios

- 6.11 Mostre o valor de `abs` depois que cada uma das seguintes instruções for realizada:
- $x = \text{fabs}(7.5)$
 - $x = \text{floor}(-7.5)$
 - $x = \text{fabs}(0.0)$
 - $x = \text{ceil}(0.0)$
 - $x = \text{fabs}(-6.4)$
 - $x = \text{ceil}(-6.4)$
 - $x = \text{ceil}(-\text{fabs}(-8 + \text{floor}(-5.5)))$
- 6.12 ~~Uma estacionária cobrará \$2.00 para estacionar por até 1 hora, e \$0.50 por hora adicional. Se nenhum carro fique estacionado por mais de 24 horas por vez. Escreva um programa que calcula e imprime os custos de estacionamento de cada um dos três clientes que estacionou o carro nessa garagem ontem. Você deve inserir as horas de estacionamento para cada cliente. Seu programa deve imprimir os resultados em um formato tabular elegante e deve calcular e imprimir o total dos recibos de ontem. Cada cliente pagará \$2.00 para a primeira hora e \$0.50 por hora adicional.~~ Crie um programa que calcule o valor da tarifa de estacionamento para determinar a tarifa para cada cliente. Suas saídas devem aparecer no seguinte formato:
- | Car | Hours | Charge |
|-------|-------|--------|
| 1 | 1.5 | 2.00 |
| 2 | 4.0 | 2.50 |
| 3 | 24.0 | 10.00 |
| TOTAL | 29.5 | 14.50 |
- 6.13 Uma aplicação da função `ceil` arredonda um valor para o inteiro mais próximo. A instrução
 $y = \text{ceil}(x + .5);$
- arredonda o número para o inteiro mais próximo e atribui o resultado. Escreva um programa que lê vários números e utiliza a instrução anterior para arredondar cada um desses números para o inteiro mais próximo. Para cada número processado, imprima ambos os números, o scinal e o arredondado.
- 6.14 A função `floor` pode ser utilizada para arredondar um número para uma casa decimal específica. A instrução
 $y = \text{floor}(x * 10 + .5) / 10;$
- arredonda para a casa decimal (a primeira posição à direita do ponto de fração decimal). A instrução
 $y = \text{floor}(x * 100 + .5) / 100$
- arredonda para a casa dos centésimos (isto é, a segunda posição à direita do ponto de fração decimal). Escreva um programa que define quatro funções para arredondar de várias maneiras:

- a) roundToInteger(number)
- b) roundToTenths(number)
- c) roundToHundredths(number)
- d) roundToThousands(number)

Para cada valor lido, seu programa deve imprimir o valor decimal, o número arredondado para o inteiro mais próximo, o número arredondado para o décimo mais próximo, o número arredondado para o centésimo mais próximo e o número arredondado para o milésimo mais próximo.

- 6.15 Responda a cada uma das seguintes perguntas:
- a) O que significa escolher números 'aleatoriamente'?
 - b) Por que a função é útil para simular jogos de azar?
 - c) Por que você poderia querer aleatorizar um programa que simula circunstâncias é desejável não aleatorizar?
 - d) Por que frequentemente é necessário escalonar ou deslocar os valores produzidos por
- 6.16 e) Por que a simulação computadorizada de situações do mundo real é uma técnica útil?
- Escreva instruções que atribuem inteiros aleatórios entre 0 e 100 a variáveis de intervalos:
- a) $1 \leq n \leq 2$
 - b) $1 \leq n \leq 100$
 - c) $0 \leq n \leq 9$
 - d) $1.000 \leq n \leq 1.112$
 - e) $-1 \leq n \leq 1$
 - f) $-3 \leq n \leq 11$
- 6.17 Para cada um dos seguintes conjuntos de inteiros, escreva uma única instrução que aleatoriamente imprime um número do conjunto:
- a) 2, 4, 6, 8, 10.
 - b) 3, 5, 7, 9, 11.
 - c) 6, 10, 14, 18, 22.
- 6.18 Escreva uma função `Power(base, exponent)` que retorna o valor de $\text{base}^{\text{exponent}}$
- Por exemplo `integerPower(3, 4) = 3 * 3 * 3 * 3`. Pressuponha que o expoente seja um inteiro não-zero positivo e que a base seja um inteiro. A função `Power` deve utilizar um loop `while` para controlar o cálculo. Não utilize funções da biblioteca matemática.
- 6.19 (Hipotenusa) Defina uma função que calcula o comprimento da hipotenusa de um triângulo reto quando os outros dois lados são dados. Utilize essa função em um programa para determinar o comprimento da hipotenusa para cada um dos triângulos mostrados abaixo. A função deve aceitar dois argumentos e retornar a hipotenusa.
- | Triângulo | Lado 1 | Lado 2 |
|-----------|--------|--------|
| 1 | 3.0 | 4.0 |
| 2 | 5.0 | 12.0 |
| 3 | 8.0 | 15.0 |
- 6.20 Escreva uma função que determina para um par de inteiros se o segundo inteiro é um múltiplo do primeiro. A função deve aceitar dois argumentos inteiros `primeiro` e `segundo` para um múltiplo de `primeiro` contrário. Utilize essa função em um programa que insere uma série de pares de inteiros.
- 6.21 Escreva um programa que insere uma série de inteiros e os passa para a função `par` para determinar se um inteiro é par. A função deve aceitar um argumento inteiro e retornar caso contrário.
- 6.22 Escreva uma função que exibe na margem esquerda da tela um quadrado sólido de asteriscos cujo lado é especificado no parâmetro tipo inteiro `side`. Por exemplo, se `for 4`, a função exibirá o seguinte:
- ```


```
- 6.23 Modifique a função criada no Exercício 6.22 para formar o quadrado a partir de qualquer caractere contido no parâmetro caractere `Character`. Portanto, se `fillCharacter` for '#, então essa função deve imprimir o seguinte:

#####  
#####  
#####  
#####  
#####

- 6.24 Utilize técnicas semelhantes àquelas desenvolvidas nos exercícios 6.22 e 6.23 para produzir um programa que representa graficamente uma ampla variedade de formas.

6.25 Escreva segmentos de programa que realizam cada uma das seguintes instruções:

  - Calcule a parte inteira do quociente quando dividido pelo inteiro
  - Calcule o resto inteiro quando dividido pelo inteiro
  - Utilize as partes do programa desenvolvido em (a) e (b) para escrever uma função que imprime um inteiro entre 1562 e 15620. Por exemplo, o resultado da seguinte maneira:

4 5 6 2

6.26 Escreva uma função que aceita a hora como três argumentos do tipo inteiro (horas, minutos e segundos) e retorna o número de segundos desde a última vez que o relógio 'deu 12'. Utilize essa função para calcular a quantidade de tempo em segundos entre duas horas, am as quais estão dentro de um ciclo de 12 horas.

6.27 (Temperaturas Celsius e Fahrenheit) Escreva as seguintes funções para trabalhar com inteiros:

  - A função  $\text{Celsius}$  retorna o equivalente em Celsius de uma temperatura em Fahrenheit.
  - A função  $\text{Fahrenheit}$  retorna o equivalente em Fahrenheit de uma temperatura em Celsius.
  - Utilize essas funções para escrever um programa que imprime gráficos para mostrar os equivalentes em Fahrenheit de todas as temperaturas em Celsius de 0 a 100 graus e os equivalentes em Celsius de todas as temperaturas em Fahrenheit de 32 a 212 graus. Imprima as saídas em um formato tabular elegante que minimiza o número de linhas de saída mas permanece legível.

6.28 Escreva um programa que insere três números de ponto flutuante com dupla precisão e os passa para uma função que retorna o menor número.

6.29 (Números perfeitos) Dizemos que um inteiro é um número perfeito se a soma de seus fatores, incluindo 1 (mas não o próprio número), é igual ao número. Por exemplo, 6 é um número perfeito porque  $1 + 2 + 3 = 6$ . Escreva uma função que determina se um número é perfeito. Utilize essa função em um programa que determina e imprime todos os números perfeitos entre 1 e 1.000. Imprima os fatores de cada número perfeito para confirmar se o número é de fato perfeito. Desafie o poder de seu computador testando números muito maiores que 1.000.

6.30 (Números primos) Dizemos que um inteiro é primo se ele é divisível somente por 1 e ele próprio. Por exemplo, 2, 3, 5 e 7 são primos, mas 4, 6, 8 e 9 não o são.

  - Escreva uma função que determina se um número é primo.
  - Utilize essa função em um programa que determina e imprime todos os números primos entre 2 e 10.000. Quantos desses 10.000 números você realmente tem de testar antes de certificar-se de que encontrou todos os primos?
  - Inicialmente você poderia ir até a raiz quadrada de quê? Reescreva o programa e execute-o de ambas as maneiras. Estime o melhor desempenho.

6.31 (Dígitos invertidos) Escreva uma função que aceita um valor inteiro e retorna o número com seus dígitos invertidos. Por exemplo, dado o número 7.631, a função deve retornar 1.367.

6.32 (Máximo divisor comum) O maior divisor comum de dois inteiros é o maior inteiro que é divisível por cada um dos dois números. Escreva uma função que retorna o máximo divisor comum de dois inteiros.

6.33 Escreva uma função que insere a média de um aluno e retorna 4 se a média do aluno for 90–100, 3 se a média for 80–89, 2 se a média for 70–79, 1 se a média for 60–69 e 0 se a média for mais baixa que 60.

6.34 Escreva um programa que simula o lançamento de uma moeda. Para cada lançamento de moeda, o programa deve imprimir Head ou Tails (cara ou coroa). Deixe o programa lançar a moeda 100 vezes e conte o número de vezes que cada lado da moeda aparece. Imprima os resultados. O programa deve chamar `separado` que não aceita nenhum argumento e retorna para.

[Nota: Se o programa simular realistamente o lançamento de uma moeda, cada lado da moeda deve aparecer aproximadamente metade das vezes.]

6.35 (Computadores na educação) Os computadores estão desempenhando um papel crescente na educação. Escreva um programa que ajuda um aluno da escola primária a aprender multiplicação. O programa deve gerar dois inteiros a partir de um algarismo positivo. Então ele deve digitar uma pergunta como

Quanto é 6 vezes 7?

Em seguida, o aluno digita a resposta. Seu programa verifica a resposta do aluno. Se estiver correto, imprima faça outra pergunta de multiplicação. Se a resposta estiver errada, imprima então deixe o aluno tentar a mesma pergunta repetidamente até que, por fim, ele consiga acertar o número.

- 6.36 (Instrução auxiliada por computador) Um problema que se desenvolve em ambientes CAI é a fadiga do aluno. Isso pode ser eliminado variando o diálogo do computador para prender a atenção do aluno. Modifique o programa do Exercício 6.35 de modo que os vários comentários sejam impressos para cada resposta correta e cada resposta incorreta como segue:

Réplicas para uma resposta correta:

Muito bom!

Excelente!

**Bom trabalho!**

Réplicas para uma resposta incorreta

Não. Tente novamente.

Errado. Tente mais uma vez.

Não desista!

Não. Continue tentando.

Utilize o gerador de números aleatórios para escolher um número de 1 a 4 a fim de selecionar uma réplica apropriada a cada resposta. Utilize uma instrução `switch` para emitir as respostas.

- 6.37 Sistemas mais sofisticados de instrução auxiliada por computador monitoram o desempenho do aluno durante um período de tempo. A decisão de iniciar um novo tópico é freqüentemente baseada no sucesso do aluno com tópicos anteriores. Modifique o programa do Exercício 6.36 para contar o número de respostas corretas e incorretas digitadas pelo aluno. Depois que o aluno digitar 10 respostas, o programa deve calcular a porcentagem de respostas corretas. Se a porcentagem for menor que 75%, seu programa deve imprimir ao seu professor uma ajuda extra" e terminar.

- 6.38 (Jogo 'Adivinhe o número') Escreva um programa que joga 'adivinhe o número' como mostrado a seguir: Seu programa escolhe o número a ser adivinhado selecionando um inteiro aleatoriamente no intervalo de 1 a 1.000. O programa então exibe o seguinte:

Tenho um número entre 1 e 1000.

Você consegue adivinhar-lo?

Digite sua primeira suposição.

O jogador então digita uma primeira suposição. O programa responde com uma das seguintes frases:

1. Excelente! Você adivinhou o número!  
Quer jogar de novo (s ou n)?
2. Muito baixo. Tente novamente.
3. Muito alto. Tente novamente.

Se a suposição do jogador estiver incorreta, o programa deve fazer um loop até o jogador por fim acertar o número. Seu programa deve continuar dizendo ao jogador "Muito baixo" ou "Muito alto" para ajudar o jogador a acertar a resposta.

- 6.39 Modifique o programa do Exercício 6.38 para contar o número de suposições que o jogador faz. Se o número for 10 ou menor, imprimir "Você sabe o segredo ou teve sorte!". Se o jogador adivinhar o número em 10 tentativas, imprimir "Parabéns! Você é o segredo!". Se o jogador fizer mais de 10 suposições, imprimir "Parece que você não é capaz de fazer melhor!". Por que não deve haver mais de 10 suposições? Bem, a cada 'boa suposição' o jogador deve ser capaz de eliminar metade dos números. Agora não por que qualquer número de 1 a 1.000 pode ser adivinhado em 10 ou menos tentativas.

- 6.40 Escreva uma função `power(base, exponent)` que, quando invocada, retorna  $base^{exponent}$

Por exemplo  $power(3, 4) = 3 * 3 * 3 * 3$ . Suponha que  $exponent$  é um inteiro maior que ou igual a 1. Dica: Faça uso de recursão utilizando o relacionamento

$$base^{exponent} = base \cdot base^{exponent-1}$$

e a condição de terminação ocorre quando porque

$$base^1 = base$$

## 6.41 Série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com os termos 0 e 1 e tem a propriedade de que cada termo sucessivo é a soma dos dois termos precedentes. (a) Escreva uma função recursiva  $fibonacci(n)$  que calcule o número de Fibonacci. (b) Determine o maior número de Fibonacci impresso no seu sistema. Modifique o programa da parte (a) para fazer de calcular e retornar números de Fibonacci e utilize esse programa modificado para repetir a parte (b).

6.42 Torres de Hanói

Neste capítulo, você estudou funções que podem ser facilmente implementadas tanto recursivamente quanto iterativamente. Neste exercício, apresentamos um problema cuja solução recursiva demonstra a elegância da recursão, e cuja solução iterativa pode ser tão evidente.

As Torres de Hanói são um dos problemas clássicos mais famosos com o qual todo cientista da computação deve lidar. Diz a lenda que, em um templo no Extremo Oriente, os sacerdotes tentavam mover uma pilha de discos de ouro a partir de um pino de diamantes para

O problema das torres de Hanói é um problema de matemática proposicional que envolve o deslocamento de discos de ouro de um pino para outro. A história diz que os discos foram criados por deuses milhares de anos atrás para desvendar o mistério do universo. Nenhuma circunstância, um disco maior poderia ser colocado em cima de um disco menor. Três pinos eram fornecidos e um deles era utilizado para armazenar discos temporariamente. Supostamente, o mundo acabará quando os sacerdotes completarem sua tarefa, por isso há pouco incentivo para facilitarmos seus esforços.

Vamos assumir que os sacerdotes estão tentando mover os discos do pino 1 para o pino 3. Desejamos desenvolver um algoritmo que imprima a seqüência precisa de transferências de discos de um pino para outro.

Se abordássemos esse problema com métodos convencionais, rapidamente ficaríamos desesperados gerenciando os discos. Em vez disso, abordar esse problema com a recursão em mente permite que os discos sejam sempre movidos em termos de mover somente discos (e daí a recursão), como segue:

- Mova 1 disco do pino 1 para o pino 2, utilizando o pino 3 como área de armazenamento temporário.
- Mova o último disco (o maior) do pino 1 para o pino 3.
- Mova 1 disco do pino 2 para o pino 3, utilizando o pino 1 como área de armazenamento temporário.

O processo termina quando a última tarefa é realizada (isso é, o caso básico). Essa tarefa é realizada simplesmente movendo o disco, sem a necessidade de uma área de armazenamento temporário.

Escreva um programa para resolver o problema Torres de Hanói. Utilize uma função recursiva com quatro parâmetros:

- O número de discos a serem movidos.
- O pino para o qual esses discos devem ser finalmente empilhados.
- O pino para o qual esses discos devem ser inicialmente empilhados.
- O pino a ser utilizado como área de armazenamento temporário.

Seu programa deve imprimir as instruções precisas que ele usará para mover os discos do pino inicial para o pino de destino. Por exemplo, para mover uma pilha de três discos do pino 1 para o pino 3, seu programa deve imprimir a seguinte série de movimentos:

- 1- 3 (Isso quer dizer mover um disco do pino 1 para o pino 3.)
- 1- 2
- 3- 2
- 1- 3

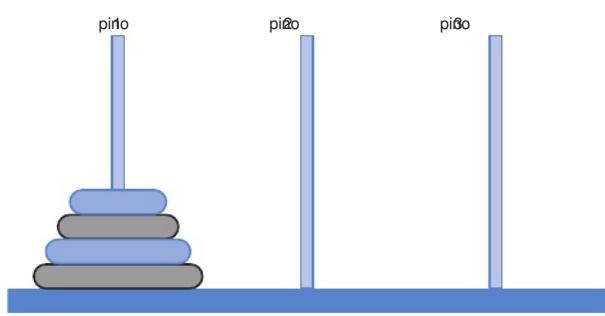


Figura 6.41 Torres de Hanói para o caso com quatro discos.

2-  
2-  
1-

- 6.43 Qualquer programa que pode ser implementado recursivamente pode ser implementado iterativamente, embora às vezes com mais dificuldade e menos clareza. Tente escrever uma versão iterativa das Torres de Hanói. Se você for bem-sucedido, compare sua versão iterativa com a versão recursiva desenvolvida no Exercício 6.42. Investigue questões de desempenho, clareza e sua capacidade de demonstrar a correção dos programas.

6.44 Visualizando a recursão é interessante observar a recursão ‘em ação’. Modifique a função factorial da Figura 6.29 para imprimir sua variável local e seu parâmetro de chamada recursiva. Para cada chamada recursiva, exiba as saídas em uma linha separada e adicione um nível de recuo. Faça o melhor que você puder para tornar a saída limpa, interessante e significativa. Seu objetivo aqui é projetar e implementar um formato de saída que ajude uma pessoa a entender melhor a recursão. Você pode querer adicionar essas capacidades de exibição aos muitos outros exemplos e exercícios de recursão ao longo de todo este texto.

6.45 (Máximo divisor comum) Escreva uma função recursiva que retorna o máximo divisor comum do menor inteiro que é divisível por x e y. Escreva uma função recursiva que retorna o máximo divisor comum dividindo recursivamente como mostrado na figura. Se  $x \geq y$ , então  $\text{mdc}(x, y)$  será caso contrário,  $\text{mdc}(y, x \% y)$ , onde o operador módulo é %. Para esse algoritmo deve ser maior que zero.

6.46 A função main pode ser chamada recursivamente em seu sistema? Escreva um programa principal iterativo que usa a função `static count` e a inicialize como 1. Pós-incremente e imprima a maior vez que for chamada. Compile seu programa. O que acontece?

6.47 Os exercícios 6.35–6.37 desenvolveram um programa de instrução auxiliada por computador para ensinar multiplicação a um aluno da escola primária. Este exercício sugere aprimoramentos nesse programa.

  - Modifique o programa para permitir que o usuário insira uma capacidade de nível de graduação. O nível 1 significa utilizar somente números de um único dígito nos problemas, o nível 2 significa utilizar números com dois dígitos etc.
  - Modifique o programa para permitir que o usuário selecione os tipos de problemas aritméticos que ele ou ela deseja estudar. A opção 1 significa problemas de adição somente, 2 significa problemas de subtração somente, 3 significa problemas de multiplicação somente, 4 significa problemas de divisão somente e 5 significa problemas de todos esses tipos misturados aleatoriamente.

6.48 Escreva uma função que calcula a distância entre dois pontos. (Todos os números e valores de retorno devem ser do tipo double.)

6.49 O que há de errado com o seguinte programa?

```
1 // Exercício 6.49: ex06_49.cpp
2 // O que há de errado com esse programa?
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 int main()
8 {
9 int c;
10
11 if ((c = cin.get()) != EOF)
12 {
13 main();
14 cout << c;
15 } // fim do if
16
17 return 0; // indica terminação bem-sucedida
18 } // fim de main
```

6.50 O que o seguinte programa faz?

```

5 using std::cin;
6 using std::endl;
7
8 int mystery(int , int); // protótipo de função
9
10 int main()
11 {
12 int x, y;
13
14 cout << "Enter two integers: " ;
15 cin >> x >> y;
16 cout << "The result is " << mystery(x, y) << endl;
17
18 return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // O parâmetro B deve ser um inteiro positivo para evitar recursão infinita
22 int mystery(int a, int b)
23 {
24 if (b == 1) // caso básico
25 return a;
26 else // passo de recursão
27 return a + mystery(a, b - 1);
28 } // fim da função mystery

```

- 6.51 Depois de determinar o que o programa do Exercício 6.50 faz, modifique o programa para funcionar adequadamente depois de remover a restrição de o segundo argumento ser não negativo.
- 6.52 Escreva um programa que testa o maior número de funções da biblioteca de matemática na Figura 6.2 que você puder. Exercite cada uma dessas funções fazendo seu programa imprimir tabelas de valores de retorno para uma diversidade de valores de argumento.
- 6.53 Localize o erro em cada um dos seguintes segmentos de programa e explique como corrigi-los:

a) ~~double cube( float number )~~; proposta: `double cube( float number )`

```

double cube(float number) // definição de função
{
 return number * number * number;
}

```

b) ~~register auto int x = 7;~~

c) ~~int randomNumber = srand();~~

d) ~~float y = 123.45678;~~

e) ~~double square( double number )~~

```

double square(double number)
{
 double number;
 return number * number;
}

```

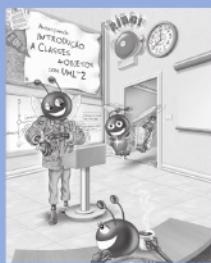
f) ~~int sum( int n )~~

```

int sum(int n)
{
 if (n == 0)
 return 0;
 else
 return n + sum(n);
}

```

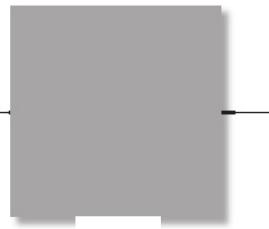
- 6.54 Modifique o programa de jogo de dados da Figura 6.11 para permitir apostas. Empacote como uma função a parte do programa que executa o jogo de dados. Inicialize a variável `bankBalance` como 1.000 dólares. Peça para o jogador inserir um valor. Utilize um loop para verificar se o valor é menor que ou igual a `bankBalance`, se não for, solicite ao usuário que inserir novamente até que um valor válido ser inserido. Depois que o valor foi inserido, execute um jogo de dados. Se o jogador ganhar, aumente `bankBalance`; poragere imprima o novo `bankBalance`. Se o jogador perder, diminua `bankBalance`; poragere imprima o novo `bankBalance` e verifique se `bankBalance` tornou-se zero e, se o for, imprima a mensagem "[Desculpe. Você perdeu!]". Enquanto o jogo continua, imprima várias mensagens para criar alguns diálogos como "broke, huh?" ["Oh, parece que você vai quebrar, hein?"] ou "Aw c'mon, take a chance!" ["Ah, vamos lá, dê uma chance para sua sorte!"] ou "You're up big. Now's the time to cash in your chips!" [Você está montado na grana. Agora é hora de trocar essas fichas e embolsar o dinheiro!"] .
- 6.55 Escreva um programa C++ que solicita ao usuário o raio de um círculo, calcule a área desse círculo.
- 6.56 Escreva um programa C++ completo com as duas funções alternativas especificadas a seguir: cada uma das quais simplesmente tripla o valor de seu argumento. Esse é o teste das abordagens. Essas duas funções são:
- a função `tripleByValue` que passa uma cópia do valor, triplica a cópia e retorna o novo valor e
  - b) a função `tripleByReference` que passa por referência via um parâmetro de referência e triplica o valor original de seu alias (isto é, o parâmetro de referência).
- 6.57 Qual é o propósito do operador unário de resolução de escopo?
- 6.58 Escreva um programa que utiliza um template para determinar o menor de dois argumentos. Teste o programa utilizando argumentos do tipo inteiro, caractere e número de ponto flutuante.
- 6.59 Escreva um programa que utiliza um template para determinar o maior de três argumentos. Teste o programa utilizando argumentos do tipo inteiro, caractere e número de ponto flutuante.
- 6.60 Determine se os seguintes segmentos de programa contêm erros. Para cada erro, explique como ele pode ser corrigido. [Cada segmento de programa particular, é possível que nenhum erro esteja presente no segmento.]
- `template < class A >`  
`int sum( int num1, int num2, int num3 )`  
`{`  
 `return num1 + num2 + num3;`  
`}`
  - `void printResults( int x, int y )`  
`{`  
 `cout << "The sum is " << x + y << '\n' ;`  
 `return x + y;`  
`}`
  - `template < A >`  
`A product( A num1A num2A num3 )`  
`{`  
 `return num1 * num2 * num3;`  
`}`
  - `double cube( int );`  
`int cube( int );`



Vai pois agora, escreve isso  
numa tábua perante eles,  
registra-o num livro.  
Isaías 30:8

Ir além é tão incerto quanto não  
alcançar o objetivo.  
Confúcio

Comece pelo começo, ... e vá  
até ao fim: então, pare.  
Lewis Carroll



## Arrays e vetores

### OBJETIVOS

Neste capítulo, você aprenderá:

A utilizar a estrutura de dados de array para representar um conjunto de itens de dados relacionados.

A utilizar arrays para armazenar, classificar e pesquisar listas e tabelas de valores.

Como declarar arrays, inicializar arrays e referenciar elementos individuais de arrays.

Como passar arrays para funções.

A utilizar técnicas básicas de pesquisa e classificação.

Como declarar e manipular arrays multidimensionais.

A utilizar o template `vector` da C++ Standard Library.

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| 7.1  | Introdução                                                                                    |
| 7.2  | Arrays                                                                                        |
| 7.3  | Declarando arrays                                                                             |
| 7.4  | Exemplos que utilizam arrays                                                                  |
| 7.5  | Passando arrays para funções                                                                  |
| 7.6  | Estudo de caso: classe GradeBook utilizando um array para armazenar notas                     |
| 7.7  | Pesquisando arrays com pesquisa linear                                                        |
| 7.8  | Classificando arrays por inserção                                                             |
| 7.9  | Arrays multidimensionais                                                                      |
| 7.10 | Estudo de caso: classe GradeBook utilizando um array bidimensional                            |
| 7.11 | Introdução ao template vector da C++ Standard Library                                         |
| 7.12 | Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional) |
| 7.13 | Síntese                                                                                       |

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) |  
[Exercícios](#) | [Exercícios com recursão](#) | [Exercícios avançados](#)

## 7.1 Introdução

Este capítulo introduz o importante [tópico de estruturas de dados](#) — coleções de itens de dados [relacionados](#) — estruturas de dados consistindo em itens de dados relacionados do mesmo tipo. Você aprendeu sobre classes no Capítulo 3. No Capítulo 9, a noção de [estruturas](#) Estruturas e classes são capazes de armazenar itens de dados relacionados de tipos possivelmente diferentes. Arrays, estruturas e classes são entidades ‘estáticas’ no sentido de que permanecem com o mesmo tamanho por toda a execução do programa. (Naturalmente, elas podem ser de uma classe de armazenamento automático e daí serem criadas e destruídas todo fluxo do programa entra e sai dos blocos em que elas são definidas.)

Depois de discutir como os arrays são declarados, criados e inicializados, este capítulo apresenta uma série de exemplos que demonstram várias manipulações comuns de array. Em seguida, explicamos como strings de caracteres (representadas até [objetos](#)) também podem ser implementadas por arrays de caracteres. Apresentaremos um exemplo de pesquisa de string, para é, colocar os dados em alguma ordem particular). Duas seções do capítulo [apresentam estudos de caso](#) da classe 3–5. Em particular, utilizamos arrays para permitir que a classe mantenha um conjunto de notas na memória e analise as notas a partir de múltiplos exames em um semestre — duas capacidades que não apareceram nas versões anteriores da classe e outros exemplos do capítulo demonstram como os arrays permitem que os programadores organizem e manipulem dados.

O estilo de arrays que utilizamos na maior parte deste capítulo é baseado em ponteiro no estilo do C. (Estudaremos ponteiro no Capítulo 8.) Na seção final deste capítulo e no Capítulo 23, “Standard Template Library (STL)”, abordaremos arrays como objetos completos, com todos os recursos, chamados vetores. Descobriremos que esses arrays baseados em objetos são mais seguros e versáteis que os arrays no estilo do C, baseados em ponteiros, que discutimos na primeira parte deste capítulo.

## 7.2 Arrays

Um array é um grupo consecutivo de posições da memória em que todas elas são do mesmo tipo. Para referir-se a uma particulação ou elemento no array, especificamos o nome do array e o elemento particular no array.

A Figura 7.1 mostra um array de inteiros chamado `contem2s`. Um programa pode referenciar qualquer um desses elementos dando o nome do array seguido pelo número da posição do elemento. Com parênteses e chaves (ou colchetes) é chamado mais formalmente de [subscrito de índice](#) (esse número especifica o número de elementos a partir do início do array). O primeiro elemento em cada array é o zero, e às vezes é chamado de [elemento 0](#). Portanto, os elementos do array

`c[0]` (pronuncia-se ‘sub zero’), `c[1]`, e assim por diante. O subscrito maior é menor que `c[12]`, o número de elementos no array. Os nomes de array seguem as mesmas convenções que outros nomes de variável, isto é, devem ser identificadores.

Um subscrito deve ser um inteiro ou uma expressão do tipo inteiro (que utiliza qualquer tipo inteiro). Se um programa utilizar expressão como um subscrito, então o programa avalia a expressão para determinar o subscrito. Por exemplo, se supormos que `a` é igual a 10, a instrução

```
c[a + b] += 2;
```

adiciona 2 ao elemento `c[11]`. Observe que o nome de um array [subscrito](#) pode ser utilizado no lado esquerdo de uma atribuição, exatamente como podem os nomes de variáveis que não pertencem a um array.

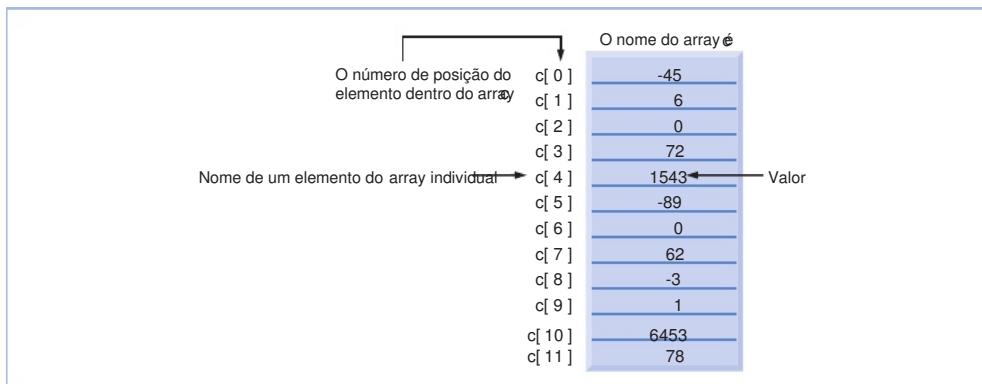


Figura 7.1 Array de 12 elementos.

Vamos examinar [Figura 7.1](#) mais atentamente. Um array de inteiros com 12 elementos das arrays referenciados como `c[ 0 ], c[ 1 ], ..., c[ 11 ]`. O valor de `c[ 0 ]` é -45, o valor de `c[ 1 ]` é 6, o valor de `c[ 2 ]` é 0, o valor de `c[ 3 ]` é 72 e o valor de `c[ 4 ]` é 1543. Para imprimir a soma dos valores contidos nos primeiros três elementos do array escreveríamos

```
cout << c[0] + c[1] + c[2] << endl;
```

Para dividir o valor depois e atribuir o resultado à variável `x` escreverímos

```
x = c[6] / 2;
```



### Erro comum de programação 7.1

É importante notar a diferença entre 'o sétimo elemento do array' e 'elemento 7 do array'. Subscritos de array iniciam em 0, portanto 'o sétimo elemento do array' tem um subscrito de 6, enquanto o 'elemento 7 do array' tem um subscrito de 7 e na realidade é o oitavo elemento do array. Infelizmente, essa distinção freqüentemente é uma fonte de erros off-by-one. Para evitar esses erros, referenciamos elementos do array específicos explicitamente pelo seu nome de array e número de subscrito (por exemplo, `c[ 7 ]`).

Os colchetes utilizados para incluir o subscrito de um array são realmente um operador em C++. Os colchetes têm o mesmo r precedência que os parênteses. A Figura 7.2 mostra a precedência e a associatividade dos operadores introduzidos até agora. C foram adicionados colchetes na primeira linha da Figura 7.2. Os operadores são mostrados de cima para baixo na ordem crescente de precedência com sua associatividade e tipo.

### 7.3 Declarando arrays

Os arrays ocupam espaço na memória. O programador especifica o tipo dos elementos e o número de elementos requeridos array como segue:

```
tipo nomeDoArray [tamanhoDoArray];
```

e o compilador reserva a quantidade apropriada de memória. Ser uma constante inteira maior que zero. Por exemplo, para instruir o compilador a reservar 12 elementos para o array de inteiros

```
int c[12]; // c é um array de 12 inteiros
```

A memória pode ser reservada para vários arrays com uma única declaração. A seguinte declaração reserva 100 elementos para de inteiros 27 elementos para o array de inteiros

```
int b[100], // b é um array de 100 inteiros
x[27]; // x é um array de 27 inteiros
```



### Boa prática de programação 7.1

Preferimos declarar um array por declaração para legibilidade, modificabilidade e facilidade de comentar.

| Operadores                         | Associatividade            | Tipo                 |
|------------------------------------|----------------------------|----------------------|
| () []                              | da esquerda para a direita | mais alto            |
| ++ -- static_cast <tipo>(operando) | da esquerda para a direita | únario(pós-fixo)     |
| ++ -- + - !                        | da direita para a esquerda | únario(prefixo)      |
| * / %                              | da esquerda para a direita | multiplicativo       |
| + -                                | da esquerda para a direita | aditivo              |
| << >>                              | da esquerda para a direita | inserção/extracão    |
| < <= > >=                          | da esquerda para a direita | relacional           |
| == !=                              | da esquerda para a direita | igualdade            |
| &&                                 | da esquerda para a direita | Elógico              |
|                                    | da esquerda para a direita | OUIógico             |
| :?                                 | da direita para a esquerda | ternário condicional |
| = += -= *= /= %=                   | da direita para a esquerda | atribuição           |
| ,                                  | da esquerda para a direita | vírgula              |

Figura 7.2 Precedência e associatividade de operadores.

Os arrays podem ser declarados para conter valores de qualquer tipo de dados que não referência. Por exemplo, um array do tipo `char` pode ser utilizado para armazenar uma string de caracteres. Até agora, utilizaram-se objetos de caracteres. A Seção 7.4 introduz a utilização de arrays de caracteres para armazenar strings. As strings de caracteres e sua semelhança (um relacionamento do C++ herdado do C), e o relacionamento entre ponteiros e arrays, são discutidos no Capítulo 8.

## 7.4 Exemplos que utilizam arrays

Esta seção apresenta muitos exemplos que demonstram como declarar arrays, como inicializar arrays e como realizar muitas operações comuns de arrays.

Declarando um array e utilizando um loop para inicializar os elementos do array

O programa na Figura 7.3 declara `array de inteiros` (linha 12). As linhas 15–16 utilizam a instrução `for` para inicializar os elementos do array como zeros. A primeira instrução de saída (linha 18) exibe os títulos de coluna para as colunas i e j. A instrução subsequente (linhas 21–22), que imprime o array em formato tabular, é precedida da seguinte linha em que somente o próximo valor será enviado para a saída.

Inicializando um array em uma declaração com uma lista inicializadora

Os elementos de um array também podem ser inicializados na declaração do array colocandose depois do nome do array um sinal de igual (`=`) e uma lista separada por vírgulas (incluídas entre chaves). O programa na Figura 7.4 utiliza uma declaração de array para inicializar um array de inteiros com 10 valores (linha 13) e imprime o array em formato tabular (linhas 15–19).

Se houver menos inicializadores que elementos no array, os elementos do array restantes são inicializados como zero. Por isso, os elementos do array na Figura 7.3 poderiam ter sido inicializados como zero com a declaração

```
int n[10] = { 0 }; // inicializa elementos do array n como 0
```

A declaração inicializa explicitamente o primeiro elemento zero e inicializa implicitamente os demais novo elementos como zero. Se houver menos inicializadores que elementos no array, arrays automáticos não são implicitamente inicializados como zero, embora static ou sejam. O programador deve pelo menos inicializar o primeiro elemento como zero com uma lista inicializadora para os elementos restantes implicitamente serem configurados como zero. O método de inicialização mostrado na Figura 7.3 pode ser repetidamente enquanto um programa executa.

Se o tamanho do array for omitido de uma declaração com uma lista inicializadora, o compilador determina o número de elementos no array contando o número de elementos na lista inicializadora. Por exemplo,

```
int n[] = { 1, 2, 3, 4, 5};
cria um array de cinco elementos.
```

```

1 // Figura 7.3: fig07_03.cpp
2 // Inicializando um array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 int n[10]; // n é um array de 10 inteiros
13
14 // inicializa elementos do array n como 0
15 for (int i = 0; i < 10; i++)
16 n[i] = 0; // configura elemento na posição i como 0
17
18 cout << "Element" << setw(13) << "Value" << endl;
19
20 // gera saída do valor de cada elemento do array
21 for (int j = 0; j < 10; j++)
22 cout << setw(7) << j << setw(13) << n[j] << endl;
23
24 return 0; // indica terminação bem-sucedida
25 } // fim de main

```

| Element | Value |
|---------|-------|
| 0       | 0     |
| 1       | 0     |
| 2       | 0     |
| 3       | 0     |
| 5       | 0     |
| 6       | 0     |
| 7       | 0     |
| 8       | 0     |
| 9       | 0     |

Figura 7.3 Inicializando elementos de um array como zeros e imprimindo o array.

```

1 // Figura 7.4: fig07_04.cpp
2 // Inicializando um array em uma declaração.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8
9
10 int main()
11 {
12 // utiliza lista inicializadora para inicializar o array n
13 int n[10] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14

```

Figura 7.4 Inicializando os elementos de um array em sua declaração.

(continua)

```

15 cout << "Element" << setw(13) << "Value" << endl;
16 // gera saída do valor de cada elemento do array
17 for (int i = 0; i < 10; i++)
18 cout << setw(7) << i << setw(13) << n[i] << endl;
19
20 return 0; // indica terminação bem-sucedida
21
22 } // fim de main

```

| Element | Value |
|---------|-------|
| 0       | 32    |
| 1       | 27    |
| 2       | 64    |
| 3       | 95    |
| 4       | 14    |
| 5       | 90    |
| 6       | 70    |
| 7       | 60    |
| 8       | 37    |

Figura 7.4 Inicializando os elementos de um array em sua declaração.

(continuação)

Se o tamanho do array e uma lista inicializadora forem especificados em uma declaração de array, o número de inicializador ser menor que ou igual ao tamanho do array. A declaração de array

```
int n[5] = { 32, 27, 64, 18, 95, 14};
```

causa um erro de compilação, porque há seis inicializadores e somente cinco elementos do array.



#### Erro comum de programação 7.2

Fornecer mais inicializadores em uma lista inicializadora de array do que o número de elementos existentes no array é um erro de compilação.



#### Erro comum de programação 7.3

Esquecer de inicializar os elementos de um array cujos elementos deveriam ser inicializados é um erro de lógica.

Especificando o tamanho de um array com uma variável constante e configurando elementos do array com cálculos. A Figura 7.5 configura os elementos de um array ~~para 10 elementos~~ para 20 (linhas 17–18) e imprime o array em formato tabular (linhas 20–24). Esses números são gerados (linha 18) multiplicando-se cada sucessivo valor do loop por ~~o~~ adicionando ~~o~~.

A linha 13 utiliza `const` para declarar a chamada `arraySize` com o valor `Variáveis constantes`. Devem ser inicializadas com uma expressão constante quando são declaradas e não podem ser modificadas depois (como na Figuras 7.6 e 7.7). Variáveis constantes também são chamadas `Variáveis de leitura-only`.



#### Erro comum de programação 7.4

Não atribuir um valor a uma variável constante quando ela é declarada é um erro de compilação.



#### Erro comum de programação 7.5

Atribuir um valor a uma variável constante em uma instrução executável é um erro de compilação.

Variáveis constantes podem ser colocadas em qualquer lugar em que uma expressão constante é esperada. Na Figura 7.5, `arraySize` especifica o tamanho do array 15.

```
1 // Figura 7.5: fig07_05.cpp
2 // Configura o array s para os inteiros pares de 2 a 20.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // uma variável constante pode ser utilizada para especificar o tamanho do array
13 const int arraySize = 10;
14
15 int s[arraySize]; // array s tem 10 elementos
16
17 for (int i = 0; i < arraySize; i++) // configura os valores
18 s[i] = 2 + 2 * i;
19
20 cout << "Element" << setw(13) << "Value" << endl;
21
22 // gera saída do conteúdo do array s em formato tabular
23 for (int j = 0; j < arraySize; j++)
24 cout << setw(7) << j << setw(13) << s[j] << endl;
25
26
27 } // fim de main
```

| Element | Value |
|---------|-------|
| 0       | 2     |
| 1       | 4     |
| 2       | 6     |
| 3       | 8     |
| 4       | 10    |
| 5       | 12    |
| 6       | 14    |
| 7       | 16    |
| 8       | 18    |
| 9       | 20    |

**Figura 7.5** Gerando valores para serem colocados em elementos de um array.



Erro comum de programação 7.6

Somente constantes podem ser utilizadas para declarar o tamanho de arrays automáticos e estáticos. Não utilizar uma constante para esse propósito é um erro de compilação.

Utilizar variáveis constantes para especificar tamanhos de arrays torna um código mais legível e mais fácil de manter.

instância padávial para tratar 1000 elementos é muito maior que a instância padávial para tratar 100. O uso de funções é uma técnica útil para escalar o programa a fim de tratar 1000 elementos do array. A medida que os programas crescem, essa técnica torna-se mais útil para programas mais claros e mais fáceis de modificar.



Observação de engenharia de software 7.1

Definir o tamanho de cada array como uma variável constante em vez de uma constante literal pode tornar os programas mais escalonáveis.

```

1 // Figura 7.6: fig07_06.cpp
2 // Utilizando uma variável constante adequadamente inicializada.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 const int x = 7; // variável constante inicializada
10
11 cout << "The value of constant variable x is: " << x << endl;
12
13 return 0; // indica terminação bem-sucedida
14 } // fim de main

```

The value of constant variable x is: 7

Figura 7.6 Inicializando e utilizando uma variável constante.

```

1 // Figura 7.7: fig07_07.cpp
2 // Uma variável const deve ser inicializada.
3
4 int main()
5 {
6 const int x; // Erro: x deve ser inicializado
7
8 x = 7; // Erro: não pode modificar a variável const
9
10 } // fim de main // indica terminação bem-sucedida

```

Mensagem de erro do compilador de linha de comando Borland C++:

```

Error E2304 fig07_07.cpp 6: Constant variable 'x' must be initialized
in function main()
Error E2024 fig07_07.cpp 8: Cannot modify a const object in function main()

```

Mensagem de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphtp5_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object
must be initialized if not extern
C:\cpphtp5_examples\ch07\fig07_07.cpp(8) : error C2166: l-value specifies
const object

```

Mensagem de erro do compilador GNU C++:

```

fig07_07.cpp:6: error: uninitialized const or read-only variable `x'

```

Figura 7.7 Variáveis const devem ser inicializadas.



### Boa prática de programação 7.2

Definir o tamanho de um array como uma variável constante em vez de uma constante literal torna os programas mais claros. Essa técnica elimina os chamados mágicos. Por exemplo, mencionar repetidamente o tamanho 10 em código de processamento de array para um array de 10 elementos dá para o número 10 uma importância artificial e infelizmente pode confundir o leitor quando o programa incluir outros números 10 que não têm nada que ver com o tamanho do array.

#### Somando os elementos de um array

Freqüentemente, os elementos de um array representam uma série de valores a serem utilizados em um cálculo. Por exemplo, os elementos de um array representam as notas de exame, um professor pode querer somar os elementos do array e utilizar essa calculadora a média da classe. Os exemplos utilizados na classe no capítulo, nomeadamente as figuras 7.16–7.17 e as figuras 7.23–7.24, utilizam essa técnica.

O programa na Figura 7.8 soma os valores contidos em um array. O código que o programa declara, cria e inicializa o

também poderia ser escrito de forma mais concisa. A figura 7.9 realiza o mesmo processo, mas com arquivos e discos rígidos. Consulte o Capítulo 14, “Processamento de arquivo”. Por exemplo, a instrução

```
for (int j = 0; j < arraySize; j++)
 cin >> a[j];
```

lê um valor por vez do teclado e armazena o valor no elemento

#### Utilizando gráficos de barras para exibir dados de array graficamente

Muitos programas apresentam dados graficamente aos usuários. Por exemplo, os valores numéricos são freqüentemente exibidos em um gráfico de barras. Nesse gráfico, as barras mais longas representam os valores numéricos proporcionalmente. Uma maneira simples de exibir os dados numéricos graficamente é utilizar um gráfico de barras que mostra cada valor numérico com uma barra de asteriscos (\*).

Os professores freqüentemente gostam de examinar a distribuição de notas de um exame. Um professor poderia plotar o número de notas em cada uma das várias categorias para visualizar a distribuição das notas. Suponha que as notas fossem 87, 68, 94, 78, 85, 91, 76 e 87. Observe que há uma nota 100, duas notas nos 90s, quatro notas nos 80s, duas notas nos 70s, uma nota 68 e nenhuma nota abaixo de 60. Nossa próximo programa (Figura 7.9) armazena esses dados de distribuição de notas em um array de elementos, cada um correspondente a uma categoria de nota. Por exemplo, a nota 100 aparece no intervalo 0–9, indica o número de notas no intervalo 70–79 e o número de notas 100. As duas versões da classe

```
1 // Figura 7.8: fig07_08.cpp
2 // Calcula a soma dos elementos do array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 const int arraySize = 10; // variável constante indicando o tamanho do array
10 int a[arraySize] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11 int total = 0;
12
13 // soma o conteúdo do array a
14 for (int i = 0; i < arraySize; i++)
15 total += a[i];
16
17 cout << "Total of array elements: " << total << endl;
18
19 return 0; // indica terminação bem-sucedida
20 } // fim de main
```

Total of array elements: 849

Figura 7.8 Calculando a soma dos elementos de um array.

no capítulo (figuras 7.16–7.17 e figuras 7.23–7.24) contêm código que calcula essas freqüências de nota com base em um array de notas. Por enquanto, criamos manualmente o array examinando o conjunto de notas.

O programa lê os números a partir do array e representa as informações graficamente como um gráfico de barras. O programa exibe cada intervalo de notas seguido por uma barra de asteriscos que indica o número de notas nesse intervalo. Para rotular as linhas 21–26 geram saída de um intervalo de notas (por exemplo, na variável `coGrade`)

```

1 // Figura 7.9: fig07_09.cpp
2 // Programa de impressão de gráfico de barras.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 const int arraySize = 11;
13 int n[arraySize] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
14
15 cout << "Grade distribution:" << endl;
16
17 // para cada elemento do array n, gera saída de uma barra do gráfico
18 for (int i = 0; i < arraySize; i++)
19 {
20 // gera a saída do rótulo das barras ("0-9:", ..., "90-99:", "100:")
21 if (i == 0)
22 cout << " 0-9: ";
23 else if (i == 10)
24 cout << " 100: ";
25 cout << i * 10 << "-" << (i * 10) + 9 << ":" ;
26
27 // imprime a barra de asteriscos
28 for (int stars = 0; stars < n[i]; stars++)
29 cout << "*";
30
31 cout << endl; // inicia uma nova linha de saída
32 } // fim do for externo
33
34 return 0; // indica terminação bem-sucedida
35 }
36 } // fim de main

```

```

Grade distribution:
0-9:
10-19:
20-29:

30-39:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

Figura 7.9 Programa de impressão de gráfico de barras.

aninhada (linhas 29–30) gera a saída das barras. Observe a condição de continuação do loop, na linha 29 (que o programa alcança tecnicamente, o loop continua), utilizando assim um valor para determinar o número de asteriscos a exibir. Nesse exemplo, contém zeros porque nenhum aluno recebeu uma nota abaixo de 60. Portanto, o programa não exibe nenhum asterisco perto dos seis primeiros intervalos de notas.



### Erro comum de programação 7.7

Embora seja possível utilizar a mesma variável de controle em mais de uma estrutura aninhada dentro dela, isso é confuso e pode levar a erros de lógica.

Utilizando os elementos de um array como contadores

Às vezes, os programas utilizam as variáveis de contador para resumir dados, como os resultados de uma enquete. Na Figura 1 zamos os contadores separados em nosso programa de lançamento de dados para monitorar o número de ocorrências de cada

Figura 7.10 o programa de rolagem de dados (listado 6) para 6.000.000 vezes, dividido em seções de código. A estrutura de instruções desse programa é mostrada na Figur desse programa substitui as instruções linhas 30–52 da Figura 6.9. A linha 26 utiliza um valor aleatório para determinar qual elemento frequency deve ser incrementado durante cada iteração do loop. O cálculo na linha 26 produz um subscrito aleatório de 1 a 6, assim array frequency deve ser grande o bastante para armazenar seis contadores. Entretanto, utilizamos um array de sete elementos em ignorarmos frequency[ 0 ] — é mais lógico fazer o valor 1 da face do dado incrementar frequency[ 0 ]. Portanto, o valor de cada face do dado é utilizado como índice de freqüência frequency[ face ]. Por fim, substituímos as linhas 56–61 da Figura 6.9 fazendo um loop pelo array para gerar saída dos resultados (linhas 31–33).

```

1 // Figura 7.10: fig07_10.cpp
2 // Rola um dado de seis lados 6.000.000 vezes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16
17 int main()
18 {
19 const int arraySize = 7; // ignora o elemento zero
20 int frequency[arraySize] = { 0 };
21
22 srand(time(0)); // semeia o gerador de número aleatório
23
24 // lança o dado 6.000.000 vezes; usa o valor do dado como índice de freqüência
25 for (int roll = 1; roll <= 6000000; roll++)
26
27 frequency[1 + rand() % 6]++;
28 cout << "Face" << setw(13) << "Frequency" << endl;
29
30 // gera a saída do valor de cada elemento do array
31 for (int face = 1; face < arraySize; face++)
32 cout << setw(4) << face << setw(13) << frequency[face]
33 << endl;

```

Figura 7.10 Programa de rolagem de dados utilizando um array em `setw`.

(continua)

```

34
35 return 0; // indica terminação bem-sucedida
36 } // fim de main

```

| Face | Frequency |
|------|-----------|
| 1    | 1000167   |
| 2    | 1000149   |
| 3    | 1000152   |
| 4    | 998748    |
| 5    | 999626    |
| 6    | 1001158   |

Figura 7.10 Programa de rolagem de dados utilizando um array em `seu código`.

(continuação)

Utilizando arrays para resumir resultados de uma enquete

Nosso próximo exemplo (Figura 7.11) utiliza arrays para resumir os resultados de dados coletados em uma enquete. Considere declaração do problema:

Foi pedido a quarenta alunos que avaliassem a qualidade da comida na cantina estudantil em uma escala de 1 a 10 (com 1 signi cando péssimo e, 10, excelente). Coloque as 40 respostas em um array de inteiros e resuma os resultados da enquete.

Essa é uma típica aplicação de processamento de array. Queremos resumir o número de respostas de cada tipo (isto é, 1 a 10). `responses` (linhas 17–19) é um array de inteiros de 40 elementos das respostas dos alunos à enquete. Observe que o array declarado `const`, uma vez que seus valores não mudam (e não devem mudar). Utilizamos `frequency` (linhas 22) para contar o número de ocorrências de cada resposta. Cada elemento do array é utilizado como contador para uma das da enquete e inicializado como zero. Como na Figura 7.10 ignoramos



#### Observação de engenharia de software 7.2

O qualificador `const` deve ser utilizado para impor o princípio do menor privilégio. Utilizar o princípio do menor privilégio adequadamente para projetar software pode reduzir significativamente o tempo de depuração e os efeitos colaterais indevidos e por



#### Boa prática de programação 7.3

Empenhe-se na clareza do programa. Às vezes vale a pena trocar utilização mais eficiente da memória ou tempo de processador a favor de escrever programas mais claros.



#### Dica de desempenho 7.1

Às vezes considerações de desempenho superam as considerações de clareza.

A primeira instrução (linhas 26–27) aceita as respostas, uma por vez, incrementa um dos 10 contadores no array `frequency` (`frequency[ 1 ]` a `frequency[ 10 ]`). A instrução-chave no loop é a linha 27, que incrementa o contador adequado, dependendo do valor de `answer`.

Vamos considerar várias iterações. Qual tipo variável de controle o valor de `responses[ answer ]` é o valor de `responses[ 0 ]` (isto é, na linha 17), então o programa interpreta

`frequency[ 1 ]++`

Quando `answer` é qual valor, o array `frequency` é alterado? Para analisar a execução, devemos para linha 26, que irá incrementar o elemento de coluna 1. Aí, a instrução é executada, incrementando o valor de `frequency[ 1 ]`. Depois, a instrução é executada, e avalia o próximo conjunto externo de colchetes (isto é, que é um valor selecionado das respostas linhas 17–19). Então, utilize o valor resultante como o subscritor para especificar qual contador incrementar.

Quando `answer` é 1, `responses[ answer ]` é o valor de `responses[ 1 ]`, que é então o programa interpreta `responses[ answer ] ]++` como:

`frequency[ 2 ]++`

que incrementa o elemento do array 2.

```

1 // Figura 7.11: fig07_11.cpp
2 // Programa de enquete de alunos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // define o tamanho do arrays
13 const int responseSize = 40; // tamanho do array responses
14 const int frequencySize = 11; // tamanho do array frequency
15
16 // coloca as respostas da enquete no array responses
17 const int responses[responseSize] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21 // inicializa contadores de freqüência como 0
22 int frequency[frequencySize] = { 0 };
23
24 // para cada resposta, seleciona o elemento de respostas e utiliza esse valor
25 // como subscrito de freqüência para determinar o elemento a incrementar
26 for (int answer = 0; answer < responseSize; answer++)
27 frequency[responses[answer]]++;
28
29 cout << "Rating" << setw(17) << "Frequency" << endl;
30
31 // gera saída do valor de cada elemento do array
32 for (int rating = 1; rating < frequencySize; rating++)
33 cout << setw(6) << rating << setw(17) << frequency[rating]
34 << endl;
35
36 return 0; // indica terminação bem-sucedida
37 } // fim de main

```

| Rating | Frequency |
|--------|-----------|
| 1      | 2         |
| 2      | 2         |
| 3      | 2         |
| 4      | 2         |
| 5      | 5         |
| 6      | 11        |
| 7      | 5         |
| 8      | 7         |
| 9      | 1         |
| 10     | 3         |

Figura 7.11 Programa de análise de enquete.

Quando `answer` é 2, `responses[ answer ]` é o valor de `responses[ 2 ]`, que é então o programa interpreta `responses[ answer ]++` como:

`frequency[ 6 ]++`

que incrementa o elemento de array 6 e assim por diante. Independentemente do número de respostas processadas na enquete só exige um array de 11 elementos (ignorando o elemento zero) para resumir o resultado, porque todos os valores de resposta 1 e 10 e os valores de subscrito de um array de 11 elementos são 0 a 10.

Se os dados no array contivessem um valor inválido, como 13, o programa teria que tentar adicionar que está fora dos limites. O array tem nenhuma verificação de limite de array para evitar que o computador referencie um elemento que não existe, um programa que executa pode ‘ultrapassar’ qualquer uma das extremidades de um array sem avisar. O programador deve assegurar que todas as referências de array permaneçam dentro dos limites do array.



### Erro comum de programação 7.8

Referenciar um elemento fora dos limites de array é um erro de lógica de tempo de execução. Não é um erro de sintaxe.



### Dica de prevenção de erro 7.1

Ao fazer um loop por um array, o subscrito de array nunca deveria ir abaixo de 0 e sempre deveria ser menor que o número total de elementos no array (um menor que o tamanho do array). Certifique-se de que a condição de terminação do loop impede o acesso a elementos fora desse intervalo.



### Dica de portabilidade 7.1

Os efeitos (normalmente sérios) de referenciar elementos fora dos limites do array são dependentes do sistema. Freqüentemente resulta em alterações no valor de uma variável não relacionada ou em um erro fatal que termina a execução do programa.

O C++ é uma linguagem extensível. A Seção 7.11 apresenta o `Container Library`, que permite que os programadores realizem muitas operações que não estão disponíveis nos arrays predefinidos do C++. Por exemplo, seremos de comparar diretamente e atribuir um array a outro. No Capítulo 11, estendemos o C++ mais ainda implementando um array como uma classe definida pelo próprio usuário. Essa nova definição de array nos permitirá inserir e gerar saída de arrays inteiros, inicializar arrays quando eles são criados, evitar acesso a elementos do array fora do intervalo e alterar o intervalo de sul (e até seu tipo de subscrito) de modo que o primeiro elemento de um array não seja obrigado a ser o elemento 0. Seremos capazes de utilizar subscritos não inteiros.



### Dica de prevenção de erro 7.2

No Capítulo 11, veremos como desenvolver uma classe que representa um ‘array inteligente’, que verifica que todas as referências a subscritos estejam dentro dos limites em tempo de execução. Utilizar tipos de dados assim inteligentes ajuda a eliminar bugs.

Utilizando arrays de caracteres para armazenar e manipular strings

Até este ponto, discutimos somente arrays de inteiros. Entretanto, os arrays podem ser de qualquer tipo. Agora introduziremos o armazenamento de strings de caracteres em arrays de caracteres. Lembre-se de que, desde o Capítulo 3, utilizamos objetos de strings de caracteres, como o nome `courseName`. Um string é na realidade um array de caracteres. Embora os strings sejam convenientes para utilizar e reduzir o potencial para erros, arrays de caracteres que representam strings têm vários recursos únicos, que discutimos nesta seção. A medida que continua seu estudo de C++, você pode descobrir novas capacidades do C++ que exigem utilizar arrays de caracteres. Seja paciente e lembre-se de que é sempre possível solicitar a atualização do código existente utilizando arrays de caracteres.

Um array de caracteres pode ser inicializado utilizando um literal string. Por exemplo, a declaração

```
char string1[] = "first" ;
```

initializa os elementos do array para os caracteres individuais no literal. O comprimento do array na declaração precedente é determinado pelo compilador com base no comprimento da string. É importante observar que a string contém cinco caracteres, mas o caractere especial de terminação de string, o `\0`, na realidade contém seis elementos. A representação de constante de caracteres é dividida em seguidas por zeros. Todas as strings representadas por arrays de caracteres acabam com esse caractere. Um array de caracteres representa uma string, sempre deve ser declarado com um tamanho grande o suficiente para armazenar o número de caracteres na string e o nulo de terminação.

Os arrays de caracteres também podem ser inicializados com constantes de caractere individuais em uma lista inicializada. A declaração precedente é equivalente à forma mais tediosa:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Observe o uso de aspas únicas para delinear cada caractere constante. Além disso, observe que fornecemos explicitamente o nulo de terminação como o último valor inicializador. Sem ele, esse array simplesmente representaria um array de caracteres, não uma string. Como discutimos no Capítulo 8, não fornecer um caractere nulo de terminação a uma string pode causar erros de lógica.

Como uma string é um array de caracteres, podemos acessar caracteres individuais em uma string diretamente com a notação crito de array. Por exemplo, é o caractere, é o caractere, é o caractere nulo.

Também podemos inserir uma string diretamente em um array de caracteres e, para isso, podemos utilizar a declaração

```
char string2[20];
```

cria um array de caracteres capaz de armazenar uma string de 19 caracteres e um caractere nulo de terminação. A instrução

```
cin >> string2;
```

lê uma string inserida pelo teclado para prescrever o caractere nulo ao fim da entrada de string feita pelo usuário. Observe que a instrução precedente fornece somente o nome do array e nenhuma informação sobre o tamanho do array. É de responsabilidade do programador assegurar que o array para o qual a string é lida é capaz de armazenar qualquer string que o usuário digite no teclado. Por padrão, lê caracteres do teclado até que o primeiro caractere de espaço em branco seja encontrado — independentemente do tamanho do array. Portanto, inserir dados pode inserir dados além do fim do array (consulte a Seção 8.13 para informações

sobre como evitar inserção além do final de um array



#### Erro comum de programação 7.9

Não fornecer um array de caracteres suficientemente grande para armazenar uma string digitada no teclado pode resultar em perda de dados em um programa e outros erros sérios de tempo de execução.

Um array de caracteres representando uma string terminada por caractere nulo pode ser enviado para a saída com a instrução

```
cout << string2;
```

imprime o array. Observe que, assim como <<, não se importa com o tamanho do array de caracteres. Os caracteres da string são enviados para a saída até que um caractere nulo de terminação seja encontrado. [Aços de carregamento de caracteres devem ser processados como strings terminadas por ocorrências em los.]

A Figura 7.12 demonstra como inicializar um array de caracteres com um literal string, ler uma string para um array de caracteres e imprimir um array de caracteres como uma string e acessar caracteres individuais de uma string.

As linhas 23–24 da Figura 7.12 utilizam a instrução `cin >> string1` para ler a string para o array e imprime os caracteres individuais separados por espaços. A condição de saída da instrução `for`, é verdadeira até que o loop encontra o caractere nulo de terminação da string.

Arrays locais estáticos e arrays locais automáticos

O Capítulo 6 discutiu o especificador de classe `static` para variáveis locais em uma definição de função. Essa classe existe até o final do programa, mas é visível somente no corpo da função.



#### Dica de desempenho 7.2

Podemos aplicar para uma declaração local de array de modo que o array não seja criado e inicializado toda vez que o programa chama a função e não seja destruído toda vez que a função termina no programa. Isso pode melhorar o desempenho especialmente quando utilizando arrays grandes.

Um programa inicializa arrays locais quando suas declarações são encontradas pela primeira vez. Se um array é inicializado explicitamente pelo programador, cada elemento desse array é inicializado como zero pelo compilador quando o array é criado. Lembre-se de que o C++ não realiza essa inicialização-padrão para variáveis automáticas.

A Figura 7.13 demonstra a função `arrayInit` (linhas 25–41) com um array local (linha 28) e a função `staticArrayInit` (linhas 44–60) com um array local automático (linha 47).

A função `staticArrayInit` é chamada duas vezes (linhas 13 e 17). O array é inicializado como zero pelo compilador na primeira vez que a função é chamada. A função imprime o array, adiciona 5 a cada elemento e imprime o array novamente. Na vez que a função é chamada a segunda vez, os valores modificados armazenados durante a primeira chamada da função. A função `arrayInit` também é chamada duas vezes (linhas 14 e 18). Os elementos do array local automático são inicializados cada vez que a função é chamada, os elementos do array são reinicializados como 1, 2 e 3. O array tem classe de armazenamento automatico, então o array é recriado durante cada chamada.



#### Erro comum de programação 7.10

Supor que elementos de um array de uma função são inicializados cada vez que a função é chamada pode levar a erros de lógica em um programa.

```

1 // Figura 7.12: fig07_12.cpp
2 // Tratando arrays de caracteres como strings.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10 char string1[20]; // reserva 20 caracteres
11 char string2[] = "string literal" ; // reserva 15 caracteres
12
13 // lê a string fornecida pelo usuário para o array string1
14 cout << "Enter the string \"hello there\": " ;
15 cin >> string1; // lê "hello" [o espaço termina a entrada]
16
17 // gera a saída de strings
18 cout << "string1 is: " << string1 << "\nstring2 is: " << string2;
19
20 cout << "\nstring1 with spaces between characters is:\n" ;
21
22 // caracteres de saída até que caractere nulo é alcançado
23 for (int i = 0; string1[i] != '\0' ; i++)
24 cout << string1[i] << ' ';
25
26 cin >> string1; // lê "there"
27 cout << "\nstring1 is: " << string1 << endl;
28
29 return 0; // indica terminação bem-sucedida
30 } // fim de main

```

```

Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there

```

Figura 7.12 Os arrays de caracteres processados como strings.

```

1 // Figura 7.13: fig07_13.cpp
2 // Arrays estáticos são inicializados como zero.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void staticArrayInit(); // protótipo da função
8
9
10 int main()
11 {
12 cout << "First call to each function:\n" ;
13 staticArrayInit();
14 automaticArrayInit();

```

Figura 7.13 Inicialização de arrays.

(continua)

```

15 cout << "\n\nSecond call to each function:\n" ;
16 staticArrayInit();
17 automaticArrayInit();
18 cout << endl;
19
20 return 0; // indica terminação bem-sucedida
21 } // fim de main
22
23 // função para demonstrar um array local estático
24 void staticArrayInit(void)
25 {
26 // inicializa elementos como 0 na primeira vez que a função é chamada
27 static int array1[3]; // array local estático
28
29 cout << "\nValues on entering staticArrayInit:\n" ;
30
31 // gera saída do conteúdo de array1
32 for (int i = 0; i < 3; i++)
33 cout << "array1[" << i << "] = " << array1[i] << " ";
34
35 cout << "\nValues on exiting staticArrayInit:\n" ;
36
37 // modifica e gera saída do conteúdo de array1
38 for (int j = 0; j < 3; j++)
39 cout << "array1[" << j << "] = " << (array1[j] += 5) << " ";
40
41 } // fim da função staticArrayInit
42
43 // função para demonstrar um array local automático
44 void automaticArrayInit(void)
45 {
46 // inicializa elementos toda vez que a função é chamada
47 int array2[3] = { 1, 2, 3 }; // array local automático
48
49 cout << "\n\nValues on entering automaticArrayInit:\n" ;
50
51 // gera saída do conteúdo de array2
52 for (int i = 0; i < 3; i++)
53 cout << "array2[" << i << "] = " << array2[i] << " ";
54
55 cout << "\nValues on exiting automaticArrayInit:\n" ;
56
57 // modifica e gera saída do conteúdo de array2
58 for (int j = 0; j < 3; j++)
59 cout << "array2[" << j << "] = " << (array2[j] += 5) << " ";
60 } // fim da função automaticArrayInit

```

First call to each function:

Values on entering staticArrayInit:  
array1[0] = 0 array1[1] = 0 array1[2] = 0  
Values on exiting staticArrayInit:  
array1[0] = 5 array1[1] = 5 array1[2] = 5

Values on entering automaticArrayInit:  
array2[0] = 1 array2[1] = 2 array2[2] = 3

Figura 7.13 Inicialização de array estático e inicialização de array automático.

(continua)

```

Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

```

Figura 7.13 Inicialização de array estático. Inicialização de array automático.

(continuação)

## 7.5 Passando arrays para funções

Para passar um argumento array a uma função, especifique o nome do array ~~sem colchetes~~. Por exemplo, se o array tures foi declarado como

```

int hourlyTemperatures[24];
a chamada de função
 modifyArray(hourlyTemperatures, 24);

```

passa o array `hourlyTemperatures` e seu tamanho à função. Ao passar um array a uma função, o tamanho do array normalmente é passado também, assim a função pode processar o número específico de elementos no array. (Caso contrário ríamos embutir esse conhecimento na própria função chamada ou, pior ainda, colocar o tamanho do array em uma variável global.) Secção 7.11, quando apresentarmos da Standard Library para representar um tipo de array mais robusto, você verá que o tamanho é um predefinido — cada objeto conhece seu próprio tamanho, que pode ser obtido invocando-se a função-membro `size` do vector. Portanto, quando passarmos um array a uma função, não teremos de passar o tamanho do array como um argumento.

O C++ passa arrays a funções por referência — as funções chamadas podem modificar os valores dos elementos nos arrays dos chamadores. O valor do nome do array é o endereço na memória do computador do primeiro elemento do array. Como o array inicial do array é passado, a função chamada sabe precisamente onde o array está armazenado na memória. Portanto, quando a função modifica elementos do array no seu corpo, ela está modificando os elementos reais do array em suas posições na memória.



### Dica de desempenho 7.3

Passar arrays por referência faz sentido por razões de desempenho. Se arrays fossem passados por valor, uma cópia de cada array seria passada. Para grandes arrays freqüentemente passados, isso seria demorado e exigiria armazenamento considerável para cópias dos elementos do array.



### Observação de engenharia de software 7.3

É possível passar um array por valor (utilizando um truque simples que explicamos no Capítulo 22) — mas isso raramente é feito.

Embora arrays inteiros sejam passados por referência, elementos do array individuais são passados por valor exatamente porque são simples. Esses fragmentos de dados são chamados de `simples`. Para passar um elemento de um array para uma função, utilize o nome subscrito do elemento do array como um argumento na chamada de função. No Exemplo 6, mostramos como passar escalares (isto é, variáveis individuais e elementos do array) por referência com ponteiros. No C++, mostramos como passar escalares por referência com ponteiros.

Para uma função receber um array por meio de uma chamada de função, a lista de parâmetros da função deve especificar que espera receber um array. Por exemplo, o código abaixo deve ser escrito como:

```
void modifyArray(int b[], int arraySize)
```

indicando que a função `modifyArray` espera receber o endereço de um array de inteiros n parâmetros. Os elementos do array no parâmetro `arraySize`. O tamanho do array não é requerido entre os colchetes do array. Se for incluído, o compilador irá ignorá-lo.

Como o C++ passa arrays a funções por referência, quando a função `modifyArray` é chamada, ela faz o referenciamento ao array real no chamador (isto é, para o array original, discutido no começo desta seção).

Observe a aparência estranha do protótipo da função para

```
void modifyArray(int [], int);
```

Esse protótipo poderia ter sido escrito assim:

```
void modifyArray(int anyArray[], int anyVariableName);
```

mas, como aprendemos no Capítulo 3, compiladores C++ ignoram nomes de variáveis em protótipos. Lembre-se, o protótipo irá ao compilador o número de argumentos e o tipo de cada argumento (na ordem em que se espera que os argumentos apareçam).

O programa na Figura 7.14 demonstra a diferença entre passar um array inteiro e passar um elemento do array. As linhas 2–5 imprimem os cinco elementos srcinais do array `a`, que tem seu tamanho à função `array` (linhas 45–50), que multiplica cada um dos elementos pelo meio do parâmetro. Depois, as linhas 32–33 imprimem novamente `a` em `main`. Enquanto mostra a saída, os elementos são modificados por `modifyArray`. Em seguida, a linha 36 imprime o valor

**do resultado** dentro de **o bloco de memória** que **chamou** a função. Isso prova que **o array é modificado**, porque os **parâmetros** por **elementos do array individuais** são **passados por valor**.

Há situações em seus programas nas quais uma função não deve ter permissão para modificar elementos do array. O C++ fornece qualificador `const`, que pode ser utilizado para evitar modificação de valores do array no chamador por meio de código em uma função chamada. Quando uma função especifica um parâmetro de array que é precedido pelo qualificador `const`, tornam-se constantes no corpo da função, e qualquer tentativa de modificar um elemento do array no corpo da função resulta em erro de compilação. Isso permite ao programador evitar modificação acidental de elementos do array no corpo da função.

```

1 // Figura 7.14: fig07_14.cpp
2 // Passando arrays e elementos de array individuais a funções.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
10 void modifyArray(int [], int); // parece estranho
11 void modifyElement(int);
12
13 int main()
14 {
15 const int arraySize = 5; // tamanho do array a
16 int a[arraySize] = { 0, 1, 2, 3, 4 }; // inicializa o array a
17
18 cout << "Effects of passing entire array by reference:"
19 << endl
20
21 // gera saída de elementos do array srcinal
22 for (int i = 0; i < arraySize; i++)
23 cout << setw(3) << a[i];
24
25 cout << endl;
26
27 // passa o array a para modifyArray por referência
28 modifyArray(a, arraySize);
29 cout << "The values of the modified array are:\n" ;
30
31 // gera saída de elementos do array modificado
32 for (int j = 0; j < arraySize; j++)
33 cout << setw(3) << a[j];

```

Figura 7.1 Passando arrays e elementos de array individuais a funções.

(continua)

```

34
35 cout << "\n\n\nEffects of passing array element by value:"
36 << "\n\na[3] before modifyElement: " << a[3] << endl;
37
38 modifyElement(a[3]); // passa elemento do array a[3] por valor
39 cout << "a[3] after modifyElement: " << a[3] << endl;
40
41 return 0; // indica terminação bem-sucedida
42 } // fim de main
43
44 // na função modifyArray, "b" aponta para o array srclinal "a" na memória
45 void modifyArray(int b[], int sizeOfArray)
46 {
47 // multiplica cada elemento do array por 2
48 for (int k = 0; k < sizeOfArray; k++)
49 b[k] *= 2;
50 } // fim da função modifyArray
51
52 // na função modifyElement, "e" é uma cópia local do
53 // elemento do array a[3] passado de main
54 void modifyElement(int e)
55 {
56 // multiplica parâmetro por 2
57 cout << "Value of element in modifyElement: " << (e *= 2) << endl;
58 } // fim da função modifyElement

```

Effects of passing entire array by reference:

The values of the srclinal array are:

0 1 2 3 4

The values of the modified array are:

Effects of passing array element by value:

a[3] before modifyElement: 6  
 Value of element in modifyElement: 12  
 a[3] after modifyElement: 6

Figura 7.14 Passando arrays e elementos de array individuais a funções.

(continuação)

A Figura 7.15 demonstra o **qualificação de array**. A linha 21, `int *ToModifyArray`, é definida como parâmetro `b[]`, que especifica que é **apontante** e não pode ser modificado. Cada uma das três tentativas por parte da função de modificar os elementos do array (linhas 23–25) resultam em um erro de compilação. O compilador Microsoft Visual C++ .NET, por exemplo, produz o erro `lvalue specifies const object`. [Nota: O padrão C++ define um 'objeto' como qualquer 'região de armazenamento', incluindo assim variáveis ou elementos de array dos tipos de dados fundamentais, bem como instâncias de classes (ou chamadas de objetos).] Essa mensagem indica que é impossível modificar o conteúdo de um operando de atribuição. Observe que as mensagens de erro de compilador variam entre compiladores (como mostrado na Figura 7.15). Consulte a documentação no Capítulo 10.



#### Erro comum de programação 7.11

Esquecer que arrays no chamador são passados por referência, e daí poderem ser modificados em funções chamadas, pode resultar em erros de lógica.



### Observação de engenharia de software 7.4

Aplicar o qualificador `const` um parâmetro de array em uma definição de função para impedir que o array ser modificado no corpo da função é outro exemplo do princípio de menor privilégio. As funções não devem receber a capacidade de modificar um array a menos que seja absolutamente necessário.

```

1 // Figura 7.15: fig07_15.cpp
2 // Demonstrando o qualificador de tipo const.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void tryToModifyArray(const int a[]); // protótipo de função
8
9 int main()
10 {
11 int a[] = { 10, 20, 30 };
12
13 tryToModifyArray(a);
14 cout << a[0] << ' ' << a[1] << ' ' << a[2] << '\n' ;
15
16 return 0; // indica terminação bem-sucedida
17 } // fim de main
18
19 // Na função tryToModifyArray, "b" não pode ser utilizado
20 // para modificar o array serializado "a" em main.
21 void tryToModifyArray(const int b[])
22 {
23 b[0] /= 2; // error
24 b[1] /= 2; // error
25 } // fim da função tryToModifyArray

```

Mensagem de erro do compilador de linha de comando Borland C++:

```

Error E2024 fig07_15.cpp 23: Cannot modify a const object
in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 24: Cannot modify a const object
in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 25: Cannot modify a const object
in function tryToModifyArray(const int * const)

```

Mensagem de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphtp5_examples\ch07\fig07_15.cpp(23) : error C2166: l-value specifies
 const object
C:\cpphtp5_examples\ch07\fig07_15.cpp(24) : error C2166: l-value specifies
 const object
C:\cpphtp5_examples\ch07\fig07_15.cpp(25) : error C2166: l-value specifies
 const object

```

Mensagem de erro do compilador GNU C++:

```

fig07_15.cpp:23: error: assignment of read-only location
fig07_15.cpp:24: error: assignment of read-only location
fig07_15.cpp:25: error: assignment of read-only location

```

Figura 7.15 Qualificador de tipo const aplicado a um parâmetro de array.

## 7.6 Estudo de caso: classe GradeBook utilizando um array para armazenar notas

Esta seção desenvolve ainda mais a classe introduzida no Capítulo 3 e expandida nos capítulos 4–6. Lembre-se de que essa classe representa um livro de notas utilizado por um professor para armazenar e analisar um conjunto de notas de aluno anteriores da classe processavam um conjunto de notas inseridas pelo usuário, mas não mantinham os valores das notas individuais membros de dados da classe. Portanto, os cálculos de repetição exigem que o usuário insira as mesmas notas novamente. Urge resolver esse problema seria armazenar cada nota inserida em um membro de dados da classe. Por exemplo, poderíamos declarar 10 membros de dados grade<sub>1</sub>...grade<sub>10</sub> na classe GradeBook para armazenar 10 notas de alunos. Entretanto, o código para somar as notas e determinar a média da classe seria complicado. Nesta seção, resolvemos esse problema armazenando as notas em um array.

Armazenando notas de aluno em um array na classe GradeBook

A versão da classe GradeBook (Figuras 7.16–7.17) apresentada aqui utiliza um array de inteiros para armazenar as notas de vários alunos.

**em memória** O código da Figura 7.16 define a classe GradeBook.

Observe que o tamanho do array na linha 29 da Figura 7.16 é especificado pelo membro de dados (declarado na linha 13). Esse membro de dados é acessível aos clientes da classe. Logo veremos um exemplo de um programa-cliente que utiliza essa constante. O operador const indica que esse membro de dados é constante — seu valor não pode ser alterado depois de ser inicializado. A sintaxe constante de variável indica que o membro de dados é compartilhado por todos os objetos da classe GradeBook para as notas de todos os alunos para o mesmo número de alunos. Lembre-se, a partir da Seção 3.6, de que, quando cada objeto de uma classe mantém sua própria cópia de um atributo que representa o atributo também é conhecida como membro de dados — cada objeto (instância) da classe tem uma cópia separada. Esse é o caso com os membros de classe static, que também são conhecidos como class variables. Quando objetos de uma classe contendo membros de dados static são criados, todos os objetos dessa classe compartilham uma única cópia dos membros de dados.

```

1 // Figura 7.16: GradeBook.h
2 // Definição da classe GradeBook que usa um array para armazenar notas de teste.
3 // As funções-membro são definidas em GradeBook.cpp
4
5 #include <string> // o programa utiliza a classe string da C++ Standard Library
6 using std::string;
7
8 // Definição da classe GradeBook
9 class GradeBook
10 {
11 public :
12 // constante -- número de alunos que fizeram o teste
13 const static int students = 10; // note os dados públicos
14
15 // construtor inicializa o nome do curso e o array de notas
16 GradeBook(string, const int []);
17
18 void setCourseName(string); // função para configurar o nome do curso
19 string getCourseName(); // função para recuperar o nome do curso
20 void displayMessage(); // exibe uma mensagem de boas-vindas
21 void processGrades(); // realiza várias operações nos dados
22 int getMinimum(); // localiza a nota mínima para o teste
23 int getMaximum(); // localiza a nota máxima para o teste
24 float getAverage(); // determina a nota média para o teste
25 void outputBarChart(); // gera saída do gráfico de barras de distribuição de notas
26 void outputGrades(); // gera a saída do conteúdo do array de notas
27 private :
28 string courseName; // nome do curso para esse livro de notas
29 int grades[students]; // array de notas de aluno
30 }; // fim da classe GradeBook

```

Figura 7.16 Definição da classe GradeBook utilizando um array para armazenar notas de teste.

```

1 // Figura 7.17: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // utiliza um array para armazenar notas de teste.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12 using std::setw;
13
14 #include "GradeBook.h"// definição da classe GradeBook
15
16 // o construtor inicializa courseName e o array grades
17 GradeBook::GradeBook(string name, const int gradesArray[])
18 {
19 setCourseName(name); // inicializa courseName
20
21 // copia notas de gradeArray para membro de dados grades
22 for (int grade = 0; grade < students; grade++)
23 grades[grade] = gradesArray[grade];
24 } // fim do construtor GradeBook
25
26 // função para configurar o nome do curso
27 void GradeBook::setCourseName(string name)
28 {
29 courseName = name; // armazena o nome do curso
30 } // fim da função setCourseName
31
32 // função para recuperar o nome do curso
33 string GradeBook::getCourseName()
34 {
35 return courseName;
36 } // fim da função getCourseName
37
38 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
39 void GradeBook::displayMessage()
40 {
41 // essa instrução chama getCourseName para obter o
42 // nome do curso que esse GradeBook representa
43 cout << "Welcome to the grade book for\n" << getCourseName() << "!"
44 << endl;
45 } // fim da função displayMessage
46
47 // realiza várias operações nos dados
48 void GradeBook::processGrades()
49 {
50 output();
51
52 // chama função getAverage para calcular a nota média
53 cout << "\nClass average is " << setprecision(2) << fixed <<
54 getAverage() << endl;
55
56

```

Figura 7.17 Funções-membro da classe GradeBook manipulam um array de notas.

(continua)

```

57 // chama funções getMinimum e getMaximum
58 cout << "Lowest grade is " << getMinimum() << "\nHighest grade is "
59 << getMaximum() << endl;
60
61 // chama outputBarChart para imprimir gráfico de distribuição de notas
62 outputBarChart();
63 } // fim da função processGrades
64
65 // localiza nota mínima
66 int GradeBook::getMinimum()
67 {
68 int lowGrade = 100 // supõe que a nota mais baixa é 100
69
70 // faz um loop pelo array de notas
71 for (int grade = 0; grade < students; grade++)
72 {
73 // se nota for mais baixa que lowGrade, ela é atribuída a lowGrade
74 if (grades[grade] < lowGrade)
75 lowGrade = grades[grade]; // nova nota mais baixa
76 } // fim do for
77
78 return lowGrade; // retorna nota mais baixa
79 } // fim da função getMinimum
80
81 // localiza nota máxima
82 int GradeBook::getMaximum()
83 {
84 int highGrade = 0; // supõe que a nota mais alta é 0
85
86 // faz um loop pelo array de notas
87 for (int grade = 0; grade < students; grade++)
88 {
89 // se a nota atual for mais alta que highGrade, ela é atribuída a highGrade
90 if (grades[grade] > highGrade)
91 highGrade = grades[grade]; // nova nota mais alta
92 } // fim do for
93
94 return highGrade; // retorna nota mais alta
95 } // fim da função getMaximum
96
97 // determina média para o teste
98 double GradeBook::getAverage()
99 {
100 int total = 0; // inicializa o total
101
102 // soma notas no array
103 for (int grade = 0; grade < students; grade++)
104 total += grades[grade];
105
106 // retorna média de notas
107 return static_cast<double>(total) / students;
108 } // fim da função getAverage
109
110 // gera a saída do gráfico de barras exibindo distribuição de notas
111 void GradeBook::outputBarChart()
112 {

```

Figura 7.17 Funções-membro da classe GradeBook manipulam um array de notas.

(continua)

```

113 cout << "\nGrade distribution:" << endl;
114
115 // armazena freqüência de notas em cada intervalo de 10 notas
116 const int frequencySize = 11;
117 int frequency[frequencySize] = { 0};
118
119 // para cada nota, incrementa a freqüência apropriada
120 for (int grade = 0; grade < students; grade++)
121 frequency[grades[grade] / 10]++;
122
123 // para cada freqüência de nota, imprime barra no gráfico
124 for (int count = 0; count < frequencySize; count++)
125 {
126 // gera a saída do rótulo das barras ("0-9:", ..., "90-99:", "100:")
127 if (count == 0)
128 cout << " 0-9: ";
129 else if (count == 10)
130 cout << " 100: ";
131 else
132 cout << count * 10 << "-" << (count * 10) + 9 << ": ";
133
134 // imprime a barra de asteriscos
135 for (int stars = 0; stars < frequency[count]; stars++)
136 cout << "*";
137
138 cout << endl; // inicia uma nova linha de saída
139 } // fim do for externo
140 } // fim da função outputBarChart
141
142 // gera a saída do conteúdo do array de notas
143 void GradeBook::outputGrades()
144 {
145 cout << "\n\nThe grades are:\n\n";
146
147 // gera a saída da nota de cada aluno
148 for (int student = 0; student < students; student++)
149 cout << "Student " << setw(2) << student + 1 << ":" << setw(3)
150 << grades[student] << endl;
151 } // fim da função outputGrades

```

Figura 7.17 Funções-membro da classe GradeBook manipulam um array de notas.

(continuação)

Um membro de `dados` pode ser acessado a partir da definição da classe e das definições de funções-membro exatamente como qualquer outro membro de dados. Como você logo verá, um membro pode ser acessado fora da classe, mesmo quando nenhum objeto da classe existe, utilizando o nome de classe seguido pelo operador binário de solução `(::)` e pelo nome do membro de dados. Você aprenderá mais sobre o Capítulo 10 sobre Dados.

O construtor da classe (declarado na linha 16 da Figura 7.16 e definido nas linhas 17–24 da Figura 7.17) tem dois parâmetros — o nome do curso e um array de notas. Quando um programa (por exemplo, a Figura 7.18, linha 13 de `main`), o programa passa um array para o construtor, o `GradeBook` os valores no array `passa` para o membro de dados (linhas 22–23 da Figura 7.17). Os valores de nota no array passado poderiam ter sido inseridos a partir do usuário ou lidos de um disco (como discutido no Capítulo 17, “Processamento de arquivo”). Em nosso programa de teste, simplesmente inicializar o array com um conjunto de valores de nota (Figura 7.18, linhas 10–11). Uma vez que as notas estão armazenadas no membro de dados da classe `GradeBook`, todas as funções-membro da classe podem acessar a mesma memória necessária para realizar vários cálculos.

A função-membro `processGrades` (declarada na linha 21 da Figura 7.16 e definida nas linhas 48–63 da Figura 7.17) contém uma série de chamadas de função-membro que geram como saída um relatório que resume as notas. A linha 51 chama a função-membro `outputGrades` para imprimir o conteúdo do array. As linhas 148–150 na função-membro `processGrades` utilizam uma instrução

para gerar saída de cada nota de aluno. Embora os índices de array iniciem em 0, em geral, um professor numeraria os alunos em 1. Portanto, as linhas 149–150 geram saída de o número de aluno para produzir os títulos de nota 1: “Student 2:” e assim por diante.

```

1 // Figura 7.18: fig07_18.cpp
2 // Cria um objeto GradeBook utilizando um array de notas.
3
4 #include "GradeBook.h"// definição da classe GradeBook
5
6 // a função main inicia a execução do programa
7 int main()
8 {
9 // array de notas de aluno
10 int gradesArray[GradeBook::students] =
11 { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13 GradeBook myGradeBook(
14 "CS101 Introduction to C++ Programming"gradesArray);
15 myGradeBook.displayMessage();
16 myGradeBook.processGrades();
17 return 0;
18 } // fim de main

```

Welcome to the grade book for  
CS101 Introduction to C++ Programming!

The grades are:

Student 1: 87

Student 2: 68

Student 4: 100

Student 5: 83

Student 6: 78

Student 7: 85

Student 8: 91

Student 9: 76

Student 10: 87

Class average is 84.90

Lowest grade is 68

Highest grade is 100

Grade distribution:

0-9:

10-19:

20-29:

30-39:

50-59:

60-69: \*

70-79: \*\*

80-89: \*\*\*\*

90-99: \*\*

100: \*

Figura 7.18 Cria um objeto GradeBook utilizando um array de notas, e então invoca a função-membro processGrades para analisá-los.

A função-membro `getAverage` em seguida chama a função-membro `getMinum` (linhas 54–55) para obter a média das notas no array. A função-membro `getMaxim` (declarada na linha 24 da Figura 7.16 e definida nas linhas 98–108) utiliza uma instrução `for` para somar os valores de todos os elementos de calcular a média. Observe que o cálculo da média na linha 107 utiliza membro de dados `const static students` para determinar o número de notas cuja média está sendo calculada.

As linhas 58–59 na função-membro `getAverage` chamam as funções-membro `getMinimum` e `getMaximum` para determinar as notas mais baixas e mais altas de qualquer aluno no exame, respectivamente. Vamos examinar como a função-membro localiza a nota mais baixa. Como a nota mais alta permitida é 100, iniciamos supondo que 100 é a nota mais baixa (linha 68). compararmos cada um dos elementos no array com a nota mais baixa, procurando valores menores. As linhas 71–76 na função- `getMinimum` fazem um loop pelo array, e as linhas 74–75 compara a nota com a menor nota encontrada. Se a nota for menor que `lowGrade`, configurado como essa nota. Quando a linha 78 é executada, é a nota mais baixa no array. A função-membro `getMaximum` (linhas 82–95) funciona de maneira semelhante à função-membro `getMinimum`.

Por fim, a linha 62 na função-membro `getAverage` chama a função-membro `getBarChart` para imprimir um gráfico de distribuição dos dados de notas utilizando uma técnica semelhante àquela na Figura 7.9. Naquele exemplo, calculamos manualmente os valores das linhas 120–121 para imprimir a técnica de 90 e 90 e 100. As figuras 7.10 e 7.11 exibem a saída do programa para os dados de notas de cada categoria. A linha 117 declara a variável `days` para armazenar a frequência de notas em cada categoria de nota. Para cada nota no array `grades`, as linhas 120–121 incrementam o elemento `appropriateDay` para determinar qual elemento incrementar, a linha 122 divide por 10 utilizando a divisão de inteiro. Por exemplo, a linha 121 incrementa `frequency[8]` para atualizar a contagem de notas no intervalo 80–89. As linhas 124–139 a seguir imprimem o gráfico de barras (ver Figura 7.18) com base nos valores. Comparando as linhas 29–30 da Figura 7.9, as linhas 135–136 da Figura 7.17 utilizam um valor `newBar` para determinar o número de asteriscos a exibir em cada barra.

Testando a classe `GradeBook`

O programa da Figura 7.18 cria um objeto da classe `GradeBook` (Figuras 7.16–7.17) utilizando o array `grades` (declarado e inicializado nas linhas 10–11). Observe que utilizamos o operador binária `new` dentro do escopo (`GradeBook`) para acessar a constante `days` da classe `GradeBook`. Utilizamos essa constante aqui para criar um array que tem o mesmo tamanho que `grades`, armazenado como um membro de `GradeBook`. As linhas 13–14 passam um nome do campo `array` para o construtor `GradeBook`. A linha 15 exibe uma mensagem de boas-vindas e a linha 16 invoca a função-membro `processGrades` do objeto `GradeBook`. A saída revela o resumo das notas em

## 7.7 Pesquisando arrays com pesquisa linear

Um programador costuma trabalhar com grandes quantidades de dados armazenados em arrays. Pode ser necessário determinar se um array contém um valor que corresponde a uma chave. O processo de localizar um elemento particular de um array é chamado de **pesquisa**. Nesta seção discutimos a pesquisa linear simples. O Exercício 7.33 no final deste capítulo pede para você implementar uma versão recursiva da pesquisa linear. No Capítulo 20, “Pesquisa e classificação”, apresentamos a pesquisa binária, que é mais eficiente mas mais complexa.

Pesquisa linear

A pesquisa linear (Figura 7.19, linhas 37–44) compara cada elemento de um array (linhas 10). Como o array não está em nenhuma ordem particular, é apenas provável que o valor localizado esteja no primeiro elemento. Em média, o programa deve comparar a chave de pesquisa com metade dos elementos do array. Para determinar que um valor não está no array, o programa deve comparar a chave de pesquisa com cada elemento no array.

O método de pesquisa linear funciona bem para arrays pequenos ou para arrays não classificados (isto é, arrays cujos elementos estão em uma ordem particular). Contudo, para arrays grandes, a pesquisa linear é ineficiente. Se o array é classificado (por exemplo, se os seus elementos estão em ordem crescente), você pode utilizar a técnica de pesquisa binária de alta velocidade sobre a qual apresentamos no Capítulo 20, “Pesquisa e classificação”.

## 7.8 Classificando arrays por inserção

Importância da classificação dos dados. Os sistemas de classificação dos dados permitem que uma organização conte de maneira eficiente a sua população existente. Por exemplo, muitas empresas individuais no final de cada mês. As empresas de telefonia classificam suas listas telefônicas por sobrenome e, dentro desse nome, por primeiro nome para facilitar a localização de números de telefone. Praticamente todas as organizações devem classificar algum tipo de dados e, em muitos casos, quantidades maciças de dados. Classificar dados é um problema intrigante que tem atraído alguns dos mais intensos de pesquisa no campo de ciência da computação. Neste capítulo, discutimos um esquema simples de classificação, e no Capítulo 20, “Pesquisa e classificação”, investigamos esquemas mais complexos que resultam em melhor desempenho. Introduzimos a noção de caracterizar o grau de dificuldade que cada esquema apresenta para realizar sua tarefa.

**Dica de desempenho 7.4**

Às vezes, algoritmos simples fornecem um desempenho pobre. Sua virtude é que eles são fáceis de escrever, testar e depurar. Às vezes, são necessários algoritmos mais complexos para alcançar um desempenho ótimo.

```

1 // Figura 7.19: fig07_19.cpp
2 // Pesquisa linear de um array.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int linearSearch(const int[], int, int); // protótipo
9
10 int main()
11 {
12 const int arraySize = 100 // tamanho do array a
13 int a[arraySize]; // cria o array a
14 int searchKey; // valor a localizar no array a
15
16 for (int i = 0; i < arraySize; i++)
17 a[i] = 2 * i; // cria alguns dados
18
19 cout << "Enter integer search key: " ;
20 cin >> searchKey;
21
22 // tenta localizar searchKey no array a
23 int element = linearSearch(a, searchKey, arraySize);
24
25 // exibe os resultados
26 if (element != -1)
27 cout << "Found value in element " << element << endl;
28 else
29 cout << "Value not found" << endl;
30
31 return 0; // indica terminação bem-sucedida
32 } // fim de main
33
34 // compara a chave com cada elemento do array até que a localização seja
35 // encontrada ou até que o fim do array seja alcançado; retorna o subscrito do
36 // elemento se a chave for encontrada ou -1 caso contrário
37 int linearSearch(const int array[], int key, int sizeOfArray)
38 {
39 for (int j = 0; j < sizeOfArray; j++)
40 if (array[j] == key) // se localizado,
41 return j; // retorna a localização da chave
42
43 return -1; // chave não-localizada
44 } // fim da função linearSearch

```

Enter integer search key: 36  
Found value in element 18

Enter integer search key: 37  
Value not found

Figura 7.19 Pesquisa linear de um array.

### Classificação por inserção

O programa na Figura 7.20 classifica os valores do array de elementos. A técnica que utilizamos é chamada **classificação por inserção**, um algoritmo de classificação simples, mas ineficiente. A primeira iteração desse algoritmo pega o segundo elemento e, se ele for menor que o primeiro, troca-o (**inserindo o elemento** na frente do primeiro elemento). A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos que todos os três elementos estejam na ordem correta. Nesse algoritmo, os elementos do array sencinal estarão classificados.

A linha 13 da Figura 7.20 declara e inicializa os seguintes valores:

```
34 56 4 10 77 51 93 30 5 52
```

O programa primeiro **consulta** a `data[ 1 ]`, cujos valores são 56 e 30, respectivamente. Esses dois elementos já estão em ordem, então o programa continua — se estivessem fora de ordem, o programa trocaria um pelo outro.

Na segunda iteração, o programa **consulta** o `val`. Esse valor é menor que o programa **armazena**

um elemento temporário. O programa **compara** o `val` com o elemento `data[ 1 ]`.

```
4 34 56 10 77 51 93 30 5 52
```

Na terceira iteração, o programa **armazena** o `val` em uma variável temporária. Depois o programa **compara** esse mesmo elemento para a direita porque o programa **move** para a direita um elemento. Quando o programa **compara** que é maior que coloca em `data[ 1 ]`. O array agora está

```
4 10 34 56 77 51 93 30 5 52
```

Utilizando esse algoritmo, na quarta iteração, os primeiros cinco elementos do array sencinal são classificados. Entretanto, talvez eles não estejam nas suas localizações finais porque os menores valores podem ser localizados mais tarde no array.

A classificação é realizada pelas **instruções** 24–39 que itera pelos elementos do array. A cada iteração, a linha 26 temporariamente armazena (**vara** declarada na linha 14) o valor do elemento que será inserido na parte classificada do array. A linha 28 declara e inicializa a variável que monitora onde inserir o elemento. As linhas 31–36 fazem um loop para localizar a posição correta onde o elemento deve ser inserido. O loop termina quando o programa alcançar o início do array ou alcançar um elemento menor que o valor a ser inserido. A linha 34 move um elemento para a direita, e a linha 35 decremente a em que inserir o próximo elemento. Depois que a linha 38 insere o elemento na posição. Quando a instrução nas linhas 24–39 termina, os elementos do array estão classificados.

A principal virtude da classificação por inserção é que ela é fácil de programar; no entanto, sua execução é lenta. Isso fica evidente quando classificamos arrays grandes. Nos exercícios, investigaremos alguns algoritmos alternativos para classificar um array. Informações sobre classificação e pesquisa em profundidade maior no Capítulo 20.

```

1 // Figura 7.20: fig07_20.cpp
2 // Este programa classifica valores de um array em ordem crescente.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 const int arraySize = 10; // tamanho do array a
13 int data[arraySize] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
14 int insert; // variável temporária para armazenar o elemento a inserir
15 cout << "Unsorted array:\n" ;
16
17 // gera saída do array sencinal
18 for (int i = 0; i < arraySize; i++)
19 cout << setw(4) << data[i];
20
21 }
```

Figura 7.20 Classificando um array com classificação por inserção.

(continua)

```

22 // classificação por inserção
23 // itera pelos elementos do array
24 for (int next = 1; next < arraySize; next++)
{
25 insert = data[next]; // armazena o valor no elemento atual
26
27 int moveItem = next; // inicializa a localização para colocar elemento
28
29 // procura a localização em que colocar o elemento atual
30 while ((moveItem > 0) && (data[moveItem - 1] > insert))
31 {
32 // desloca o elemento uma posição para a direita
33 data[moveItem] = data[moveItem - 1];
34 moveItem--;
35 } // fim do while
36
37 data[moveItem] = insert; // lugar em que o elemento é inserido no array
38 } // fim do for
39
40 cout << "\nSorted array:\n" ;
41
42 // gera a saída do array classificado
43 for (int i = 0; i < arraySize; i++)
44 cout << setw(4) << data[i];
45
46 cout << endl;
47 return 0; // indica terminação bem-sucedida
48 } // fim de main

```

Unsorted array:  
34 56 4 10 77 51 93 30 5 52  
Sorted array:  
4 5 10 30 34 51 52 56 77 93

Figura 7 Classificando um array com classificação por inserção.

(continuação)

## 7.9 Arrays multidimensionais

Os arrays multidimensionais com duas dimensões costumam ser utilizados para representar dados organizados de maneira tabular. Para identificar um elemento particular da tabela, devemos especificar dois subscritos. Por convenção, o primeiro identifica a linha do elemento e o segundo identifica a coluna do elemento. Os arrays que requerem dois subscritos para identificar um elemento particular são chamados de arrays 2-D. Observe que arrays multidimensionais podem ter mais que duas dimensões (por exemplo, subscritos). A Figura 7.21 ilustra um array tridimensional, com três dimensões. As linhas e quatro colunas, então dizemos que ele é um array de 3 por 4. O nome geral das arrays é array m por n.

Cada elemento no array é identificado na Figura 7.21 por um nome de elemento, onde o nome do array é *array* e os subscritos que identificam de maneira única o elemento. Os elementos nos nomes dos elementos na linha 0 têm um primeiro subscrito, os nomes dos elementos na coluna 3 têm um segundo subscrito de



### Erro comum de programação 7.12

Referenciar um elemento de array bidimensional corretamente, como `array[0][0]`, é um erro. Na realidade, `array[0][0]` é tratado como `array[0,0]`, porque o C++ avalia a expressão (só considerando um operador vírgula) simplesmente como duas expressões separadas por vírgulas.

Um array multidimensional pode ser inicializado em sua declaração de maneira muito semelhante a um array unidimensional. Por exemplo, um array bidimensional com valores 1 e 2 nos elementos da sua linha 0 e os valores 3 e 4 nos elementos da sua linha 1 poderiam ser declarados e inicializados com

```
int b[2][2] = { { 1, 2 }, { 3, 4 } };
```

Os valores são agrupados por linha entre chaves. Então,  $b[ 0 ][ 1 ]$ , respectivamente, inicializa  $b[ 1 ][ 0 ]$  e  $b[ 1 ][ 1 ]$ , respectivamente. Se houver inicializadores suficientes para uma dada linha, os elementos restantes dessa linha são inicializados como zero.

```
int b[2][2] = { { 1 }, { 3, 4 } };
```

inicializa  $b[ 0 ][ 0 ]$ ,  $b[ 0 ][ 1 ]$ ,  $b[ 1 ][ 0 ]$ ,  $b[ 1 ][ 1 ]$  e  $b[ 2 ][ 2 ]$ .

A Figura 7.22 demonstra a inicialização de arrays bidimensionais em declarações. As linhas 11–13 declaram três arrays, cada com duas linhas e três colunas.

A declaração `a[0]` (linha 11) fornece seis inicializadores em duas sublistas. A primeira sublista inicializa a linha 0 do array com os valores 1, 2 e 3; e a segunda sublista inicializa a linha 1 do array com os valores para 4, 5 e 6. Se as chaves em torno da sublista forem removidas da lista `initializadores`, o compilador inicializa os elementos da linha 0 seguidos pelos elementos da linha 1, produzindo o mesmo resultado.

A declaração `a[1]` (linha 12) fornece somente cinco inicializadores. Os inicializadores são atribuídos à linha 0 e depois à linha 1. Qualquer elemento que não tem um inicializador explícito é inicializado como zero.

A declaração `a[2]` (linha 13) fornece três inicializadores em duas sublistas. A sublista para a linha 0 inicializa explicitamente os primeiros dois elementos da linha 0 como 1 e 2; o terceiro elemento é implicitamente inicializado como zero. A sublista para a linha 1 inicializa explicitamente o primeiro elemento como 4 e inicializa implicitamente os dois últimos elementos como zero.

O programa chama `printf` para gerar a saída dos elementos de cada array. Observe que a definição de função (linhas 27–38) especifica o parâmetro `array[ ][ 3 ]`. Quando uma função recebe um array unidimensional como um argumento, os colchetes do array estão vazios na lista de parâmetros da função. O tamanho da primeira dimensão (isto é, o número de linhas) do array bidimensional também não é requerido, mas todos os tamanhos subsequentes de dimensão são requeridos. O compilador usa esses tamanhos para determinar as localizações na memória de elementos em arrays multidimensionais. Todos os elementos são armazenados consecutivamente na memória, independentemente do número de dimensão. Em um array bidimensional, `array` é armazenada na memória seguida pela linha 1. Em um array bidimensional, cada linha é um array unidimensional. Para localizar um elemento em uma linha particular, a função deve conhecer exatamente quantos elementos estão em cada linha para assim poder calcular adequadamente de posições da memória quando acessar `array`. Portanto, a função deve pular três elementos da linha 0 na memória para obter a linha 1. Entretanto, a função acessa o elemento 2 dessa linha.

Muitas manipulações de array comuns utilizam as instruções `for` e `for` para percorrer a estrutura de array. Por exemplo, a seguinte instrução percorre todos os elementos na linha 2 da Figura 7.21 como zero:

```
for (column = 0; column < 4; column++)
 a[2][column] = 0;
```

A instrução varia apenas o segundo subscrito (isto é, o subscrito de coluna). A seguir, compare a seguinte instrução às seguintes instruções de atribuição:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

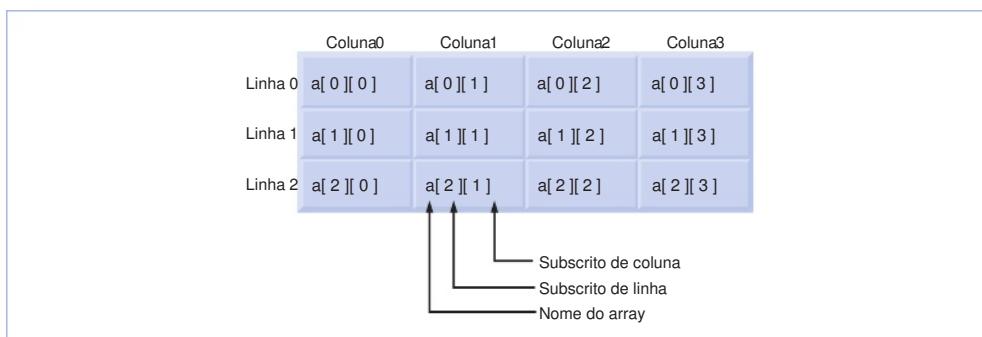


Figura 7.21 O array bidimensional com três linhas e quatro colunas.

```

1 // Figura 7.22: fig07_22.cpp
2 // Inicialização de arrays multidimensionais.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void printArray(const int a[][3]); // protótipo
8
9 int main()
10 {
11 int array1[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
12 int array2[2][3] = { { 1, 2, 3, 4, 5 } };
13 int array3[2][3] = { { 1, 2 }, { 4 } };
14
15 cout << "Values in array1 by row are:" << endl;
16 printArray(array1);
17
18 cout << "\nValues in array2 by row are:" << endl;
19 printArray(array2);
20
21 cout << "\nValues in array3 by row are:" << endl;
22 printArray(array3);
23 return 0; // indica terminação bem-sucedida
24 } // fim de main
25
26 // gera saída do array com duas linhas e três colunas
27 void printArray(const int a[][3])
28 {
29 // faz um loop pelas linhas do array
30 for (int i = 0; i < 2; i++)
31 {
32 // faz um loop pelas colunas da linha atual
33 for (int j = 0; j < 3; j++)
34 cout << a[i][j] << ' ';
35
36 cout << endl; // inicia nova linha de saída
37 } // fim do for externo
38 } // fim da função printArray

```

Values in array1 by row are:

1 2 3  
4 5 6

Values in array2 by row are:

1 2 3  
4 5 0

Values in array3 by row are:

1 2 0  
4 0 0

Figura 7.22 Inicializando arrays multidimensionais.

A seguinte instrução soma o total de todos os elementos no array

```
total = 0;
for (row = 0; row < 3; row++)
 for (column = 0; column < 4; column++)
 total += a[row][column];
```

A instrução soma os elementos do array uma linha por vez. A instrução configura (index), o subscrito da linha) ~~contém~~ assim os elementos da linha 0 podem ser somados. A instrução interna então incrementa ~~row~~ para assim os elementos da linha 1 podem ser somados. ~~Ela~~ assim os elementos da linha 2 podem ser somados. Quando a instrução finaliza, a contém a soma de todos os elementos do array.

## 7.10 Estudo de caso: classe GradeBook utilizando um array bidimensional

Na Seção 7.6, apresentamos a classe `GradeBook` (Figuras 7.16–7.17), que utilizou um array unidimensional para armazenar notas de alunos em um único exame. Na maioria dos semestres, os alunos fazem vários exames. É provável que os professores queiram

Armazenando notas de aluno em um array bidimensional na classe `GradeBook`

As Figuras 7.23–7.24 contêm uma versão da classe que utiliza um array bidimensional para armazenar as notas de vários alunos em múltiplos exames. Cada linha do array representa as notas de um único aluno para o curso inteiro e cada coluna representa todas as notas que os alunos tiraram em um exame particular. ~~Uma~~ Um programa cliente como `main` como um argumento para `getAverage`. Nesse exemplo, utilizamos um array dez por três contendo as notas de três exames de dez alunos.

```
1 // Figura 7.23: GradeBook.h
2 // Definição da classe GradeBook que utiliza um
3 // array bidimensional para armazenar notas de teste.
4 // As funções-membro são definidas em GradeBook.cpp
5 #include <string> // o programa utiliza a classe string da C++ Standard Library
6 using std::string;
7
8 // definição da classe GradeBook
9 class GradeBook
10 {
11 public :
12 // constantes
13 const static int students = 10; // número de alunos
14 const static int tests = 3; // número de testes
15
16 // o construtor inicializa o nome do curso e o array de notas
17 GradeBook(string, const int [[tests]]);
18
19 void setCourseName(string); // função para configurar o nome do curso
20 string getCourseName(); // função para recuperar o nome do curso
21 void displayMessage(); // exibe uma mensagem de boas-vindas
22 void processGrades(); // realiza várias operações nos dados
23 int getMinimum(); // localiza a nota mínima no livro de notas
24 int getMaximum(); // localiza a nota máxima no livro de notas
25 double getAverage(const int [[const int]]); // encontra a média das notas
26 void outputGrades(); // gera a saída do conteúdo do array de notas
27
28 private :
29 string courseName; // nome do curso para esse livro de notas
30 int grades[students][tests]; // array bidimensional de notas
31 };
```

Figura 7.23 Definição da classe `GradeBook` com um array bidimensional para armazenar notas.

```

1 // Figura 7.24: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // utiliza um array bidimensional para armazenar notas.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip> // manipuladores de fluxo parametrizados
11 using std::setprecision; // configura a precisão da saída numérica
12 using std::setw; // configura a largura de campo
13
14 // inclui a definição da classe GradeBook de GradeBook.h
15 #include "GradeBook.h"
16
17 // o construtor de dois argumentos inicializa courseName e array de notas
18 GradeBook::GradeBook(string name, const int gradesArray[][tests])
19 {
20 setCourseName(name); // inicializa courseName
21
22 // copia notas de gradeArray para grades
23 for (int student = 0; student < students; student++)
24
25 for (int test = 0; test < tests; test++)
26 grades[student][test] = gradesArray[student][test];
27 } // fim do construtor GradeBook de dois argumentos
28
29 // função para configurar o nome do curso
30 void GradeBook::setCourseName(string name)
31 {
32 courseName = name; // armazena o nome do curso
33 } // fim da função setCourseName
34
35 // função para recuperar o nome do curso
36 string GradeBook::getCourseName()
37 {
38 return courseName;
39 } // fim da função getCourseName
40
41 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
42 void GradeBook::displayMessage()
43 {
44 // essa instrução chama getCourseName para obter o
45 // nome do curso que esse GradeBook representa
46 cout << "Welcome to the grade book for" << getCourseName() << "!"
47 << endl;
48 } // fim da função displayMessage
49
50 // realiza várias operações nos dados
51 {
52 // gera saída de array de notas
53 outputGrades();
54
55 // chama funções getMinimum e getMaximum

```

Figura 7.24 Definições de função-membro da classe GradeBook manipulam um array bidimensional de notas.

(continua)

```

57 cout << "\nLowest grade in the grade book is " << getMinimum()
58 << "\nHighest grade in the grade book is " << getMaximum() << endl;
59
60 // gera saída do gráfico de distribuição de notas de todas as notas em todos os testes
61 outputBarChart();
62 } // fim da função processGrades
63
64 // localiza nota mínima
65 int GradeBook::getMinimum()
66 {
67 int lowGrade = 100 // supõe que a nota mais baixa é 100
68
69 // faz um loop pelas linhas do array de notas
70 for (int student = 0; student < students; student++)
71 {
72 // faz um loop pelas colunas da linha atual
73 for (int test = 0; test < tests; test++)
74 {
75 // se a nota for menor que lowGrade, atribui a nota a lowGrade
76 if (grades[student][test] < lowGrade)
77 lowGrade = grades[student][test]; // nova nota mais baixa
78 } // fim do for interno
79 } // fim do for externo
80
81 return lowGrade; // retorna nota mais baixa
82 } // fim da função getMinimum
83
84 // localiza nota máxima
85 int GradeBook::getMaximum()
86 {
87 int highGrade = 0; // supõe que a nota mais alta é 0
88
89 // faz um loop pelas linhas do array de notas
90 for (int student = 0; student < students; student++)
91 {
92 // faz um loop pelas colunas da linha atual
93 for (int test = 0; test < tests; test++)
94 {
95 // se a nota atual for maior que highGrade, atribui essa nota a highGrade
96 if (grades[student][test] > highGrade)
97 highGrade = grades[student][test]; // nova nota mais alta
98 } // fim do for interno
99 } // fim do for externo
100
101 return highGrade; // retorna nota mais alta
102 } // fim da função getMaximum
103
104 // determina a média de conjunto particular de notas
105 double GradeBook::getAverage(const int setOfGrades[], const int grades)
106 {
107 int total = 0; // inicializa o total
108
109 // soma notas no array
110 for (int grade = 0; grade < grades; grade++)
111 total += setOfGrades[grade];
112

```

Figura 7.24 Definições de função-membro da classe `GradeBook` manipulam um array bidimensional de notas.

(continua)

```

113 // retorna média de notas
114 return static_cast < double >(total) / grades;
115 } // fim da função getAverage
116
117 // gera a saída do gráfico de barras exibindo distribuição de notas
118 void GradeBook::outputBarChart()
119 {
120 cout << "\nOverall grade distribution:" << endl;
121
122 // armazena freqüência de notas em cada intervalo de 10 notas
123 const int frequencySize = 11;
124 int frequency[frequencySize] = { 0 };
125
126 // para cada nota, incrementa a freqüência apropriada
127 for (int student = 0; student < students; student++)
128
129 for (int test = 0; test < tests; test++)
130 ++frequency[grades[student][test] / 10];
131
132 // para cada freqüência de nota, imprime uma barra no gráfico
133 for (int count = 0; count < frequencySize; count++)
134 {
135 // gera saída do rótulo da barra ("0-9:", ..., "90-99:", "100:")
136 if (count == 0)
137 cout << " 0-9: ";
138 else if (count == 10)
139 cout << " 100: ";
140 else
141 cout << count * 10 << "-" << (count * 10) + 9 << ":" ;
142
143 // imprime a barra de asteriscos
144 for (int stars= 0; stars < frequency[count]; stars++)
145 cout << '*';
146
147 cout << endl; // inicia uma nova linha de saída
148 } // fim do for externo
149 } // fim da função outputBarChart
150
151 // gera a saída do conteúdo do array de notas
152 void GradeBook::outputGrades()
153 {
154 cout << "\nThe grades are:\n\n" ;
155 cout << " " ; // alinha títulos de coluna
156
157 // cria um título de coluna para cada um dos testes
158 for (int test = 0; test < tests; test++)
159 cout << "Test " << test + 1 << " " ;
160
161 cout << "Average" << endl; // título da coluna de média do aluno
162
163 // cria linhas/colunas de texto que representam notas de array
164 for (int student = 0; student < students; student++)
165 {
166 cout << "Student " << setw(2) << student + 1;
167
168 // gera saída de notas do aluno

```

Figura 7.24 Definições de função-membro da classe GradeBook manipulam um array bidimensional de notas.

(continua)

```

169 for (int test = 0; test < tests; test++)
170 cout << setw(8) << grades[student][test];
171
172 // chama a função-membro getAverage para calcular a média do aluno;
173 // passa linha de notas e o valor dos testes como argumentos
174 double average = getAverage(grades[student], tests);
175 cout << setw(9) << setprecision(2) << fixed << average << endl;
176 } // fim do for externo
177 } // fim da função outputGrades

```

**Figura 7.24** Definições de função-membro da classe `GradeBook` manipulam um array bidimensional de notas.

(continuação)

Cinco funções-membro (declaradas nas linhas 23–27 da Figura 7.23) realizam manipulações de array para processar as notas de todos os alunos no semestre. Cada uma dessas funções-membro é semelhante à sua contraparte na versão anterior, baseada em um array unidimensional GradeBook (figuras 7.16–7.17). A função `getMemberGrade` (definida nas linhas 65–82 da Figura 7.24) determina a nota mais baixa de qualquer aluno no semestre. A função `getMemberGrade` (definida nas linhas 85–102 da Figura 7.24) determina a nota mais alta de qualquer aluno no semestre. A função `getAverage` (linhas 105–115 da Figura 7.24) determina a média de um aluno particular no semestre. A função `getMemberBarDinhas` (linhas 118–149 da Figura 7.24) gera saída de um gráfico de barras da distribuição de todas as notas de aluno no semestre. A função `getMemberTabularOutput` (linhas 150–177 da Figura 7.24) gera saída do array bidimensional em um formato tabular, juntamente com a média de cada aluno no semestre.

As funções-membro `getMaximumOutputBarChart` e `outputGrades` iteram pelo array utilizando instruções `for` aninhadas. Por exemplo, considere a função `getMembro` (linhas 70–79). A instrução `for` interna começa configurando o subscrito de linha, assumos que os elementos da linha 0 podem ser comparados com a variável `newGrade` no corpo da instrução externa. A instrução interna itera pelas notas de uma linha particular e compara cada nota com `newGrade`. Se uma nota for menor que `newGrade`, é configurado como essa nota. A extensão não incrementa o subscrito da linha. Os elementos da linha 1 são comparados com `newGrade` na instrução externa, então incrementa o subscrito de linha. Os elementos da linha 2 são comparados com `newGrade` e assim se repete até que todas as linhas tenham sido percorridas. Quando a execução da instrução `getMembro` é encerrada, mais baixa no array bidimensional. A função menciona de maneira semelhante à função `membro`

A função de notas de um setor interno, a qual é idêntica àquela na Figura 7.17, é obtida para gerar saída da distribuição unidimensional com base em todas as notas no array bidimensional. O restante do código em cada uma das funções-membro outputBarChart que exibem o gráfico é idêntico.

A função-membro `printGrades` (linhas 152–177) também utiliza `imprime`s para gerar a saída de valores do array `grades` além da média semestral de cada aluno. A saída na Figura 7.25 mostra o resultado, que é semelhante ao formato tabular livro de notas de um professor de física. As linhas 158–159 imprimem os títulos de coluna para cada teste. Utilizamos uma `for` controlada por contador para podermos identificar cada teste com um número. De maneira semelhante, a instrução 164–176 primeiro gera a saída de um rótulo de linha utilizando uma variável contadora para identificar cada aluno (linha 166). E os índices de array iniciem em 0, observe que as linhas 159 e 160 geram saídas respectivamente, para produzir números de teste e de aluno que iniciam em 1 (ver Figura 7.25). As linhas 160–170 utilizam a variável contadora dentro da instrução externa para fazer um loop por uma linha específica de saída da nota de teste de cada aluno. Por fim, a linha 174 obtém a média do semestre de cada aluno (utilizando a linha atual de `dent ]`) para a função-membro `getAverage`.

```
1 // Figura 7.25: fig07_25.cpp
2 // Cria objeto GradeBook utilizando um array bidimensional de notas.
3 //include "GradeBook.h"// Definição da classe GradeBook
4
5 // a função main inicia a execução do programa
6 int main()
7 {
```

**Figura 7.25** Cria um objeto GradeBook utilizando um array bidimensional de notas, então invoca a função-membro `getGrades` para analisá-los.

(continua)

```

8 {
9 // array bidimensional de notas de aluno
10 int gradesArray[GradeBook::students][GradeBook::tests] =
11 { { 87, 96, 70},
12 { { 68, 87, 90},
13 { { 94, 100, 90},
14 { { 100, 81, 82},
15 { { 83, 65, 85},
16 { { 78, 87, 65},
17 { { 85, 75, 83},
18 { { 91, 94, 100},
19 { { 76, 72, 84},
20 { { 87, 93, 73}};

21 GradeBook myGradeBook(
22 "CS101 Introduction to C++ Programming"gradesArray);
24 myGradeBook.displayMessage();
25 myGradeBook.processGrades();
26 return 0; // indica terminação bem-sucedida
27 } // fim de main

```

Welcome to the grade book for  
CS101 Introduction to C++ Programming!

The grades are:

|                  | Test 1     | Test 2    | Test 3    | Average      |
|------------------|------------|-----------|-----------|--------------|
| Student 1        | 87         | 96        | 70        | 84.33        |
| Student 2        | 68         | 87        | 90        | 81.67        |
| Student 3        | 94         | 100       | 90        | 94.67        |
| <b>Student 5</b> | <b>180</b> | <b>65</b> | <b>82</b> | <b>87.67</b> |
| Student 6        | 78         | 87        | 65        | 76.67        |
| Student 7        | 85         | 75        | 83        | 81.00        |
| Student 8        | 91         | 94        | 100       | 95.00        |
| Student 9        | 76         | 72        | 84        | 77.33        |
| Student 10       | 87         | 93        | 73        | 84.33        |

Lowest grade in the grade book is 65

Highest grade in the grade book is 100

Overall grade distribution:

- 0-9:
- 10-19:
- 20-29:
- 30-39:
- 40-49:
- 50-59:
- 60-69: \*\*\*
- 70-79: \*\*\*\*\*
- 80-89: \*\*\*\*\*
- 90-99: \*\*\*\*\*
- 100: \*\*\*

Figura 7.25 Cria um objeto GradeBook utilizando um array bidimensional de notas, então invoca a função-membro processGrades para analisá-los.

(continuação)

A função-membro `average` (linhas 105–115) aceita dois argumentos — um array unidimensional de resultados de teste para um aluno particular e o número de resultados de teste no array. Quando a dimensão é chamada por [student], que especifica que uma linha particular do array deve ser passada para. Por exemplo, baseado no array criado na Figura 7.25, os argumentos representam os três valores (um array unidimensional de notas) armazenados na `l1d`. Um array bidimensional pode ser considerado um array cujos elementos são arrays de uma dimensão. A função `average` calcula a soma dos elementos do array, divide o total pelo número de resultados do teste e retorna o resultado de ponto flutuante (`float`) Valor.

Testando a classe `GradeBook`

O programa na Figura 7.25 cria um objeto da classe 7.23–7.24) utilizando o array bidimensional gradesArray (declarado e inicializado nas linhas 10–20). Observe que a linha 21 chama as rotinas classe `GradeBook` para indicar o tamanho de cada dimensão do array. As linhas 22–23 passam um nome do curso e o gradesArray para o construtor `GradeBook`. As linhas 24–25 então invocam as funções `welcome` e `processGrades`.

`welcome` para exibir uma mensagem de boas-vindas e obter um relatório que resume as notas semestrais dos alunos, respe-

## 7.11 Introdução ao template vector da C++ Standard Library

Agora, introduzimos o template C++ Standard Library, que representa um tipo de array mais robusto que possui muitas capacidades adicionais. Como você verá em capítulos posteriores e em cursos mais avançados de C++, arrays baseados em p-estilo do C (isto é, o tipo de array apresentado até aqui) têm grande potencial para erros. Por exemplo, como mencionado anteriormente, um programa pode facilmente ‘ultrapassar’ qualquer uma das extremidades de um array, porque o C++ não verifica se os sub-caem fora do intervalo de um array. Dois arrays não podem ser significativamente comparados com operadores de igualdade nem operadores relacionais. Como você aprenderá no Capítulo 8, variáveis de ponteiro (conhecidas mais comumente como ponteiros) têm endereços de memória como seus valores. Os nomes de array são simplesmente ponteiros para onde os arrays iniciam na memória, naturalmente, dois arrays sempre estarão em posições de memória diferentes. Quando um array é passado para uma função geral projetada para lidar com arrays de qualquer tamanho, o tamanho do array deve ser passado como um argumento adicional. Isso, um array não pode ser atribuído a outro com o(s) operador(es) de atribuição. As notações de array são ponteiros e, você aprenderá no Capítulo 8, um ponteiro constante não pode ser utilizado no lado esquerdo do operador de atribuição. Essas capacidades certamente parecem ‘naturais’ para lidar com arrays, mas o C++ não fornece tais capacidades. Entretanto, a C++ Standard Library fornece o template de classe `vector` para permitir que os programadores criem uma alternativa mais poderosa e menos propensa a erros para arrays. No Capítulo 11, “Sobrecarga de operadores; objetos string e array”, apresentamos os modos como implementar capacidades de array da maneira como se você estiver se perguntando a personalizar operadores para utilizar com suas próprias classes (uma conhecida técnica de sobrecarga de operador).

O template de classe `vector` está disponível para qualquer pessoa que constrói aplicativos com C++. As notações que o exemplo `vector` utiliza talvez sejam pouco conhecidas para você, já que a notação de template. Lembre-se de que a Seção 6.18 discutiu templates de função. No Capítulo 14, discutimos templates de classe. Por enquanto, você deve se sentir confortável com a notação de template de classe, simulando a sintaxe no exemplo que mostramos nesta seção. Você aprofundará seu entendimento à medida que estudarmos templates de classe no Capítulo 14. O Capítulo 23 apresenta templates de classe contêiner do padrão C++ em detalhes.

O programa da Figura 7.26 demonstra as capacidades fornecidas pelo Standard Library que não estão disponíveis para arrays baseados em ponteiro no estilo do C. O template de classe `vector` nos recursos que a classe `vector`, que construímos no Capítulo 11, “Sobrecarga dos operadores; objetos string e array”. O template de classe padrão `vector` é definido no cabeçalho (linha 11) e pertence ao namespace (2). O Capítulo 23 discute a funcionalidade completa do template de classe padrão.

As linhas 19–20 criam dois objetos que armazenam valores de tipo `int`. A classe `vector` contém sete elementos e contém 10 elementos. Por padrão, todos os elementos são do tipo `int`. Observe que os valores podem ser definidos para armazenar qualquer tipo de dados, substituindo pelo tipo de dados apropriado. Essa notação, que especifica o tipo armazenado, é semelhante à notação de template que a Seção 6.18 introduziu com templates de função. Novamente, o Capítulo 14 discute essa sintaxe em detalhe.

A linha 23 utiliza a função `operator[]` (linhas 28–30), que é utilizada para obter e alterar o elemento do array como um valor que pode ser utilizado para saída. Observe a semelhança dessa notação com a notação utilizada para acessar um elemento do array. As linhas 28 e 30 realizam as seguintes tarefas para:

A função-membro template de classe `vector` retorna o número de elementos comum valor de tipo (que representa o tipo `int` em muitos sistemas). Como resultado, a linha 90 também declara uma variável de controle `size_t`. Em alguns compiladores, `cout << m` faz com que o compilador emita uma mensagem de advertência, uma vez que a condição de continuação do loop (linha 92) não indica um valor válido. Isso ocorre porque `m` é um `unsigned` (isto é, um valor do tipo `size_t` retornado pela função `size`).

```

1 // Figura 7.26: fig07_26.cpp
2 // Demonstrando o template de classe vector da C++ Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <vector>
12 using std::vector;
13
14 void outputVector(const vector< int > &); // exibe o vetor
15 void inputVector(vector< int > &); // insere valores no vetor
16
17 int main()
18 {
19 vector< int > integers1(7); // vector< int > de 7 elementos
20 vector< int > integers2(10); // vector< int > de 10 elementos
21
22 // imprime o tamanho e o conteúdo de integers1
23 cout << "Size of vector integers1 is " << integers1.size()
24 << endl;
25 outputVector(integers1);
26
27 // imprime o tamanho e o conteúdo de integers2
28 cout << "Size of vector integers2 is " << integers2.size()
29 << endl;
30 outputVector(integers2);
31
32 // insere e imprime integers1 e integers2
33 cout << "\nEnter 17 integers:" << endl;
34 inputVector(integers1);
35 inputVector(integers2);
36
37 cout << "\nAfter input, the vectors contain:\n"
38 << "integers1:" << endl;
39 outputVector(integers1);
40 cout << "integers2:" << endl;
41 outputVector(integers2);
42
43 // utiliza operador de desigualdade (!=) com objetos vector
44 cout << "\nEvaluating: integers1 != integers2" << endl;
45
46 if (integers1 != integers2)
47 cout << "integers1 and integers2 are not equal" << endl;
48
49 // cria o vector integers3 utilizando integers1 como um
50 // inicializador; imprime tamanho e conteúdo
51 // inicializador; imprime tamanho e conteúdo // construtor de cópia
52
53 cout << "Size of vector integers3 is " << integers3.size()
54 << endl;
55 outputVector(integers3);
56

```

Figura 7.26 Template vector da Standard Library.

(continua)

```

57 // utiliza operador atribuição (=) sobrecarregado
58 cout << "nAssigning integers2 to integers1:" << endl;
59 integers1 = integers2; // integers1 é maior que integers2
60
61 cout << "integers1:" << endl;
62 outputVector(integers1);
63 cout << "integers2:" << endl;
64 outputVector(integers2);
65
66 // utiliza operador de igualdade (==) com objetos vector
67 cout << "nEvaluating: integers1 == integers2" << endl;
68
69 if (integers1 == integers2)
70 cout << "integers1 and integers2 are equal" << endl;
71
72 // utiliza colchetes para criar rvalue
73 cout << "nintegers1[5] is " << integers1[5];
74
75 // utiliza colchetes para criar lvalue
76 cout << "n\nAssigning 1000 to integers1[5]" << endl;
77 integers1[5] = 1000
78 cout << "integers1:" << endl;
79 outputVector(integers1);
80
81 // tentativa de utilizar subscrito fora do intervalo
82 cout << "nAttempt to assign 1000 to integers1.at(15)" << endl;
83 integers1.at(15) = 1000; // ERRO: fora do intervalo
84 return 0;
85 } // fim de main
86
87 // gera saída do conteúdo do vetor
88 void outputVector(const vector< int > &array)
89 {
90 size_t i; // declara a variável de controle
91
92 for (i = 0; i < array.size(); i++)
93 {
94 cout << setw(12) << array[i];
95
96 if ((i + 1) % 4 == 0) // 4 números por linha de saída
97 cout << endl;
98 } // fim do for
99
100 if (i % 4 != 0)
101 cout << endl;
102 } // fim da função outputVector
103
104 // insere o conteúdo de vetor
105 void inputVector(vector< int > &array)
106 {
107 for (size_t i = 0; i < array.size(); i++)
108 cin >> array[i];
109 } // fim da função inputVector

```

Figura 7.26 Tela da Standard Library.

(continua)

```

Size of vector integers1 is 7
vector after initialization:
 0 0 0 0
 0 0 0 0

Size of vector integers2 is 10
vector after initialization:
 0 0 0 0
 0 0 0 0
 0 0 0 0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
 1 2 3 4
 5 6 7 8
integers2:
 8 9 10 11
 12 13 14 15
 16 17 18 19

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
 1 2 3 4
 5 6 7 8

Assigning integers2 to integers1:
integers1:
 8 9 10 11
 12 13 14 15
 16 17 18 19
integers2:
 8 9 10 11
 12 13 14 15
 16 17 18 19

Evaluating: integers1 == integers2
integers1 and integers2 are equal
integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
 8 9 10 11
 12 1000 14 15
 16 17 18 19

Attempt to assign 1000 to integers1.at(15)

abnormal program termination

```

Figura 7.26 Template vector.

(continuação)

As linhas 34–35 passam `integers2` à função `putVector` (linhas 105–109) para ler os valores dos elementos de cada vetor fornecidos pelo usuário. A função utiliza colchetes para obter os valores que podem ser utilizados para armazenar os valores de entrada em cada elemento do vetor.

A linha 46 demonstra que objetos podem ser comparados diretamente se os operadores de comparação dos dados não forem iguais, o operador `retorna` contrário, o operador `retorna` retorna.

O template da C++ Standard Library permite que os programadores criem um novo objeto com o conteúdo de um existente. A linha 51 cria um `vector<integers3>` e o inicializa com uma cópia desse invoca o chamado construtor de cópia para realizar a operação de cópia. Você aprenderá sobre construtores de cópia em detalhe no Capítulo 11. As linhas 53 e 55 geram saída do conteúdo para mostrar que ele foi inicializado corretamente.

A linha 59 atribui `integers2` a `integers1` para demonstrar que o operador `=` pode ser utilizado com objetos. As linhas 62 e 64 geram saída do conteúdo de ambos os objetos para mostrar que eles agora contêm valores idênticos. A linha 66 compara `integers1` com `integers2` utilizando o operador de igualdade para determinar se os conteúdos dos dois objetos são iguais.

As linhas 73–77 dão trabalho! Linha 59 (`integers1`) programa para utilizar o conteúdo como um `value` não modificável como `value` modifiable respectivamente. `value` modifiable é uma expressão que identifica um objeto na memória (como um elemento de `vector`), mas não pode ser utilizado para modificar esse objeto. `value` também identifica um objeto na memória, mas pode ser utilizado para modificar o objeto. Como é o caso com arrays baseados em ponteiro do C, o C++ não realiza qualquer verificação de limite quando se usam colchetes. Portanto, o programador deve assegurar que operações de membro não tentem manipular elementos `fora` dos limites do template de classe. No entanto, na realidade fornece limite para essa verificação, que é a função-membro ‘`exception`’ (consulte o Capítulo 16, “Tratamento de exceções”) se seu argumento é um subscrito inválido. Por padrão, isso faz com que o programa C++ termine. Se o subscrito for válido, a localização na localização especificada é alterada. Em outras palavras, se o subscrito for inválido, dependendo do contexto, nem que a chamada aparece. A linha 83 demonstra uma chamada à função `value` com um subscrito inválido.

Nesta seção, demonstramos o template Standard Library, uma classe reutilizável e robusta que pode substituir arrays baseados em ponteiro no estilo do C. No Capítulo 11, `arrays` e suas capacidades ‘sobrerecarregando’ operadores predefinidos do C++ e aprenderá a personalizar operadores para utilizar com suas próprias classes de maneiras ser. Por exemplo, criamos uma classe `array`, como o template da classe `vector`, para aumentar as capacidades básicas de array. Nossa classe `Array` também fornece recursos adicionais, como a capacidade de realizar entrada e saída de arrays inteiros com operadores respectivamente.

## 7.12 Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional)

Nesta seção, nós nos concentramos nas colaborações (interações) entre objetos em nosso sistema de ATM. Quando dois objetos precisam para realizar uma tarefa, diz-se que eles são colaboradores. Os objetos fazem isso invocando as operações um do outro. Uma colaboração consiste em um objeto de uma classe enviar uma mensagem para um objeto de outra classe. As mensagens são enviadas em C++ via chamadas de função-membro.

Na Seção 6.18, determinamos muitas das operações das classes em nosso sistema. Nesta seção, focalizamos as mensagens que invocam essas operações. Para identificar as colaborações no sistema, retornamos ao documento de requisitos da Seção 2.8. Isto é de que esse documento especifica a série de atividades que ocorre durante uma sessão ATM (por exemplo, autenticar um usuário, efetuar transações). Os passos utilizados para descrever como o sistema deve realizar cada uma dessas tarefas são nossa primeira indicação das colaborações em nosso sistema. À medida que avançamos por esta e pelas seções restantes do “Estudo de caso de engenharia de software”, podemos descobrir colaborações adicionais.

Identificando as colaborações em um sistema

Identificamos as colaborações no sistema lendo cuidadosamente as seções do documento de requisitos que especificam o que deve fazer para autenticar um usuário e realizar todo tipo de transação. Para cada ação ou passo descrito no documento de requisitos, decidimos quais objetos em nosso sistema devem interagir para alcançar o resultado desejado. Identificamos um objeto como o emissor (isto é, o objeto que envia a mensagem) e outro como o objeto receptor (isto é, o objeto que oferece essa operação para a classe). Então selecionamos uma das operações do objeto receptor (identificadas na Seção 6.18) que devem ser invocadas pelo emissor para produzir o comportamento adequado. Por exemplo, o ATM exibe uma mensagem de boas-vindas quando desliga. Sabemos que um objeto da classe `ATM` envia uma mensagem para o usuário via `playMessage`. Portanto, decidimos que o sistema pode exibir uma mensagem de boas-vindas empregando o método `playMessage` da classe `ATM`. A mensagem `playMessage` para a screen é invocando a operação `playMessage` da classe `Screen`. [Nota: Para evitar repetir a frase ‘um objeto da classe...’, nós nos referimos a cada objeto simplesmente utilizando seu nome de classe precedido (ou não) por um ponto (‘um’/‘uma’ ou ‘o’/‘a’) — por exemplo, ‘referir-se a um objeto da classe’]

A Figura 7.27 lista as colaborações que podem ser derivadas do documento de requisitos. Para cada objeto emissor, lista colaborações na ordem em que elas são discutidas no documento de requisitos. Listamos cada colaboração envolvendo um metente, uma única mensagem e um único destinatário somente uma vez, mesmo que a colaboração possa ocorrer várias vez uma sessão de ATM. Por exemplo, a primeira linha na Figura 7.27 indica que sempre que precisar exibir uma mensagem para o usuário.

Vamos considerar as colaborações na Figura 7.27. Antes de permitir que um usuário realize qualquer transação, o ATM dev um prompt pedindo para o usuário inserir um número de conta e um PIN. Ele realiza cada uma dessas tarefas enviando uma n displayMessageParaScreen. Essas duas ações referem-se à mesma colaboração que já está listada na Figura 7.27. Obtém a entrada em resposta a um prompt enviado para o usuário. Em seguida, o ATM deve determinar se o número de conta especificado pelo usuário e o PIN correspondem aqueles de uma conta no banco de dados. Enviando uma mensagem authenticateUser para BankDatabase. Lembre-se de que o banco de dados não pode autenticar um usuário diretamente — somente o usuário (isto é, o que contém o número da conta especificado pelo usuário) pode acessar o PIN do usuário para autenticar o usuário. Portanto, a Figura 7.27 lista uma colaboração com mensagem

Depois de inserir o PIN para autenticar o usuário, o ATM manda o menu principal enviando uma série de mensagens para Screen e obtém entrada contendo uma seleção de menu enviando uma mensagem para Keypad. Vamos explicar essas colaborações. Depois que o usuário escolher o tipo de transação enviando uma mensagem para o objeto da classe de transação apropriada (BalanceInquiry, Withdrawal ou Deposit). Por exemplo, se o usuário escolher realizar uma consulta de saldo, envia uma mensagem para BalanceInquiry.

O exame adicional do documento de requisitos revela as colaborações envolvidas na execução de cada tipo de transação. BalanceInquiry recupera a quantia de dinheiro disponível na conta do usuário enviando uma mensagem para BankDatabase que responde enviando uma mensagem para Account. De maneira semelhante, BalanceInquiry recupera o valor do depósito enviando uma mensagem para BankDatabase que envia a mesma mensagem ao usuário. Para exibir os dois tipos de saldo do usuário em seu tempo, mensagem displayMessageParaScreen.

Withdrawal envia uma série de mensagens para Screen para exibir um menu de valores de saque-padrão (isto é, \$ 20, \$ 40, \$ 60, \$ 100, \$ 200). Withdrawal envia uma mensagem para Keypad para obter a seleção de menu do usuário. Em seguida, Withdrawal determina se o valor do saque solicitado é menor que ou igual ao saldo disponível no usuário.

| Um objeto da classe... | envia a mensagem...                                                                                     | para um objeto da classe...                                                        |
|------------------------|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| ATM                    | displayMessage<br>getInput<br>authenticateUser<br>execute<br>execute<br>execute                         | Screen<br>Keypad<br>BankDatabase<br>BalanceInquiry<br>Withdrawal<br>Deposit        |
| BalanceInquiry         | getAvailableBalance<br>getTotalBalance<br>displayMessage                                                | BankDatabase<br>BankDatabase<br>Screen                                             |
| Withdrawal             | displayMessage<br>getInput<br>getAvailableBalance<br>isSufficientCashAvailable<br>debit<br>dispenseCash | Screen<br>Keypad<br>BankDatabase<br>CashDispenser<br>BankDatabase<br>CashDispenser |
| Deposit                | displayMessage<br>getInput<br>depositReceived<br>credit                                                 | Screen<br>DepositSlot<br>BankDatabase                                              |
| BankDatabase           | validatePIN<br>getAvailableBalance<br>getTotalBalance<br>debit<br>credit                                | Account<br>Account<br>Account<br>Account<br>Account                                |

Figura 7.27 Colaborações no sistema ATM.

valor disponível na conta do usuário enviando uma mensagem para `BankDatabaseWithdrawal`, então testa se o dispensador de cédulas contém dinheiro suficiente enviando uma mensagem para `CashDispenserWithdrawal` envia uma mensagem para `BankDatabase` para diminuir o saldo da conta do usuário. Por sua vez, envia a mesma mensagem para apropriada. Lembre-se de que debita `BankDatabase` `availableBalance` como `availableBalance`. Para liberar o valor solicitado, envia uma mensagem para `CashDispenser` por fim, `Withdrawal` envia uma mensagem para `Screen`, que instrui o usuário a pegar o dinheiro.

`Deposit` responde a uma mensagem enviando primeiro uma mensagem para `Screen` para solicitar ao usuário o valor do depósito. Envia uma mensagem para `Keypad` para obter a entrada do usuário. Então envia uma mensagem para `Screen` para instruir o usuário a inserir um envelope de depósito. Para determinar se a abertura de depósito recebeu um envelope de depósito, uma mensagem para `DepositSlot`. `Deposit` atualiza a conta do usuário enviando uma mensagem para `BankDatabase`, que subsequente mente envia uma mensagem do usuário. Lembre-se de que creditará automaticamente `availableBalance` mas não `availableBalance`.

#### Diagramas de interação

Agora que identificamos um conjunto de colaborações possíveis entre os objetos em nosso sistema ATM, vamos modelar graficamente essas interações utilizando a UML. A UML fornece vários tipos de diagramas que modelam o comportamento de um sistema modelando a maneira como os objetos interagem entre si. [Os diagramas de comunicação foram chamados de 'colaborações' nas versões anteriores da UML.] Como o diagrama de comunicação mostra as colaborações entre objetos, é útil pensar que as mensagens são enviadas entre objetos ao longo do tempo.

#### Diagramas de comunicação

A Figura 7.28 mostra um diagrama de comunicação que modela a operação `BalanceInquiry`. Os objetos são modelados na UML como retângulos que contêm nomes de forma `NomeDaClasse`. Nesse exemplo, que envolve apenas um objeto de cada tipo, não damos importância ao nome de objeto e listamos apenas dois-ponto `Nota Acima` pelo nome da classe. [múltiplos objetos do mesmo tipo, recomenda-se especificar o nome de cada objeto em um diagrama de comunicação.] Os objetos comunicam-se conectados com linhas sólidas e as mensagens são passadas entre os objetos ao longo dessas linhas na direção por setas. O nome da mensagem, que aparece junto à seta, é o nome de uma operação (isto é, uma função-membro) que pelo objeto receptor — considere o nome como um serviço que o objeto receptor fornece para enviar objetos (seus 'clientes').

A seta de preenchimento sólido na Figura 7.28 representa uma mensagem UML e uma chamada de função em C++.

Vista que essa é uma chamada síncrona, o objeto emissor pode não enviar outra mensagem, ou não fazer coisa alguma, até que receptor processe a mensagem e retorne o controle para o objeto emissor. O emissor simplesmente espera. Por exemplo, na Figura 7.28, a função-membro `BalanceInquiry` é chamada e retorna o controle para o objeto emissor. Se isso fosse uma chamada assíncrona, representada por uma seta, o objeto emissor não teria de esperar o objeto receptor para retornar o controle — ele continuaria enviando mensagens adicionais imediatamente após a assíncrona. Chamadas assíncronas frequentemente podem ser implementadas em C++ utilizando bibliotecas específicas de pluggable fornecidas com seu compilador. Tais técnicas estão além do escopo deste livro.]

#### Seqüência de mensagens em um diagrama de comunicação

A Figura 7.29 mostra um diagrama de comunicação que modela as interações entre objetos no sistema quando um objeto de `BalanceInquiry` é executado. Supomos que o atributo do objeto contém o número da conta do usuário atual. As colaborações na Figura 7.29 iniciam depois de uma mensagem para `BalanceInquiry` (isto é, a interação modelada na Figura 7.28). O número à esquerda de um nome de mensagem indica a ordem numérica do menor para o maior. Nesse diagrama, a numeração de mensagens em um diagrama de comunicação progride em ordem numérica do menor para o maior. Nesse diagrama, a numeração inicia com a mensagem 1 com a mensagem `BalanceInquiry` primeiro envia uma mensagem para `BankDatabase` para `BankDatabase` (mensagens de envio de mensagem). TotalBalance para `BankDatabase` (mensagem). Dentro dos parênteses que se seguem a um nome de mensagem, podemos especificar uma lista separada por vírgulas dos nomes dos parâmetros que são enviados com a mensagem (isto é, argumentos em uma chamada de função em C++). O atributo `BankDatabase` para indicar quais informações devem ser recuperar. A partir da Figura 6.33,

lembre-se de que as operações `getAvailableBalance` e `getTotalBalance` da classe `BankDatabase` exigem ambas um parâmetro para

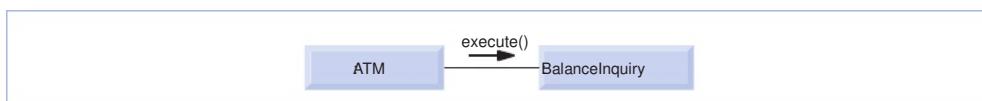


Figura 7.28 Diagrama de comunicação do ATM que executa uma consulta de saldo.

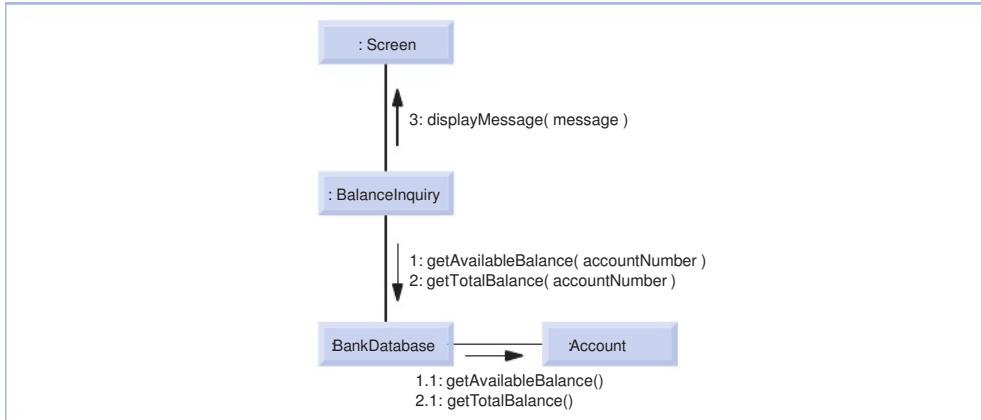


Figura 7.29 Diagrama de comunicação para executar uma consulta de saldo.

identificar uma conta. O parâmetro `accountNumber` é passado para `getAvailableBalance` e `getTotalBalance` para o usuário, passando uma mensagem `displayMessage` para o `Screen` que inclui um parâmetro que indica o que deve ser exibido.

Note, porém, que a Figura 7.29 modela duas mensagens aninhadas que são passadas dentro do mesmo objeto. Para fornecer os dois saldos da conta ao usuário (como solicitado pelas mensagens), deve passar `getAvailableBalance` e uma mensagem `getTotalBalance` para a conta do usuário. Essas mensagens passadas dentro do tratamento de outra mensagem são chamadas de `mensagens aninhadas`. A UML recomenda utilizar um esquema de numeração decimal para indicar mensagens aninhadas. Por exemplo, a primeira mensagem aninhada na mesma classe passa uma mensagem `getAvailableBalance` durante o processamento de uma mensagem com o mesmo nome. Se o `BankDatabase` precisasse passar uma segunda mensagem aninhada durante o processamento de uma mensagem, por exemplo, `getTotalBalance`, uma mensagem só poderia passar quando todas as mensagens aninhadas da mensagem anterior forem passadas.

O esquema aninhado de numeração utilizado nos diagramas de comunicação ajuda a esclarecer precisamente quando e contexto cada mensagem é passada. Por exemplo, se numerássemos as mensagens na Figura 7.29 utilizando um esquema de simples (isto é, 1, 2, 3, 4, 5), alguém examinando o diagrama poderia não ser capaz de determinar qual mensagem `getAvailableBalance` (mensagem 1) é passada para a conta durante o processamento da mensagem 2. Da mesma forma, a oposição decimal aninhados deixam claro que a segunda mensagem `getTotalBalance` (mensagem 2) é passada para a conta dentro do tratamento da primeira mensagem 1 (mensagem 1) pelo `BankDatabase`.

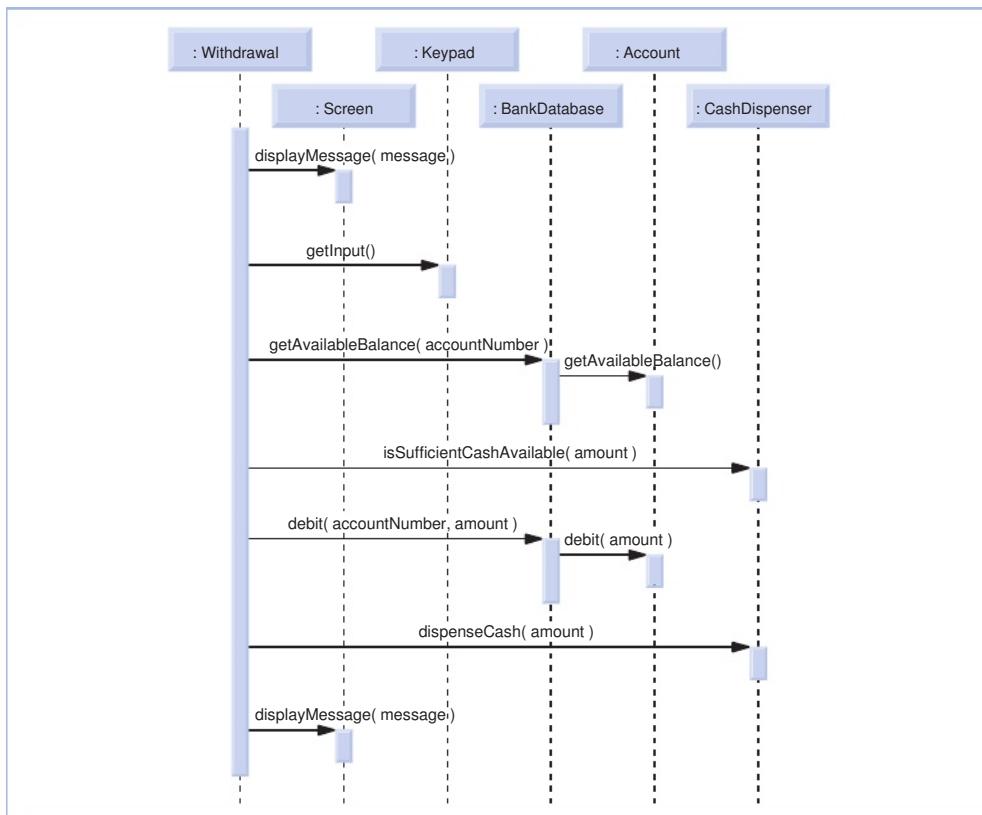
#### Diagramas de seqüências

Os diagramas de comunicação enfatizam os participantes de colaborações, mas modelam sua sincronização de uma maneira complicada. Um diagrama de seqüências ajuda a modelar a sincronização de colaborações mais claramente. A Figura 7.30 é um diagrama de seqüências modelando a seqüência de interações que ocorre quando o usuário clica no botão "Saque" no formulário que se estende para baixo do retângulo de um objeto, que representa a progressão de tempo. As ações geralmente ocorrem ao longo da linha da vida de um objeto na ordem cronológica de cima para baixo — uma ação próxima da parte superior acontece antes de uma próxima da parte inferior.

A passagem de mensagem em diagramas de seqüências é semelhante à passagem de mensagem em diagramas de comunicação.

**Este é o diagrama de seqüência que mostra a execução de uma operação entre os objetos. Ele mostra o controle de retorno de uma operação entre os objetos. Quando um objeto retorna o controle, uma mensagem de retorno, representada como uma linha com uma seta, estende-se desde a ativação do objeto que está retornando o controle até a ativação do objeto que inicialmente enviou a mensagem. Para eliminar a poluição visual, omitimos as setas de retorno de mensagem — a UML permite essa prática para diagramas mais legíveis. Como os diagramas de comunicação, os diagramas de seqüências podem indicar parâmetros de mensagem entre os parênteses que se seguem a um nome de mensagem.**

A seqüência de mensagens na Figura 7.30 inicia quando o usuário que escolha um valor de saque enviando uma mensagem `displayMessage` para o `Screen`. O `Screen` então envia uma mensagem `withdrawalValue` para o `Keypad` que obtém a entrada do usuário. Já modelamos a lógica de controle no diagrama de atividades da Figura 5.28, então não mostraremos mais.



**Figura 7.30** Diagrama de seqüências que modela a withdrawalem execução.

mos essa lógica no diagrama de seqüências da Figura 7.30. Em vez disso, modelamos o cenário mais favorável em que o saldo do usuário é maior que ou igual ao valor retirado escolhido e o dispensador de cédulas contém uma quantia suficiente de dinheiro para atender à solicitação. Para informações sobre como modelar a lógica de controle em um diagrama de seqüências, consulte os links da Web e as leituras recomendadas listadas no fim da Seção 2.8.

Depois de obter o valor de `availableBalance`, envia uma mensagem `getBalance` para `BankDatabase`, que por sua vez envia uma mensagem `availableBalance` para o `Account` do usuário. Supondo que a conta do usuário tem dinheiro suficiente disponível para permitir a transação, em seguida envia uma mensagem `cashAvailable` para o `CashDispenser`. Supondo que há dinheiro suficiente disponível nua o saldo da conta do usuário (`availableBalance` é maior ou igual a `availableBalance`) enviando uma mensagem `getBalance` para `BankDatabase`. `BankDatabase` responde enviando uma mensagem `getBalance` para o `Account` do usuário. Por fim, o `BankDatabase` envia uma mensagem `cashAvailable` para o `CashDispenser`, e este envia uma mensagem `getBalance`

Mesmo com os primeiros passos feitos, é hora de pensar no que mais podemos fazer para aprimorar o sistema. Vamos analisar as interações entre os objetos ATM e Banco. Para isso, vamos descrever a interação entre esses dois sistemas.

Exercícios de revisão do estudo de caso de engenharia de software

- 7.1 Um(a)\_\_\_\_\_ consiste em um objeto de uma classe que envia uma mensagem para um objeto de outra classe.

  - a) associação
  - b) agregação
  - c) colaboração
  - d) composição

- 7.2 Que forma de diagrama de interações ocorrem? Que forma de colaborações ocorrem?
- 7.3 Crie um diagrama de seqüências que modela as interações entre os objetos no sistema ATM que ocorrem quando um com sucesso e explique a seqüência de mensagens modeladas pelo diagrama.

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 7.1 C.
- 7.2 Os diagramas de comunicação modelizam interações. Os diagramas de seqüências modelizam colaborações ocorrem.
- 7.3 A Figura 7.31 apresenta um diagrama de seqüências que modela as interações entre objetos do sistema ATM que ocorrem quando um depósito com sucesso. A Figura 7.31 apresenta um envio de uma mensagem para pedir ao usuário que insira o valor do depósito. Em seguida, a mensagem é enviada para receber a entrada do usuário. Depois, o usuário inseriu o envelope de depósito em seguida. Em seguida, a mensagem é enviada para confirmar que o envelope de depósito foi recebido pela ATM. Por fim, aumenta o saldo da conta (mas não o ativo) enviando uma mensagem para o banco de dados. O banco de dados responde enviando a mesma mensagem para o usuário.

### 7.13 Síntese

Este capítulo iniciou nossa introdução às estruturas de dados, explorando arrays e arrays e recuperar dados de listas e tabelas de valores. Os exemplos do capítulo demonstraram como declarar um array, inicializar um array e elementos individuais de um array. Também ilustramos como passar arrays a funções e como utilizar o qualificador o princípio do menor privilégio. Os exemplos do capítulo também apresentaram técnicas de pesquisa de classificação básica aprendeu a declarar e manipular arrays multidimensionais. Por fim, demonstramos o uso da Standard Library, que fornece uma alternativa mais robusta aos arrays.

Continuamos nossa cobertura de estruturas de dados no Capítulo 14, “Templates”, no qual construímos um template de pilha, e no Capítulo 21, “Estruturas de dados”, que introduz estruturas de dados dinâmicas, como listas, filas, pilhas e árvores, dem crescer e encolher à medida que os programas executam. O Capítulo 23, “Standard Template Library (STL)”, introduz vá-

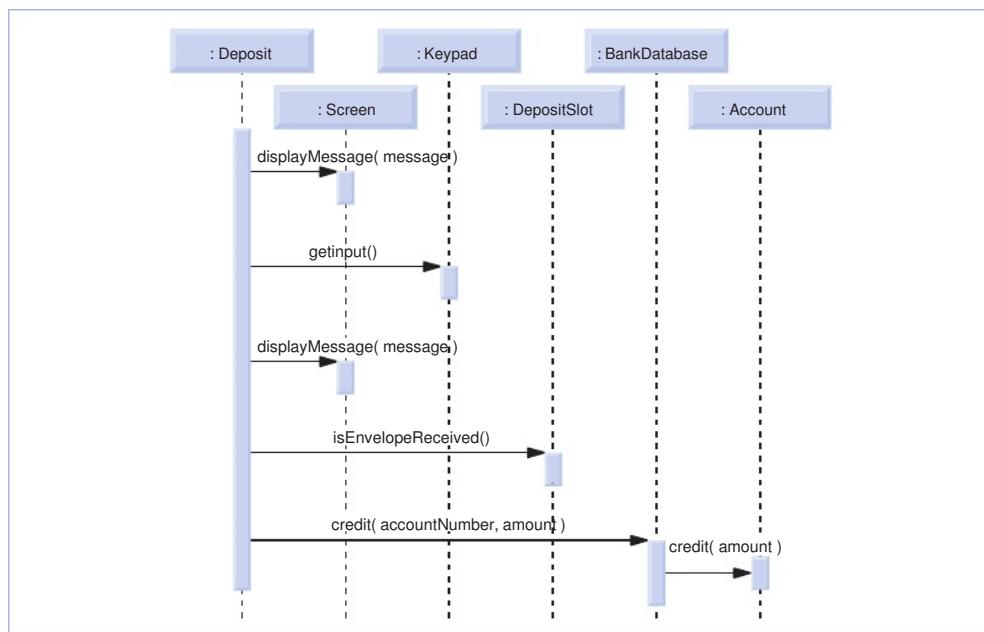


Figura 7.31 Diagrama de seqüências que modela o deposit em execução.

estruturas de dados predefinidas da C++ Standard Library, as quais os programadores podem utilizar em vez de construir suas | O Capítulo 23 apresenta a funcionalidade completa de ~~de tempo de vida~~ outras classes de estrutura de dados, incluindo ~~edique~~ que são estruturas de dados do tipo array que podem crescer e encolher em resposta à mudança dos requisitos de armazenamento de um programa.

Agora introduzimos os conceitos básicos de classes, objetos, instruções de controle, funções e arrays. No Capítulo 8, apresentamos os recursos mais poderosos do C++ — o ponteiro. Os ponteiros monitoram onde dados e funções são armazenados na memória, que nos permite manipular esses itens de maneiras interessantes. Depois de introduzir conceitos básicos de ponteiro, examinaremos detalhe o íntimo relacionamento entre arrays, ponteiros e strings.

### Resumo

- As estruturas de dados são coleções de itens de dados relacionados. Arrays são estruturas de dados consistindo em itens de dados do tipo relacionados. Os arrays são entidades ‘estáticas’ no sentido de que permanecem com o mesmo tamanho ao longo de toda a execução do programa. (Naturalmente, elas podem ser de uma classe de armazenamento automático e daí serem criadas e destruídas toda vez que o programa entra e sai dos blocos em que elas são definidas.)
- Um array é um grupo consecutivo de posições da memória que compartilham o mesmo tipo.
- Para referir-se a uma localização ou elemento particular em um array, especificamos o nome do array e o número de posição do elemento particular no array.
- Um programa referencia qualquer um dos elementos de um array dando o nome do array seguido pelo número de posição do elemento entre colchetes. O número de posição mais formalmente é chamado subscrito ou índice. (Esse número especifica o número de elementos a partir do início do array.)
- O primeiro elemento em cada array tem subscrito zero e às vezes é chamado de zero-ésimo elemento.
- Um subscrito deve ser um inteiro ou uma expressão do tipo inteiro (que utiliza qualquer tipo inteiro).
- É importante notar a diferença entre o ‘sétimo elemento do array’ e o ‘elemento 7 do array’. Subscritos de array iniciam em 0, portanto o ‘sétimo elemento do array’ tem um subscrito de 6, enquanto o ‘elemento 7 do array’ tem um subscrito de 7 e na realidade é o oitavo elemento do array. Essa distinção costuma ser uma ~~fonte de erros~~.
- Os colchetes utilizados para incluir o subscrito de um array são na realidade um operador em C++. Os colchetes têm o mesmo nível de precedência que os parênteses.
- Os arrays ocupam espaço na memória. O programador especifica o tipo de cada elemento e o número de elementos requeridos por um comando `new`:

```
tipo nomeDoArray[tamanhoDoArray];
```

e o compilador reserva a quantidade apropriada de memória.

- Arrays podem ser declarados para conter qualquer tipo de dados. ~~Por exemplo, se utilizar array para armazenar uma string de caracteres.~~
- Os elementos de um array podem ser inicializados na declaração do array colocando depois do nome de array um sinal de igual e uma inicializadora — uma lista separada por vírgulas (incluída entre chaves) de constantes inicializadoras. Ao inicializar um array com uma lista inicializadora, se houver menos inicializadores que elementos no array, os elementos restantes são inicializados como zero.
- Se o tamanho do array for omitido de uma declaração com uma lista inicializadora, o compilador determina o número de elementos no array contando o número de elementos na lista inicializadora.
- Se o tamanho do array e uma lista inicializadora forem especificados em uma declaração de array, o número de inicializadores deve ser menor ou igual ao tamanho do array. Fornecer mais inicializadores em uma lista inicializadora de array do que o número de elementos existe no array é um erro de compilação.
- Constantes devem ser inicializadas com uma expressão constante ao serem declaradas e não podem ser modificadas depois. Constantes podem ser colocadas em qualquer lugar em que uma expressão constante é esperada.
- ~~Programas que excedem a capacidade de armazenamento de um array podem gerar resultados imprevisíveis. Portanto, é importante garantir que as referências de array permaneçam dentro dos limites do array.~~
- Um array de caracteres pode ser inicializado utilizando um literal string. O tamanho de um array de caracteres é determinado pelo compilador com base no comprimento da string. Um caractere especial de terminação de string chamado caractere nulo (representado pela constante de caractere `\0`).
- Todas as strings representadas por arrays de caracteres acabam com o caractere nulo. Um array de caracteres representando uma string deve ser declarado com um tamanho grande o suficiente para armazenar o número de caracteres na string e o caractere nulo de terminação.
- Os arrays de caracteres também podem ser inicializados com constantes de caractere individuais em uma lista inicializadora.

- Caracteres individuais em uma string podem ser acessados diretamente com a notação de subscrito de array.
- Uma string pode ser inserida diretamente em um array de caracteres ~~a partir do teclado utilizando~~
- Um array de caracteres representando uma string terminada por caractere nulo ~~pode~~ ser enviado para a saída com
- Uma variável local em uma definição de função existe até o fim do programa, mas é visível somente no corpo da função.
- Um programa inicializa arrays declarando suas declarações ~~são encontradas pela pesquisa não. Se~~ zado explicitamente pelo programador, cada elemento desse array é inicializado como zero pelo compilador quando o array é criado.
- Para passar um argumento de array a uma função, especifique o nome do array sem colchetes. Para passar um elemento de um array para uma função, utilize o nome subscrito do elemento do array como um argumento na chamada de função.
- Os arrays são passados a funções por referência — as funções chamadas podem modificar o valor dos elementos nos arrays ~~sr~~ maiores. O valor do nome do array é o endereço na memória do computador do primeiro elemento do array. Como o endereço inicial do é passado, a função chamada sabe precisamente onde o array está armazenado na memória.
- Elementos individuais de um array são passados por valor exatamente como variáveis simples o são. Esses fragmentos de dados simplesmente ~~valores ou quantidades ressaltares~~.
- Para receber um argumento de array, a lista de parâmetros da função deve especificar que a função espera receber um array. O tamanho não é requerido entre os colchetes do array.
- O C++ fornece o qualificador ~~de tipo~~ pode ser utilizado para evitar modificação de valores do array no chamador por meio de código em uma função chamada. Quando um parâmetro de array é ~~processado pelo qualificador~~ tornam-se constantes no corpo da função e qualquer tentativa de modificar um elemento do array no corpo da função resulta em um erro de compilação.
- A pesquisa linear compara cada elemento de um array com uma chave de pesquisa. Como o array não está em nenhuma ordem particular, apenas provável que o valor localizado esteja no primeiro elemento. Em média, portanto, um programa deve comparar a chave de pesc com metade dos elementos do array. Para determinar que um valor não está no array, o programa deve comparar a chave de pesquisa com o elemento no array. O método de pesquisa linear funciona bem para arrays pequenos e é aceitável para arrays não classificados.
- Um array pode ser classificado utilizando a classificação por inserção. A primeira iteração desse algoritmo pega o segundo elemento e, se menor que o primeiro, troca-o por este (~~isso é feito para garantir que o elemento na frente do primeiro elemento~~). A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos, de modo que todos os três elementos na ordem ~~na~~ ~~mais~~ ~~uma~~ iteração desse algoritmo ~~o array~~ ~~o array~~ ~~sr~~ ~~o array~~ estarão classificados. Para arrays pequenos, a classificação por inserção é aceitável, mas para arrays maiores é ineficiente comparada com outros algoritmos de classificação mais sofisticados.
- Os arrays multidimensionais com duas dimensões costumam ser utilizados para representar tabelas de valores consistindo em inform organizadas em linhas e colunas.
- Os arrays que requerem dois subscritos para identificar um elemento particular são chamados ~~sr~~ ~~array~~ bidimensionais. Um array com n colunas é chamado ~~de~~ ~~array~~.
- O template `vector` da C++ Standard Library representa uma alternativa mais robusta aos arrays por apresentar muitas capacidades que não são oferecidas pelos arrays baseados em ponteiro no estilo do C.
- Por padrão, todos os elementos ~~de um array~~ ~~o array~~ configurados como 0.
- `Unvector` pode ser definido para armazenar qualquer tipo de dados utilizando uma declaração na forma:

  - `vector< tipo > nome( tamanho );`
  - A função-membro `template` da classe `vector` retorna o número de elementos que é invocada.
  - O valor de um elemento `at` pode ser acessado ou modificado utilizando colchetes (`[ ]`).
  - Os objetos do template de classe `vector` podem ser comparados diretamente com os operadores de igualdade (`==`).
  - O operador de atribuição (`=`) também pode ser utilizado com objetos.
  - Univalueável modificável é uma expressão que identifica um objeto na memória (como um elemento em um vetor), mas não pode ser utilizada para modificar esse objeto. Unidifável também identifica um objeto na memória, mas pode ser utilizado para modificar o objeto.
  - O template de classe `vector` ~~o array~~ ~~sr~~ ~~o array~~ ~~sr~~ ~~o array~~ ~~sr~~ ~~o array~~ ~~sr~~ ~~o array~~ exceção' se seu argumento for um subscrito inválido. Por padrão, isso faz com que um programa C++ termine.

## Terminologia

|                          |                                 |                                  |
|--------------------------|---------------------------------|----------------------------------|
| array 2-D                | array multidimensional          | classificar um array             |
| <code>a[ i ]</code>      | array unidimensional            | chetes                           |
| <code>a[ i ][ j ]</code> | <code>at</code> , função-membro | coluna de um array bidimensional |
| array                    | <code>char( nulo )</code>       | const, qualificador de tipo      |
| array bidimensional      | chave de pesquisa               | constante identificada           |
| <code>arraypon</code>    | classificação por inserção      | declarar um array                |

|                                 |                                                |                                                |
|---------------------------------|------------------------------------------------|------------------------------------------------|
| elemento de um array            | lvalor modifíável                              | string representada por um array de caracteres |
| erro off-by-one                 | lvalor não modifíável                          | subscrito                                      |
| escalabilidade                  | nome de um array                               | subscrito de coluna                            |
| escalar                         | número de posição                              | subscrito de linha                             |
| estrutura de dados              | número mágico                                  | tabela de valores                              |
| formato tabular                 | passando arrays a funções                      | 'ultrapassar' os limites de um array           |
| índice                          | passar por referência                          | valor de um elemento                           |
| índice zero                     | pesquisa linear de um array                    | valor-chave                                    |
| initializador                   | pesquisar um array                             | variáveis de leitura                           |
| inicializar um array            | quantidade escalar                             | variável constante                             |
| linha de um array bidimensional | size, função-membro                            | vector (template da C++ Standard Library)      |
| lista inicializadora            | static, membro de dados                        | verificação de limites                         |
| lista inicializadora de array   | string representada por um array de caracteres | ultimo elemento                                |

### Exercícios de revisão

- 7.1 Complete cada uma das seguintes sentenças:
- Listas e tabelas de valores podem ser armazenadas em \_\_\_\_\_ ou \_\_\_\_\_.
  - Os elementos de um array são relacionados pelo fato de eles terem o mesmo \_\_\_\_\_ e \_\_\_\_\_.
  - O número utilizado para referenciar um elemento particular de um array é chamado de seu \_\_\_\_\_.
  - Um(a) \_\_\_\_\_ deve ser utilizado(a) para declarar o tamanho de um array, porque torna o programa mais escalonável.
  - O processo de colocar os elementos de um array em ordem é chamado de \_\_\_\_\_ o array.
  - O processo de determinar se um array contém um valor-chave particular é chamado de \_\_\_\_\_ o array.
  - Um array que utiliza dois subscritos é referido como um array \_\_\_\_\_.
- 7.2 Determine se as seguintes sentenças são verdadeiras ou falsas. Se a resposta for falsa, explique por quê.
- Um array pode armazenar muitos tipos de valores diferentes.
  - Um subscrito de array normalmente deve ser do tipo de dados.
  - Se houver menos inicializadores em uma lista inicializadora que o número de elementos no array, os elementos restantes são inicializados com o último valor na lista inicializadora.
  - É um erro se uma lista inicializadora contiver mais inicializadores que o número de elementos no array.
  - Um elemento do array individual que é passado para uma função e modificado nessa função conterá o valor modificado quando a função terminar sua execução.
- 7.3 Escreva uma ou mais instruções que realizam as seguintes tarefas para um array chamado `array`:
- Defina uma variável `constante` inicializada como \_\_\_\_\_.
  - Declare um array `array` de elementos do tipo \_\_\_\_\_ e inicialize os elementos como \_\_\_\_\_.
  - Nomeie o quarto elemento do array.
  - Referencie o elemento 4 do array.
  - Atribua o valor 7 ao elemento 9 do array.
  - Atribua o valor 3 ao sétimo elemento do array.
  - Imprima os elementos do array 6 e 9 com dois dígitos de precisão à direita do ponto de fração decimal e mostre a saída que realmente é exibida na tela.
  - Imprima todos os elementos do array utilizando a instrução de controle `for` com uma variável de controle para o loop. Mostre a saída.
- 7.4 Responda às seguintes perguntas relacionadas a um array chamado `table`:
- Declare o array como um array de inteiros e tendo 3 linhas e 3 colunas. Suponha que a constante definida como 3.
  - Quantos elementos o array contém?
  - Utilize a instrução de controle para adicionar o elemento do array com a soma de seus subscritos. Suponha que as variáveis `int size` e `int sum` declaradas como variáveis de controle. Mostre a saída.
  - Escreva um segmento de programa para imprimir os valores de cada elemento tabular em 3 linhas e 3 colunas. Suponha que o array foi inicializado com a declaração
- ```
int table[ arraySize ][ arraySize ] = { { 1, 8}, { 2, 4, 6}, { 5 } };
```
- e que as variáveis `int size` declaradas como variáveis de controle. Mostre a saída.
- 7.5 Localize o erro em cada um dos seguintes segmentos de programa e corrija-o:
- #include <iostream>;

```

b) arraySize = 10; // arraySize foi declarado const
c) Suponha que b[ 10 ] = { 0 };
   for ( int i = 0; i <= 10; i++ )
     b[ i ] = 1;
d) Suponha que a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
   a[ 1, 1 ] = 5;

```

Respostas dos exercícios de revisão

- 7.1 a) array~~s~~s. b) nome, tipo. c) subscrito (ou índice). d) variável constante. e) classificar. f) pesquisar. g) bidimensional.
- 7.2 a) Falsa. Um array pode armazenar apenas valores do mesmo tipo.
 b) Falsa. Um subscrito de array deve ser um inteiro ou uma expressão do tipo inteiro.
 c) Falsa. Os elementos restantes são inicializados como zero.
 d) Verdadeira.
 e) Falsa. Elementos individuais de um array são passados por valor. Se o array inteiro é passado para uma função, então quaisquer modificações serão refletidas no scrcinal.
- 7.3 a) `const int arraySize = 10;`
 b) `double fractions[arraySize] = { 0.0 };`
 c) `fractions[3]`
 d) `fractions[4]`
 e) `fractions[9] = 1.667;`
 f) `fractions[6] = 3.333`
 g) `cout << fixed << setprecision(2);
 cout << fractions[6] << ' ' << fractions[9] << endl;`
 Saída:
 33 1.67.
- h) `for (int i = 0; i < arraySize; i++)
 cout << "fractions[" << i << "] = " << fractions[i] << endl;`
 Saída:
 fractions[0] = 0.0
 fractions[1] = 0.0
 fractions[2] = 0.0
 fractions[3] = 0.0
 fractions[4] = 0.0
 fractions[5] = 0.0
 fractions[6] = 3.333
 fractions[7] = 0.0
 fractions[8] = 0.0
 fractions[9] = 1.667
- 7.4 a) `int table[arraySize][arraySize];`
 b) Nove.
 c) `for (i = 0; i < arraySize; i++)

 for (j = 0; j < arraySize; j++)
 table[i][j] = i + j;`
 d) `cout << "[0] [1] [2]" << endl;

 for (int i = 0; i < arraySize; i++) {
 cout << '[' << i << "] " ;

 for (int j = 0; j < arraySize; j++)
 cout << setw(3) << table[i][j] << " " ;

 cout << endl;
 }`
 Saída:
 [0] [1] [2]
 [0] 1 8 0

```
[1] 2 4 6
[2] 5 0 0
```

- 7.5 a) Erro: O ponto-e-vírgula no final da diretiva de processador

Correção: Elimine o ponto-e-vírgula.

- b) Erro: Atribuir um valor a uma variável constante utilizando uma instrução de atribuição.

Correção: Inicialize a variável constante em sua declaração.

- c) Erro: Referenciando um elemento de array além dos limites do array (

Correção: Altere o valor final da variável de controle para

- d) Erro: Atribuição de subscrito de array feita incorretamente.

Correção: Altere a instrução para 5;

Exercícios

- 7.6 Preencha as lacunas em cada uma das seguintes sentenças:

- a) Os nomes dos quatro elementos de um array são _____, _____, _____ e _____.
- b) Nomear um array, declarar seu tipo e especificar o número de dimensões no array é chamado de _____ o array.
- c) Por convenção, o primeiro subscrito em um array bidimensional identifica um elemento _____ um elemento _____.
- d) Um array tem _____ linhas, _____ colunas e _____ elementos.
- e) O nome do elemento na linha 3 e na coluna 5 do array

- 7.7 Determine se cada uma das seguintes afirmações é verdadeira ou falsa e explique por quê.

- a) Para referir-se a uma localização particular ou elemento dentro de um array, especificamos o nome do array e o valor do elemento particular.
- b) Uma declaração de array reserva espaço para o array.
- c) Para indicar que 100 localizações devem ser reservadas para o array, deve-se fazer a declaração
`p[100];`
- d) Uma instrução deve ser utilizada para inicializar os elementos de um array de 15 elementos como zero.
- e) Instruções aninhadas devem ser utilizadas para somar os elementos de um array bidimensional.

- 7.8 Escreva instruções C++ para realizar cada uma das seguintes tarefas:

- a) Exiba o valor do elemento 6 do array de caracteres
- b) Insira um valor no elemento 4 do array unidimensional de ponto flutuante
- c) Inicialize cada um dos 5 elementos do array de inteiros unidimensional
- d) Some e imprima os elementos do array de ponto flutuante.
- e) Copie o array a primeira parte do array. Considere a [11], b[34].
- f) Determine e imprima os maiores e menores valores contidos nesse array de ponto flutuante

- 7.9 Considere um array de inteiros 2 por 3

- a) Escreva uma declaração para
- b) Quantas linhas tem
- c) Quantas colunas tem
- d) Quantos elementos tem
- e) Escreva os nomes de todos os elementos na linha 1 de
- f) Escreva os nomes de todos os elementos na coluna 2 de
- g) Escreva uma única instrução que configura o elemento de coluna 2 como zero.
- h) Escreva uma série de instruções que inicializam todos os elementos de um loop.
- i) Escreva uma instrução que inicializa cada elemento de
- j) Escreva uma instrução que insere os valores para os elementos de
- k) Escreva uma série de instruções que determinam e imprimem o menor valor no array
- l) Escreva uma instrução que exibe os elementos na linha 0 de
- m) Escreva uma instrução que soma os elementos na coluna 3 de
- n) Escreva uma série de instruções que manipule a matriz. Liste os subscritos de coluna como títulos ao longo do topo da tabela e liste os subscritos de linha à esquerda de cada linha.

- 7.10 Utilize um array unidimensional para resolver o seguinte problema. Uma empresa paga seu pessoal de vendas por comissão. Os vendedores recebem \$ 200 por semana mais 9% de suas vendas brutas por semana. Por exemplo, um vendedor que vende \$ 5.000 brutos em uma semana recebe \$ 200 mais 9% de \$ 5.000 ou um total de \$ 650. Escreva um programa (utilizando um array de contadores) que determine quanto o pessoal de vendas ganhou em cada um dos seguintes intervalos (suponha que o salário de cada vendedor foi truncado para quantidade do tipo inteiro) :

- a) \$ 200–\$ 299
- b) \$ 300–\$ 399
- c) \$ 400–\$ 499
- d) \$ 500–\$ 599
- e) \$ 600–\$ 699
- f) \$ 700–\$ 799
- g) \$ 800–\$ 899
- h) \$ 900–\$ 999
- i) \$ 1.000 e acima

- 7.11 (Classificação por borbulhamento ([Bubble sort](#)) de classificação por borbulhamento) valores menores gradualmente sobem para a parte superior do array como bolhas de ar subindo na água, enquanto as bolhas maiores afundam. A classificação por borbulhamento faz várias passagens pelo array. Em cada passagem, sucessivos pares de elementos são comparados. Se um par de valores não está em ordem, é trocado. Quando todos os pares de elementos estiverem em ordem, a classificação por borbulhamento termina.
- 7.12 A classificação por borbulhamento descrita no Exercício 7.11 é ineficiente para arrays grandes. Faça as seguintes modificações simples para aprimorar o desempenho da classificação por borbulhamento:
- Depois da primeira passagem, garante-se que o maior número está no elemento de número mais alto do array; após a segunda passagem, os dois números mais altos estão ‘no lugar’; e assim por diante. Em vez de fazer nove comparações em cada passagem, modifique a classificação por borbulhamento para fazer oito comparações na segunda passagem, sete na terceira passagem e assim por diante.
 - Os dados no array já podem estar na ordem adequada ou ordem quase adequada, então por que fazer nove passagens se menos são suficientes? Modifique a classificação para verificar no fim de cada passagem se alguma troca foi feita. Se nenhuma troca tiver sido feita, então os dados já devem estar na ordem apropriada; portanto, o programa deve terminar. Se trocas foram feitas, então pelo menos uma passagem é necessária.
- 7.13 Escreva instruções simples que realizam as seguintes operações de um array unidimensional:
- Inicialize os 10 elementos do array de inteiros.
 - Adicione 1 a cada um dos 15 elementos do array de inteiros.
 - Leia 12 valores para a array monthlyTemperatures a partir do teclado.
 - Imprima os 5 valores do array monthlyTemperatures no formato de coluna.
- 7.14 Localize o(s) erro(s) em cada uma das seguintes instruções:
- Suponha que str[5];
cin >> str; // usuário digita “hello”
 - Suponha que[3];
cout << a[1] << “ “ << a[2] << “ “ << a[3] << endl;
 - double f[3] = { 1.1, 10.01, 100.001, 1000.0001 };
 - Suponha que d[2][10];
d[1, 9] = 2.345;
- 7.15 Utilize um array unidimensional para resolver o seguinte problema. Leia 20 números, cada um dos quais está entre 10 e 100, inclusivamente. À medida que cada número é lido, valide-o e armazene-o no array somente se ele não for uma duplicata de um número já lido. Depois de ler todos os valores, exiba somente os valores únicos que o usuário inseriu. Previna-se para o ‘pior caso’ em que todos os 20 números são diferentes. Utilize o menor array possível para resolver esse problema.
- 7.16 Rotule os elementos de um array unidimensional de 3 por 5 de acordo com a ordem em que eles são configurados como zero pelo seguinte segmento de programa:
[See 7.16 for details.](#)
- ```
for (row = 0; row < 3; row++)
 for (column = 0; column < 5; column++)
 sales[row][column] = 0;
```
- 7.17 Escreva um programa que simula a rolagem de dois dados. [And for more information about rolling two dice, see 7.17.](#) O programa deve gerar um dado e deve utilizar rand novamente para rolar o segundo dado. A soma dos dois valores deve ser contada e pode aparecer um valor inteiro de 2 a 12, portanto a soma dos dois valores variará de 2 a 12, com 7 sendo a soma mais frequente, e 2 e 12 sendo as somas mais raras. A Figura 7.32 mostra as 36 possíveis combinações de dois dados. Seu programa deve rolar os dois dados 36.000 vezes. Use um array unidimensional para contar o número de vezes que cada possível soma aparece. Imprima os resultados em um formato tabular. Além disso, determine se os totais são razoáveis (isto é, há seis maneiras de rolar um 7, então aproximadamente um sexto de todas as rolagens deve ser 7.).]

|   |   |   |   |    |    |    |
|---|---|---|---|----|----|----|
|   | 1 | 2 | 3 | 4  | 5  | 6  |
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Figura 7.32 Os 36 possíveis resultados de rolar dois dados.

## 7.18 O que o seguinte programa faz?

```

1 // Ex. 7.18: Ex07_18.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int whatIsThis(int [], int); // protótipo de função
8
9 int main()
10 {
11 const int arraySize = 10;
12 int a[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14 int result = whatIsThis(a, arraySize);
15
16 cout << "Result is " << result << endl;
17 return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?
21 int whatIsThis(int b[], int size)
22 {
23 if (size == 1) // caso básico
24 return b[0];
25 else // passo recursivo
26 return b[size - 1] + whatIsThis(b, size - 1);
27 } // fim da função whatIsThis

```

- 7.19 Modifique o programa da Figura 6.11 para jogar 1000 partidas. O programa deve monitorar as estatísticas e responder às seguintes perguntas:
- Quantas partidas são ganhas na primeira rolagem dos dados, na segunda, ..., na vigésima e depois da vigésima rolagem dos dados?
  - Quantas partidas são perdidas na primeira rolagem dos dados, na segunda, ..., na vigésima e depois da vigésima rolagem dos dados?
  - Quais são as chances de ganhar no Jogo de dados? Descubra qual é o percentual dos jogos mais comuns de cassino.
  - O que você supõe que isso significa?
  - Qual é o comprimento médio de uma partida de dados?
  - As chances de ganhar aumentam com o comprimento do jogo?
- 7.20 (Sistema de reservas de passageiros) Uma companhia aérea acabou de comprar um computador para seu novo sistema automatizado de reservas. Você foi solicitado a programar o novo sistema. Você escreverá um programa para atribuir assentos em cada vôo da companhia aérea (capacidade: 10 assentos). Seu programa deve exibir o seguinte menu:
- Please alternative "First Class"   Please type 2 for "Economy." Se a pessoa digitar 1, o programa deve atribuir um assento na primeira classe (assento 10) e se a pessoa digitar

deve atribuir um assento na classe econômica (assentos 6-10). Seu programa deve imprimir um bilhete de embarque indicando o número do assento da pessoa e se ela está na primeira classe ou na classe econômica do avião.

Utilize um array unidimensional para representar o gráfico de assentos do avião. Inicialize todos os elementos do array como 0 para indicar que todos os assentos estão vazios. À medida que cada assento é atribuído, configure os elementos correspondentes do array como 1 para indicar que o assento não está mais disponível.

Naturalmente, seu programa nunca deve atribuir um assento que já foi atribuído. Quando a primeira classe estiver lotada, seu programa deve perguntar à pessoa se ela aceita ficar na classe econômica (e vice-versa). Se ela aceitar, faça a atribuição apropriada de assentos contrário, imprima a mensagem "leaves in 3 hours."

7.21 O que o seguinte programa faz?

```

1 // Ex. 7.21: Ex07_21.cpp
2 // O que esse programa faz?

3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void someFunction(int [], int , int); // protótipo de função
8
9 int main()
10 {
11 const int arraySize = 10;
12 int a[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14 cout << "The values in the array are:" << endl;
15 someFunction(a, 0, arraySize);
16 cout << endl;
17 return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?

21 void someFunction(int b[], int current, int size)
22 {
23 if (current < size)
24 {
25 someFunction(b, current + 1, size);
26 cout << b[current] << " ";
27 } // fim do if
28 } // fim da função someFunction

```

7.22 Utilize um array bidimensional para resolver o seguinte problema. Uma empresa tem quatro equipes de vendas (1 a 4) que vendem cinco produtos diferentes (1 a 5). Uma vez por dia, cada vendedor passa uma nota de cada tipo de produto vendido. Cada nota contém o seguinte:

- O número do vendedor
- O número do produto
- O valor total em reais desse produto vendido nesse dia

Portanto, cada vendedor passa entre 0 e 5 notas de vendas por dia. Suponha que as informações a partir de todas as notas durante o mês estão disponíveis. Escreva um programa que lerá todas essas informações para as vendas do último mês e resumirá as vendas por vendedor e produto. Todos os totais devem ser armazenados. Depois de processar todas as informações

durante o último mês, imprima os resultados em formato tabular com cada uma das colunas representando um vendedor específico. Cada uma das linhas representa um produto específico. Cruze cada linha de total para obter as vendas totais de cada produto durante o último mês; cruze cada coluna de total para obter as vendas totais por vendedor durante o último mês. Sua impressão tabular deve incluir os totais cruzados à direita das linhas totalizadas e na parte inferior das colunas totalizadas.

7.23 (Gráficos de tartarugas) A linguagem Logo, que é popular entre crianças do ensino fundamental, famoso o conceito de tartarugas. Imagine uma tartaruga mecânica que caminha em um ambiente sob o controle de um programa C++. A tartaruga segura uma caneta em uma de duas posições, para cima ou para baixo. Enquanto a caneta está para baixo, a tartaruga desenha formas à medida que se move; enquanto a caneta está para cima, a tartaruga move-se quase livremente sem escrever nada. Nesse problema você só precisa se preocupar com a operação da tartaruga e criar gráficos.

Utilize um array de 20 por 20 que é inicializado como zeros. Leia comandos a partir de um array que contenha esses comandos. Monitore a posição atual da tartaruga todas as vezes e se a caneta está atualmente para cima ou para baixo. Suponha que a tartaruga sempre inicia na posição (0, 0) do chão com sua caneta para cima. O conjunto de comandos da tartaruga que seu programa deve processar é mostrado na Figura 7.33.

Suponha que a tartaruga esteja em algum lugar próximo ao centro do chão. O seguinte 'programa' desenharia e imprimaria um quadrado de 12 por 12 e terminaria com a caneta na posição para cima:

```

2
5,12
3
5,12
3
5,12
8,12
1
6
9

```

À medida que a tartaruga se move com a caneta para baixo, configure os elementos saídos do array comandos (imprimir) é dado, onde quer que haja exiba um asterisco ou algum caractere diferente que você escolher. Onde quer que haja um zero, exiba uma lacuna. Escreva um programa para implementar as capacidades dos gráficos de tartaruga discutidas aqui. Escreva vários programas de gráfico de tartaruga para desenhar formas interessantes. Adicione outros comandos para aumentar as capacidades da linguagem de gráfico de tartaruga.

- 7.24 (O Passeio do Cavalo) quebra-cabeças mais interessantes dos entusiastas do xadrez é o problema do Passeio do Cavalo. Esta é a pergunta: a peça de xadrez chamada cavalo pode mover-se em um tabuleiro vazio e tocar cada um dos 64 quadrados uma vez e uma vez? Estudamos esse problema intrigante profundamente nesse exercício.

O cavalo move-se em um caminho em forma de L (duas posições em uma direção e então uma em uma direção perpendicular). Portanto, a partir de um quadrado no meio de um tabuleiro vazio, o cavalo pode fazer oito movimentos diferentes (numerados de 0 a 7) como mostrado na Figura 7.34.

- a) Desenhe um tabuleiro de 8 por 8 em uma folha de papel e experimente fazer o Passeio do Cavalo à mão. Coloque um asterisco no quadrado que o cavalo toca quando ele faz o seu passeio. A medida que o passeio é feito, note se o cavalo foi próximo de sua estimativa?

- b) Agora vamos desenvolver um programa que moverá o cavalo por um tabuleiro. O tabuleiro é representado por um array bidimensional de 8 por 8 chamado tabuleiro. Cada um dos quadrados é inicializado como zero. Descrevemos cada um dos oito possíveis movimentos em termos de seus componentes verticais e horizontais. Por exemplo, um movimento do tipo 0 como mostrado na Figura 7.34 consiste em mover dois quadrados horizontalmente para direita e um quadrado verticalmente para cima. O movimento 2 consiste em mover um quadrado horizontalmente para a esquerda e dois quadrados verticalmente para cima. Movimentos horizontais para a esquerda e movimentos verticais para cima são indicados com números negativos. Os oito movimentos podem ser descritos por dois arrays unidimensionais horizontal e vertical, como segue:

```

horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1

```

| Comando | Significado                                                  |
|---------|--------------------------------------------------------------|
| 1       | Canetaparâcima                                               |
| 2       | Canetaparâbaixo                                              |
| 3       | Viraparâdireita                                              |
| 4       | Viraparâesquerda                                             |
| 5,10    | Mova para a frente 10 espaços (ou um número diferente de 10) |
| 6       | Imprimaarray20por20                                          |
| 9       | Fimdosdados(sentinela)                                       |

Figura 7.33 Comandos dos gráficos de tartaruga.

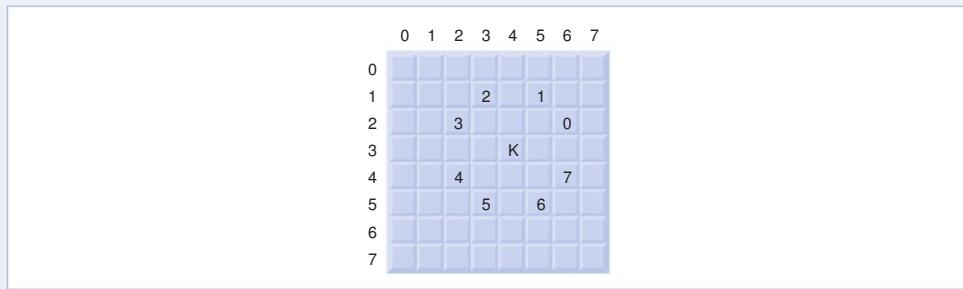


Figura 7.34 Os oito possíveis movimento do cavalo.

Faça com que as variáveis `currentRow` e `currentColumn` indiquem a linha e a coluna da posição atual do cavalo. Para fazer um movimento do tipo `moveNumber`, onde `moveNumber` está entre 0 e 7, seu programa utiliza as instruções

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Mantenha um contador que conta os movimentos. Registre a última contagem em cada quadrado para o qual o cavalo se move. Lembre-se de testar cada potencial movimento para ver se o cavalo já visitou esse quadrado e, naturalmente, teste cada potencial movimento para certificar-se de que o cavalo não cai fora do tabuleiro. Agora escreva um programa para mover o cavalo pelo tabuleiro. Execute o programa. Quantos movimentos o cavalo fez?

- c) Depois de tentar escrever e executar um programa para o Passeio do Cavalo, provavelmente alguns insights valiosos foram desse dos. Utilizaremos esses insights para desenvolver uma estratégia para mover o cavalo. A heurística não garante sucesso, mas uma heurística cuidadosamente desenvolvida aprimora significativamente a chance de sucesso. Você pode ter observado que quadrados externos são mais incômodos que os quadrados próximos do centro do tabuleiro. De fato, os quadrados mais problemáticos ou inacessíveis, são os quatro cantos.

A intuição pode sugerir que você deve tentar mover o cavalo para os quadrados mais problemáticos primeiro e deixar abertos aqueles que são fáceis de alcançar de modo que, quando o tabuleiro ficar congestionado próximo do fim do passeio, haja maior chance de sucesso.

Podemos desenvolver uma 'heurística de acessibilidade' classificando cada quadrado de acordo com seu grau de acessibilidade e sempre mover o cavaleiro para o quadrado (dentro do movimento em forma de L do cavalo, naturalmente) que seja mais inacessível. Rotulamos um array bidimensional `accessibility`, com números indicando a partir de quantos quadrados cada quadrado particular é acessível. Em um tabuleiro vazio, os 16 quadrados centrais ~~terão~~ têm a maior classificação, e os outros quadrados têm números de acesso ~~maior~~ menor.

```

2 3 4 4 4 4 3 2
3 4 6 6 6 4 3
4 6 8 8 8 6 4
4 6 8 8 8 6 4
4 6 8 8 8 6 4
4 6 8 8 8 6 4
3 4 6 6 6 4 3
2 3 4 4 4 3 2

```

Agora escreva uma versão do programa do Passeio do Cavalo utilizando a heurística de acessibilidade. Em qualquer dado momento o cavaleiro deve mover-se para o quadrado com o número mais baixo de acessibilidade. Em caso de um impasse, o cavalo pode mover-se para qualquer quadrado já visitado. Portanto, o passeio pode iniciar em ~~qualquer~~ ~~uma~~ ~~medida~~ das quatro cantos. [O cavalo se move pelo tabuleiro de xadrez, seu programa deve reduzir os números de acessibilidade porque cada vez mais quadrados tornam ocupados. Dessa maneira, em qualquer dado tempo durante o passeio, o número de acessibilidade de cada quadrado disporá permanecer igual ao número de quadrados a partir dos quais esse quadrado pode ser alcançado.] Execute essa versão de seu programa. Você obteve um passeio completo? Agora modifique o programa para executar 64 passeios, um iniciando a partir de cada quadrado do tabuleiro de xadrez. Quantos passeios completos você obteve?

- d) Escreva uma versão do programa Passeio do Cavalo que, diante de um impasse entre dois ou mais quadrados, decide qual quadrado escolher olhando para a frente aqueles quadrados alcançáveis a partir dos quadrados geradores do impasse. Seu programa deve mover-se para o quadrado por meio do qual seu próximo movimento chegaria ao quadrado com o número de acessibilidade mais baixo.

7.25 (Passeio do Cavalo: abordagens de força bruta) 7.24 desenvolvemos uma solução para o problema do Passeio do Cavalo. A abordagem utilizada, chamada de ‘acessibilidade heurística’, gera muitas soluções e executa eficientemente.

À medida que os computadores continuam aumentando em potência, seremos capazes de resolver cada vez mais problemas com a capacidade do computador e algoritmos relativamente simples. Essa é a abordagem de ‘força bruta’ para resolução de problemas.

- a) Utilize geração de números aleatórios para permitir que o cavalo ande pelo tabuleiro de xadrez (em seus movimentos válidos em forma de L, naturalmente) de maneira aleatória. Seu programa deve executar um passeio e imprimir o tabuleiro de xadrez final. Até onde o cavalo chegou?  
b) Muito provavelmente, o programa precedente produziu um passeio relativamente curto. Agora modifique seu programa para tentar 1.000 passeios. Utilize um array unidimensional para monitorar o número de passeios de cada comprimento. Quando seu programa tiver de tentar os 1.000 passeios, ele deve imprimir essas informações em formato tabular organizado. Qual foi o melhor resultado?  
c) Muito provavelmente, o programa precedente deu-lhe alguns passeios ‘respeitáveis’, mas nenhum passeio completo. Agora ‘soltar todas as amarras’ e simplesmente deixe seu programa executar até que seja encontrado um passeio completo. Em seguida, imprima a tabela do número de passeios de cada comprimento e imprima essa tabela quando o primeiro passeio completo for encontrado. Quantos passeios seu programa tentou antes de produzir um passeio completo? Quanto tempo ele levou?

- d) Compare a versão de força bruta do Passeio do Cavalo com a versão de acessibilidade heurística. Qual exigiu um estudo mais cidadoso do problema? Qual algoritmo foi mais difícil de desenvolver? Qual exigiu mais capacidade do computador? Poderíamos ter certeza (antecipadamente) de obter um passeio completo com a abordagem de acessibilidade heurística? Poderíamos ter certeza (com antecedência) de obter um passeio completo com a abordagem de força bruta? Argumente as vantagens e desvantagens de resolver o problema de força bruta em geral.

7.26 (Oito Rainhas) Outro problema difícil para fãs do xadrez é o das Oito Rainhas. Eis o problema: É possível colocar oito rainhas em um tabuleiro de xadrez vazio de modo que nenhuma rainha esteja ‘atacando’ qualquer outra, isto é, sem que duas rainhas estejam na mesma linha, na mesma coluna ou na mesma diagonal? Utilize a consideração desenvolvida no Exercício 7.24 a fim de formular uma heurística para resolver o problema das Oito Rainhas. Execute o seu programa para atribuir um valor para cada quadrado do tabuleiro de xadrez que indica quantos quadrados de um tabuleiro de xadrez vazio ‘são eliminados’ se uma rainha for colocada nesse quadrado; cada um dos cantos seria atribuído o valor 22, como na Figura 7.35.] Uma vez que esses ‘números de eliminação’ são colocados em todos os 64 quadrados, uma heurística apropriada talvez seja: coloque a próxima rainha no quadrado com o menor número de eliminação. Fique com essa estratégia é intuitivamente atraente?

7.27 (Oito Rainhas: abordagens de força bruta) Neste exercício você desenvolverá várias abordagens de força bruta para resolver o problema das Oito Rainhas introduzido no Exercício 7.26.

- a) Resolva o exercício das Oito Rainhas utilizando a técnica de força bruta aleatória desenvolvida no Exercício 7.25.  
b) Utilize uma técnica exaustiva, isto é, tente todas as combinações possíveis de oito rainhas no tabuleiro de xadrez.  
c) Por que você supõe que a abordagem de força bruta exaustiva pode não ser apropriada para resolver o problema do Passeio do Cavalo?  
d) Compare e contraste as abordagens de força bruta aleatória e de força bruta exaustiva em geral.

7.28 (Passeio do Cavalo: Teste do Passeio do Cavalo) Um passeio completo ocorre quando o cavalo move-se tocando cada um dos 64 quadrados do tabuleiro de xadrez uma vez e apenas uma vez. Um passeio fechado ocorre quando o 64

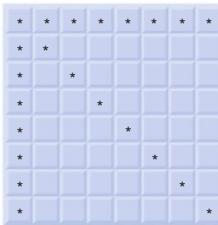


Figura 7.35 Os 22 quadrados foram eliminados posicionando uma rainha no canto esquerdo superior.

cai no quadrado em que o cavalo iniciou o passeio. Modifique o programa do Passeio do Cavalo que você escreveu no Exercício 7.2 fim de testar para um passeio fechado se um passeio completo ocorreu.

- 7.29 (O Crivo de Eratóstenes é um inteiro primo é qualquer inteiro que seja divisível apenas por si mesmo e por 1. O crivo, ou peneira, de Eratóstenes é um método para localizar números primos. Ele opera como segue:
- Crie um array com todos os elementos inicializados como 1 (verdadeiro). Os elementos do array com subscritos primos permanecem como 1. Todos os outros elementos do array acabarão sendo configurados como zero. Você ignorará os elementos 0 e 1 nesse exemplo.
  - Iniciando com o subscrito de array 2 (o subscrito 1 deve ser primo), toda vez que for localizado um elemento do array cujo valor é 1, faça um loop pelo restante do array e configure como zero cada elemento cujo subscrito é um múltiplo do subscrito para o elemento com valor 1. Para o subscrito de array 2, todos os elementos além de 2 no array que são múltiplos de 2 serão configurados como 0 (subscritos 4, 6, 8, 10 etc.); para o subscrito de array 3, todos os elementos além de 3 no array que são múltiplos de 3 serão configurados como zero (subscritos 6, 9, 12, 15 etc.); e assim por diante.

Quando esse processo estiver completo, os elementos do array que ainda estiverem configurados como 1 indicam que o subscrito é um número primo. Esses subscritos então podem ser impressos. Escreva um programa que utiliza um array de 1.000 elementos para determinar e imprimir os números primos entre 2 e 999. Ignore o elemento 0 do array.

- 7.30 (Bucket Sort) A classificação [Bucket Sort](#) inicia com um array unidimensional de inteiros positivos a ser classificado e um array bidimensional de inteiros com linhas indexadas de 0–9 encolhendo dezenas de zeros no array a serem classificados. Cada linha do array bidimensional é chamada de 'passagem'. Sort que aceita um array de inteiros e o tamanho do array como argumentos e funciona da seguinte maneira:
- Coloque cada valor do array unidimensional em `bucket[linha][dígito]` das unidades do valor. Por exemplo, 97 é colocado na linha 7, 3 é colocado na linha 3, e 100 é colocado na linha 0. Isso é chamado de 'passagem de distribuição'.
  - Realize um loop pelo array de bucket linha por linha e copie os valores de volta para o array original. Isso é chamado de 'passagem de coleta'. A nova ordem dos valores precedentes no array unidimensional é 100, 3 e 97.
  - Repita esse processo para a posição de cada dígito subsequente (dezenas, centenas, milhares etc.).

Na segunda passagem, 100 é colocado na linha 0, 3 é colocado na linha 0 (porque 3 não tem dígito de dezenas) e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no array unidimensional é 100, 3 e 97. Na terceira passagem, 100 é colocado na linha 1, 3 é colocado na linha zero, e 97 é colocado na linha zero (depois do 3). Depois da última passagem de coleta, o array `srcin` está agora na ordem classificada.

Observe que o array de buckets bidimensional tem 10 vezes o tamanho do array de inteiros sendo classificado. Essa técnica de classificação fornece melhor desempenho que uma classificação por inserção, mas exige muito mais memória. A classificação por inserção requer espaço para apenas um elemento de dados adicional. Esse é um exemplo de classificação com mais espaço-tempo: A memória que a classificação por inserção, mas seu desempenho é muito menor.

Voltar para o array principal a cada passagem. Outra possibilidade é copiar o array de buckets para o array principal a cada passagem.

### Exercícios com recursão

- 7.31 (Classificação por seleção) A classificação [seleção](#) pesquisa um array procurando o menor elemento. Então, o elemento menor é trocado com o primeiro elemento do array. O processo é repetido para o subarray que inicia com o segundo elemento do array. Cada passagem do array resulta em um elemento sendo colocado em sua localização adequada. O desempenho dessa classificação é com paralelo ao da classificação por inserção — para  $n$  elementos,  $\frac{n(n+1)}{2}$  passagens devem ser feitas e para  $n$  dados no array,

ções devem ser feitas para localizar o menor valor. Quando o subarray sendo processado contiver um elemento, o array será classificado. Escreva a função `recursiveSort` para realizar esse algoritmo.

- 7.32 (Palíndromo) Um palíndromo é uma string que é lida da mesma maneira, quer da esquerda para a direita, quer da direita para a esquerda. Alguns exemplos de palíndromos são 'radar', 'Atlas salta' e (se espaços forem ignorados) 'erro comum ocorre'. Escreva uma função recursiva `isPalindrome` que retorna se a string armazenada no array for um palíndromo. A função deve ignorar espaços e pontuação na string.
- 7.33 (Pesquisa linear) Modifique o programa na Figura 7.19 de modo que ele utilize `separador` para realizar uma pesquisa linear do array. A função deve receber um array de inteiros e o tamanho do array como argumentos. Se a chave de pesquisa encontrada, retorne o subscrito do array; caso contrário, retorne -1.
- 7.34 (Oito Rainhas) Modifique o programa das Oito Rainhas criado no Exercício 7.26 para resolver o problema recursivamente.
- 7.35 (Imprimir um array) Escreva uma função `printArray` que aceita um array, um subscrito inicial e um subscrito final como argumentos e nada retorna. A função deve parar o processamento e retornar quando o subscrito inicial for igual ao subscrito final.
- 7.36 (Inverter uma string) Modifique a função `reverse` de modo que aceite um array de caracteres contendo uma string e um subscrito inicial como argumento, imprime a string de trás para frente e nada retorna. A função deve parar o processamento e retornar quando o caractere nulo de terminação for encontrado.
- 7.37 (Localizar o valor mínimo em um array) Escreva uma função `recursiveMin` que aceita um array de inteiros, um subscrito inicial e um subscrito final como argumentos e retorna o menor elemento do array. A função deve parar o processamento e retornar o subscrito inicial for igual ao subscrito final.

### Exercícios com vetor

- 7.38 Utilize um vetor de inteiros para resolver o problema descrito no Exercício 7.10.
- 7.39 Modifique o programa de jogo de dados que você criou no Exercício 7.17 de modo que ele utilize numeros de vezes que cada possível soma dos dois dados aparece.
- 7.40 (Localizar o valor mínimo em um array) Modifique sua solução do Exercício 7.37 para localizar o valor mínimo em um vez de em um array.

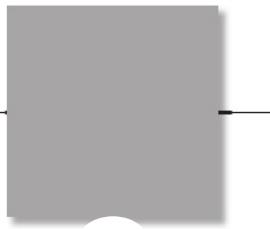


Os endereços nos são dados para ocultar nosso paradeiro.  
Saki (H. H. Munro)

Quando os saudamos ao caminho por desvios.  
William Shakespeare

Muitas coisas, quando tudo parece apontar para o consenso, podem funcionar de maneira adversa.  
William Shakespeare

Você descobrirá que é uma prática muito boa sempre verificar suas referências, senhor!  
Dr. Routh



## Ponteiros e strings baseadas em ponteiro

### OBJETIVOS

Neste capítulo, você aprenderá:

O que são ponteiros.

As semelhanças e diferenças entre ponteiros e referências e quando utilizar cada um.

Como utilizar ponteiros para passar argumentos a funções por referência.

Como utilizar strings no estilo C baseadas em ponteiro.

Os estreitos relacionamentos entre ponteiros, arrays e strings no estilo C.

Como utilizar ponteiros para funções.

A declarar e utilizar arrays de strings no estilo C.

|                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>P</b><br><b>á</b><br><b>m</b><br><b>S</b> | <ul style="list-style-type: none"> <li>8.1 Introdução</li> <li>8.2 Declarações de variável ponteiro e inicialização</li> <li>8.3 Operadores de ponteiro</li> <li>8.4 Passando argumentos para funções por referência com ponteiros</li> <li>8.5 Utilizando const com ponteiros</li> <li>8.6 Classificação por seleção utilizando passagem por referência</li> <li>8.7 Operador sizeof</li> <li>8.8 Expressões e aritmética de ponteiro</li> <li>8.9 Relacionamento entre ponteiros e arrays</li> <li>8.10 Arrays de ponteiros</li> <li>8.11 Estudo de caso: simulação de embaralhamento e distribuição de cartas</li> <li>8.12 Ponteiros de função</li> <li>8.13 Introdução ao processamento de string baseada em ponteiro           <ul style="list-style-type: none"> <li>8.13.1 Fundamentos de caracteres e strings baseadas em ponteiro</li> <li>8.13.2 Funções de manipulação de string da biblioteca de tratamento de strings</li> </ul> </li> <li>8.14 Síntese</li> </ul> |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

[Resumo](#) | 
 [Terminologia](#) | 
 [Exercícios de revisão](#) | 
 [Respostas dos exercícios de revisão](#) | 
 [Exercícios](#) | 
 [Seção Especial: construindo seu próprio computador](#) | 
 [Mais exercícios sobre ponteiros](#) | 
 [Exercícios de manipulação de string](#) | 
 [Seção especial: exercícios avançados de manipulação de string](#) | 
 [Um projeto desafiador de manipulação de string](#)

## 8.1 Introdução

Este capítulo discute um dos recursos mais poderosos da linguagem de programação C++, o ponteiro. No Capítulo 6, vimos como referências podem ser utilizadas para realizar a passagem por referência. Os ponteiros também permitem a passagem por referência, mas podem ser utilizados para criar e manipular estruturas de dados dinâmicas (isto é, estruturas de dados que podem crescer e expandir) como listas vinculadas, filas, pilhas e árvores. Este capítulo explica conceitos básicos do ponteiro e reforça o relacionamento entre arrays e ponteiros. A visão de arrays como ponteiros deriva da linguagem de programação C. Como vimos no Capítulo 7, a *C++ Standard Library* fornece uma implementação de arrays como objetos completos.

De modo semelhante, o C++ realmente oferece dois tipos de strings que utilizamos desde o Capítulo 3: `string` (que é baseada em arrays de caractere) e `string*` (baseadas em ponteiros no estilo C). Este capítulo sobre ponteiros para strings fornece conhecimento de ponteiros. De fato, as strings terminadas por caractere nulo introduzidas na Seção 7.4 e utilizadas na Figura 7.12 são strings baseadas em ponteiros. Este capítulo também inclui uma coleção considerável de exercícios de processamento de string que utilizam `string*`. As strings baseadas em ponteiros no estilo C são amplamente utilizadas em sistemas C e C++ legados. Então, se trabalha com sistemas C ou C++ legados, você pode ser solicitado a usar strings baseadas em ponteiros.

Examinaremos o uso de ponteiros com classes no Capítulo 13, “Programação orientada a objetos: polimorfismo”, em que veremos que o chamado ‘processamento polimórfico’ de programação orientada a objetos é realizado com ponteiros e referências. O Capítulo 21, “Estruturas de dados”, apresenta exemplos de como criar e utilizar estruturas de dados dinâmicas que são implementadas com ponteiros.

## 8.2 Declarações de variável ponteiro e inicialização

As variáveis ponteiro contêm endereços de memória como seus valores. Normalmente, uma variável contém diretamente um valor.

**Atenção!** Mas uma variável ponteiro contém o endereço de memória de uma variável que (Figura 8.1) pode conter um valor específico. Nesse sentido, um ponteiro é freqüentemente chamado de seta, porque os diagramas normalmente representam um ponteiro como uma seta da variável que contém um endereço para a variável localizada nesse endereço na memória.

Os ponteiros, como qualquer outra variável, devem ser declarados antes de ser utilizados. Por exemplo, para o ponteiro na Seção 8.1, a declaração

```
int *countPtr, count;
declara a variável countPtr como do tipo int (isto é, um ponteiro para int) e declara count como
```

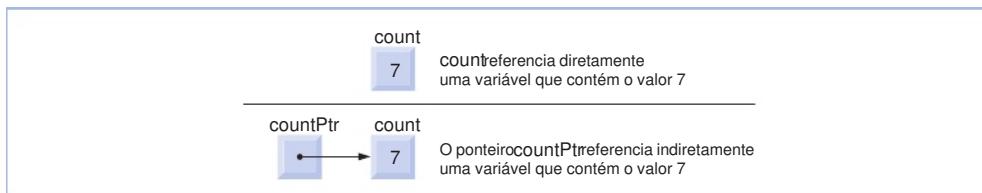


Figura 8.1 Referenciando direta e indiretamente uma variável.

um ponteiro para `int` na declaração se aplica a `int`. Toda variável sendo declarada como um ponteiro deve ser precedida por um asterisco. Por exemplo, a declaração `int *count; count = 7;` indica que tanto `count` quanto `*count` são ponteiros para `int`. Quando `*` aparece em uma declaração, ele não é um operador; em vez disso, indica que a variável sendo declarada é um ponteiro. Os ponteiros podem ser declarados para apontar para o qualquer tipo de dados.

**Erro comum de programação 8.1**

Supor que `*` é usado para declarar um ponteiro distribui-se por todos os nomes de variável na lista de variáveis separadas por vírgulas de uma declaração pode levar a erros. Cada ponteiro deve ter uma declaração separada (com ou sem espaço entre eles — o compilador ignora o espaço). Declarar apenas uma variável por declaração ajuda a evitar esse tipo de erro e melhora a legibilidade do programa.

**Boa prática de programação 8.1**

Embora não seja um requisito, incluir asletas de variáveis ponteiro torna claro que essas variáveis são ponteiros e devem ser tratadas apropriadamente.

Os ponteiros devem ser inicializados quando forem declarados ou em uma atribuição. Um ponteiro pode ser inicializado com `NULL` ou como um endereço. Um ponteiro que não aponta para nada é conhecido como `apontante nulo`. O símbolo `NULL` é definida no arquivo de cabeçalho `<cs50.h>` em vários outros arquivos de cabeçalho da biblioteca-padrão para representar o valor nulo. É equivalente a inicializar um ponteiro com `0`, que é utilizado por convenção. Quando atribuído, ele é convertido em um ponteiro do tipo `apontante nulo`. O tipo inteiro que pode ser atribuído diretamente a uma variável ponteiro sem primeiro fazer coerção do inteiro em um tipo ponteiro. A atribuição de um número inteiro a um ponteiro é discutida na Seção 8.3.

**Dica de prevenção de erro 8.1**

Inicialize ponteiros para impedir apontar para áreas da memória desconhecidas ou não inicializadas.

### 8.3 Operadores de ponteiro

O operador de endereço (`&`) é um operador unário que retorna o endereço de memória de seu operando. Por exemplo, considerando as declarações

```
int y = 5; // declara variável y
int *yPtr; // declara variável ponteiro yPtr
```

a instrução

```
yPtr = &y; // atribui o endereço de y a yPtr
```

atribui o endereço da variável ponteiro. Então dizemos que a variável `yPtr` agora aponta para o valor da variável `y`. Note que o resultado da atribuição anterior não é imediatamente visível, porque a variável de referência, que sempre é precedida por um nome de tipo de dados.

A Figura 8.2 mostra uma representação esquemática da memória depois da atribuição precedente. O ‘relacionamento de apontamento’ é indicado desenhando uma seta a partir da caixa que representa o ponteiro para a caixa que representa a variável memória.

A Figura 8.3 mostra outra representação do ponteiro na memória, supondo que a variável de tipo inteiro `y` está armazenada na posição da memória que a variável `yPtr` armazena na posição da memória do operando do operador de

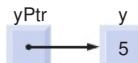


Figura 8.2 Representação gráfica de um ponteiro que aponta para uma variável na memória.



Figura 8.3 Representação de y e yPtr na memória.

endereço deve ser **valido** (isto é, algo a que um valor pode ser atribuído, como um nome de variável ou uma referência); o operador de endereço não pode ser aplicado a constantes ou expressões que não resultam em referências.

O operador `*`, comumente referido como **operador de indireção** ou **operador de desreferenciamento**, retorna um sinônimo (isto é, um alias ou um apelido) ao objeto para o qual seu operando de ponteiro aponta. Por exemplo (referindo-se novamente à Figura 8.1):

```
cout << *yPtr << endl;
imprime o valor de y
```

Utilizar dessa maneira é chamado de **desreferenciar um ponteiro**. Observe que um ponteiro desreferenciado também pode ser

utilizado no lado esquerdo de uma instrução de atribuição, como em

```
*yPtr = 9;
```

que atribui 9 na Figura 8.3. O ponteiro desreferenciado também pode ser utilizado para receber um valor de entrada como em

```
cin >> *yPtr;
```

o que coloca o valor de entradaponteiro desreferenciado em



#### Erro comum de programação 8.2

Desreferenciar um ponteiro que não foi inicializado de modo adequado ou que não foi atribuído para apontar a uma posição específica na memória poderia produzir um erro fatal em tempo de execução, ou poderia modificar accidentalmente dados importantes e permitir que o programa executasse até a conclusão, possivelmente com resultados incorretos.



#### Erro comum de programação 8.3

Uma tentativa de desreferenciar uma variável que não é um ponteiro é um erro de compilação.



#### Erro comum de programação 8.4

Desreferenciar um ponteiro nulo é normalmente um erro fatal em tempo de execução.

O programa na Figura 8.4 demonstra os operadores de ponteiro. Neste ponto, as posições da memória são enviadas para a saída por como inteiros hexadecimais (isto é, base 16). (Consulte o Apêndice D, "Sistemas de numeração", para obter informações adicionais sobre inteiros hexadecimais.) Observe que a saída de yPtr é diferente da saída de y, porque a saída de yPtr é desreferenciada, enquanto a saída de y é diretamente o valor da variável.



#### Dica de portabilidade 8.1

O formato em que um ponteiro é enviado para a saída depende do compilador. Alguns sistemas geram saída de valores de ponteiros como inteiros hexadecimais, enquanto outros sistemas utilizam inteiros decimais.

Note que o endereço (base 15) e o valor de y (base 16) são idênticos na saída, confirmando que o endereço de yPtr foi atribuído à variável y. Os operadores `*` e `&` são o oposto um do outro — quando os dois são aplicados consecutivamente a yPtr em qualquer ordem, eles se 'cancelam' e o mesmo resultado é impresso:

```

1 // Figura 8.4: fig08_04.cpp
2 // Utilizando os operadores & e *.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int a; // a é um inteiro
10 int *aPtr; // aPtr é um ponteiro int * -- para um inteiro
11
12 a = 7; // atribuiu 7 a a
13 aPtr = &a; // atribui o endereço de a ao aPtr
14
15 cout <<"The address of a is " << &a
16 << endl;
17 cout << "\n\nThe value of a is " << a
18 << endl;
19 cout << "\n\nThe value of *aPtr is " << *aPtr;
20 << endl;
21 cout << "\n\nShowing that * and & are inverses of "
22 << "each other.\n&aPtr = " << &aPtr
23 << endl;
24 << "\n*aPtr = " << *aPtr << endl;
25
26 return 0; // indica terminação bem-sucedida
27 }

```

The address of a is 0012F580  
 The value of aPtr is 0012F580

The value of a is 7  
 The value of \*aPtr is 7

Showing 0012F580 & are inverses of each other.  
 \*&aPtr = 0012F580

Figura 8.4 Operadores de ponteiro & e \*.

A Figura 8.5 lista a precedência e a associatividade dos operadores introduzidos até agora. Observe que o operador de encadernação é o de desreferência, ou seja, os operadores unários no terceiro nível de precedência no gráfico.

#### 8.4 Passando argumentos para funções por referência com ponteiros

Há três maneiras em C++ de passar argumentos para uma função: [passagem por valor](#), [passagem por referência](#) e [passagem por ponteiro](#).

O Capítulo 6 comparou e contrastou a passagem por valor e a passagem por referência com argumentos de referência. Nesta seção, explicamos como passar por referência com argumento de ponteiro.

Como vimos no Capítulo 6, pode ser utilizado para retornar um valor de uma função chamada para um chamador (ou retornar o controle de uma função chamada sem passar um valor de volta). Vimos também que os argumentos podem ser passados para uma função usando argumentos de referência. Esses argumentos permitem que a função chamada modifique os valores dos argumentos no chamador. Os argumentos de referência também permitem aos programas passar grandes objetos de dados entre função e evitam o overhead de passar os objetos por valor (o que, naturalmente, exige a produção de uma cópia do objeto). Os argumentos de referência, também podem ser utilizados para modificar uma ou mais variáveis no chamador ou passar ponteiros para grandes conjuntos de dados a fim de evitar o overhead de passar os objetos por valor.

Em C++, os programadores podem utilizar ponteiros e [apenas realizar a passagem por referência](#) (exatamente como a passagem por referência é feita em programas C, porque o C não tem referências). Ao chamar uma função com um argumento que deve ser modificado, o endereço do argumento é passado. Isso normalmente é realizado aplicando o operador de endereçamento da variável cujo valor será modificado.

| Operadores                              | Associatividade            | Tipo                 |
|-----------------------------------------|----------------------------|----------------------|
| () []                                   | da esquerda para a direita | mais alto            |
| ++ -- static_cast < tipo > ( operando ) | da esquerda para a direita | unário (pós-fixo)    |
| ++ -- + - ! & *                         | da direita para a esquerda | unário (prefixo)     |
| * / %                                   | da esquerda para a direita | multiplicativo       |
| + -                                     | da esquerda para a direita | aditivo              |
| << >>                                   | da esquerda para a direita | inserção/extração    |
| < <= > >=                               | da esquerda para a direita | relacional           |
| == !=                                   | da esquerda para a direita | igualdade            |
| &&                                      | da esquerda para a direita | E lógico             |
|                                         | da esquerda para a direita | OU lógico            |
| ? :                                     | da direita para a esquerda | ternário condicional |
| = += -= *= /= %=                        | da direita para a esquerda | atribuição           |
| ,                                       | da esquerda para a direita | vírgula              |

Figura 8.5 Precedência e associatividade de operadores.

Como vimos no Capítulo 7, os arrays não são passados como parâmetro array é a posição inicial na memória do array (isto é, um nome de array já é um ponteiro). A variável é equivalente a `arrayName[0]`. Quando o endereço de uma variável é passado para uma função, \*é possível utilizá-lo diretamente\* — é a função para formar um sinônimo para o nome da variável — isso por sua vez pode ser utilizado para modificar o valor da variável naquela localização na memória chamador.

As figuras 8.6 e 8.7 possuem duas versões presentes de uma função `cubeByValueByReference` — `cubeByValue` — que A Figura 8.6 passa `number` por valor para a função `cubeByValue` (linhas 15). A função `cubeByValue` (linhas 21–24) eleva seu argumento ao cubo e passa o novo valor de volta para a instrução (linha 23). O novo valor é atribuído a `number` (linha 15). Observe que a função chamadora tem a oportunidade de examinar o resultado da chamada de função ante de modificar o valor da variável. Por exemplo, nesse programa, poderíamos ter armazenado o resultado de `cubeByValue` em uma variável, examinado seu valor e atribuído esse resultado depois de determinar que o valor retornado era razoável.

A Figura 8.7 passa `number` para a função `cubeByReference` utilizando a passagem por referência com um argumento de ponteiro (linha 15) — o endereço é passado para a função `cubeByReference` (linhas 22–25) especifica o parâmetro `nPtr` (um ponteiro) para receber seu argumento. A função desreferencia o ponteiro e eleva ao cubo o valor para o qual aponta (linha 24). Isso altera diretamente o valor de

```

1 // Figura 8.6: fig08_06.cpp
2 // Eleva uma variável ao cubo utilizando passagem por valor.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 int cubeByValue(int); // protótipo
7
8
9 int main()
10 {
11 int number = 5;
12

```

Figura 8.6 Passagem por valor utilizada para elevar o valor de uma variável ao cubo.

(continua)

```

13 cout << "The scinal value of number is " << number;
14 number = cubeByValue(number);
 // passa number por valor ao cubeByValue
15 cout << "\nThe new value of number is << number << endl;
16 return 0; // indica terminação bem-sucedida
17 } // fim de main
18
19 // calcula e retorna o cubo do argumento inteiro
20 int cubeByValue(int n)
21 {
22 return n * n * n; // eleva a variável local n ao cubo e retorna o resultado
23 } // fim da função cubeByValue

```

The scinal value of number is 5  
 The new value of number is 125

Figura 8.6 Passagem por valor utilizada para elevar o valor de uma variável ao cubo.

(continuação)

```

1 // Figura 8.7: fig08_07.cpp
2 // Eleva uma variável ao cubo usando passagem por referência com um argumento nPtr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void cubeByReference(*); // protótipo
8
9 int main()
10 {
11
12 int number = 5;
13 cout << "The scinal value of number is " << number;
14
15 cubeByReference(&number);
 // passa endereço de number para cubeByReference
16
17 cout << "\nThe new value of number is << number << endl;
18 return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // calcula o cubo de *nPtr; modifica a variável number em main
22 void cubeByReference(*nPtr)
23 {
24 *nPtr = *nPtr * *nPtr * *nPtr; // eleva *nPtr ao cubo
25 } // fim da função cubeByReference

```

The scinal value of number is 5  
 The new value of number is 125

Figura 8.7 Passagem por referência com um argumento de ponteiro utilizado para elevar ao cubo o valor de uma variável.



### Erro comum de programação 8.5

Não desreferenciar um ponteiro quando é necessário fazer isso para obter o valor para o qual o ponteiro aponta é um erro.

Uma função que recebe um endereço como um argumento deve definir um parâmetro de ponteiro para receber o endereço. Por exemplo, o cabeçalho `#include <cslibs.h>` (linha 22) especifica que `cubeByReference` recebe o endereço de uma variável `int` (isto é, um ponteiro para `int`) como um argumento, armazena o endereço localmente e, em seguida, modifica o valor.

O protótipo de função `cubeByValue` (Referência 7) contém entre parênteses. Como com outros tipos de variável, não é necessário incluir nomes de parâmetros de ponteiro em protótipos de função. Os nomes de parâmetro incluídos para propós documentação são ignorados pelo compilador.

As figuras 8.8 e 8.9 analisam graficamente a execução dos programas das figuras 8.6 e. 8.7, respectivamente.



### Observação de engenharia de software 8.1

Utilize a passagem por valor para passar argumentos para uma função a menos que o chamador requeira explicitamente que a função chamada modifique diretamente o valor da variável do argumento no chamador. Esse é outro exemplo do princípio do menor privilégio.

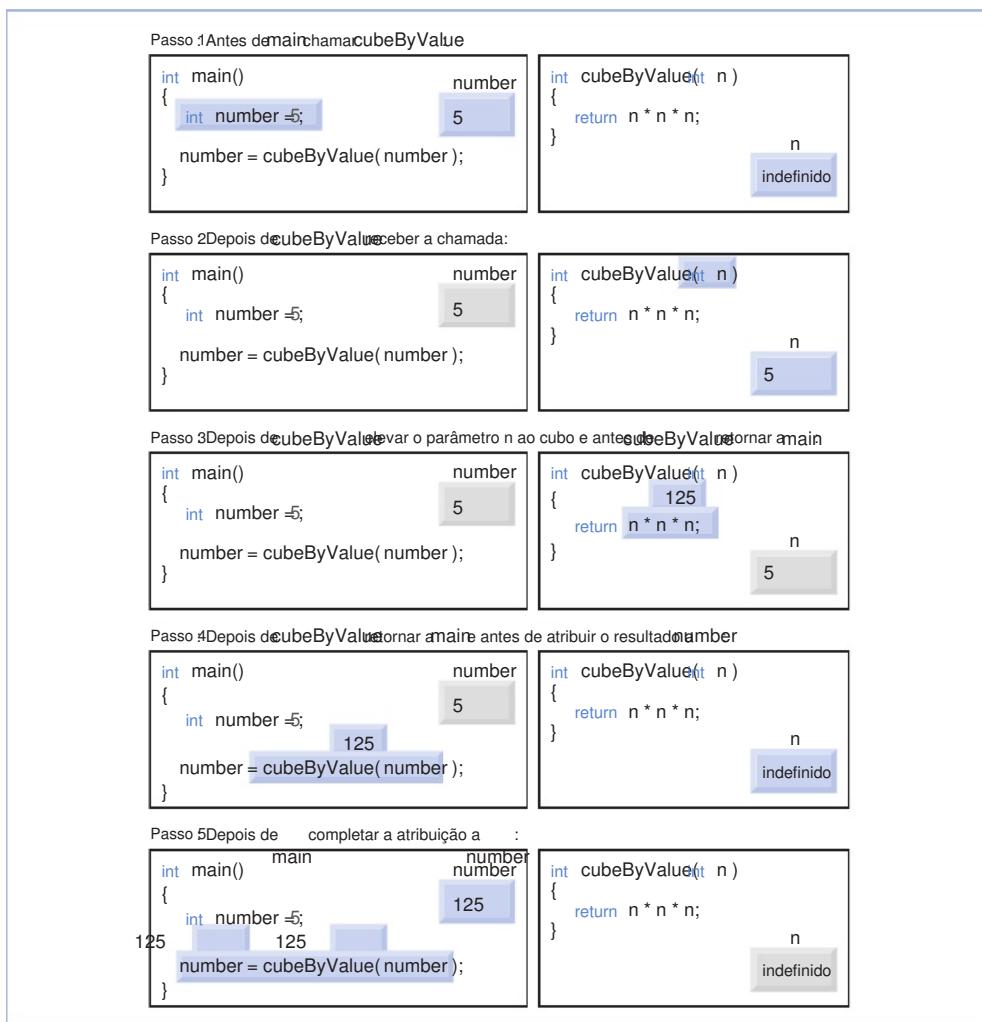


Figura 8.8 Análise da passagem por valor do programa da Figura 8.6.

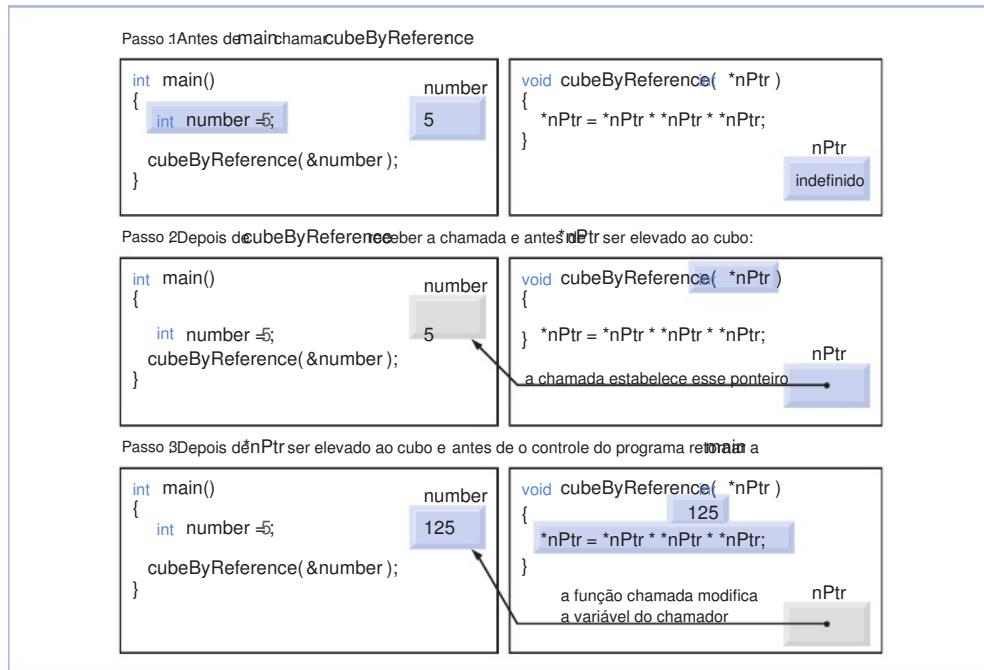


Figura 8.9 Análise da passagem por referência (com um argumento de ponteiro) do programa da Figura 8.7.

No cabeçalho e no protótipo de uma função que espera um array unidimensional como um argumento, pode-se utilizar a notação `arrayByValue`. O compilador não diferencia entre uma função que recebe um ponteiro e uma que recebe um array unidimensional. Isso, naturalmente, quer dizer que a função deve 'saber' quando está recebendo um array e plesamente uma única variável pela qual realizar a passagem por referência. Quando o compilador encontra um parâmetro de função com o tipo `array`, ele converte o parâmetro em notação de ponteiro ('um ponteiro para um inteiro'). Ambas as formas de declarar um parâmetro de função como um array dimensional são intercambiáveis.

## 8.5 Utilizando `const` com ponteiros

Lembre-se de que o `const` permite ao programador informar ao compilador que o valor de uma variável particular não deve ser modificado.



### Dica de portabilidade 8.2

Embora `const` seja bem definido em ANSI C e C++, alguns compiladores não o implementam adequadamente. Portanto, uma boa regra é: 'Conheça seu compilador'.

Ao longo dos anos, nas primeiras versões do C, foi escrita uma grande base de código que não utilizava `const` (ou `const` era ignorado). Ainda assim, muitos programadores oportunidades de programar de forma mais segura e eficiente. No entanto, é importante lembrar que o uso de `const` não garante a segurança do código, porque começaram a programar nas primeiras versões de C. Esses programadores estão perdendo muitas oportunidades de aplicar boa engenharia de software.

Há muitas possibilidades para utilizar `const` em parâmetros de função. Como escolher a mais adequada dessas possibilidades? Deixe o princípio do menor privilégio ser seu guia. Sempre dê a uma função acesso suficiente aos dados em râmetros para realizar sua tarefa especificada, porém não mais. Esta é a segunda das regras de ponteiro para impor o princípio do menor privilégio.

O Capítulo 6 explicou que, quando uma função é chamada utilizando passagem por valor, uma cópia do argumento (ou argumento) é feita e passada à função. Se a cópia é modificada na função, o valor original é mantido no chamador sem

Em muitos casos, um valor passado a uma função é modificado para que a função possa realizar sua tarefa. Entretanto, em algum valor não deve ser alterado na função chamada, mesmo que a função chamada manipule apenas uma cópia do valor original.

Por exemplo, considere uma função que aceita um array unidimensional e seu tamanho como argumentos e, subsequenteemente imprime o array. Tal função deve fazer loop pelo array e gerar saída de cada elemento de array individualmente. O tamanho do array é utilizado no corpo da função para determinar o subscrito mais alto do array a fim de que o loop possa terminar quando a impressão é concluída. O tamanho do array não muda no corpo de função, então o resultado é sempre o mesmo. Isso é importante porque um array inteiro é sempre passado por referência e poderia ser facilmente alterado na função chamada.



### Observação de engenharia de software 8.2

Se um valor não muda (ou não deve mudar) no corpo de uma função para o qual ele é passado, o parâmetro deve ser declarado com `const` para assegurar que ele não seja modificado acidentalmente.

Se houver uma tentativa de modificar um valor ou um erro é emitido, dependendo do compilador particular.



### Dica de prevenção de erro 8.2

Antes de utilizar uma função, verifique seu protótipo de função para determinar os parâmetros que ele pode modificar.

Há quatro maneiras de passar um ponteiro para uma função: um ponteiro não constante para dados não constantes (Figura 8.10), um ponteiro não constante para dados constantes (Figuras 8.11 e 8.12), um ponteiro constante para dados não constantes (Figura 8.13) e um ponteiro constante para dados constantes (Figura 8.14). Cada combinação fornece um nível diferente de privilégios de acesso.

Ponteiro não constante para dados não constantes

O acesso mais alto é concedido por um ponteiro não constante para dados não constantes. Os dados podem ser modificados pelo ponteiro desreferenciado e o ponteiro pode ser modificado para apontar para outros dados. A declaração de um ponteiro não constante para dados não constantes é a seguinte. Esse ponteiro pode ser utilizado para receber uma string terminada por caractere nulo em uma função que altera o valor do ponteiro para processar (e, possivelmente, modificar) cada caractere na string. A partir daí, lembre-se de que uma string terminada com caractere nulo pode ser colocada em um array de caracteres que contém os caracteres da string e um caractere nulo que indica onde a string termina.

```

1 // Figura 8.10: fig08_10.cpp
2 // Convertendo minúsculas em maiúsculas
3 // utilizando um ponteiro não constante para dados não constantes.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <cctype> // protótipos para islower e toupper
9 using std::islower;
10 using std::toupper;
11
12 void convertToUppercase(ar *);
13
14 int main()
15 {
16 char phrase[] = "characters and $32.98"
17
18 convertToUppercase(phrase);
19 cout << "Conversion is: "<< phrase;
20 cout << "\nThe phrase after conversion is: " << phrase << endl;
21 return 0; // indica terminação bem-sucedida
22 } // fim de main
23
24 // converte string em letras maiúsculas
25 void convertToUppercase(ar *sPtr)

```

Figura 8.10 Convertendo uma string em letras maiúsculas.

(continua)

```

26 {
27 while (*sPtr != '\0') // faz loop enquanto caractere atual não é '\0'
28 {
29 if (islower(*sPtr)) // se o caractere estiver em minúsculas,
30 *sPtr = toupper(*sPtr); // converte em maiúsculas
31
32 sPtr++; // move sPtr para o próximo caractere na string
33 } // fim do while
34 } // fim da função convertToUppercase

```

The phrase before conversion is: characters and \$32.98  
The phrase after conversion is: CHARACTERS AND \$32.98

Figura 8.10 Convertendo uma string em letras maiúsculas.

(continuação)

Na Figura 8.10, a função `convertToUppercase` (linhas 25–34) declara o parâmetro `sPtr` (linha 25) como sendo um ponteiro não constante para dados não constantes (`const` não mencionado). A função processa um caractere por vez a partir da string terminada por caractere nulo armazenada no parâmetro `sPtr` (linhas 27–38). Tenha em mente que o nome de um array de caracteres na realidade equivale a um ponteiro para o primeiro caractere desse array. Portanto, é possível passar para a função `convertToUppercase` a função `islower` (linha 29) que aceita um argumento de caractere e retorna verdadeiro se o caractere for uma letra minúscula e falso, caso contrário. Os caracteres são interconvertidos em suas letras maiúsculas correspondentes pela função `toupper` (linha 30); outros permanecem inalterados. O operador `++` é usado para incrementar o ponteiro `sPtr` (linha 32) para o próximo caractere na string. Se o operador `++` fosse aplicado ao caractere correspondente, o resultado seria o caractere anterior. As funções `islower` e `toupper` fazem parte da biblioteca de tratamento de caracteres (veja Capítulo 22, “Bits, caracteres, strings C e structs”). Depois de processar um caractere, a lista `for(;;)` (linhas 22–26) é executada. Quando o operador `++` é aplicado a um ponteiro que aponta para um array, o endereço de memória armazenado no ponteiro é modificado para apontar para o próximo elemento do array (nesse caso, o próximo caractere na string). Adicionar um a um pode ser feito com a operação `sPtr++`, que é discutida em detalhes nas seções 8.8 e 8.9.

**Ponteiro não constante para dados constantes** Um ponteiro não constante para dados constantes é um ponteiro que pode ser modificado para apontar para qualquer item de dados do tipo apropriado, mas os dados para os quais ele aponta não podem ser modificados por esse ponteiro. Esse ponteiro poderia ser usado para receber um argumento de array para uma função que irá processar cada elemento do array, mas não deve ter permissão para modificar os dados. Por exemplo, a função `printCharacters` (linhas 22–26 da Figura 8.11) declara o parâmetro `sPtr` para ser do tipo `const char *`, para que ele possa receber uma string baseada em ponteiro terminada por caractere nulo. A declaração é lida da seguinte forma: “`sPtr` é um ponteiro para uma constante do tipo caractere”. O corpo da função realiza uma instrução `for(;;)` (linhas 24–25) para gerar saída de cada caractere na string até que o caractere nulo seja encontrado. Depois que cada caractere é impresso, o ponteiro `sPtr` é incrementado para apontar para o próximo caractere na string (íso funciona). A função `printCharacters` é mainçaria a array phrase para ser passada para printCharacters. Novamente, podemos passar `phrase` para printCharacters porque o nome do array é, na realidade, um ponteiro para o primeiro caractere no array.

```

1 // Figura 8.11: fig08_11.cpp
2 // Imprimindo uma string um caractere por vez utilizando
3 // um ponteiro não constante para dados constantes.
4 #include <iostream>
5
6 using std::endl;
7
8 void printCharacters(const char *);
9
10 int main()
11 {
12 const char phrase[] = "print characters of a string" ;

```

Figura 8.11 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes.

(continua)

```

13 cout << "The string is:\n" ;
14 printCharacters(phrase); // imprime caracteres em phrase
15 cout << endl;
16 return 0; // indica terminação bem-sucedida
17 } // fim de main
18
19 // sPtr pode ser modificado, mas não pode modificar o caractere para o qual
20 // ele aponta, isto é, sPtr é um ponteiro 'de leitura'
21 void printCharacters(const char *sPtr)
22 {
23 for (; *sPtr != '\0' ; sPtr++) // nenhuma inicialização
24 cout << *sPtr; // exibe caractere sem modificação
25
26 } // fim da função printCharacters

```

The string is:  
print characters of a string

Figura 8.11 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes.

(continuação)

A Figura 8.12 demonstra as mensagens de erro de compilação produzidas ao tentar compilar uma função que recebe um ponteiro não constante para dados constantes e, então, tenta utilizar esse ponteiro para modificar os dados. [As mensagens de erro de compilador variam entre compiladores.]

Como sabemos, os arrays são tipos de dados agregados que armazenam itens de dados relacionados do mesmo tipo sob um único nome. Quando uma função é chamada com um array como um argumento, o array é passado à função por referência. Entretanto, os arrays são sempre passados por valor — uma cópia do objeto inteiro é passada. Isso requer o overhead em tempo de execução de uma cópia de cada item de dados no objeto e armazená-la na pilha de chamada de função. Quando um objeto deve ser passado para uma função, podemos utilizar um ponteiro para dados constantes (ou uma referência para dados constantes) para obter o desempenho da passagem por referência e a proteção da passagem por valor. Quando um ponteiro for passado para um objeto, deve-se fazer uma cópia do endereço do objeto; o objeto em si não é copiado. Em uma máquina com endereços de quatro bytes, uma cópia de 4 bytes de memória é feita em vez de uma cópia de um objeto possivelmente grande.

```

1 // Figura 8.12: fig08_12.cpp
2 // Tentando modificar dados por meio de um
3 // ponteiro não-constante para dados constantes.
4
5 void f(const int *); // protótipo
6
7 int main()
8 {
9 int y;
10
11 f(&y); // f tenta modificação não-válida
12 return 0; // indica terminação bem-sucedida
13 } // fim de main
14
15 // xPtr não pode modificar o valor da variável constante para a qual ele aponta
16 void f(const int *xPtr)
17 {
18 *xPtr = 100; // erro: não é possível modificar objeto const
19 } // fim da função f

```

Figura 8.12 Tentando modificar dados por meio de um ponteiro não-constante para dados constantes.

(continua)

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_12.cpp 18:
 Cannot modify a const object in function f(const int *)
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphttp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
error C2166: l-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

Figura 8.12 Tentando modificar dados por meio de um ponteiro não constante para dados constantes.

(continuação)



#### Dica de desempenho 8.1

Se eles não precisarem ser modificados pela função chamada, passe os objetos grandes utilizando ponteiros para dados constantes ou referências para dados constantes, para obter os benefícios de desempenho da passagem por referência.



#### Observação de engenharia de software 8.3

Passe objetos grandes utilizando ponteiros para dados constantes, ou referências para dados constantes, para obter a segurança da passagem por valor.

Ponteiro constante para dados não constantes

Um **ponteiro constante para dados não constantes** é um ponteiro que sempre aponta para a mesma posição da memória; os dados

**passa a posição de memória**. Toda modificação feita por esse tipo de ponteiro pode ser vista por todos os outros. Essa é a razão de se utilizar o termo **const** para um ponteiro constante para dados não constantes. Ele pode ser utilizado para receber um array como um argumento para uma função. O ponteiro constante para array é útil quando os elementos do array são constantes. Os ponteiros que são inicializados quando são declarados. (Se o ponteiro for um parâmetro de função, ele será inicializado com um ponteiro que é passado para a função.) O da Figura 8.13 tenta modificar um ponteiro constante. A linha 10 é a que causa o erro. A declaração na figura é lida da direita para a esquerda: ‘ponteiro constante para um inteiro não constante’. O ponteiro é inicializado

```
1 // Figura 8.13: fig08_13.cpp
2 // Tentar modificar um ponteiro constante para dados não constantes.
3
4 int main()
5 {
6 int x, y;
7
8 // ptr é um ponteiro constante para um inteiro que pode
9 // ser modificado por este ponto. ptr sempre aponta para a
10 // mesma posição da memória.
11 int * const ptr = &x; // ponteiro const deve ser inicializado
12
13 *ptr = 7; // permitido: *ptr não é const
14 ptr = &y; // erro: ptr é constante; não é possível atribuí-lo a um novo endereço
15 return 0; // indica terminação bem-sucedida
16 } // fim de main
```

Figura 8.13 Tentando modificar um ponteiro constante para dados não constantes.

(continua)

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
I-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

Figura 8.13 Tentando modificar um ponteiro constante para dados não constantes.

(continuação)

com o endereço de variável ~~delimberb4~~ tenta atribuir o endereço ao compilador gera uma mensagem de erro. Observe que nenhum erro ocorre quando a linha ~~após atribuir valor não-constante para ponteiro~~ pode ser modificada utilizando ~~desreferenciado~~, mesmo que ~~desreferenciado~~ declarado como



#### Erro comum de programação 8.6

Não inicializar um ponteiro que é declarado durante a compilação.

Ponteiro constante para dados constantes

A menor quantidade de privilégio de acesso é concedida por ~~arrays~~ para dados constantes. Esse ponteiro sempre aponta para a mesma posição da memória, e os dados nessa posição da memória não podem ser modificados utilizando o ponteiro. A maneira como um array deve ser passado para uma função que somente lê o array, usando a notação de subscrito de array, não modifica o array. O programa da Figura 8.14 declara para ~~arr~~ para ~~arr~~ ponteiro \* const (linha 14). Essa declaração é lida da direita para a esquerda: 'ponteiro constante para um inteiro constante'. A figura mostra as mensagens de erro geradas quando se tenta modificar os dados na linha 18, quando se tenta modificar o endereço armazenado na variável ponteiro (linha 19). Observe que não ocorre nenhum erro quando o programa tenta desreferenciar saída do valor para a ponteira (linha 16), porque nem o ponteiro nem os dados para os quais ele aponta estão sendo modificados nessa instrução.

## 8.6 Classificação por seleção utilizando passagem por referência

Nesta seção, definimos um programa de classificação para demonstrar a passagem de arrays e elementos de array individuais por referência. Utilizamos o algoritmo de classificação por seleção, que é um algoritmo de classificação fácil de programar, mas, infelizmente, ineficiente. A primeira iteração do algoritmo seleciona o menor elemento no array e o troca pelo primeiro elemento. A segunda iteração seleciona o segundo menor elemento (que é o menor dos elementos restantes) e o troca pelo segundo elemento. O algoritmo continua a última iteração seleciona o segundo maior elemento e permite que seja trocado pelo penúltimo índice, deixando o maior elemento no último índice. Depois das ~~últimas~~ iterações, todos os itens do array serão classificados pela ordem dos elementos dos primeiros

Como um exemplo, considere o array

```
34 56 4 10 77 51 93 30 5 52
```

Um programa que implementa classificação por seleção determina o menor elemento (4) desse array, que está contido no elemento 2. O programa troca o 4 pelo elemento 0 (34), resultando em

```
4 56 34 10 77 51 93 30 5 52
```

[Nota: Utilizamos negrito para destacar os valores que foram trocados.] O programa então determina o menor valor dos elementos restantes (todos os elementos exceto 4), que é 5, contido no elemento 8. O programa troca o 5 pelo elemento 1 (56), resultando em

```
4 5 34 10 77 51 93 30 56 52
```

Na terceira iteração, o programa determina o próximo menor valor (10) e o troca pelo elemento 2 (34).

```
4 5 10 34 77 51 93 30 56 52
```

O processo continua até que o array seja completamente classificado.

```
4 5 10 30 34 51 52 56 77 93
```

```

1 // Figura 8.14: fig08_14.cpp
2 // Tentando modificar um ponteiro constante para dados constantes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int x = 5, y;
10
11 // ptr é um ponteiro constante para um inteiro constante.
12 // ptr sempre aponta para a mesma posição; o inteiro
13 // nessa posição não pode ser modificado.
14 const int * const ptr = &x;
15
16 cout << *ptr << endl;
17
18 *ptr = 7; // erro: *ptr é const; não é possível atribuir novo valor
19 ptr = &y; // erro: ptr é const; não é possível atribuir endereço
20
21 } // fim de main

```

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphttp5e_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
I-value specifies const object
c:\cpphttp5e_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
I-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

**Figura 8.14** Tentando modificar um ponteiro constante para dados constantes.

Observe que, depois da primeira iteração, o menor elemento estará na primeira posição. Depois da segunda iteração, os dois elementos estarão na ordem nas duas primeiras posições. Depois da terceira iteração, os três menores elementos estarão nas três primeiras posições.

A Figura 8.15 implementa a classificação por seleção utilizando a função `selectionSort` (linhas 36–53) classifica o array. A linha 38 `smallest = arr[0]` armazenará o índice do menor elemento no array restante.

~~As linhas 41–52 realizam a troca de elementos entre a linha 24 e a linha 43 cada vez que o menor elemento, armazenado na linha 18 para a iteração atual, é menor que o elemento atual. Assim, ao final das 52 iterações, o menor elemento sempre permanecerá no topo do array.~~

Fazemos agora um exame mais minucioso sobre o que o C++ impõe o ocultamento de informações entre funções, porque não tem acesso aos elementos de array individualmente. A função `selectionSort` que tem acesso aos elementos do array a serem ordenados, a cada um desses elementos — a referência — o endereço de cada elemento de array é passado explicitamente. Embora os arrays inteiros sejam passados por referência, os elementos individuais são escalares e comumente passados por valor. Parte operador de endereçamento para o elemento

do array na chamada (linha 51) para produzir a passagem por referência (linhas 57 e 62) recebe `i` na variável ponteiro `*element1Ptr` ocultando de informações importantes que nome `[ i ]`, mas `swap` pode utilizar `*element1Ptr` como um sinônimo de `a[ i ]`. Portanto, quando `*element1Ptr` é referenciado na realidade está referenciando `array[ i ]` e `selectionSort` de maneira semelhante à preferência de `*element2Ptr`, na realidade está referenciando `array[ smallest ]` e `selectionSort`.

Ainda que `swap` não tenha permissão de utilizar as instruções

```
hold = array[i];
array[i] = array[smallest];
array[smallest] = hold;
```

precisamente o mesmo efeito é alcançado por

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
```

na função `swap` (Figura 8.15).

```

1 // Figura 8.15: fig08_15.cpp
2 // Este programa coloca valores em um array, classifica os valores em
3 // ordem crescente e imprime o array resultante.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 void selectionSort(int * const, const int); // protótipo
12 void swap(int * const, int * const); // protótipo
13
14 int main()
15 {
16 const int arraySize = 10;
17 int a[arraySize] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19 cout << "Data items in scinal order\n" ;
20
21 for (int i = 0; i < arraySize; i++)
22 cout << setw(4) << a[i];
23
24 selectionSort(a, arraySize); // classifica o array
25
26 cout << "\nData items in ascending order\n"
27
28 for (int j = 0; j < arraySize; j++)
29 cout << setw(4) << a[j];
30
31 cout << endl;
32 } // fim de main // indica terminação bem-sucedida
33
34 // função para classificar um array
35 void selectionSort(int * const array, const int size)
36 {
37 int smallest; // índice do menor elemento
38
39 }
```

Figura 8.15 Classificação por seleção com passagem por referência.

(continua)

```

40 // itera sobre size - 1 elementos
41 for (int i = 0; i < size - 1; i++)
42 {
43 smallest = i; // primeiro índice do array remanescente
44
45 // faz um loop para localizar o índice do menor elemento
46 for (int index = i + 1; index < size; index++)
47
48 if (array[index] < array[smallest])
49 smallest = index;
50
51 swap(&array[i], &array[smallest]);
52 } // fim do for
53 } // fim da função selectionSort
54
55 // troca os valores nas posições da memória para as quais
56 // element1Ptr e element2Ptr apontem
57 void swap(int * const element1Ptr, int * const element2Ptr)
58 {
59 int hold = *element1Ptr;
60 *element1Ptr = *element2Ptr;
61 *element2Ptr = hold;
62 } // fim da função swap

```

Data items in srcinal order  
 2 6 4 8 10 12 89 68 45 37  
 Data items in ascending order  
 2 4 6 8 10 12 37 45 68 89

Figura 8 Classificação por seleção com passagem por referência.

(continuação)

Vários recursos das funções de classificação devem ser observados. O cabeçalho de função `selectionSort` declara `const array` em vez de `array[]`, para indicar que a função `selectionSort` recebe um array unidimensional como um argumento. Tanto o ponteiro do parâmetro `array` quanto o parâmetro `size` são declarados `const` para impor o princípio do menor privilégio. Embora o parâmetro `array` receba uma cópia de um array, é importante modificar a cópia não possa alterar o valor. A função `selectionSort` não precisa alterar `array` para realizar sua tarefa — o tamanho do array permanece fixo durante a execução. O parâmetro `size` é declarado `const` para assegurar que ele não seja modificado. Se o tamanho do array fosse modificado durante o processo de classificação, o algoritmo de classificação não teria executado corretamente.

Observe que a função `selectionSort` recebe o tamanho do array como um parâmetro, porque a função deve ter essas informações para classificar o array. Quando um array baseado em ponteiro é passado para uma função, somente o endereço de memória com o elemento do array é recebido pela função; o tamanho do array deve ser passado separadamente para a função.

Definindo a função `selectionSort` para receber o tamanho de array como um parâmetro, permitimos que a função seja utilizada por qualquer programa que classifique arrays unidimensionais de tamanho arbitrário. O tamanho do array poderia ter sido programado diretamente na função, mas isso restringiria a função a processar um array de um tamanho específico e reduziria a reusabilidade da função — somente os programas que processam arrays unidimensionais específicos codificados diretamente na função poderiam utilizar a função.



#### Observação do engenheiro de software 8.4

Ao passar um array para uma função, passe também o tamanho do array (em vez de construir na função o conhecimento sobre o tamanho do array). Isso torna a função mais reutilizável.

## 8.7 Operadores sizeof

O C++ fornece o operador `sizeof` para determinar o tamanho de um array (ou de qualquer outro tipo de dados, variável ou constante) em bytes durante a compilação de programa. Quando aplicado ao nome de um array, como na Figura 8.16 (linha 1), o operador `sizeof` retorna o número total de bytes no array `const int values[10]`. Na maioria dos

compiladores). Observe que esse é diferente de `int >`, por exemplo, que é o número de elementos do tipo inteiro no vetor. O computador que utilizamos para compilar esse programa armazena 8 bytes de memória, e array é declarado para ter 20 elementos (linha 12) ocupando 160 bytes na memória. Quando aplicado a um parâmetro de ponteiro (linha 24) em uma função que recebe um array como argumento, tamanho do ponteiro em bytes (4), não o tamanho do array.



### Erro comum de programação 8.7

Utilizar o operador `of` em uma função para localizar o tamanho em bytes de um parâmetro de array resulta no tamanho em bytes de um ponteiro, não no tamanho em bytes do array.

[Nota] Quando o compilador Borland C++ é utilizado para compilar a Figura 8.16, o compilador gera a mensagem de advertência "Parameter 'ptr' is never used in function getSize(double \*)". Esse aviso ocorre porque a realidade um operador em tempo de compilação; portanto não é visualizada no corpo da função em tempo de execução. Muitos compiladores emitem

aviso semelhante para quando se usa o operador `of` em variáveis não utilizadas de modo que não figura na lista de erros.

O número de elementos em um array também pode ser determinado utilizando o operador `of`. Por exemplo, duas operações considera a seguinte declaração de array:

```
double realArray[22];
```

Se variáveis do tipo `double` são armazenadas em oito bytes de memória, terá um total de 176 bytes. Para determinar o número de elementos no array, a seguinte expressão pode ser utilizada:

```
sizeof realArray / sizeof (double) // calcula o número de elementos
```

A expressão determina o número de bytes (176) e divide esse valor pelo número de bytes utilizados na memória para armazenar um valor (8); o resultado é o número de elementos (22).

```

1 // Figura 8.16: fig08_16.cpp
2 // Operador Sizeof quando utilizado em um nome de array
3 // retorna o número de bytes no array.
4 #include <iostream>
5
6 using std::cout;
7
8 size_t getSize(double *); // protótipo
9
10 int main()
11 {
12 double array[20]; // 20 doubles; o que ocupa 160 bytes em nosso sistema
13
14 cout << "The number of bytes in the array is " << sizeof (array);
15
16 cout << "\nThe number of bytes returned by getSize is "
17 << getSize(array) << endl;
18 return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // retorna o tamanho de ptr
22 size_t getSize(double *ptr)
23 {
24 return sizeof (ptr);
25 } // fim da função getSize

```

The number of bytes in the array is 160  
The number of bytes returned by getSize is 4

Figura 8.16 O operador `sizeof` quando aplicado a um nome de array retorna o número de bytes no array.

Determinando os tamanhos dos tipos fundamentais, um array e um ponteiro

O programa da Figura 8.17 utiliza o operador sizeof para calcular o número de bytes utilizados para armazenar a maioria dos tipos de dados padrão. Note que, na saída, todos possuem o mesmo tamanho. Os tipos podem ter tamanhos diferentes com base no sistema em que o programa é executado. Em ~~outro sistema pode ser~~, definidos com tamanhos diferentes.

```

1 // Figura 8.17: fig08_17.cpp
2 // Demonstrando o operador sizeof.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 char C; // variável de tipo char
10 short S; // variável de tipo short
11 int i; // variável de tipo int
12 long l; // variável de tipo long
13 float f; // variável de tipo float
14 double d; // variável de tipo double
15 long double ld; // variável de tipo long double
16 int array[20]; // array de int
17 int *ptr = array; // variável de tipo int *
18
19 cout << "sizeof c = " << sizeof c
20 << "\nsizeof(char) = " << sizeof(char)
21 << "\nsizeof s = " << sizeof s
22 << "\nsizeof(short) = " << sizeof(short)
23 << "\nsizeof i = " << sizeof i
24 << "\nsizeof(int) = " << sizeof(int)
25 << "\nsizeof(long) = " << sizeof(long)
26 << "\nsizeof(f) = " << sizeof f
27 << "\nsizeof(float) = " << sizeof(float)
28 << "\nsizeof(double) = " << sizeof(double)
29 << "\nsizeof(long double) = " << sizeof(long double)
30 << "\nsizeof(array) = " << sizeof array
31 << "\nsizeof(ptr) = " << sizeof ptr << endl;
32
33 return 0; // indica terminação bem-sucedida
34 }
35 // fim de main
36

```

```

sizeof c = 1 sizeof(char) = 1
sizeof s = 2 sizeof(short) = 2
sizeof i = 4 sizeof(int) = 4
sizeof l = 4 sizeof(long) = 4
sizeof f = 4 sizeof(float) = 4
sizeof d = 8 sizeof(double) = 8
sizeof ld = 8 sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4

```

Figura 8.17 Operador sizeof utilizado para determinar tamanhos de tipo de dados padrão.



Dica de portabilidade 8.3

O número de bytes utilizado para armazenar um tipo de dados particular pode variar entre sistemas. Ao escrever programas que dependem de tamanhos de tipo de dados e que executarão em vários sistemas, é importante que o programa utilize o número de bytes utilizados para armazenar os tipos de dados.

O operador `of` pode ser aplicado a qualquer nome de variável, nome de tipo ou literal constante. Quando um nome de variável (que não seja um nome de array) ou um valor constante, o número de bytes utilizados para armazenar o variável específica ou constante é retornado. Observe que o tipo desse resultado é o nome de um tipo (por exemplo, `for`) fornecido como seu operando. Os parentêses são necessários quando o operando de `sizeof` for um nome de variável ou constante. Lembre-se de que `sizeof` é operador, não uma função e que ele tem seu efeito em tempo de compilação, não em tempo de execução.



Erro comum de programação 8.8

Omitir os parênteses em uma expressão quando o operando é um nome de tipo é um erro de compilação.



Dica de desempenho 82

Como o `of` é um operador unário em tempo de compilação, não um operador ensinado no tempo de execução, utilizar negativamente o desempenho da execução.



Dica de prevenção do erro 8.3

Para evitar erros associados com a omissão dos parênteses em torno de operadores, os programadores colocam cada operador entre parênteses.

## 8.8 Expressões e aritmética de ponteiros

Os ponteiros são operandos válidos em expressões aritméticas, expressões de atribuição e expressões de comparação. Entre todos os operadores normalmente utilizados nessas expressões são válidos com variáveis ponteiro. Esta seção descreve os que podem ter ponteiros como operandos e como esses operadores são utilizados com ponteiros.

um Mário pode ser salvo de um Léo, mas o Léo não pode ser salvo de um Mário (o Léo é dependente de ser subtraído de outro).

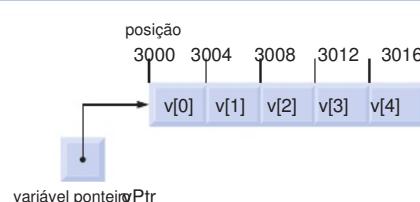
Suponha que `array` tenha sido declarado e que seu primeiro elemento esteja na memória. Suponha que o ponteiro `p` tenha sido inicializado para apontar para o valor de `array[0]`. A Figura 8.18 diagrama essa situação para uma máquina com inteiros de quatro bytes. `p` pode ser inicializado para apontar para qualquer uma das seguintes instruções (porque o nome de um array é equivalente ao endereço de seu primeiro elemento):

```
int *vPtr = v;
int *vPtr = &v[0];
```



Dica de portabilidade 8.4

A maioria dos computadores hoje tem inteiros de dois bytes ou quatro bytes. Algumas máquinas mais novas utilizam inteiros de oito bytes. Como os resultados da aritmética de ponteiros dependem do tamanho dos objetos para os quais um ponteiro aponta, a aritmética de ponteiros é dependente da máquina.



**Figura 8.18** O array `V` e uma variável ponteiro `Ptr` que aponta para

Na aritmética convencional, ao adicionar ou subtrair um inteiro a um ponteiro resulta no valor da posição que o ponteiro aponta no valor. Normalmente não é esse o caso com a aritmética de ponteiros. Quando um inteiro é adicionado a, ou subtraído de, um ponteiro, o ponteiro simplesmente não é incrementado ou decrementado esse inteiro, mas por esse inteiro vezes o tamanho do objeto que o ponteiro referencia. O número de bytes depende do tipo de objeto. Por exemplo, a instrução

```
vPtr += 2;
```

produziria 3008 ( $3000 + 2 * 4$ ) supondo que vPtr armazenado em quatro bytes de memória. No array para [2] (Figura 8.19). Se um inteiro fosse armazenado em dois bytes de memória, então o cálculo precedente resultaria na posição 3004 ( $3000 + 2 * 2$ ). Se o array fosse de um tipo de dados diferente, a instrução precedente incrementaria o ponteiro por duas vezes o número de bytes que ele aceita para armazenar um objeto desse tipo de dados. Ao realizar a aritmética de ponteiros de caracteres, os resultados serão consistentes com a aritmética regular, porque cada caractere tem um byte de comprimento.

Se vPtr tiver sido incrementado duas vezes para apontar para a instrução

```
vPtr -= 4;
```

operaria da mesma forma, subtraindo 8 bytes de memória para o ponteiro. Caso o ponteiro seja incrementado ou decrementado por um, os

```
+ +vPtr;
vPtr++;
```

incrementa o ponteiro para apontar para o próximo elemento do array. Cada uma das instruções

```
--vPtr;
vPtr--;
```

decrementa o ponteiro para apontar para o elemento anterior do array.

As variáveis ponteiro que apontam para o mesmo array podem ser subtraídas para obter a diferença entre endereços. Por exemplo, se 3000 e v2Ptr contiverem o endereço da instrução

```
x = v2Ptr - vPtr;
```

atribuiria a x o número de elementos do array – nesse caso, a aritmética de ponteiros não faz sentido a menos que essa operação seja realizada em um ponteiro que aponta para um array. Não podemos pressupor que duas variáveis do mesmo tipo estejam armazenadas contiguamente na memória a menos que elas sejam elementos adjacentes de um array.



### Erro comum de programação 8.9

Utilizar aritmética de ponteiros em um ponteiro que não referencia um array de valores é um erro de lógica.



### Erro comum de programação 8.10

Subtrair ou comparar dois ponteiros que não referenciam elementos do mesmo array é um erro de lógica.



### Erro comum de programação 8.11

Utilizar aritmética de ponteiros para incrementar ou decrementar um ponteiro de modo que esse ponteiro referencie um elemento depois do fim, ou antes do começo do array, é normalmente um erro de lógica.

Um ponteiro pode ser atribuído a outro ponteiro se ambos forem do mesmo tipo. Caso contrário, um operador de coerção deve ser utilizado para converter o valor do ponteiro à direita da atribuição no tipo de ponteiro à esquerda da atribuição. A exceção a essa regra é o ponteiro `void *`, que é um ponteiro genérico capaz de representar qualquer tipo de ponteiro. Um ponteiro de

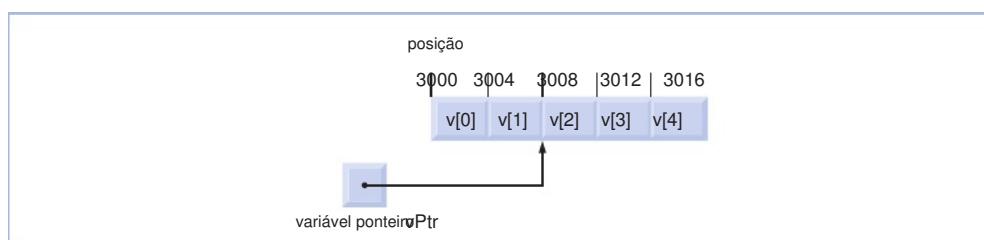


Figura 8.19 Ponteiro vPtr depois da aritmética de ponteiros.

tipo/oid \* sem coerção pode ser atribuído a todos os tipos de ponteiro. Entretanto, o tipo de ponteiro pode ser alterado diretamente a um ponteiro de outro tipo — o ponteiro de tipo original sofrer coerção para o tipo de ponteiro adequado.



### Observação de engenharia de software 8.5

Os argumentos de ponteiro não-constantes podem ser passados para parâmetros de ponteiro constantes. Isso é útil quando o código de um programa utiliza um ponteiro não-constante para acessar dados, mas não quer que os dados sejam modificados por uma chamada de função no corpo do programa.

Um ponteiro `void *` não pode ser desreferenciado. Por exemplo, o compilador ‘sabe’ que o tipo `void *` para bytes de memória em uma máquina com inteiros de quatro bytes, mas não sabe qual é o tipo de dados armazenados na memória para um tipo de dados desconhecido — o número preciso de bytes que o ponteiro referencia e o tipo dos dados não são conhecidos pelo compilador. O compilador deve conhecer o tipo de dados para determinar o número de bytes a ser desreferenciado para um ponteiro particular — para um ponteiro, o número de bytes não pode ser determinado a partir do tipo.



### Erro comum de programação 8.12

Atribuir um ponteiro de um tipo a um ponteiro de outro (`void * para int *`) faz a coerção do primeiro ponteiro para o tipo do segundo ponteiro é um erro de compilação.



### Erro comum de programação 8.13

Todas as operações em ~~um ponteiro~~ erros de compilação, exceto comparar ~~ponteiros~~ outros ponteiros, fazendo coerção dos ~~ponteiros~~ para tipos de ponteiros válidos e atribuindo ~~endereços~~ a ponteiros

Os ponteiros podem ser comparados utilizando operadores de igualdade e operadores relacionais. As comparações que usam operadores relacionais não têm sentido a menos que os ponteiros apontem para membros do mesmo array. As comparações devem comparar os endereços armazenados nos ponteiros. Uma comparação de dois ponteiros que apontam para o mesmo array pode tratar, por exemplo, que um ponteiro aponta para um elemento de número maior alto no array do que o outro ponteiro. Uma utilização da comparação de ponteiro é determinar se um ponteiro é 0 (isto é, o ponteiro é um ponteiro nulo — ele não aponta para nada).

## 8.9 Relacionamento entre ponteiros e arrays

~~Os arrays e os ponteiros estão intimamente relacionados. Os ponteiros podem ser utilizados para manipular os elementos de um array.~~

Suponha as seguintes declarações:

```
int b[5]; // cria array int b de 5 elementos
int *bPtr; // cria ponteiro int bPtr
```

Como o nome de array (sem subscrito) é um ponteiro (constante) para o primeiro elemento do array, podemos configurar o endereço do primeiro elemento como array-instrução

```
bPtr = b; // atribui o endereço de array b ao bPtr
```

Isso é equivalente a aceitar o endereço do primeiro elemento do array como mostrado a seguir:

```
bPtr = &b[0]; // também atribui o endereço do array b ao bPtr
```

O elemento do array pode ser alternativamente referenciado com a expressão de ponteiro

```
*(bPtr + 3)
```

O na expressão precedente é o ponteiro para o ponteiro. Quando o ponteiro aponta para o começo de um array, o deslocamento indica que elemento do array deve ser referenciado e o valor de deslocamento é idêntico ao subscrito de array. A notação anterior é a notação de ponteiro/deslocamento. Os parênteses são necessários, porque é preciso a clareza de precedência de

Sem os parênteses, a expressão `a[ 3 ] + 3` é tratada como `a[ 3 + 3 ]` (isto é, seria adicionado 3 ao subscrito). Assim como o elemento do array pode ser referenciado com uma expressão de ponteiro, o endereço

```
&b[3]
```

pode ser escrito com a expressão de ponteiro

```
bPtr + 3
```

O nome do array pode ser tratado como um ponteiro e utilizado na aritmética de ponteiros. Por exemplo, a expressão

```
*(b + 3)
```

também referencia o elemento do array. Em geral, todas as expressões de array subscritas podem ser escritas com um ponteiro e um deslocamento. Nesse caso, a notação de ponteiro/deslocamento foi utilizada com o nome do array como um ponteiro. Obs a expressão precedente não modifica o nome do array, ele sempre aponta para o primeiro elemento no array.

Os ponteiros podem ser indexados com subscritos exatamente como arrays. Por exemplo, a expressão

`bPtr[ 1 ]`

referencia o elemento do array. A expressão utiliza

Lembre-se de que um nome do array é um ponteiro constante; ele sempre aponta para o começo do array. Portanto, a expr

`b += 3`

causa um erro de compilação, porque tenta modificar o valor do nome do array (uma constante) com a aritmética de ponteiros



#### Erro comum de programação 8.14

Embora os nomes das variáveis ponteiras para arrays sejam ponteiros constantes em expressões aritméticas, eles podem ser modificados em expressões aritméticas.



#### Boa prática de programação 8.2

Por questão de clareza, use notação de array em vez de notação de ponteiro ao manipular arrays.

A Figura 8.20 utiliza as quatro notações discutidas nesta seção para referenciar elementos de um array — a notação de s de array, a notação de ponteiro/deslocamento com o nome de array como um ponteiro, a notação de subscrito de ponteiro e a de ponteiro/deslocamento com um ponteiro — para realizar a mesma tarefa, isto é, imprimir os quatro elementos do array de ii

Para ilustrar ainda mais a intercambiabilidade de arrays e ponteiros, vejamos as opções de copiar strings — programa da Figura 8.21. Ambas as funções copiam uma string em um array de caracteres. Depois de uma comparação dos pro função `copy1` e `copy2` as funções parecem idênticas (por causa da intercambiabilidade de arrays e ponteiros). Essas funções realizam a mesma tarefa, mas são implementadas de modo diferente.

```

1 // Figura 8.20: fig08_20.cpp
2 // Utilizando notações de subscrito e de ponteiro com arrays.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int b[] = { 10 20 30 40}; // cria o array b de 4 elementos
10 int *bPtr = b; // configura bPtr para apontar para o array b
11
12 // gera saída do array b utilizando notação de subscrito de array
13 cout << "Array b printed with:\n\nArray subscript notation\n";
14
15 for (int i = 0; i < 4; i++)
16 cout << "b[" << i << "] = " << b[i] << '\n' ;
17
18 // gera saída do array b utilizando a notação de nome de array e a de ponteiro/deslocamento
19 cout << "\nPointer/offset notation where "
20 <<"the pointer is the array name\n";
21
22 for (int offset1 = 0; offset1 < 4; offset1++)
23 cout << "*(" << b + " << offset1 << ")" = " << *(b + offset1) << '\n' ;
24
25 // gera saída do array b utilizando bPtr e notação de subscrito de array
26 cout << "\nPointer subscript notation\n";
27

```

Figura 8.20 Referenciando elementos do array com o nome do array e com ponteiros.

(continua)

```

28 for (int j = 0; j < 4; j++)
29 cout << "bPtr[" << j << "] = " << bPtr[j] << '\n' ;
30
31 cout << "\nPointer/offset notation\n" ;
32
33 // gera saída do array b utilizando bPtr e notação de ponteiro/deslocamento
34 for (int offset2 = 0; offset2 < 4; offset2++)
35 cout << *(bPtr + offset2) << offset2 << " = "
36 << *(bPtr + offset2) << '\n' ;
37
38 return 0; // indica terminação bem-sucedida
39 } // fim de main

```

Array b printed with:

Array subscript notation

```

b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

```

Pointer/offset notation where the pointer is the array name

```

*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

```

Pointer subscript notation

```

bPtr[0] = 10
bPtr[1] = 20

```

bPtr[2] = 30

Pointer/offset notation

```

*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

```

Figura 8.20 Referenciando elementos do array com o nome do array e com ponteiros.

(continuação)

A função `copy1` (linhas 26–31) utiliza a notação de subscrito de array para copiar arrays de caracteres. A função declara uma variável contadora `i` utilizada como o subscrito de array. O cabeçalho `#include <iostream.h>` realiza toda a operação de cópia — seu corpo é a instrução vazia. O loop incrementado por um a cada iteração do loop. A condição `i < 4` é a condição de parada. Quando o caractere nulo é encontrado, o loop termina, porque o caractere nulo é igual a `'\0'`. Lembre-se de que o valor de uma instrução de atribuição é o valor atribuído a seu operando esquerdo.

Novamente, o cabeçalho `#include <iostream.h>` realiza a operação de cópia entre os ponteiros `s1` e `s2`, sem precisar de nenhuma variável de inicialização. Como na função `copy1`, a condição `*s1 != '\0'` realiza a operação de cópia. O deslocamento `i` é incrementado e o caractere resultante é atribuído ao ponteiro `s2`. Depois da atribuição na condição, o loop incrementa ambos os ponteiros, então eles apontam para o próximo caractere de destino respectivamente. Quando o loop encontra o caractere `\0`, o caractere nulo é atribuído ao ponteiro `s2` e o loop termina. Observe que a ‘parte de incremento’ dessa instrução é duas expressões de incremento separadas por um operador vírgula.

O primeiro argumento da função `copy2` deve ser um array grande o bastante para armazenar a string no segundo argumento. Caso contrário, pode ocorrer um erro quando for feita uma tentativa de escrever em uma posição da memória além dos limites (`c` (lembre-se de que ao utilizar arrays baseados em ponteiro, não há nenhuma verificação de limite ‘integrada’). Além disso, obse-

```

1 // Figura 8.21: fig08_21.cpp
2 // Copiando uma string utilizando a notação de array e a notação de ponteiro.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void copy1(char *, const char *); // protótipo
8 void copy2(char *, const char *); // protótipo
9
10 int main()
11 {
12 char string1[10];
13 char *string2 = "Hello";
14 char string3[10];
15 char string4[] = "Good Bye"
16
17 copy1(string1, string2); // copia string2 para string1
18 cout << "string1 = " << string1 << endl;
19
20 copy2(string3, string4); // copia string4 para string3
21 cout << "string3 = " << string3 << endl;
22 return 0; // indica terminação bem-sucedida
23 } // fim de main
24
25 // copia s2 para s1 utilizando notação de array
26 void copy1(char * s1, const char * s2)
27 {
28 // a cópia ocorre no cabeçalho do for
29 for (int i = 0; (s1[i] = s2[i]) != '\0' ; i++)
30 ; // não faz nada no corpo
31 } // fim da função copy1
32
33 // copia s2 para s1 utilizando notação de ponteiro
34 void copy2(char *s1, const char *s2)
35 {
36 // a cópia ocorre no cabeçalho do for
37 for (; (*s1 = *s2) != '\0' ; s1++, s2++)
38 ; // não faz nada no corpo
39 } // fim da função copy2

```

```

string1 = Hello
string3 = Good Bye

```

Figura 8.21 Cópia de strings utilizando a notação de array e a notação de ponteiro.

o segundo parâmetro de cada função é declarado como ponteiro para um caractere constante — isto é, uma string constante). Em ambas as funções, o segundo argumento é copiado para o primeiro argumento — os caracteres são copiados

~~para o direto para o argumento, mas só os caracteres privados são modificados. Isso significa que se modifica diretamente o array de que o argumento é apontado para a função recebe a permissão de modificar o segundo argumento.~~

## 8.10 Arrays de ponteiros

Os arrays podem conter ponteiros. Uma utilização comum dessa estrutura de dados é formar um array de strings baseadas em referido simplesmente **comodumring**. Toda entrada no array é uma string, mas em C++ uma string é essencialmente um ponteiro para seu primeiro caractere, então cada entrada em um array de strings é simplesmente um ponteiro para o primeiro de uma string. Considere a declaração **string1, string2, string3** que é muito útil na representação de um baralho:

```
const char *suit[4] =
{ "Hearts", "Diamonds", "Clubs", "Spades"};
```

A parte `suit[4]` da declaração indica um array de quatro **elementos**. A declaração indica que cada elemento de `array` é do tipo ‘ponteiro para strings’. Os quatro valores a ser colocados são “Hearts”, “Diamonds”, “Clubs” e “Spades”. Cada um é armazenado na memória como uma string de caracteres terminada por caractere nulo, que é um caractere mais longo que o número de caracteres entre aspas. As quatro strings são os caracteres de tamanho sete, nove, seis e sete (seus caracteres nulos de terminação), respectivamente. Embora pareça que essas strings estão sendo colocadas no array, os ponteiros são realmente armazenados no array, como mostrado na Figura 8.22. Cada ponteiro aponta para o primeiro caractere de sua string correspondente. Portanto, mesmo que a string seja fixa, ele fornece acesso a strings de qualquer comprimento. Essa flexibilidade é um exemplo das poderosas capacidades da estrutura de dados do C++.

As strings de naipe poderiam ser colocadas em um array bidimensional, em que cada linha representa um naipe e cada coluna uma das letras de um nome de naipe. Essa estrutura de dados deve ter um número fixo de colunas por linha, e essa quantidade deve ser tão grande quanto a maior string. Portanto, uma quantidade considerável de memória é desperdiçada quando armazena grande quantidade de strings iguais, quais a maioria é mais curta que a string mais longa. Utilizamos arrays de strings para ajudar a re-

utilizar strings comuns. Os arrays de string são comumente utilizados com **argumentos de linha de comando**, que são passados para a função `main` quando um programa inicia a execução. Esses argumentos seguem o nome de programa quando um programa é executado da linha de comando. Uma utilização típica de argumentos de linha de comando é passar opções para um programa. Por exemplo, a partir da linha de comando em um computador Windows, o usuário pode digitar

```
dir /P
```

para listar o conteúdo do diretório atual e pausar depois de cada tela de informações. O comando `/P` é o argumento passado para o comando `dir` como um argumento de linha de comando. Esses argumentos são colocados em um array de string que é tratado como um argumento. Discutimos os argumentos de linha de comando no Apêndice E, “Tópicos sobre o código C legado”.

## 8.11 Estudo de caso: simulação de embaralhamento e distribuição de cartas

Esta seção utiliza a geração de números aleatórios para desenvolver um programa de simulação de embaralhamento e distribuição de cartas. Esse programa pode então ser utilizado como uma base para implementar programas que reproduzem jogos de cartas e outras aplicações. Para revelar alguns problemas de desempenho sutis, utilizamos intencionalmente algoritmos de embaralhamento e distribuição que são mais lentos que os algoritmos mais eficientes.

Utilizando a abordagem de refinamento passo a passo de cima para baixo, desenvolvemos um programa que irá embaralhar

maiores 52 cartas em sequência, meganícos capazes de destruir o mundo. A abordagem de cima para baixo é particularmente útil para atacar problemas complexos, como este. Utilizamos um array bidimensional para representar o baralho (Figura 8.23). As linhas correspondem aos naipes — a linha 0 corresponde ao naipe de copas, a linha 1 ao de ouros, a linha 2 ao de paus e a linha 3 ao de espadas. As colunas correspondem aos valores de face das cartas — as colunas de 0 a 9 correspondem às faces de ás a 10, respectivamente, e as colunas de 10 a 12 correspondem ao valete, dama e rei, respectivamente. Carregaremos o array de strings de caracteres que representam os quatro naipes (como na Figura 8.22) e o array de strings de caracteres que representam os 13 valores de face.

Esse baralho simulado pode ser embaralhado como mostrado na Figura 8.23. Primeiramente, os naipes (que são `row[0–3]`) e os valores (`column[0–12]`) são escolhidos aleatoriamente. O número 1 é inserido no elemento `array[0][0]` do array `array` para indicar que essa será a primeira carta distribuída do conjunto de cartas embaralhadas. Esse processo continua com os números 2, 3, ..., 52 sendo inseridos aleatoriamente para dirigir as cartas que devem ser colocadas em segundo lugar, terceiro lugar, ..., e 52º lugar no baralho embaralhado. A matriz `array` preenchido com os números de carta, é possível que uma carta seja selecionada duas vezes (o elemento `array[0][column]` será não-zero quando for selecionada). Essa seleção simplesmente é ignorada, e outras cartas são repetidamente escolhidas de forma aleatória até que uma carta não selecionada seja localizada. Por fim, os números 1 a 52 ocuparão os elementos `array[0][0] ... array[0][51]`. Nesse ponto, o baralho está completamente embaralhado.

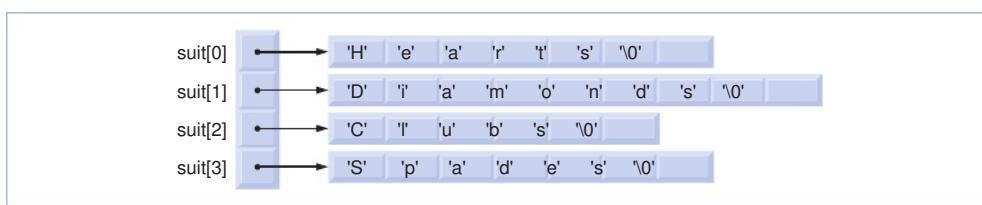


Figura 8.22 Representação gráfica do array `suit`.

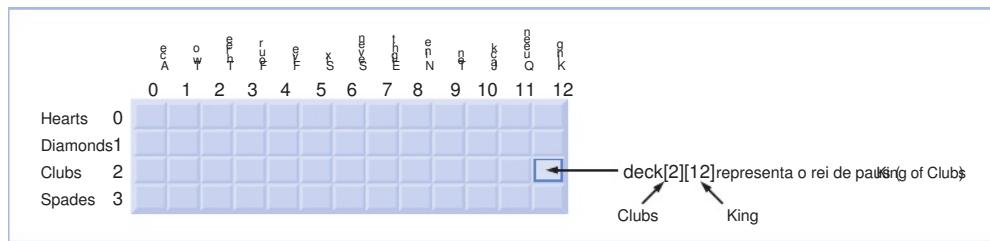


Figura 8.23 A representação de array bidimensional de um baralho.

Esse algoritmo de embaralhamento poderia executar por um período indefinidamente longo se as cartas que já foram embaralhadas fossem repetidamente selecionadas a esmo. Esse fenômeno é conhecido como [adiamento indefinido](#). Nos exercícios, discutimos um algoritmo de embaralhamento melhor que elimina a possibilidade de adiamento indefinido.



### Dica de desempenho 8.3

Às vezes os algoritmos que emergem de modo ‘natural’ podem conter problemas de desempenho sutis como o adiamento indefinido. Busque algoritmos que evitem adiamento indefinido.

Para distribuir a primeira carta, procuramos ~~o array element que corresponde ao slot~~ que é realizado com uma instrução ~~array~~ ~~array~~ de 0 a 51. A que slot esse array corresponde? O array pré-carregado com os quatro naipes, portanto, para obter o naipe, imprimimos. ~~Detalhe: se estivermos~~, para obter o valor de face da carta, imprimimos ~~até que o valor~~ e também a string de caracteres de ". Imprimir essas informações na ordem adequada permite ~~imprimir cada carta~~ imprimir a carta de forma eficiente.

Diamonds[As de ouros] e assim por diante.

Vamos prosseguir com o processo de refinamento passo a passo de cima para baixo. A parte superior é simplesmente

Embaralhe e distribua as 52 cartas

Nosso primeiro refinamento resulta em:

```

Initialize array suit
Initialize o array face
Initialize o array deck
Embaralhe as cartas
Distribua as 52 cartas

```

‘Embaralhe as cartas’ pode ser expandido como mostrado a seguir:

```

Para cada uma das 52 cartas
 Coloque o número da carta no slot de baralho vazio selecionado aleatoriamente

```

‘Distribua as 52 cartas’ pode ser expandido como mostrado a seguir:

```

Para cada uma das 52 cartas
 Localize o número no array deck e imprima a face e o naipe da carta

```

Incorporar essas expansões produz nosso segundo refinamento completo:

```

Initialize o array suit
Initialize o array face
Initialize o array deck

Para cada uma das 52 cartas
 Coloque o número da carta no slot vazio selecionado aleatoriamente
Para cada uma das 52 cartas
 Localize o número no array deck e imprima a face e o naipe da carta

```

‘Coloque o número de carta no slot vazio selecionado aleatoriamente’ pode ser expandido como mostrado a seguir:

```

Escolha o slot aleatoriamente
Enquanto o slot escolhido tiver sido previamente escolhido
 Escolha o slot aleatoriamente
Coloque o número de carta no slot escolhido do baralho

```

'Localize o número de carta no array deck e imprima a face e o naipe da carta' pode ser expandido como mostrado a seguir:

Para cada slot do array deck  
    Se o slot contiver o número da carta  
        Imprima a face e o naipe da carta

Incorporar essas expansões produz nosso terceiro refinamento (Figura 8.24).

Isso completa o processo de refinamento. As figuras 8.25–8.27 contêm o programa de embaralhamento e distribuição de cartas para uma execução de exemplo. As linhas 6–16 (Figura 8.26) implementam as linhas 1–2 da Figura 8.24. O construtor ([linhas 22–35](#) da Figura 8.26) implementa as linhas 1–3 da [Figura 8.24](#). As [linhas 38–55](#) da Figura 8.26) implementam as linhas 5–11 da Figura 8.24. A ([função 58–88](#) da Figura 8.26) implementa as linhas 13–16 da Figura 8.24. Observe a formatação de saída utilizada ([linhas 81–83](#) da Figura 8.26). A instrução de saída gera saída da face alinhada à direita em um campo de cinco caracteres e gera saída do naipe alinhado à esquerda em um campo de oito caracteres (Figura 8.27). I impressa em formato de duas colunas — se a carta cuja saída está sendo gerada estiver na primeira coluna, uma tabulação é gerada.

a síntese de exercícios da cava e fazer mola ligar para desigualdade de ártica. 8.3) na aço quietaria, como a provável é encadada para a esquerda encontrada na primeira tentativa, as tentativas continuam procurando uma correspondência nos elementos restantes de deck Nos exercícios, corrigimos essa deficiência.

- 1 Initialize o array suit
- 2 Initialize o array face
- 3 Initialize o array deck
- 4
- 5 Para cada uma das 52 cartas
- 6     Escolha o slot aleatoriamente
- 7
- 8     Enquanto o slot tiver sido previamente escolhido
- 9         Escolha o slot aleatoriamente
- 10
- 11     Coloque o número de carta no slot escolhido do baralho
- 12 Para cada uma das 52 cartas
- 13     Para cada slot do array deck
- 14         Se o slot contiver o número de carta desejado
- 15             Imprima a face e o naipe da carta
- 16

**Figura 8.24** O algoritmo em pseudocódigo para o programa de embaralhamento e distribuição de cartas.

```
1 // Figura 8.25: DeckOfCards.h
2 // Definição da classe DeckOfCards que
3 // representa um baralho.
4
5 // Definição da classe DeckOfCards
6 class DeckOfCards
7 {
8 public :
9 DeckOfCards(); // construtor inicializa deck
10 void shuffle(); // embaralha as cartas do baralho
11 void deal(); // distribui as cartas do baralho
12 private :
13 int deck[4][13]; // representa o baralho de cartas
14 }; // fim da classe DeckOfCards
```

Figura 8.25 Arquivo de cabeçalho DeckOfCards

```

1 // Figura 8.26: DeckOfCards.cpp
2 // Definições de função-membro para a classe DeckOfCards que simula
3 // o embaralhamento e distribuição de um baralho.
4 #include <iostream>
5 using std::cout;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // protótipos para rand e srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // protótipo para time
17 using std::time;
18
19 #include "DeckOfCards.h" // definição da classe DeckOfCards
20
21 // construtor-padrão DeckOfCards inicializa deck
22 DeckOfCards::DeckOfCards()
23 {
24 // itera pelas linhas do baralho
25 for (int row = 0; row <= 3; row++)
26 {
27 // itera pelas colunas do baralho para linha atual
28 for (int column = 0; column <= 12; column++)
29 {
30 deck[row][column] = 0; // inicializa slot de deck como 0
31 } // fim do for interno
32 } // fim do for externo
33 srand(time(0)); // semeia o gerador de número aleatório
34 } // fim do construtor-padrão DeckOfCards
35
36 // embaralha as cartas do baralho
37 void DeckOfCards::shuffle()
38 {
39 int row; // representa o valor do naipe da carta
40 int column; // representa o valor da face da carta
41
42 // para cada uma das 52 cartas, escolhe um slot aleatoriamente
43 for (int card = 1; card <= 52; card++)
44 {
45 do // escolhe uma nova localização aleatória até um slot vazio ser encontrado
46 {
47 row = rand() % 4; // seleciona a linha aleatoriamente
48 column = rand() % 13; // seleciona a coluna aleatoriamente
49
50 } while (deck[row][column] != 0); // fim da instrução do...while
51 // coloca o número de carta no slot escolhido
52 deck[row][column] = card;
53 } // fim do for
54 } // fim da função shuffle
55 } // fim da função shuffle
56
57 // distribui as cartas do baralho

```

Figura 8.2 Definições de funções-membro para embaralhamento e distribuição.

(continua)

```

58 void DeckOfCards::deal()
59 {
60 // inicializa o array suit
61 static const char *suit[4] =
62 { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64 // inicializa o array face
65 static const char *face[13] =
66 { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67 "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
68
69 // para cada uma das 52 cartas
70 for (int card = 1; card <= 52; card++)
71 {
72 // itera pelas linhas do baralho
73 for (int row = 0; row <= 3; row++)
74 {
75 // itera pelas colunas de baralho para linha atual
76 for (int column = 0; column <= 12; column++)
77 {
78 // se o slot contiver a carta atual, exibe a carta
79 if (deck[row][column] == card)
80 {
81 cout << setw(5) << right << face[column]
82 << " of " << setw(8) << left << suit[row]
83 << (card % 2 == 0 ? '\n' : '\t');
84 } // fim do if
85 } // fim do for mais interno
86 } // fim do for interno
87 } // fim do for externo
88 } // fim da função deal

```

Figura 8.2 Definições de funções-membro para embaralhamento e distribuição.

(continuação)

```

1 // Figura 8.27: fig08_27.cpp
2 // Programa de embaralhamento e distribuição de cartas.
3 #include "DeckOfCards.h" // Definição da classe DeckOfCards
4
5 int main()
6 {
7 DeckOfCards deckOfCards; // cria objeto DeckOfCards
8
9 deckOfCards.shuffle(); // embaralha as cartas
10 deckOfCards.deal(); // distribui as cartas
11 return 0; // indica terminação bem-sucedida
12 } // fim de main

```

|                   |                   |
|-------------------|-------------------|
| Nine of Spades    | Seven of Clubs    |
| Five of Spades    | Eight of Clubs    |
| Queen of Diamonds | Three of Hearts   |
| Jack of Spades    | Five of Diamonds  |
| Jack of Diamonds  | Three of Diamonds |
| Three of Clubs    | Six of Clubs      |
| Ten of Clubs      | Nine of Diamonds  |

Figura 8.2 Programa de embaralhamento e distribuição de cartas.

(continua)

|                  |                   |
|------------------|-------------------|
| Ace of Hearts    | Queen of Hearts   |
| Seven of Spades  | Deuce of Spades   |
| Six of Hearts    | Deuce of Clubs    |
| Ace of Clubs     | Deuce of Diamonds |
| Nine of Hearts   | Seven of Diamonds |
| Six of Spades    | Eight of Diamonds |
| Ten of Spades    | King of Hearts    |
| Four of Clubs    | Ace of Spades     |
| Ten of Hearts    | Four of Spades    |
| Eight of Hearts  | Eight of Spades   |
| Jack of Hearts   | Ten of Diamonds   |
| Four of Diamonds | King of Diamonds  |
| Seven of Hearts  | King of Spades    |
| Queen of Spades  | Four of Hearts    |
| Nine of Clubs    | Six of Diamonds   |
| Deuce of Hearts  | Jack of Clubs     |
| King of Clubs    | Three of Spades   |
| Queen of Clubs   | Five of Clubs     |
| Five of Hearts   | Ace of Diamonds   |

Figura 8.8 Programa de embaralhamento e distribuição de cartas.

(continuação)

## 8.12 Ponteiros de função

Um ponteiro para uma função contém o endereço da função na memória. No Capítulo 7, vimos que o nome de um array é, na realidade, o endereço na memória do primeiro elemento do array. De maneira semelhante, o nome de uma função é o endereço inicial na memória que realiza a tarefa da função. Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros de função.

Classificação por seleção para múltiplos propósitos utilizando ponteiros de função

Para ilustrar o uso de ponteiros para funções, a Figura 8.8 modifica o programa de classificação por seleção da Figura 8.15. A classe `Card` (linhas 7–30) e suas subclasses (`Heart`, `Spade`, `Club` e `Diamond`) permanecem inalteradas (linhas 33–37). A função `Sort` (linhas 38–52) — como um argumento além do array de inteiros a classificar e o tamanho do array — retorna um ponteiro para a função que implementa a ordem da classificação. O programa pede para o usuário escolher se o array deve ser classificado em ordem crescente ou decrescente (linhas 24–26). Se a entrada do usuário for 1, um ponteiro é passado para a função `Sort` (linha 37), fazendo com que o array seja classificado na ordem crescente. Se a entrada do usuário for 0, é passado para a função `Sort` (linha 45), fazendo com que o array seja classificado em ordem decrescente.

O parâmetro a seguir aparece na linha 60 do código: é o endereço de função de

```
bool (*compare)(int, int)
```

Esse parâmetro especifica um ponteiro para uma função que aceita dois inteiros e retorna um valor

O texto `*compare` indica o nome do ponteiro para `indice` (que é um parâmetro de tipo ponteiro).

O tipo `int` indica que a função apontada por `compare` aceita dois argumentos do tipo inteiro. Os parênteses são necessários em `*compare` para indicar que `compare` é um ponteiro para uma função. Se não tivéssemos incluído os parênteses, a declaração teria sido

```
bool *compare(int, int)
```

que declara uma função que recebe dois inteiros como parâmetros e retorna um ponteiro para um valor

O parâmetro correspondente no protótipo da função é

```
bool (*)(int, int)
```

Observe que só foram incluídos tipos. Como sempre, para propósitos de documentação, o programador pode incluir nomes que o compilador irá ignorar.

A função passada para `Sort` é chamada na linha 71 como mostrado a seguir:

```
(*compare)(work[smallestOrLargest], work[index])
```

Assim como um ponteiro para uma variável é desreferenciado para acessar o valor da variável, um ponteiro para uma função é desreferenciado para executar a função. Os parênteses são irrelevantes — se não fossem incluídos, o operador `*` tentaria desreferenciar o valor retornado a partir da chamada de função. A chamada à função poderia ter sido feita sem desreferenciar o ponteiro, como em

compare( work[ smallestOrLargest ], work[ index ] )

que utiliza o ponteiro diretamente como o nome de função. Preferimos o primeiro método de chamar uma função por um ponteiro, ilustra explicitamente que é um ponteiro para uma função que é desreferenciado para chamar a função. O segundo método de chamar uma função por um ponteiro faz pensar que é o nome de uma função real no programa. Isso pode ser confuso para um usuário do programa que gostaria de ver a definição da função que ela não está definida no arquivo.

```

1 // Figura 8.28: fig08_28.cpp
2 // Programa de classificação para múltiplos propósitos usando ponteiros de função.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 // protótipos
12 void selectionSort(int[], const int, bool (*)(int, int));
13 void swap(int * const, int * const);
14 bool ascending(int, int); // implementa ordem crescente
15 bool descending(int, int); // implementa ordem decrescente
16
17 int main()
18 {
19 const int arraySize = 10;
20 int order; // 1 = crescente, 2 = decrescente
21 int counter; // índice do array
22 int a[arraySize] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
23
24 cout << "Enter 1 to sort in ascending order,\n";
25 cin >> order;
26 cout << "\nData items in scrcinal order\n" ;
27
28 // gera saída do array scrcinal
29 for (counter = 0; counter < arraySize; counter++)
30 cout << setw(4) << a[counter];
31
32 // classifica o array em ordem crescente; passa a função ascending
33 // como um argumento para especificar a ordem de classificação ascendente
34 if (order == 1)
35 {
36
37 selectionSort(a, arraySize, ascending);
38 cout << "\nData items in ascending order\n"
39 } // fim do if
40
41 // classifica o array em ordem decrescente; passa a função descending
42 else // como um argumento para especificar a ordem de classificação descendente
43 {
44
45 selectionSort(a, arraySize, descending);
46 cout << "\nData items in descending order\n"
47 } // fim da parte else do if...else
48

```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função.

(continua)

```

49 // gera a saída do array classificado
50 for (counter = 0; counter < arraySize; counter++)
51 cout << setw(4) << a[counter];
52
53 cout << endl;
54 return 0; // indica terminação bem-sucedida
55 } // fim de main
56
57 // classificação por seleção para múltiplos propósitos; o parâmetro compare é um ponteiro para
58 // a função compare que determina a ordem de classificação
59 void selectionSort(int work[], const int size,
60 bool (*compare)(int , int))
61 {
62 int smallestOrLargest;/ índice do menor (ou maior) elemento
63
64 // itera sobre size - 1 elementos
65 for (int i = 0; i < size - 1; i++)
66 {
67 smallestOrLargest = i; // primeiro índice do vetor restante
68
69 // itera para localizar o índice do menor (ou maior) elemento
70 for (int index = i + 1; index < size; index++)
71 if (!(*compare)(work[smallestOrLargest], work[index]))
72 smallestOrLargest = index;
73
74 swap(&work[smallestOrLargest], &work[i]);
75 } // fim do if
76 } // fim da função selectionSort
77
78 // troca os valores nas posições da memória para as quais
79 // element1Ptr e element2Ptr apontem
80 void swap(int * const element1Ptr,int * const element2Ptr)
81 {
82 int hold = *element1Ptr;
83 *element1Ptr = *element2Ptr;
84 *element2Ptr = hold;
85 } // fim da função swap
86
87 // determina se o elemento a é menor que o
88 // elemento b para uma classificação em ordem crescente
89 bool ascending(int a, int b)
90 {
91 return a < b; // retorna true se a for menor que b
92 } // fim da função ascending
93
94 // determina se o elemento a é maior que o
95 // elemento b para uma classificação em ordem decrescente
96 bool descending(int a, int b)
97 {
98 } // fim da função descending

```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função.

(continua)

```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in srcinal order
2 6 4 8 10 12 89 68 45 37
Data items in ascending order
2 4 6 8 10 12 37 45 68 89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in srcinal order

Data items in ascending order
89 68 45 37 12 10 8 6 4 2

```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função.

(continuação)

#### Arrays de ponteiros para funções

Uma utilização de ponteiros de função é em sistemas baseados em menus. Por exemplo, um programa poderia pedir para um selecionar uma opção de um menu inserindo valores de um inteiro. A escolha do usuário pode ser utilizada como um subscrito à array de ponteiros de função e o ponteiro no array pode ser utilizado para chamar a função.

A Figura 8.29 fornece um exemplo mecânico que demonstra como declarar e utilizar um array de ponteiros para funções. O

grama define três funções—function1 e function2— cada uma aceita um argumento de inteiro e não retorna um valor. A linha 17 armazena ponteiros para essas três funções, todas as funções para as quais o array aponta devem ter o mesmo tipo de retorno e os mesmos tipos de parâmetro. A declaração na linha 17 é lida começando no conjunto de parênteses à esquerda com array de três ponteiros para funções. Isso indica que o argumento é vetorQarray é inicializado com os nomes das três funções (que, novamente, são ponteiros). O programa pede para o usuário inserir um número 0 ou 2 ou 8 para terminar. Quando o usuário inserir 0, o valor é utilizado como o subscrito no array de ponteiros para

funções. A linha 29 invoca uma das funções chamadas por índice 1, seleciona o ponteiro na localização array[1] e desreferenciado para chama-la passando como argumento a função. Cada função imprime o valor do seu argumento e seu nome de função para indicar que a função foi chamada corretamente. Nos exercícios, você desenvolverá um sistema baseado em menus. Veremos no Capítulo 13, “Programação orientada a objetos: polimorfismo”, que os arrays de ponteiros para funções são utilizados por desenvolvedores de compiladores para implementar os mecanismos que suportam as funções tecnologia-chave por trás do polimorfismo.

```

1 // Figura 8.29: fig08_29.cpp
2 // Demonstrando um array de ponteiros para funções.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // protótipos de função -- cada função realiza ações semelhantes
9 void function0(int);
10 void function1(int);
11 void function2(int);
12
13 int main()
14 {
15 // inicializa array de 3 ponteiros para funções que
16 // aceitam um argumento int e retornam void

```

Figura 8.29 Array de ponteiros para funções.

(continua)

```

17 void (*f[3])(int) = { function0, function1, function2 };
18
19 int choice;
20
21 cout << "Enter a number between 0 and 2, 3 to end: "
22 cin >> choice;
23
24 // processa escolha do usuário
25 while ((choice >= 0) && (choice < 3))
26 {
27 // invoca a função na localização choice no
28 // array f e passa choice como um argumento
29 (*f[choice])(choice);
30
31 cout << "Enter a number between 0 and 2, 3 to end: "
32 cin >> choice;
33 } // fim do while
34
35 cout << "Program execution completed." < endl;
36 return 0; // indica terminação bem-sucedida
37 } // fim de main
38
39 void function0(int a)
40 {
41 cout << "You entered << a << so function0 was called\n\n";
42 } // fim da função function0
43
44 void function1(int b)
45 {
46 cout << "You entered << b << so function1 was called\n\n";
47 } // fim da função function1
48
49 void function2(int c)
50 {
51 cout << "You entered << c << so function2 was called\n\n";
52 } // fim da função function2

```

Enter a number between 0 and 2, 3 to end: 0  
 You entered 0 so function0 was called

Enter a number between 0 and 2, 3 to end: 1  
 You entered 1 so function1 was called

Enter a number between 0 and 2, 3 to end: 2  
 You entered 2 so function2 was called

Enter a number between 0 and 2, 3 to end: 3  
 Program execution completed.

**Figura 8.13** Array de ponteiros para funções.

(continuação)

### 8.13 Introdução ao processamento de string baseada em ponteiro

Nesta seção, introduzimos algumas funções comuns da C++ Standard Library que facilitam o processamento de string. As técnicas discutidas são apropriadas para desenvolver editores de texto, processadores de texto, software de layout de página, sistemas padronizados de composição e outros tipos de software de processamento de texto. Já vimos que esse é um assunto complexo, mas vamos tentar abordá-lo de forma simples e direta. Vamos ver como os strings podem ser representados por arrays de ponteiros para strings individuais, e como essas strings podem ser manipuladas usando operações de inserção e remoção de caracteres.

GradeBôos capítulos 3–7 representam um nome de curso que **Não é o Capítulo 18** apresenta a classe em detalhes. Embora utilizados normalmente simples e direto, nesta seção, utilizamos strings baseadas em ponteiro terminadas por caractere nulo. Muitas funções da C++ Standard Library operam apenas em strings baseadas em ponteiro por caractere nulo, que são mais complicadas de utilizar. Além disso, se você trabalha com programas C++ legados, pode ser solicitado a manipular essas strings baseadas em ponteiro.

### 8.13.1 Fundamentos de caracteres e strings baseadas em ponteiro

Os caracteres são os blocos de construção fundamentais do código-fonte dos programas C++. Cada programa é composto de uma série de caracteres que — quando agrupados entre si significativamente — é interpretada pelo compilador como uma série de instruções utilizadas para realizar uma tarefa. Um **programa pode conter** um caractere constante é um valor de inteiro representado como caractere entre aspas simples. O valor de uma constante de caractere é o valor inteiro do caractere no conjunto de caracteres da máquina. Por exemplo, a constante 'A' é o valor inteiro 65, que é o código ASCII para o caractere 'A'. Um literal string é um conjunto de caracteres ASCII; ver Apêndice B) e representa o valor inteiro de nova linha (10 no conjunto de caracteres ASCII).

Uma string é uma série de caracteres tratada como uma unidade única. Uma string pode incluir letras, dígitos e vários especiais, como ',', ',' e '\$'. Literais string ou constantes string em C++ são escritos em aspas duplas como mostrado a seguir:

```
"John Q. Doe" (um nome)
"9999 Main Street" (um endereço)
"Maynard, Massachusetts" (uma cidade e um estado)
"(201) 555-1212" (um número de telefone)
```

Uma string baseada em ponteiro em C++ é um array de caracteres que **acabará com caractere nulo**. A string é armazenada na memória. Uma string é acessada via um ponteiro para seu primeiro caractere. O valor de uma string é o endereço de seu primeiro caractere. Portanto, em C++, é apropriado dizer que **é um ponteiro constante**, um ponteiro para o primeiro caractere da string. Nesse sentido, as strings são como arrays, porque um nome de array é também um ponteiro para seu primeiro elemento.

Um literal string pode ser utilizado como um inicializador na declaração de um array de caracteres ou de uma variável de tipo char \*. Cada uma das declarações

```
char color[] = "blue";
const char *colorPtr = "blue";
```

inicializa uma variável com uma string. A primeira declaração cria um array de cinco elementos contendo o caractere 'b', 'l', 'u', 'e' e '\0'. A segunda declaração cria o ponteiro de variável que aponta para a string "blue" (que acaba em '\0') em algum lugar da memória. Os literais string têm a classe de vida constante (até o fim do programa) e

assim, os literais strings em C++ são constantes. Seus caracteres não podem ser modificados. A declaração char[] = "blue"; também poderia ser escrita

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

Ao declarar um array de caracteres para conter uma string, o array deve ser suficientemente grande para armazenar a string e seu caractere nulo. A declaração anterior determina o tamanho do array, com base no número de inicializadores fornecidos na inicializadora.



#### Erro comum de programação 8.15

É um erro não alocar espaço suficiente em um array de caracteres para armazenar o caractere nulo que termina uma string.



#### Erro comum de programação 8.16

Criar ou utilizar uma string no estilo C que não contém um caractere nulo de terminação pode levar a erros de lógica.



#### Dica de prevenção de erro 8.4

**Armazene uma string de caracteres em um array de caracteres** quando o seu comprimento é maior que o comprimento do array. Se para armazenar uma string que é mais longa que o array de caracteres em que ela deve ser armazenada, os caracteres além do final do array sobrescreverão os dados na memória seguinte ao array, resultando em erros de lógica.

Uma string pode ser lida em um array de caracteres utilizando a função `cin`. Por exemplo, a seguinte instrução pode ser utilizada a fim de ler uma string para o array de caracteres

```
cin >> word;
```

A string inserida pelo usuário é armazenada na região precedente lê caracteres até que um caractere de espaço em branco ou indicador de fim do arquivo seja encontrado. Observe que a string não deve ter mais que 19 caracteres para deixar espaço

caractere de terminação nulo. O manipulador `setw(20)` é utilizado para assegurar que o caractere de tamanho do array. Por exemplo, a instrução

```
cin >> setw(20) >> word;
```

especifica que deve ler um máximo de 19 caracteres e salvar na posição no array para armazenar o caractere de terminação nulo para a string. O manipulador se aplica ao próximo valor sendo inserido. Se mais de 19 caracteres forem inseridos, os caracteres restantes não são armazenados e podem ser armazenados em outra variável.

Em alguns casos, é desejável inserir uma linha inteira de texto em um array. Para esse propósito, o C++ fornece a função `getline` no arquivo de cabeçalho `<iostream.h>`. No Capítulo 3 você foi apresentado à função `getline`, que lia a entrada até que um caractere de nova linha fosse inserido e armazenava a entrada (sem o caractere de nova linha) em uma string especificada como um argumento. A função `getline` aceita três argumentos — um array de caracteres em que a linha de texto será armazenada, um comprimento e um caractere delimitador. Por exemplo, o segmento de programa

```
char sentence[80];
```

declaração `getline(sentence, 80, '\n')` e lê uma linha de texto do teclado no array. A função pára de ler caracteres quando o caractere delimitador é encontrado, quando o indicador de fim do arquivo é inserido ou quando o número de caracteres lidos até agora é menor que o comprimento especificado no segundo argumento. (O último caractere no array é reservado para o caractere de tamanho.) Se o caractere delimitador é encontrado, ele é lido e descartado. O terceiro argumento para valor-padrão, então a chamada de função precedente poderia ter sido escrita como mostrado a seguir:

```
cin.getline(sentence, 80);
```

O Capítulo 15, “Entrada/saída de fluxo”, fornece uma discussão detalhada sobre as funções de entrada/saída.



### Erro comum de programação 8.17

Processar um único caractere como argumento pode levar a um erro fatal de tempo de execução. Uma string ponteiro — provavelmente um inteiro bem grande. Entretanto, um caractere é um inteiro pequeno (valores ASCII variam de 0–255). Em muitos sistemas, desreferenciar `char` causa um erro, porque os endereços de memória baixa são reservados para usos especiais como handlers de interrupção de sistemas operacionais — portanto, ocorrem as ‘violações de acesso de memória’.



### Erro comum de programação 8.18

É um erro de compilação passar uma string como um argumento para uma função quando um caractere é esperado.

#### 8.13.2 Funções de manipulação de string da biblioteca de tratamento de strings

A biblioteca de tratamento de strings fornece muitas funções úteis para manipular dados de string, comparar strings, pesquisar strings e outras strings em strings, tokenizar strings (separá-las em partes lógicas como as palavras separadas em uma frase) e seu comprimento. Esta seção apresenta algumas funções de manipulação de strings comuns da biblioteca de tratamento de strings padrão C++. As funções são resumidas na Figura 8.30; em seguida, cada uma delas é utilizada em um exemplo de código. Os protótipos para essas funções são encontrados no arquivo de cabeçalho `<iomanip.h>`.

Observe que várias funções na Figura 8.30 contêm parâmetros. Esse tipo é definido no arquivo de cabeçalho `<iomanip.h>` como um tipo integral sem sinal, como `unsigned long`.



### Erro comum de programação 8.19

Esquecer de incluir o arquivo de cabeçalho `<iomanip.h>` ao usar funções da biblioteca de tratamento de strings causa erros de compilação.

#### Copiando strings `constncpy` e `strncpy`

A função `strcpy` sempre copia a string e seu caractere de terminação para o array. A função `strncpy` copia caracteres que deve ser copiado para o array. Observe que a função `strncpy` não copia necessariamente o caractere de terminação nulo de seu segundo argumento — um caractere de terminação só é escrito se o número de caracteres a ser copiado for pelo menos um maior que o comprimento da string. Por exemplo, se o segundo argumento, um caractere de terminação nulo só é escrito se o terceiro argumento for maior que o comprimento da string (mais um caractere de terminação nulo). Se o terceiro argumento for menor que o comprimento da string, os caracteres que serão acrescentados ao array até que o número total de caracteres especificado pelo terceiro argumento seja escrito.

| Protótipo da função                                      | Descrição da função                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char *strcpy( char *s1, const char *s2 );                | Copia a string para o array de caractere s1. O endereço é retornado.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| char *strncpy( char *s1, const char *s2, size_t n );     | Copia no máximo n caracteres da string s2 para o array de caractere s1. O endereço é retornado.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| char *strcat( char *s1, const char *s2 );                | Acrescenta a string s2 ao final da string s1. O primeiro caractere de s2 sobrescreve o caractere de terminação nulo de s1. O endereço de s1 é retornado.                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| char *strncat( char *s1, const char *s2, size_t n );     | Acrescenta no máximo n caracteres da string s2 à string s1. O primeiro caractere de s2 sobrescreve o caractere de terminação nulo de s1. O endereço é retornado.                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| int strcmp( const char *s1, const char *s2 );            | Compara a string s1 com a string s2. A função retorna um valor zero, menor que zero (normalmente -1) ou maior que zero (normalmente 1) se as strings forem igual, maior ou menor respectivamente.                                                                                                                                                                                                                                                                                                                                                                                                |
| int strncmp( const char *s1, const char *s2, size_t n ); | Comparar n caracteres das strings s1 e s2. A função retorna zero, menor que zero ou maior que zero se a parte de n caracteres de s1 for igual a, maior ou menor que a parte correspondente de s2 respectivamente.                                                                                                                                                                                                                                                                                                                                                                                |
| char *strtok( char *s1, const char *s2 );                | Uma seqüência de chamadas sobre a string ‘tokens’ — partes lógicas como as palavras em uma linha de texto. A string é dividida com base nos caracteres s2. Pode-se, por exemplo, se quisermos quebrar a string “is:a:string” em tokens com base no caractere ‘:’. As partes resultantes seriam “is”, “a” e “string”. Entretanto, a função retorna somente um token por vez. A primeira chamada contém o endereço do primeiro argumento. Cada chamada subsequente que não é a última retorna o endereço do próximo argumento. Cada chamada subsequente que é a última retorna o endereço de NULL. |
| size_t strlen( const char *s );                          | Determina o comprimento da string de caracteres que precedem o caractere de terminação nulo é retornado.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

Figura 8.30 Funções de manipulação de string da biblioteca de tratamento de strings.



### Erro comum de programação 8.20

Ao utilizar `strncpy`, o caractere de terminação nulo do segundo argumento é copiado se o número de caracteres especificado pelo terceiro argumento for maior que o comprimento do segundo argumento. Nesse caso, pode ocorrer um erro fatal se o programador não terminar manualmente a string com um caractere nulo.

A Figura 8.31 utiliza (linha 17) para copiar a string inteira para o array utilizada `strncpy` (linha 23) para copiar os primeiros caracteres do array o array a linha 24 acrescenta um caractere nula porque a chamada a `strncpy` no argumento não escreve um caractere de terminação nulo. (O terceiro argumento é menor que o comprimento de string.)

Concatenando strings com `strcat` e `strncat`

A função `strcat` acrescenta seu segundo argumento (uma string) a seu primeiro argumento (um array de caracteres contendo uma string). O primeiro caractere do segundo argumento substitui o caractere nulo na string no primeiro argumento. O programador deve assegurar que o array utilizado para armazenar a primeira string é suficientemente grande para armazenar a combinação da primeira e da segunda strings e do caractere de terminação nulo (copiado a partir da `strcat` da string). A função `strncat` acrescenta o número especificado de caracteres da segunda string para a primeira string e acrescenta um caractere de terminação nulo ao final da string (linhas 19 e 29) e a função (linha 24).

```

1 // Figura 8.31: fig08_31.cpp
2 // Utilizando strcpy e strncpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos para strcpy e strncpy
8 using std::strcpy;
9 using std::strncpy;
10
11 int main()
12 {
13 char x[] = "Happy Birthday to You"// o comprimento da string é 21
14 char y[25];
15 char z[15];
16
17 strcpy(y, x); // copia conteúdo de x para y
18
19 cout <<"The string in array x is: " << x
20 << "\nThe string in array y is: " << y << "\n" ;
21
22 // copia os primeiros 14 caracteres de x para z
23 strncpy(z, x, 14); // não copia o caractere nulo
24 z[14] = '\0'; // acrescenta '\0' ao conteúdo de z
25
26 cout <<"The string in array z is: " << z << endl;
27 return 0; // indica terminação bem-sucedida
28 } // fim de main

```

The string in array x is: Happy Birthday to You  
 The string in array y is: Happy Birthday to You  
 The string in array z is: Happy Birthday

Figura 8.31 strcpy e strncpy

```

1 // Figura 8.32: fig08_32.cpp
2 // Utilizando strcat e strncat.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos para strcat e strncat
8 using std::strcat;
9 using std::strncat;
10
11 int main()
12 {
13 char s1[20]= "Happy " // comprimento 6
14 char s2[] = "New Year"; // comprimento 9
15 char s3[40] = "";
16
17 cout <<"s1 = " << s1 << "\ns2 = " << s2;
18
19 strcat(s1, s2); // concatena s2 com s1 (comprimento 15)

```

Figura 8.32 strcat e strncat.

(continua)

```
20
21 cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23 // concatena os 6 primeiros caracteres de s1 a s3
24 strcat(s3, s1, 6); // coloca '\0' depois de último caractere
25
26 cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27 << "\ns3 = " << s3;
28
29 strcat(s3, s1); // concatena s1 a s3
30 cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31 << "\ns3 = " << s3 << endl;
32
33 } // fim de main
```

s1 = Happy  
s2 = New Year

After strcat(s1, s2):  
s1 = Happy New Year  
s2 = New Year

After `strncat(s3, s1, 6)`:  
`s1 = Happy New Year`  
`s3 = Happy`

```
After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

Figura 8.32 strcat e strncat.

(continuação)

Comparando strings com strcmp e strncmp

A Figura 8.33 compara três strings utilizando `strcmp` (linhas 21, 22 e 23) e `strcmpi` (linhas 26, 27 e 28). A função `strcmp` compara, caractere por caractere, seu primeiro argumento de string com seu segundo argumento de string. A função retornará zero se as strings forem iguais, um valor negativo se a primeira string for menor que a segunda string e um valor positivo se a primeira string for maior que a segunda. A função é equivalente à função `strcmp` exceto pelo fato de que `strcmpi` compara até um número especificado de caracteres. A função `strcmp` não pode ser usada para comparar caracteres se alcançar o caractere nulo em um de seus argumentos de string. O programa imprime o valor inteiro retornado em cada chamada de função.



Erro comum de programação 8.21

Pressupor que `strcmp` retornam 1 (valor verdadeiro) quando seus argumentos são iguais é um erro de lógica. Ambas as funções retornam zero (valor falso do C++) para igualdade. Portanto, ao testar a igualdade de duas strings, o resultado da função `strcmp` ou `strcasecmp` deve ser comparado com zero para determinar se as strings são iguais.

Para entender exatamente o que significa uma string ser ‘maior que’ ou ‘menor que’ outra string, considere o processo de alfabetização de sobrenomes. O leitor iria, sem dúvida, colocar ‘Jones’ antes de ‘Smith’ porque a primeira letra de ‘Jones’ vem antes da primeira letra de ‘Smith’ no alfabeto. Mas o alfabeto é mais que uma simples lista de 26 letras — é uma lista ordenada de caracteres. Cada letra ocorre em uma posição específica dentro da lista. O ‘z’ é mais que apenas uma letra do alfabeto; ‘z’ é especificamente a última letra do alfabeto.

Como o computador sabe que uma letra vem antes de outra? Todos os caracteres são representados dentro do computador como numéricos; quando o computador compara duas strings, na realidade, ele compara os códigos numéricos dos caracteres nas strings.

Em um esforço de padronizar representações de caractere, a maioria dos fabricantes de computador projetou suas máquinas para utilizar um dos dois esquemas de codificação populares. Lembre-se que ASCII significa 'American Standard Code for Information Interchange'. EBCDIC quer dizer 'Extended Binary Coded Decimal Interchange Code'. Existem outros esquemas de codificação, mas esses dois são os mais populares.

O ASCII e o EBCDIC são chamados de **conjuntos de caracteres**. A maioria dos leitores deste livro utilizará computadores desktop ou notebooks que empregam o conjunto de caracteres ASCII. Os computadores mainframe da IBM utilizam um conjunto de caracteres EBCDIC. Como o uso da Internet e da World Wide Web torna-se intenso no mundo todo, o mais novo conjunto de caracteres Unicode está crescendo rapidamente em popularidade. Para informações adicionais sobre o Unicode, visite [de.org](http://de.org). As manipulações de strings e caracteres, na realidade, envolvem a manipulação dos códigos numéricos apropriados para os caracteres em si. Isso explica a intercambiabilidade entre caracteres e inteiros pequenos no C++. Visto que é significativo dizer que o código numérico é maior que, menor que ou igual a outro código numérico, torna-se possível relacionar vários caracteres ou inteiros entre si para se referir aos códigos de caractere. O Apêndice B contém os códigos de caractere ASCII.



#### Dica de portabilidade 8.5

Os códigos numéricos internos utilizados para representar caracteres podem ser diferentes em computadores diferentes, porque os computadores podem utilizar conjuntos de caracteres diferentes.



#### Dica de portabilidade 8.6

Não teste explicitamente os códigos ASCII `‘A’` ou `‘B’`; em vez disso, use a constante de caractere correspondente, como `‘\n’ == ‘A’`.

[Nota] Com alguns compiladores, as funções `strcmp` e `strncpy` sempre retornam 0, como na saída de exemplo da Figura 8.33. Com outros compiladores, essas funções retornam 0 ou a diferença entre os códigos numéricos dos primeiros caracteres das strings sendo comparadas. Por exemplo, se os primeiros caracteres que diferem entre eles são os primeiros caracteres da segunda palavra (`“Happy”` tem código numérico 70, e `“Year”` tem código numérico 72), respectivamente. Nesse caso, o valor de `strcmp` será comparado com 70.

```

1 // Figura 8.33: fig08_33.cpp
2 // Utilizando strcmp e strncmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // protótipos para strcmp e strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16 char *s1 = "Happy New Year"
17 char *s2 = "Happy New Year"
18 char *s3 = "Happy Holidays"
19
20 cout << s1 << s2 << s3 << endl
21 << "\n\nstrcmp(s1, s2) = " << setw(2) << strcmp(s1, s2)
22 << "\nstrcmp(s1, s3) = " << setw(2) << strcmp(s1, s3)
23 << "\nstrcmp(s3, s1) = " << setw(2) << strcmp(s3, s1);
24
25 cout << "\n\nstrncmp(s1, s3, 6) = " << setw(2)
26 << strncmp(s1, s3, 6) << "\nstrncmp(s1, s3, 7) = " << setw(2)
27 << strncmp(s1, s3, 7) << "\nstrncmp(s3, s1, 7) = " << setw(2)
28 << strncmp(s3, s1, 7) << endl;
29
30 } // fim de main

```

Figura 8.33 strcmp e strncmp

(continua)

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) = 0
strcmp(s1, s3) = 1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) = 1
strncmp(s3, s1, 7) = -1
```

Figura 8.33 strcmp e strncmp

(continuação)

Tokenizando uma string com strtok

A função `tok` divide uma string em tokens. Token é uma sequência de caracteres **separados por** (normalmente espaços ou sinais de pontuação). Por exemplo, em uma linha de texto, cada palavra pode ser considerada um token. Espaços que separam as palavras podem ser considerados delimitadores.

São necessárias múltiplas chamadas para dividir uma string em tokens (supondo que a string contenha mais de um token). A primeira chamada `amadara` também tem dois argumentos, uma string a ser tokenizada e uma string contendo caracteres que separam os tokens (isto é, delimitadores). A linha 19 na Figura 8.34 **aponta para** o primeiro token em `sentence`. O segundo argumento indica que os tokens são separados por espaços. A função `procura` o primeiro caractere na sentença que não seja um caractere delimitador (espaço). Isso no primeiro token. A função então localiza o próximo caractere delimitador na string e o substitui por um caractere terminal. A função (**em uma variável**) `atual` é um ponteiro para o próximo caractere que **se segue a token**, ou seja, em ponteiro para o token atual.

As chamadas subsequentes **contêm** continuam tokenizando o conteúdo NUL como o primeiro argumento (linha 25).

O argumento `ultimo` indica que a chamada deve continuar tokenizando a partir da localização da **última** chamada `strtok`. Observe que `ultimo` mantém essas informações salvas de modo que não fiquem visíveis ao programador. Se nenhum token restar quando a chamada `strtok` retorna NUL, o programa da Figura 8.34 para de tokenizar a string "This is a sentence with 7 tokens". O programa imprime cada token em uma linha separada. A linha 28 gera saída de sentença depois da tokenização. Observe que a string de saída é uma cópia da string, deve ser feita se o programa exigir a string depois das chamadas. A sentença enviada para a saída depois da tokenização, observe que somente a palavra impressa, porque substituiu cada espaço em branco por um caractere nulo ( ) durante o processo de tokenização.

```
1 // Figura 8.34: fig08_34.cpp
2 // Utilizando strtok.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo para strtok
8 using std::strtok;
9
10 int main()
11 {
12 char sentence[] = "This is a sentence with 7 tokens";
13 char *tokenPtr;
14
15 cout << "The string to be tokenized is:\n" << sentence
16 << "\n\nThe tokens are:\n\n";
17 }
```

Figura 8.34 strtok.

(continua)

```

18 // inicia a tokenização da frase
19 tokenPtr = strtok(sentence, " ");
20
21 // continua tokenizando a frase até tokenPtr tornar-se NULL
22 while (tokenPtr != NULL){
23 {
24 cout << tokenPtr << '\n' ;
25 tokenPtr = strtok(NULL,L" "); // obtém o próximo token
26 } // fim do while
27
28 cout << "\nAfter strtok, sentence = " << sentence << endl;
29 return 0; // indica terminação bem-sucedida
30 } // fim de main

```

The string to be tokenized is:  
This is a sentence with 7 tokens

The tokens are:

This  
is  
a  
sentence  
with  
7  
tokens

After strtok, sentence = This

Figura 8.34 strtok.

(continuação)



### Erro comum de programação 8.22

Não perceber que modifica a string tokenizada e, então, tentar utilizar essa string como se ela fosse a string original não modificada é um erro de lógica.

Determinando comprimentos de string

A função `len` aceita uma string como um argumento e retorna o número de caracteres na string — o caractere de terminação não é incluído no comprimento. O comprimento também é o índice do caractere nulo. O programa da Figura 8.35 demonstra `strlen`.

```

1 // Figura 8.35: fig08_35.cpp
2 // Utilizando strlen.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 #include <cstring> // protótipo para strlen
7 using std::strlen;
8
9
10 int main()
11 {

```

Figura 8.35 strlen retorna o comprimento de uma stringar \*.

(continua)

```

12 char *string1 = "abcdefghijklmnopqrstuvwxyz"
13 char *string2 = "four";
14 char *string3 = "Boston";
15
16 cout << "The length of '"
17 << string1 << "' is " << strlen(string1)
18 << endl;
19 << endl;
20 return 0; // indica terminação bem-sucedida
21 } // fim de main

```

The length of "abcdefghijklmnopqrstuvwxyz" is 26

The length of "four" is 4

Figura 8.35 `strlen` retorna o comprimento de uma `string`.

(continuação)

## 8.14 Síntese

Neste capítulo fornecemos uma introdução detalhada sobre ponteiros ou variáveis que contêm endereços de memória como seus valores. Começamos demonstrando como declarar e inicializar ponteiros. Você viu como passar o operador de endereço (`&`) para obter o endereço de uma variável a um ponteiro e o operador de deslocamento (`*`) para obter os dados armazenados na variável indiretamente referenciada por um ponteiro. Discutimos como passar argumentos por referência utilizando tanto os argumentos de ponteiro como os argumentos de referência.

Você aprendeu a utilizar `const` com ponteiros para impor o princípio do menor privilégio. Demonstramos como utilizar ponteiros constantes para dados não constantes, ponteiros não constantes para dados constantes, ponteiros constantes para dados não constantes e ponteiros constantes para dados constantes. Em seguida, utilizamos a classificação por seleção para demonstrar como passar elementos do array individuais por referência. Discutimos o operador utilizado para determinar o tamanho, em bytes, de um tipo de dados durante a compilação de programa.

Ponteiros podem ser usados para manipular expressões aritméticas de memória. Vou explicar como é possível manipular arrays de ponteiros, especificamente, arrays de string (uma string de arrays). Então continuamos discutindo os ponteiros de função, que permitem a programadores passar funções como parâmetros. Concluímos o capítulo com uma discussão sobre as várias funções C++ que manipulam strings baseadas em ponteiro. Você aprendeu as capacidades de processamento de string, por exemplo, copiar strings, tokenizar e determinar seu comprimento.

No próximo capítulo, iniciamos nosso tratamento aprofundado de classes. Você aprenderá sobre o escopo de membros de classe e sobre como manter os objetos em um estado consistente. Também aprenderá a utilizar funções-membro especiais como construtores e destrutores, que executam quando um objeto é criado e destruído, respectivamente.

### Resumo

- Os ponteiros são variáveis que contêm como seus valores endereços de memória de outras variáveis.
- A declaração
 

```
int *ptr;
```

 Declara `ptr` como um ponteiro para uma variável de tipo `int`. O escopo de `ptr` é um ponteiro para `int`, do modo usado aqui em uma declaração, indica que a variável é um ponteiro.
- Há três valores que podem ser utilizados para inicializar um ponteiro de um objeto do mesmo tipo. A inicialização de um ponteiro com a inicialização desse mesmo objeto é a convenção em C++.
- O único inteiro que pode ser atribuído a um ponteiro sem coerção é zero.
- O operador (`&`) retorna o endereço de memória de seu operando.
- O operando do operador de endereço deve ser um nome de variável. O operador de endereço não pode ser aplicado a constantes ou expressões que não retornam uma referência.

- O operador `&` (referido como o operador de indireção (ou desreferenciação), retorna um sinônimo, alias ou apelido para o nome do objeto para o qual seu operando aponta na memória. Isso é chamado de desreferenciar o ponteiro.
- Ao chamar uma função com um argumento que o chamador quer que a função chamada modifique, o endereço do argumento pode ser passado para a função chamada. A função chamada então utiliza o operador `&` para desreferenciar o ponteiro e modificar o valor do argumento na função chamadora.
- Uma função que recebe um endereço como um argumento deve ter um ponteiro como seu parâmetro correspondente.
- O operador `const` permite que o programador informe ao compilador que o valor de uma variável particular não pode ser modificado pelo identificador especificado. Se uma tentativa de modificar o valor é feita, o compilador emite um aviso ou um erro, dependendo do compilador particular.
- Há quatro maneiras de passar um ponteiro para uma função — um ponteiro não constante para dados não constantes, um ponteiro não constante para dados constantes, um ponteiro constante para dados não constantes e um ponteiro constante para dados constantes.
- O valor do nome do array é o endereço de (um ponteiro para) um primeiro elemento do array.
- Para passar um único elemento de um array por referência utilizando ponteiros, passe o endereço do elemento do array específico.
- O C++ fornece o operador `[ ]` para determinar o tamanho de um array (ou de qualquer outro tipo de dados, variável ou constante) em bytes em tempo de compilação.
- Quando aplicado ao nome de um array, o resultado é o número total de bytes no array como um inteiro.
- As operações aritméticas que podem ser realizadas em ponteiros são o incremento (`+=`) e o decremento (`-=`) um inteiro a um ponteiro, somar (`+=`) um inteiro de um ponteiro e subtraír um ponteiro de outro.
- Quando um inteiro é adicionado a ou subtraído de um ponteiro, o ponteiro é incrementado ou decrementado por esse inteiro vezes o tamanho do objeto que o ponteiro referencia.
- Dois ponteiros podem ser atribuídos reciprocamente se ambos forem do mesmo tipo. Caso contrário, uma coerção deve ser utilizada. A expressão `a isso é um ponteiro`, que é um tipo genérico de ponteiro que pode armazenar valores de ponteiro de qualquer tipo. Os ponteiros para void podem ser atribuídos a ponteiros de outros tipos. O ponteiro é convertido para o tipo de ponteiro a um ponteiro de outro tipo somente com uma coerção de tipo explícita.
- As únicas operações válidas envolvendo ponteiros são a comparação de ponteiros, a atribuição de endereços a ponteiros e a coerção de ponteiros para tipos de ponteiros válidos.
- Os ponteiros podem ser comparados utilizando operadores de igualdade e relacionais. As comparações que utilizam operadores relacionais significativas se os ponteiros apontarem para membros do mesmo array.
- Os ponteiros que apontam para arrays podem ser subscritos exatamente como os nomes de array podem.
- Na notação de ponteiro/deslocamento, se o ponteiro apontar para o primeiro elemento do array, o deslocamento é o mesmo que um subscritor de array.
- Todas as expressões de array indexadas com subscritos podem ser escritas com um ponteiro e um deslocamento, utilizando o nome da array como um ponteiro ou um ponteiro separado que aponta para o array.
- Os arrays podem conter ponteiros.
- Um ponteiro para uma função é o endereço em que o código da função reside.
- Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros.
- Uma utilização comum de ponteiros de função é nos assim chamados sistemas baseados em menus. Os ponteiros de função são utilizados para selecionar qual função chamar para um item de menu particular.
- A função `strcpy` copia seu segundo argumento — uma string — para seu primeiro argumento — um array de caracteres. O programador deve assegurar que o array-alvo seja suficientemente grande para armazenar a string e seu caractere de terminação nulo.
- A função `strncpy` é equivalente à `strcpy`, exceto pelo fato de que uma `char *nbytes` especifica o número de caracteres a ser copiado da string para o array. O caractere de terminação nulo só será copiado se o número de caracteres a ser copiado for pelo menos um maior que o comprimento da string.
- A função `strcat` acrescenta seu segundo argumento de string — incluindo o caractere de terminação nulo — ao seu primeiro argumento de string. O primeiro caractere da segunda string substitui o último caractere do array-alvo. O programador deve assegurar que o array-alvo utilizado para armazenar a primeira string é suficientemente grande para armazenar a primeira e a segunda strings.
- A função `strncat` é equivalente à `strcat`, exceto pelo fato de que uma `char *nbytes` acrescenta um número especificado de caracteres da segunda string à primeira string. Um caractere de terminação nulo é acrescentado ao resultado.
- A função `strcmp` compara, caractere por caractere, seu primeiro argumento de string com seu segundo argumento de string. A função retorna zero se as strings forem iguais, um valor negativo se a primeira string for menor que a segunda string, e um valor positivo se a primeira string for maior que a segunda.

- A função `strcmp` é equivalente à `strncmp`, exceto pelo fato de que a `strcmp` compara um número especificado de caracteres. Se o número de caracteres em uma das strings for menor que o número de caracteres especificados, a `strcmp` para comparar os caracteres até o caractere nulo na string mais curta ser encontrado.
- Uma seqüência de chamadas de uma string em tokens que são separados por caracteres contidos em um segundo argumento de string. A primeira chamada especifica a string tokenizada como o primeiro argumento, e as chamadas subsequentes para continuar tokenizam a mesma string especificando o mesmo argumento. A função retorna um ponteiro para o token atual de cada chamada. Se não houver mais tokens quando for chamada, `NULL` é retornado.
- A função `strlen` aceita uma string como um argumento e retorna o número de caracteres na string — o caractere de terminação nulo não é incluído no comprimento da string.

## Terminologia

|                                         |                                                  |                                         |
|-----------------------------------------|--------------------------------------------------|-----------------------------------------|
| &(operador de endereço)                 | EBCDIC (Extended Binary Coded Decimal)           | referenciar elementos de array          |
| * (operador de desreferenciar ponteiro) | change Code)                                     | referenciar um valor indiretamente      |
| operador de indireção)                  | endereço, operador de                            | size_t, tipo                            |
| '\0' (caractere nulo)                   | funçãogetline dein                               | sizeof, operador                        |
| adiamento indefinido                    | inanição                                         | strcat, função do arquivo de cabeçalho  |
| algoritmo de classificação por seleção  | lementar um ponteiro                             | <cstring>                               |
| argumentos da linha de comando          | ndireção                                         | strcmp, função do arquivo de cabeçalho  |
| aritmética de ponteiros                 | intercambialidade de arrays e ponteiros          | <cstring>                               |
| array de ponteiros para funções         | islower, função<ctype>                           | strcpy, função do arquivo de cabeçalho  |
| array de strings                        | modificar endereço armazenado em variável        | <cstring>                               |
| ASCII (American Standard Code for       | ponteiro                                         | string sendo tokenizada                 |
| Information Interchange)                | modificar um ponteiro constante                  | string terminada por caractere nulo     |
| caractere de terminação nulo            | operador de desreferênci(a)                      | strings baseadas em ponteiro            |
| caractere delimitador                   | operador de desreferencia(ponteiro (             | strings de tokenização                  |
| caractere nulo )                        | operador de indireção (                          | strlen, função do arquivo de cabeçalho  |
| caracteres especiais                    | passagem por referência com argumento            | <string>                                |
| chamando funções por referência         | ponteiro                                         | strncat, função do arquivo de cabeçalho |
| <b>códigos de escape</b>                |                                                  |                                         |
| concatenando strings                    | passagem por referência com argumento            | string do arquivo de cabeçalho          |
| constcom parâmetros de função           | ponteiro constante                               | <cstring>                               |
| constante de caractere                  | ponteiro constante para dados constantes         | strcpy, função do arquivo de cabeçalho  |
| constante string                        | ponteiro constante para dados não-constantes     | <cstring>                               |
| cópia de string                         | ponteiro de função                               | strtok, função do arquivo de cabeçalho  |
| copiando strings                        | ponteiro não-constante para dados constantes     | <cstring>                               |
| decrementar um ponteiro                 | ponteiro não-constante para dados não-constantes | <cstring>                               |
| deslocamento para um ponteiro           | ponteiro nulo                                    | token                                   |
| desreferenciar um ponteiro              | ponteiro para uma função                         | toupper, função<ctype>                  |
| desreferenciar um ponteiro              | referência a dados constantes                    |                                         |
| desreferenciar um ponteiro              | referenciar diretamente um valor                 |                                         |

## Exercícios de revisão

- 8.1 Complete cada uma das seguintes sentenças:
- Um ponteiro é uma variável que contém como seu valor o(a) de outra variável.
  - Os três valores que podem ser utilizados para inicializar um ponteiro são , , e .
  - O único inteiro que pode ser atribuído diretamente a um ponteiro é .
- 8.2 Determine se as seguintes sentenças são verdadeiras ou falsas. Se a resposta for falsa, explique por quê.
- O operador de endereço pode ser aplicado somente a constantes e expressões.
  - Um ponteiro que é declarado constante pode ser desreferenciado.
  - Os ponteiros de tipos diferentes nunca podem ser atribuídos um ao outro sem a operação de coerção.
- 8.3 Para cada um dos itens a seguir, escreva instruções C++ que realizam a tarefa especificada. Suponha que números de dupla precisão ponto flutuante sejam armazenados em oito bytes e que o endereço inicial do array esteja na posição 1002500 na memória. Cada parte do exercício deve utilizar os resultados de partes anteriores onde apropriado.

- a) Declare um array de inteiros chamado numbers com 10 elementos, e inicialize os elementos para 28 valores 9.9. Suponha que a constante ~~CONSTANTE~~ é definida como  
 b) Declare um ponteiro que aponta para uma variável do tipo  
 c) Utilize uma instrução para imprimir os elementos subscritos, usando notação de array subscrito. Imprima cada número com uma casa decimal de precisão à direita do ponto de fração decimal.  
 d) Escreva duas instruções separadas que atribuem, cada uma, um valor ao array numbers.  
 e) Utilize uma instrução para imprimir os elementos subscritos, usando a notação de ponteiro/deslocamento com o ponteiro nPtr.  
 f) Utilize uma instrução para imprimir os elementos subscritos, usando a notação de ponteiro/deslocamento com o nome do array como o ponteiro.  
 g) Utilize uma instrução para imprimir os elementos subscritos, usando a notação de ponteiro/subscrito com o ponteiro nPtr.  
 h) Referencie o quarto elemento da matriz, usando a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como o ponteiro, a notação de subscrito de array com o ponteiro/deslocamento com o nome de array como o ponteiro.  
 i) Suponha que a variável ~~numbers~~ aponta para o começo da matriz, qual endereço é referenciado pelo que valor é armazenado nessa localização?  
 j) Suponha que a variável ~~numbers[ 5 ]~~ aponta para o endereço de ~~nptr = 4~~ ser executado? Qual valor é armazenado nessa localização?
- 8.4 Para cada uma das seguintes sentenças, escreva uma única instrução que realiza a tarefa especificada. Suponha que as variáveis d flutuante number1 e number2 foram declaradas e já estão inicializado com 10.0 e 20.0 respectivamente. Suponha que a variável tipo é do tipo char.  
 a) Suponha que os arrays, cada um, ~~char~~ 100 elementos que são inicializados com literais string.  
 a) Declare a variável ~~char~~ 100 um ponteiro para um objeto tipo  
 b) Atribua o endereço da variável ~~char~~ 100 variável ponteiro  
 c) Imprima o valor do objeto apontado por  
 d) Atribua o valor do objeto apontado por ~~char~~ 100  
 e) Imprima o valor ~~char~~ 100  
 f) Imprima o endereço ~~char~~ 100  
 g) Imprima o endereço armazenado no valor impresso é o mesmo de ~~char~~ 100 de  
 h) Copie a string armazenada na variável  
 i) Compare a string com a string e imprima o resultado.  
 j) Acrescente os primeiros 10 caracteres da string  
 k) Determine o comprimento da string e imprima o resultado.  
 l) Atribua a localização do primeiro token de delimitadores de tokens são vírgulas ( )
- 8.5 Realize a tarefa especificada em cada uma das seguintes instruções:  
 a) Escreva o cabeçalho de função para uma função que aceita dois ponteiros para números de dupla precisão com ponto flutuante como parâmetros e que não retorna um valor.  
 b) Escreva o protótipo de função para a função na parte (a).  
 c) Escreva o cabeçalho de função para uma função que aceita um inteiro e aceita como parâmetro o inteiro um ponteiro para a função. A função aceita um parâmetro do tipo inteiro e retorna um inteiro.  
 d) Escreva o protótipo de função para a função na parte (c).  
 e) Escreva duas instruções que inicializam, cada uma, o array de dupla precisão.
- 8.6 Encontre o erro em cada um dos seguintes segmentos de programa. Suponha as seguintes declarações e instruções:
- ```

int *zPtr;           // zPtr irá referenciar o array z
int *aPtr = 0;
void *sPtr = 0;
int number;
int z[ 5 ] = { 1, 2, 3, 4, 5 };

a) ++zPtr;
b) // utiliza ponteiro para obter o primeiro valor de array
   number = zPtr;
c) // atribui o array de 2 elementos (o valor 3) ao número
   number = *zPtr[ 2 ];
d) // imprime todo o array z
   for ( int i = 0; i <= 5; i++ )
       cout << zPtr[ i ] << endl;
e) // atribui o valor apontado por sPtr ao número
   number = *sPtr;
  
```

```

f) ++z;
g) char s[ 10];
   cout << strncpy( s, "hello" , 5 ) << endl;
h) char s[ 12];
   strcpy( s, "Welcome Hořje"
i) if ( strcmp( string1, string2 ) )
   cout <<"The strings are equal"<< endl;

```

- 8.7 O que (se houver algo) é impresso quando cada uma das seguintes instruções é realizada? Se a instrução contiver um erro, descreva o que indica como corrigi-la. Suponha as seguintes declarações de variável:

```

char s1[ 50] = "jack";
char s2[ 50] = "jill" ;
char s3[ 50];

```

- B) ~~cout << strcpy(s3, s2) << endl;~~
 << endl;
c) cout << strlen(s1) + strlen(s2) << endl;
d) cout << strlen(s3) << endl;

Respostas dos exercícios de revisão

- 8.1 a) endereço 0, ~~NUL~~ um endereço c)
 8.2 a) Falsa. O operando do operador de endereçamento de ponteiro endereço não pode ser aplicado a constantes ou expressões que não resultam em referências.
 b) Falsa. Um ponteiro para void não pode ser desreferenciado. Esse ponteiro não tem um tipo que permite ao compilador determinar o número de bytes de memória a desreferenciar e o tipo dos dados para os quais o ponteiro aponta.
 c) Falsa. Qualquer tipo de ponteiro pode ser atribuído a outros tipos, desde que os tipos possam ser convertidos para o tipo que o ponteiro aponta.
 8.3 a) `double numbers[SIZE] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};`
 b) `double *nPtr;`
 c) `cout << fixed << showpoint << setprecision();`
 d) `for (int i = 0; i < SIZE; i++)`
~~cout << numbers[i] << endl;~~
 d) `nPtr = numbers;`
~~nPtr = &numbers[1];~~
 e) `cout << fixed << showpoint << setprecision();`
~~for (int j = 0; j < SIZE; j++)~~
~~cout << *(nPtr + j) << endl;~~
 f) `cout << fixed << showpoint << setprecision();`
~~for (int k = 0; k < SIZE; k++)~~
~~cout << *(numbers + k) << endl;~~
 g) `cout << fixed << showpoint << setprecision();`
~~for (int m = 0; m < SIZE; m++)~~
~~cout << nPtr[m] << endl;~~
 h) `numbers[3]`
~~*(numbers + 3)~~
~~nPtr[3]~~
~~*(nPtr + 3)~~
 i) O endereço ~~1002500 + 8 * 8 = 1002564~~ valor 88.
 j) O endereço ~~numbers[5] 1002500 + 5 * 8 = 1002540~~
 O valor nessa localização é
- 8.4 a) `double *fPtr;`
 b) `fPtr = &number1;`
 c) `cout << "The value of *fPtr is " << *fPtr << endl;`
 d) `number2 = *fPtr;`
 e) `cout << "The value of number2 is " << number2 << endl;`
 f) `cout << "The address of number1 is " << &number1 << endl;`

- g) cout << "The address stored in fPtr is " << fPtr << endl;
 Sim, o valor é o mesmo.
- h) strcpy(s1, s2);
 i) cout << strcmp(s1, s2) = " << strcmp(s1, s2) << endl;
 j) strncat(s1, s2, 10);
 k) cout << strlen(s1) = " << strlen(s1) << endl;
 l) ptr = strtok(s2, " ");
- 8.5 a) void exchange(double *x, double *y)
 b) void exchange(double *, double *);
 c) int evaluate(int x, int (*poly)(int))
 d) int evaluate(int , int (*)(int));
 e) char vowel[] = "AEIOU"
- 8.6 a) Errado! Não foi inicializado. , 'O', 'U', '\0' };
 Correção: Inicialize z para *zPtr = z;
- b) Erro: O ponteiro não é desreferenciado.
 Correção: Altere a instrução para *zPtr;
- c) Errado! zPtr[2] não é um ponteiro e não deve ser desreferenciado.
 Correção: altere z[2] para zPtr[2].
- d) Erro: Referir-se a um elemento de array fora dos limites de array com subscrito de ponteiro.
 Correção: Para impedir isso, mude o operador ~~relacionada~~ para ~~relacionada~~ na instrução
- e) Erro: Desreferenciar um ponteiro void.
 Correção: Para desreferenciar o ponteiro void, ele deve primeiro sofrer coerção para um ponteiro do tipo inteiro. Mude a instrução para number = *static_cast< int >(sPtr);
- f) Erro: Tentar modificar um nome de array com a aritmética de ponteiros.
 Correção: Utilize uma variável ponteiro em vez do nome de array para realizar a aritmética de ponteiros ou utilize subscritos no nome de array para referenciar um elemento específico.
- g) Erro: A função ~~copy~~ não escreve um caractere de terminação nulo se seu segundo argumento é igual ao comprimento da string.
 Correção: Torne o terceiro argumento ~~copy~~ ou atribua as[5] para assegurar que o caractere de terminação nulo
- h) Seja o array de strings grande o bastante para armazenar o caractere de terminação nulo.
 Correção: Declare o array com mais elementos.
- i) Erro: A função ~~strcmp~~ retornará 0 se as strings forem iguais; portanto, a condição falsa na instrução de saída não será executada.
 Correção: Compare explicitamente ~~strcmp~~ para a condição da instrução
- 8.7 a) jill
 b) jack and jill
 c) 8
 d) 13
- ### Exercícios
- 8.8 Determine se as seguintes ~~sentenças~~ são falsas. Se for, explique por quê.
- a) Dois ponteiros que apontam para arrays diferentes não podem ser comparados significativamente.
- b) Como o nome de um array é um ponteiro para o primeiro elemento do array, os nomes de array podem ser manipulados precisamente da mesma maneira que os ponteiros.
- 8.9 Para cada um dos itens a seguir, escreva instruções C++ que realizam a tarefa especificada. Suponha que inteiros sem sinal estejam declarados em dobject.h, que o endereço inicial do elemento é mdp[0] e que mdp[10] é o endereço final de inteiros pares de 2 a 10. Suponha que a constante SIZE é definida como
- a) Declare um ponteiro que aponta para um objeto intoint
- b) Utilize uma instrução para imprimir os elementos de mdp usando notação de array subscrito.
- c) Escreva duas instruções separadas que atribuem o endereço variável ~~cont~~ para mdp[0].
- d) Utilize uma instrução para imprimir os elementos de mdp usando a notação de ponteiro/deslocamento.
- e) Utilize uma instrução para imprimir os elementos de mdp usando a notação de ponteiro/deslocamento com o nome de array como o ponteiro.

- g) Utilize uma instrução para imprimir os elementos do array utilizando subscritos no ponteiro para o array.
- h) Referencie o quinto elemento utilizando a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como o ponteiro, a notação de subscrito de ponteiro e a notação de ponteiro/deslocamento.
- i) Que endereço é referenciado? Que valor é armazenado nessa localização?
- j) Supondo que `aponte para` `int value[4]`, que endereço é referenciado? Que valor é armazenado nessa localização?
- 8.10 Para cada um dos itens seguintes, escreva uma única instrução que realiza a tarefa indicada. Suponha que as variáveis do tipo inteiro `value1` e `value2` tenham sido declaradas e tenha sido inicializada `20000`.
- Declare a variável `ponteiro` como um ponteiro para um objeto do tipo `int`.
 - Atribua o endereço da variável `value1` à variável `ponteiro`.
 - Imprima o valor do objeto apontado por `ponteiro`.
 - Atribua o valor do objeto apontado por `ponteiro` à variável `value2`.
- 8.11
- f)** Imprima o endereço de `value1`.
 - g)** Imprima o endereço armazenado em `value2`. O valor impresso é o mesmo que o endereço de
- Realize a tarefa especificada por cada uma das seguintes instruções:
- Escreva o cabeçalho de função para que aceite um parâmetro `long` do tipo inteiro long e que não retorna um valor.
 - Escreva o protótipo de função para a função na parte (a).
 - Escreva o cabeçalho de função para que aceite um parâmetro `long` do tipo inteiro e que retorna um inteiro.
 - Escreva o protótipo de função para a função descrita na parte (c).
- Nota: Os exercícios 8.12 a 8.15 são razoavelmente desafiadores. Uma vez que tiver resolvido esses problemas, você deve ser capaz de implementar muitos jogos de cartas populares.
- 8.12 Modifique o programa na Figura 8.27 de modo que a função de distribuição de cartas distribua uma mão de pôquer de cinco cartas. Encontre as funções para realizar cada uma das seguintes tarefas:
- Determine se a mão contém um par.
 - Determine se a mão contém dois pares.
 - Determine se a mão contém uma trinca (por exemplo, três valetes).
 - Determine se a mão contém uma quadra (por exemplo, quatro ases).
 - Determine se a mão contém cinco cartas de valores consecutivos (é, cinco cartas de valores consecutivos).
- 8.13 Utilize as funções desenvolvidas no Exercício 8.12 para escrever um programa que distribui duas mãos de pôquer de cinco cartas, avalia cada mão e determina qual é a melhor mão.
- 8.14 Modifique o programa desenvolvido no Exercício 8.13 de modo que você possa simular o carteador. A mão de cinco cartas do carteador é distribuída ‘no escuro’, então o jogador não pode vê-la. O programa deve então avaliar a mão do carteador e, com base na qualidade da mão, o carteador deve distribuir uma, duas ou três mais cartas para substituir o número correspondente de cartas desnecessárias na mão. O programa então deve reavaliar a mão. [Este é um problema difícil.]
- 8.15 Modifique o programa desenvolvido no Exercício 8.14 de modo que ele trate a mão do carteador, mas permita ao jogador decidir quais cartas ele quer substituir. O programa então deve avaliar ambas as mãos e determinar quem ganha. Agora utilize esse novo programa para disputar 20 jogos contra o computador. Quem ganha mais jogos, você ou o computador? Faça um de seus amigos disputar 20 jogos contra o computador. Quem ganha mais jogos? Com base nos resultados desses jogos, faça modificações apropriadas para refinar o programa de jogar de pôquer. [Este também é um problema difícil.] Dispute mais 20 jogos. Seu programa modificado reproduziu um jogo melhor?
- 8.16 No programa de embaralhamento e distribuição de cartas das figuras 8.25–8.27, utilizamos intencionalmente um algoritmo inefficiente de embaralhamento que introduziu a possibilidade de adiamento indefinido. Nesse problema, você criará um algoritmo de embaralhamento de alto desempenho que evita o adiamento indefinido.
- Modifique as figuras 8.25–8.27 como mostrado a seguir. Utilize a função `shuffle` na Figura 8.36. Modifique a função `shuffle` para iterar linha por linha e coluna por coluna pelo array, tocando uma vez em cada elemento. Cada elemento deve ser trocado por um elemento do array aleatoriamente selecionado. Imprima o array resultante para determinar se o baralho é embaralhado satisfatoriamente (como na Figura 8.37, por exemplo). Você pode querer que o programa volte para a saída para garantir um embaralhamento satisfatório.
- Observe que, embora a abordagem nesse problema melhore o algoritmo de embaralhamento, o algoritmo de distribuição de cartas ainda requer pesquisar de forma linear para encontrar a carta 1, depois a carta 2, a carta 3 e assim por diante. Pior ainda, mesmo depois de o

Array deck não embaralhado													
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Figura 8.36 Array deck não embaralhado.

Exemplo do array deckembaralhado													
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Figura 8.37 Exemplo do array deckembaralhado.

algoritmo de distribuição de cartas localizar e distribuir a carta, ele continua pesquisando pelo restante do baralho. Modifique o programa das figuras 8.25–8.27 de modo que, uma vez que uma carta é distribuída, nenhuma tentativa adicional de localizar esse número deve ser feita, e o programa imediatamente prossiga com a distribuição da próxima carta.

- 8.17 (Simulação: A tartaruga e a lebre) No exercício, você recriará a clássica corrida da tartaruga e da lebre. Você utilizará geração de números aleatórios para desenvolver uma simulação desse memorável evento.

Nossos competidores começam a corrida no ‘quadrado 1’ de 70 quadrados. Cada quadrado representa uma possível posição ao longo do percurso da competição. A linha de chegada está no quadrado 70. O primeiro competidor a alcançar ou passar o quadrado 70 é compensado com um cesto de cenouras frescas e alface. O percurso envolve subir uma montanha escorregadia, então ocasionalmente os competidores perdem terreno.

Há um relógio que toca uma vez por segundo. A cada tique do relógio, seu programa deve ajustar a posição dos animais de acordo com as regras na Figura 8.38.

Animal	Tipodemovimento	Porcentagendotempo	Movimentoreal
Tartaruga	Caminhadarápida	50%	3 quadrados à direita
	Escorregão	20%	6 quadrados à esquerda
	Caminhadalentamente	30%	1 quadrado à direita
Lebre	Parada	20%	Nenhum movimento
	Saltogrande	20%	9 quadrados à direita
	Escorregão grande	10%	12 quadrados à esquerda
	Salto pequeno	30%	1 quadrado à direita
	Escorregão pequeno	20%	2 quadrados à esquerda

Figura 8.38 Regras para mover a tartaruga e a lebre.

Utilize variáveis para monitorar a posição dos animais (isto é, os números de posição são 1–70). Inicie cada animal na posição 1 (isto é ‘partida’). Se um animal escorrega para a esquerda antes do quadrado 1, move o animal de volta para quadrado 1.

Gere as porcentagens da tabela precedente produzindo intervalos de tempo para tartaruga, realize uma ‘caminhada rápida’ quando um ‘escorregão’ quando uma ‘caminhada lenta’ quando use uma técnica semelhante para mover a lebre.

Inicie a corrida imprimindo

```
BANG !!!!!
AND THEY'RE OFF !!!!!
[BANG !!!!!
E LÁ SE VÃO ELES !!!!!]
```

Para cada tique do relógio (isto é, cada repetição de um loop), imprima uma linha de 70 posições para mostrar a letra

tartaruga [H] e lebre [L]. Caso alguma dessas posições das casas tenha o valor 0, nesse caso, a tartaruga morde a (no caso de um empate) devem estar em branco.

Depois de imprimir cada linha, teste se algum animal alcançou ou passou o quadrado 70. Em caso positivo, imprima o vencedor e termine a simulação. Se a tartaruga ganhar, **TARTARUGA WINS!!! YAH!** [TARTARUGA GANHOU!!! ÉHH!!!]. Se a lebre ganhar, imprima **wins. Yuch!** [A Lebre ganhou. Uhh.]. Se ambos tiverem a mesma marcação no relógio, você pode querer favorecer a tartaruga (a ‘coitadinha’) ou querer imprimir [Empate]. Se nenhum dos animais ganhar, realize o loop novamente para simular o próximo tique do relógio. Quando você estiver pronto para executar seu programa, monte um grupo de fãs para observar a corrida. Você ficará surpreso com o envolvimento da sua audiência!

Seção especial: construindo seu próprio computador

Nos próximos problemas, nós nos desviamos temporariamente do mundo da linguagem de programação de alto nível. Vamos ‘abrir’ um computador e examinar sua estrutura interna. Introduzimos programação de linguagem de máquina e escrevemos vários programas de linguagem de máquina. Para tornar essa uma experiência especialmente valiosa, construímos em seguida um computador (utilizando a simulação baseada em software) em que você pode executar seus programas de linguagem de máquina!

- 8.18 (Programação de linguagem de máquina) Construir um computador que chamaremos de Simpletron. Como seu nome implica, é uma máquina simples, mas como logo veremos também é uma máquina poderosa. O Simpletron executa programas escritos na única linguagem que ele entende diretamente, isto é, a Simpletron Machine Language ou, abreviadamente, SML.

O Simpletron contém um registrador especial em que as informações são colocadas antes de o Simpletron utilizá-las em cálculos ou examiná-las de várias maneiras. Todas as informações no Simpletron são palavras em termos de um número decimal de quatro dígitos 0000, 0001, 0002, 0003, 0004, 0005, 0006, 0007, 00001 etc. O Simpletron é equipado com uma memória de 100 palavras e essas palavras são referenciadas por 100 números de posição.

Antes de executar um programa de SML, é necessário carregar o programa na memória. A primeira instrução de cada programa de SML sempre é colocada no endereço 0000. O computador começará a executar nessa posição.

Cada instrução escrita em SML ocupa uma palavra da memória do Simpletron; portanto, as instruções são números decimais de quatro dígitos com sinal. Suponha que o sinal de uma instrução de SML seja sempre positivo, mas o sinal de uma palavra de dados pode ser positivo ou negativo. Cada localização na memória de Simpletron pode conter uma instrução, um valor de dados utilizado por um programa ou uma área da memória não-utilizada (e portanto indefinida). Os primeiros dois dígitos de cada instrução do SML são o de operação e especifica a operação a ser realizada. Os códigos de operação SML são mostrados na Figura 8.39.

Os últimos dois dígitos de uma instrução de SML são o endereço da posição da memória contendo a palavra à qual a operação se aplica.

Agora vamos considerar dois programas SML simples. O primeiro programa SML (Figura 8.40) lê dois números do teclado e calcula e imprime sua soma. A instrução 0001 lê o primeiro número do teclado e o coloca (copia) na posição inicializada como zero.

A instrução 0002 lê o próximo número na posição 0001, adiciona +2007 e coloca (copia) o primeiro número no acumulador; e

a instrução 0003 adiciona o segundo número ao número total. As operações aritméticas da SML deixam seus resultados no acumulador. A instrução 0010 coloca (copia) o resultado de volta na posição de memória write+1109 pega o número e o imprime (como um número decimal de quatro dígitos com sinal) na instrução

O programa SML na Figura 8.41 lê dois números a partir do teclado, então determina e imprime o maior valor. Note o uso da instrução +4107 como uma transferência condicional de controle, muito parecido com a instrução

Código de operação	Significado
Operações de entrada/saída	
const int READ 10;	Lê uma palavra do teclado para uma posição específica da memória.
const int WRITE 11;	Escreve na tela uma palavra de uma posição específica da memória.
Operações de carregamento e armazenamento	
const int LOAD 20;	Carrega uma palavra de uma posição específica na memória para o acumulador.
const int STORE 21;	Armazena uma palavra do acumulador para uma posição específica na memória.
Operações aritméticas	
const int ADD 30;	Adiciona uma palavra de uma posição específica na memória à palavra no acumulador (deixa o resultado no acumulador).
const int SUBTRACT 31;	Subtrai uma palavra de uma posição específica na memória da palavra no acumulador (deixa o resultado no acumulador).
const int DIVIDE 32;	Divide uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
const int MULTIPLY 33;	Multiplica uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
Operações de transferência de controle	
const int BRANCH 40;	Desvia para uma posição específica na memória.
const int BRANCHNEG 41;	Desvia para uma posição específica na memória se o acumulador for negativo.
const int BRANCHZERO 42;	Desvia para uma posição específica na memória se o acumulador for zero.
const int HALT 43;	Suspende — o programa completou sua tarefa.

Figura 8.39 Códigos de operação de Simpletron Machine Language (SML).

Posição	Número	Instrução
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Figura 8.40 Exemplo de SML 1.

Posição	Número	Instrução
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Desvio negativo para 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 8.41 Exemplo de SML 2.

Agora escreva programas de SML para realizar cada uma das seguintes tarefas:

- Utilize um loop controlado por sentinela para ler números positivos e calcular e imprimir sua soma. Termine a entrada quando um número negativo for inserido.
- Utilize um loop controlado por contador para ler sete números, alguns negativos e alguns positivos e compute e imprima sua média.
- Leia uma série de números, e determine e imprima o maior número. O primeiro número lido indica quantos números devem ser processados.

8.19 (Simulador de computador) À primeira vista, pode parecer pretensioso, mas nesse problema você construirá seu próprio computador. Não, você não irá soldar componentes. Em vez disso, você utilizará uma base técnica para construir um modelo de software do Simpletron. Você não se decepcionará. Seu Simpletron Simulator transformará o computador que você está utilizando em um Simpletron, e você realmente será capaz de executar, testar e depurar os programas de SML escritos no Exercício 8.18.

Quando você executar seu simulador Simpletron, ele deve começar imprimindo

```
*** Bem vindo ao Simpletron!
*** Por favor insira uma instrução
*** (ou data word) por vez em seu programa. Eu vou digitar ***
*** o número de alocação e o ponto de interrogação (?). ***
*** Então você digita a palavra para a alocação. ***
*** Digite o número -99999 para parar de inserir dados ***
*** no seu programa.
```

Seu programa deve simular a memória do Simpletron com um de ~~memória 100 slots~~ 100 slots. Agora assuma que o simulador está executando e vamos examinar o diálogo à medida que inserirmos o programa do Exemplo 2 do Exercício 8.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins ***
```

Observe que os números à direita da barra vertical anterior representam as instruções do programa SML inseridos pelo usuário.

O programa de SML agora foi colocado (ou carregado) na memória do Simpletron e o Simpletron executa seu programa SML. A execução inicia com a instrução `nextOp`, continua sequencialmente, a menos que dirigido para alguma outra parte do programa por uma transferência de controle.

Utilize a variável `accumulator` para representar o registrador acumulador. Utilize a variável `counter` para monitorar a posição na memória que contém a instrução sendo realizada. Utilize a variável `operationCode` para indicar a operação que está sendo atualmente realizada (isto é, os dois dígitos esquerdos da palavra da instrução). Utilize a variável `operand` para armazenar a posição da memória em que a instrução atual opera. `operationCode` os dois dígitos mais à direita da instrução sendo atualmente realizada. Não execute instruções diretamente de memória. Mais precisamente, transfira a próxima instrução que será realizada da memória para uma variável `instructionRegister`. Então 'pegue' os dois dígitos esquerdos de `instructionRegister` e 'pegue' os dois dígitos direitos e os coloque em `operationCode`. Quando o Simpletron começa a executar, todos os registradores especiais são inicializados como zero.

Agora vamos 'percorrer' a execução da primeira instrução. A posição de memória é chamada de `counter`.

O `counter` informa a posição da próxima instrução que será realizada. Realizaremos essa posição a partir da memória utilizando a instrução C++:

```
instructionRegister = memory[ counter ];
O código de operação e o operando são extraídos do registrador de instrução pelas instruções
    operationCode = instructionRegister / 100
    operand = instructionRegister % 100
```

Agora o Simpletron deve determinar que o código de operação (ver seção 8.1.1) é switch. Ele diferencia entre as 12 operações de SML.

Na instrução `switch`, o comportamento de várias instruções SML é simulado como mostrado na Figura 8.42 (deixamos os outros para o leitor).

A instrução `halt` também faz com que o Simpletron imprima o nome e o conteúdo de cada registrador, bem como o conteúdo completo da memória. Esse tipo de impressão é freqüentemente chamado de dump. Isso ajuda-o a programar sua função de dump, um formato de dump de exemplo é mostrado na Figura 8.43. Observe que um dump, depois de executar um programa Simpletron, mostra os valores reais das instruções e os valores dos dados no momento em que a execução terminasse. Para formatar os números seu sinal como mostrado no dump, utilize o manipulador `setw`. Para números que tenham menos de quatro dígitos, você pode formatar números com zeros à esquerda entre o sinal e o valor utilizando a seguinte instrução antes de gerar a saída do valor:

```
cout << setfill( '0' ) << internal;
```

O manipulador de fluxo `setfill` especifica o caractere de preenchimento que irá aparecer entre o sinal e o valor quando um número for exibido com uma largura de campo de cinco caracteres mas sem quatro dígitos. (Uma largura de campo é reservada para o sinal.) O manipulador `internal` faz os caracteres de preenchimento aparecerem entre o sinal e o valor numérico.

Vamos prosseguir com a execução da primeira instrução. De acordo com o que indicamos, a instrução `switch` simula isso executando a instrução C++:

```
cin >> memory[ operand ];
```

Um ponto de interrogação (`?`) é exibido na tela antes de a instrução para solicitar a entrada ao usuário. O Simpletron espera o usuário digitar um valor e pressionar a tecla `Enter`. O valor digitado é lido na posição

```
read:      cin >> memory[ operand ];
load:     accumulator = memory[ operand ];
add:      accumulator += memory[ operand ];
branch:   Discutiremos as instruções de desvio brevemente.
halt:     Essa instrução imprime a mensagem
           *** Simpletron execution terminated ***
```

Figura 8.42 Comportamento de instruções SML.

```

REGISTERS:
accumulator      +0000
counter          00
instructionRegister +0000
operationCode     00
operand           00

MEMORY:
  0   1   2   3   4   5   6   7   8   9
0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

```

Figura 8.43 Um dump de exemplo.

Neste ponto, a simulação da primeira instrução é concluída. Tudo o que resta é preparar o Simpletron para executar a próxima instrução que acabamos de realizar não era uma transferência de controle, portanto precisamos meramente incrementar os registradores de contadores de instruções como mostrado a seguir:

```
++counter;
```

Isso completa a execução simulada da primeira instrução. O processo inteiro (isto é, o ciclo de execução de instrução) começa de novo com a busca da próxima instrução a executar.

Agora vamos considerar como simular as instruções de desvio (isto é, as transferências de controle). Tudo o que precisamos fazer é ajustar o valor no contador de instrução apropriadamente. Portanto, a instrução de desvio condicional

```
counter = operand;
```

A instrução condicional 'desvie se acumulador for zero' é simulada como

```
if ( accumulator ==0 )
    counter = operand;
```

Nesse ponto, você deve implementar seu Simpletron Simulator e executar cada um dos programas em SML que você escreveu no Exercício 8.18. Você pode sofisticar a SML com recursos adicionais e adaptá-los ao seu simulador.

Seu simulador deve verificar vários tipos de erros. Durante a fase de carregamento do programa, por exemplo, cada número que usuário digita no Simpletron deve estar no intervalo 0–9999. Seu simulador deve utilizar loop para testar se cada número inserido está nesse intervalo e, se não tiver, continuar pedindo para ao usuário tentar novamente o número até inserir um número correto.

Durante a fase de execução, seu simulador deve verificar vários erros sérios, como tentativas de divisão por zero, tentativas de execução de códigos de operação inválidos, estouros de acumulador (isto é, operações aritméticas resultando em valores maiores que 9999) e assim por diante. Os erros sérios são fatal. Se um erro fatal é detectado, seu simulador deve imprimir uma mensagem de erro como

*** Tentou dividir por zero ***

*** A execução do Simpletron foi interrompida ***

e deve imprimir um dump de computador completo no formato discutido previamente. Isso ajudará o usuário localizar o erro no programa.

Mais exercícios sobre ponteiros

- 8.20 Modifique o programa de embaralhamento e distribuição de cartas das figuras 8.25–8.27 para que as operações de embaralhamento e distribuição sejam realizadas pela `shuffleAllCards()`. A função deve conter uma instrução de loop aninhada que seja semelhante à `single()` na Figura 8.26.

8.21 O que esse programa faz?

```

1 // Ex. 8.21: ex08_21.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 void mystery1(char *, const char *); // protótipo
9
10 int main()
11 {
12     char string1[ 80];
13     char string2[ 80];
14
15     cout << "Enter two strings: ";
16     cin >> string1 >> string2;
17     mystery1( string1, string2 );
18     cout << string1 << endl;
19     return 0; // indica terminação bem-sucedida
20 } // fim de main
21
22 // O que essa função faz?
23 void mystery1(char *s1, const char *s2 )
24 {
25     while (*s1 != '\0' )
26         ++s1;
27
28     for ( ; *s1 = *s2; s1++, s2++)
29     ;
30 } // fim da função mystery1

```

8.22 O que esse programa faz?

```

1 // Ex. 8.22: ex08_22.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int mystery2(const char *); // protótipo
9
10 int main()
11 {
12     char string1[ 80];
13
14     cout << "Enter a string: ";
15
16     cin >> string1;
17     mystery2( string1 ) << endl;
18 } // fim de main
19
20 // O que essa função faz?
21 int mystery2(const char *s )
22 {

```

```

23     int x;
24
25     for ( x = 0; *s != '\0' ; s++ )
26         ++x;
27
28     return x;
29 } // fim da função mystery2

```

8.23 Localize o erro em cada um dos seguintes segmentos de código. Se for possível corrigir o erro, explique como.

- a) `int *number;`
`cout << number << endl;`
- b) `double *realPtr;`
`long *integerPtr;`
`integerPtr = realPtr;`
- c) `int * x, y;`
`x = y;`
- d) `char s[] = "this is a character array" ;`
`for (; *s != '\0' ; s++)`
 `cout << *s << ' ' ;`
- e) `short *numPtr, result;`
`void *genericPtr = numPtr;`
`result = *genericPtr + 7;`
- f) `double x = 19.34`
`double xPtr = &x;`
`cout << xPtr << endl;`
- g) `char *s;`
`cout << s << endl;`

8.24 (Quicksort – Técnica de classificação) Vimos previamente as técnicas de classificação por seleção.

Agora apresentamos a técnica de classificação recursiva chamada Quicksort. O algoritmo básico para um array de um único subscrito de valores é como segue:

- a) Passo de particionamento: **Escolha** o primeiro elemento do array não-classificado e determine sua localização final no array classificado (isto é, todos os valores à esquerda do elemento no array são menores que o elemento e todos os valores à direita do elemento no array são maiores que o elemento). Agora temos um elemento em sua posição adequada e dois subarrays não-classificados.
- b) Passo recursivo: **Realize** Passo a) em cada subarray não-classificado.

Toda vez que Passo a) é realizado em um subarray, outro elemento é colocado em sua posição final no array classificado, e dois subarrays não-classificados são criados. Quando um subarray consiste em um elemento, esse subarray deve ser classificado; portanto, o elemento está em sua localização final.

O algoritmo básico parece suficientemente simples, mas como determinamos a posição final do primeiro elemento de cada subarray? Como um exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de particionamento — ele será colocado em sua localização final no array classificado):

37 2 6 4 89 8 10 12 68 45

- a) Iniciando do elemento mais à direita do array, compare o elemento com o elemento que é encontrado. Então troque esse elemento com o elemento que é encontrado. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 89 8 10 37 68 45

O elemento 12 está em itálico para indicar que acabou de ser permutado com 37.

- b) Iniciando à esquerda do array, mas começando com o elemento depois de 37, compare o elemento com o maior que é encontrado. Então troque esse elemento com o elemento que é encontrado. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 37 8 10 89 68 45

- c) Iniciando à direita, mas começando com o elemento antes de 89, compare cada elemento com o maior que seja localizado. Então troque esse elemento com o elemento que é encontrado. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 10 8 37 89 68 45

- d) Iniciando à esquerda, mas começando com o elemento depois de 107 até que o elemento que seja localizado. Então esse elemento. Não há mais elementos para comparar e podemos mesmo sabermos que foi colocado na sua localização final do array classificado.

Depois que o particionamento foi aplicado ao array, restam dois subarrays não classificados. O subarray com valores menores que contém 12, 2, 6, 4, 10 e 8. O subarray com valores maiores que 37 contém 89, 68 e 45. A classificação continua com ambos os subarrays sendo particionados da mesma maneira que o array `srcinal`.

Com base na discussão precedente, escreva `quickSortpara` para classificar um array de inteiros de um único subscrito. A função deve receber como argumentos um array de inteiros, um subscrito inicial `partition`, um subscrito final. A função chamada `partition` para realizar o passo de particionamento.

- 8.25 (Percorrendo um labirinto) A figura 8.44 é uma representação de um array bidimensional de um labirinto. No array bidimensional, as cerquinhos representam as paredes do labirinto e os pontos representam quadrados nos possíveis caminhos pelo labirinto. Movimentos são permitidos apenas nas posições do array que contiverem um ponto.

Há um algoritmo simples para percorrer um labirinto que garante a localização da saída (supondo que existe uma saída). Se não houver uma saída, você chegará à localização inicial novamente. Coloque a sua mão direita na parede à sua direita e comece a andar para a direita. Nunca tire a sua mão da parede. Se o labirinto virar para a direita, siga a parede à direita. Contanto que você não remova a sua mão da parede, você acabará chegando à saída do labirinto. É possível que haja um caminho mais curto do que o que você tomou, mas a solução do labirinto é garantida se o algoritmo for seguido.

Escreva a função `reverso` para percorrer o labirinto. A função deve receber argumentos que incluem um array de 12 por 12 caracteres que representa o labirinto e a localização inicial `labyrinто начальная` para saída do labirinto, ele deve colocar `X` em cada quadrado no caminho. A função deve exibir o labirinto depois de cada movimento de modo que o usuário possa observar enquanto o problema da saída do labirinto é resolvido.

- 8.26 (Gerando labirintos aleatórios) Escreva uma função `GenerarLabyrinth` que recebe como um argumento um array de 12 caracteres bidimensional e produza aleatoriamente um labirinto. A função também deve fornecer as posições inicial e final do labirinto. Experimente a função `azeTraverse` do Exercício 8.25, utilizando vários labirintos gerados aleatoriamente.

- 8.27 (Labirintos de qualquer largura e altura.) Generalize as funções `traverse` e `mazeGenerator` dos exercícios 8.25 e 8.26 para processar labirintos de qualquer largura e altura.

- 8.28 (Modificações para o Simpletron) No Exercício 8.19, você escreveu uma simulação de software de um computador que executa programas escritos em Simpletron Machine Language (SML). Nesse exercício, são propostas várias modificações e aprimoramentos

- para o Simpletron Simulator. Nos exercícios 21-26 e 21-27 [PÁGINA] os construir um compilador que converte programas escritos em linguagem de programação de alto nível (uma variação do PASCAL) em SML. Algumas das seguintes modificações e memórias podem ser necessárias para executar os programas produzidos. [NOTAS DE APOIO] As declarações podem entrar em conflito com outras e, portanto, devem ser feitas separadamente.]

- a) Estender a memória do Simpletron Simulator para conter 1.000 posições de memória, fim de permitir que o Simpletron trate programas maiores.
 - b) Permitir que o simulador realize cálculos de módulo. Isso requer uma instrução SML adicional.
 - c) Permitir que o simulador realize cálculos de exponenciação. Isso requer uma instrução SML adicional.
 - d) Modificar o simulador para utilizar valores hexadecimais em vez de valores inteiros para representar instruções SML.
 - e) Modificar o simulador para permitir saída de uma nova linha. Isso requer uma instrução SML adicional.

```
#####
# . . . # . . . . #
. . # . # . # # # . #
# # # . # . . . # . #
# . . . # # . # . .
#
# # # . # # # . # #
# # . # . # . # . #
# . . . . . . # . #
# # # # # . # # # . #
# . . . . . . # . #
# # # # # # # # # #
```

Figura 8.44 Representação de um labirinto com um array bidimensional.

- f) Modificar o simulador para processar valores de ponto flutuante além de valores inteiros.
- g) Modificar o simulador para tratar entradas de string. A palavra do Simpletron pode ser dividida em dois grupos, cada uma armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente ASCII decimal de um caractere. Adicione uma instrução de linguagem de máquina que irá inserir uma string e armazenar a string inicial em uma posição da memória específica do Simpletron. A primeira metade da palavra nessa posição será uma contagem do número de caracteres na string (isto é, comprimento da string). Cada meia-palavra sucessiva contém um caractere ASCII como dois dígitos decimais expressos. A instrução de linguagem de máquina converte cada caractere em seu equivalente ASCII e atribui a ele uma meia-palavra.]
- h) Modificar o simulador para tratar saída de strings armazenadas. Adicione uma instrução de linguagem de máquina que imprimirá uma string inicial em certa posição da memória de Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia-palavra sucessiva contém um caractere ASCII como dois dígitos decimais expressos. Cada meia-palavra sucessiva contém um caractere de ASCII como dois dígitos decimais expressos.]
- i) Modifique o simulador para incluir o comando `SIM_DUMP` que imprime um dump de memória depois que cada instrução executa.

Forneca o código de operação de palavras de depuração.

8.29

O que esse programa faz?

```

1 // Ex. 8.29: ex08_29.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 bool mystery3(const char *, const char *); // protótipo
9
10 int main()
11 {
12     char string1[ 80], string2[ 80];
13
14     cout << "Enter two strings: ";
15     cin >> string1 >> string2;
16
17     cout << "The result is " << mystery3( string1, string2 ) << endl;
18 } // fim de main
19
20 // O que essa função faz?
21 bool mystery3(const char *s1, const char *s2 )
22 {
23     for ( ; *s1 != '\0' && *s2 != '\0' ; s1++, s2++ )
24
25         if ( *s1 != *s2 )
26             return false ;
27
28     return true ;
29 } // fim da função mystery3

```

Exercícios de manipulação de string

- 8.30 Escreva um programa que utilize uma função para comparar duas strings inseridas pelo usuário. O programa deve declarar se a primeira string é menor que, igual a ou maior que a segunda string.
- 8.31 Escreva um programa que utiliza uma função para comparar duas strings inseridas pelo usuário. O programa deve inserir o número de caracteres a ser comparado. O programa deve declarar se a primeira string é menor que, igual a ou maior que a segunda string.
- 8.32 Escreva um programa que utiliza a geração de números aleatórios para criar frases. O programa deve utilizar quatro arrays de ponteiros para strings chamados `verb`, `noun`, `verb` e `preposition`. O programa deve criar uma frase selecionando aleatoriamente uma palavra de cada array na seguinte ordem: `verb noun verb preposition`. Quando cada palavra é selecionada, ela deve

ser concatenada pelas palavras anteriores em um array suficientemente grande para armazenar a frase inteira. As palavras devem ser separadas por espaços. Quando a frase final for enviada para saída, ela deve iniciar com uma letra maiúscula e terminar com um pc. O programa deve gerar 20 dessas frases.

Os arrays devem ser preenchidos como segue: o array `adjective` deve conter os valores "a", "big", "one", "some" e "any"; e array `noun` deve conter os valores "boy", "girl", "dog", "town" e "car"; o array `verb` deve conter os valores "walked", "jumped", "ran", "walked" e "skipped"; o array `preposition` deve conter as preposições "in", "over", "under" e "on".

Depois de concluir o programa, modifique-o para produzir uma pequena história com várias dessas frases. (E que tal um escritor de textos aleatório? !)

- 8.33 (Limerick) Um limerick é um poema humorístico de cinco versos em que a primeira e a segunda linha rimam com a quinta, e a terceira linha rima com a quarta. Utilizando técnicas semelhantes àquelas desenvolvidas no Exercício 8.32, escreva um programa C++ que produza limericks aleatórios. Aprimorar esse programa para produzir bons limericks é um problema desafiador, mas o resultado vale o esforço.

- 8.34 Escreva um programa que traduz frases de inglês para latim de porco. Existem muitas maneiras de se fazer isso, mas o método mais simples é regularizar a frase e depois aplicar as regras de latim de porco. Utilizando tokens para marcar as frases da língua de porco para simplificar, utilize o seguinte algoritmo: Para formar uma frase em latim de porco a partir do inglês, tokenize a frase em palavras com função `strtok`. Para traduzir cada palavra inglesa em uma palavra do latim de porco, coloque a primeira letra da palavra inglesa no final da palavra e adicione "as" ao final, a palavra "torna" torna-se "tornay", a palavra "the" torna-se "tay" e a palavra

"computer" torna-se "computercay". Os espaços entre as palavras permanecem iguais. Suponha que a escrita do inglês consista em palavras separadas por espaços, não haja nenhuma marcação de pontuação e todas as palavras tenham duas ou mais letras. A função `printLatinWord` deve exibir cada palavra. Toda vez que um token for localizado em `strtok`, chame a função `getNextToken` do token para a função `printLatinWord`. Imprima a palavra em latim de porco.]

- 8.35 Escreva um programa que insere um número de telefone com dígitos e símbolos. O programa deve utilizar a função `strtok` para extrair o código de área como um token, os três primeiros dígitos do número de telefone como um segundo token e os últimos quatro dígitos do número de telefone como um terceiro token. Os sete dígitos do número de telefone devem ser concatenados em uma string. O código de área e o número de telefone devem ser impressos.

- 8.36 Escreva um programa que insere uma linha de texto, tokenizada, inversamente, com tokens na ordem inversa.

- 8.37 Utilize as funções de comparação de string discutidas na Seção 8.13.2 e as técnicas para classificar arrays desenvolvidas no Capítulo 8 para escrever um programa que alfabetiza uma lista de strings. Utilize os nomes de 10 ou 15 bairros em sua área como dados para o programa.

- 8.38 Escreva duas versões de cada função de cópia de string e concatenação de string na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.

- 8.39 Escreva duas versões de cada função de comparação de string na Figura 8.30. A primeira versão deve utilizar o subscrito de array, a segunda, ponteiros e aritmética de ponteiros.

- 8.40 Escreva duas versões da função `strlen` na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.

Seção especial: exercícios avançados de manipulação de string

Os exercícios precedentes são voltados para texto e projetados para testar o entendimento do leitor de conceitos fundamentais de manipulação de string. Esta seção inclui uma coleção de exercícios de manipulação de string avançados e intermediários. O leitor deve acreditar que esses problemas desafiadores, e até divertidos. Os problemas variam consideravelmente em dificuldade. Alguns requerem uma hora ou duas para escrever e implementar o programa. Outros são úteis para atribuições de laboratório que talvez requeiram duas ou três sessões de estudo e implementação. Alguns são projetos de conclusão de curso desafiadores.

- 8.41 (Análise de textos) A disponibilidade de computadores com capacidades de manipulação de string resultou em algumas abordagens interessantes para analisar textos de grandes autores. Muita atenção foi dada à polêmica de que William Shakespeare não teria existido de fato. Alguns especialistas acreditam que há evidência substancial para indicar que Christopher Marlowe ou outros foram os verdadeiros autores das obras-primas atribuídas a Shakespeare. Os pesquisadores têm utilizado computadores para encontrar semelhanças nas obras de desses autores. Esse exercício examina três métodos para analisar textos com um computador. Observe que milhares de textos, incluindo os de Shakespeare, estão disponíveis em www.gutenberg.org

- a) Escreva um programa que lê várias linhas de texto do teclado e imprime uma tabela que indica o número de ocorrências de cada letra do alfabeto no texto. Por exemplo, a frase

To be, or not to be: that is the question:

contém um 'a', dois 'b', nenhum 'c' etc.

- b) Escreva um programa que lê várias linhas de texto e imprime uma tabela que indica o número de palavras de uma letra, palavras de duas letras, palavras de três letras etc. que aparecem no texto. Por exemplo, a frase

Whether 'tis nobler in the mind to suffer
 contém os seguintes comprimentos de palavra e ocorrências:

Comprimento de palavra	Ocorrências
1	0
2	2
3	1
4	(Incluindo tis)
5	0
6	2
7	1

- c) Escreva um programa que lê várias linhas de texto e imprime uma tabela que indica o número de ocorrências de cada palavra diferente no texto. A primeira versão de seu programa deve incluir as palavras na tabela na mesma ordem em que elas aparecem no texto. F

To be, or not to be: that is the question:

Whether 'tis nobler in the mind to suffer

Contém a palavra 'to' três vezes, a palavra 'be' duas vezes, a palavra 'or' uma vez etc. Então você deve tentar uma impressão mais interessante (e útil) em que as palavras são classificadas alfabeticamente.

- 8.42 (Processamento de layout) Um aspecto importante em sistemas de processamento de texto é o alinhamento à esquerda e à direita de uma página simultaneamente. Isso significa que quando a impressão de que o documento foi editorado em vez de preparado em uma máquina de escrever. A justificação de texto pode ser realizada em sistemas computador inserindo caracteres de espaço em branco entre cada uma das palavras em uma linha para que a última palavra de cada seja alinhada com a margem direita.

Escreva um programa que lê várias linhas de texto e imprime esse texto no formato alinhado. Suponha que o texto deva ser impresso no papel de 8 1/2 polegadas de largura e que as margens de uma polegada sejam adotadas nos lados esquerdo e direito. Suponha que o computador imprima 10 caracteres por polegada horizontal. Portanto, seu programa deve imprimir 6 1/2 polegadas de texto ou 65 caracteres por linha.

- 8.43 (Imprimindo datas em vários formatos) As datas são comumente impressas em vários formatos diferentes na correspondência comercial. Dois dos formatos mais comuns em inglês são

07/21/1955

July 21, 1955

Escreva um programa que lê uma data no primeiro formato e imprime essa data no segundo formato.

- 8.44 (Proteção de cheques) Computadores são freqüentemente empregados em sistemas de verificação de cheque como aplicativos de folha de pagamento e contas a pagar. Circulam muitas histórias estranhas relacionadas com cheques de pagamento de salário sem impressos (por equívoco) com quantias acima de \$ 1 milhão. Valores de cheque absurdos são impressos por sistemas computadorizados de preenchimento de cheque devido a erro humano ou falha mecânica. Os projetistas de sistemas embutem controles em seus sistemas para evitar a emissão desses cheques errados.

Outro problema sério é a alteração intencional de um valor do cheque por alguém que pretende receber um cheque fraudulentamente. Para evitar que uma quantia monetária seja alterada, a maioria dos sistemas computadorizados de preenchimento de cheque emprega uma técnica chamada de chequing.

Cheques projetados para imprimir por computador contêm um número fixo de espaços em que o computador pode imprimir uma quantidade monetária.

Suponha que um cheque de pagamento contenha oito espaços em branco em que o computador deve imprimir a quantidade de um cheque de pagamento semanal. Se a quantidade for grande, então todos os oito espaços serão preenchidos, por exemplo:

1,230.60 (valor do cheque)

12345678 (números de posição)

Por outro lado, se a quantidade for menor que \$ 1000, então vários dos espaços seriam comumente deixados em branco. Por exemplo:

99.87

12345678

contém três espaços em branco. Se um cheque é impresso com espaços em branco, é mais fácil para alguém alterar o valor do cheque. Para evitar que um cheque seja alterado, muitos sistemas de preenchimento de cheques protegem o valor como segue:

***99.87

12345678

Escreva um programa que aceita como entrada uma quantia monetária para ser impressa em um cheque e então imprime o valor em forma de cheque protegido com asteriscos iniciais se necessário. Suponha que nove espaços estão disponíveis para imprimir uma quantia.

8.45 (Valor de um cheque por extenso)

Continuando a discussão do exemplo anterior, reiteramos a importância de projetar sistemas de preenchimento de cheque para evitar alteração de seus valores. Um método comum de segurança requer que o valor do cheque seja escrito tanto em números como 'por extenso'. Mesmo se alguém for capaz de alterar o valor numérico do cheque, é extremamente difícil alterar o valor por extenso.

Escreva um programa que aceita como entrada o valor do cheque em números e gera como saída o valor por extenso correspondente. O programa deve ser capaz de tratar valores de cheques tão altos quanto \$ 99,99. Por exemplo, o valor 112,43 deve ser escrito assim:

ONE HUNDRED TWELVE and 43/100

(Código Morse) Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1844 para utilização com o sistema de telégrafo. O código Morse atribui uma série de pontos e traços a cada letra do alfabeto, a cada dígito e a alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-vírgula). Em sistemas voltados para áudio, o ponto representa um som breve e o traço representa um som longo. Outras representações de pontos e traços são utilizadas com sistemas baseados em sinais luminosos e sistemas baseados em bandeira.

A separação entre palavras é indicada por um espaço ou, simplesmente, pela ausência de um ponto ou traço. Em um sistema baseado em áudio, um espaço é indicado por um ponto breve de tempo durante o qual nenhum som é transmitido. A versão internacional do código Morse aparece na Figura 8.45.

Escreva um programa que lê uma frase em inglês e a codifica em código Morse. Escreva também um programa que lê uma frase em código Morse e a converte no equivalente em inglês. Utilize um espaço em branco entre cada letra codificada em Morse e três espaços em branco entre cada palavra codificada em Morse.

Caractere	Código	Caractere	Código	Dígito	Código
A	.-	N	-.	1	----
B	-...	O	---	2	..---
C	-.-.	P	.--.	3	...--
D	-..	Q	--.-	4-
E	.	R	.-.	5
F	..-.	S	...	6
G	--.	T	-	7	--...
H	U	--	8	---..
I	..	V	...-	9	----.
J	---	W	.--	0	-----
K	-.-	X	-..-		
L	-..	Y	-.--		
M	--	Z	--..		

Figura 8.45 Alfabeto do código Morse.

- 8.47 (Um programa de conversões métricas) Escreva um programa que ajudará o usuário a realizar conversões métricas. Seu programa deve permitir que o usuário especifique o nome das unidades como strings (isto é, centímetros, litros, gramas etc. para o sistema métrico polegadas, quartos, libras etc. para o sistema inglês) e deve responder a perguntas simples como

“Quantas polegadas temos em 2 metros?”
“Quantos litros temos em 10 quartos?”

Seu programa deve reconhecer conversões inválidas. Por exemplo, a pergunta

“Quantos pés temos em 5 quilos?”

não é significativo, pois não são unidades de comprimento que são unidades de peso.

Um projeto desafiador de manipulação de string

- 8.48 (Um gerador de palavras cruzadas) Alguns anos atrás, as pessoas já brincou de palavras cruzadas, mas poucos tentaram gerar um jogo de palavras cruzadas. Gerar um programa de palavras cruzadas é um problema difícil. Isso é sugerido aqui como um projeto de manipulação de strings. Gerar um programa de palavras cruzadas é um problema difícil. Isso é sugerido aqui como um projeto de manipulação de strings. Que exige bastante sofisticação e esforço. Há muitas questões que o programador deve resolver para fazer funcionar até mesmo o mais simples programa gerador de palavras cruzadas. Por exemplo, como se representa a grade das palavras cruzadas dentro do computador? Você deve utilizar uma série de strings ou arrays bidimensionais? O programador precisa de uma fonte de palavras (isto é, um dicionário computadorizado) que possa ser referenciado diretamente pelo programa. De que forma essas palavras devem ser armazenadas para facilitar as complexas manipulações requeridas pelo programa? O leitor realmente ambicioso vai querer gerar a parte das ‘pistas’ do jogo em que as breves dicas para as palavras ‘horizontais’ e para as palavras ‘verticais’ são impressas para o gerador de quebra-cabeça. A impressão de uma versão da parte em branco do jogo não é um problema simples.



Meu desígnio, todo sublime,
a tempo hei de alcançar.
W. S. Gilbert

~~Este é o meu condéndado qude?~~
William Shakespeare

Não seja 'consistente', mas
simplesmente autêntico.
Oliver Wendell Holmes, Jr.

Mas, sobretudo, sé a ti próprio
fiel.
William Shakespeare

Classes: um exame mais profundo, parte 1

OBJETIVOS

Neste capítulo, você aprenderá:

A utilizar um empacotador de pré-processador para evitar múltiplos erros de definição causados pela inclusão de mais de uma cópia de um arquivo de cabeçalho em um arquivo de código-fonte.

A definir o escopo de classe e acessar membros de classe via o nome de um objeto, uma referência a um objeto ou um ponteiro para um objeto.

A definir construtores com argumentos-padrão.

Como os destrutores são utilizados para realizar uma 'faxina de terminação' em um objeto antes de ele ser destruído.

Quando construtores e destrutores são chamados e a ordem em que são chamados.

Os erros de lógica que podem ocorrer quando uma função-membro `public` de uma classe retorna uma referência a dados `private`.

A atribuir os membros de dados de um objeto àqueles de outro objeto por atribuição-padrão de membro a membro.

P á m S	<ul style="list-style-type: none"> 9.1 Introdução 9.2 Estudo de caso da classe Time 9.3 Escopo de classe e acesso a membros de classe 9.4 Separando a interface da implementação 9.5 Funções de acesso e funções utilitárias 9.6 Estudo de caso da classe Time construtores com argumentos-padrão 9.7 Destruidores 9.8 Quando construtores e destruidores são chamados 9.9 Estudo de caso da classe Time uma armadilha sutil — retornar uma referência a um membro de dados private 9.10 Atribuição-padrão de membro a membro 9.11 Reusabilidade de software 9.12 Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional) 9.13 Síntese
------------------	--

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

9.1 Introdução

Nos capítulos anteriores, introduzimos muitos termos e conceitos básicos da programação C++ orientada a objetos. Além disso, cutimos nossa metodologia de desenvolvimento de programa: selecionamos atributos e comportamentos apropriados para cada e especificamos a maneira como os objetos de nossas classes colaboravam com os objetos de classes da C++ Standard Library para realizar os objetivos gerais de cada programa.

Neste capítulo, fazemos um exame mais profundo das classes. Utilizamos um [estudo de caso](#) integrado da classe Time (três exemplos) e no Capítulo 10 (dois exemplos) para demonstrar vários recursos da construção de classes. Iniciamos com uma revisão que revisa vários dos recursos apresentados nos capítulos anteriores. O exemplo também demonstra um importante conceito de engenharia de software do C++ — utilizar um ‘empacotador de pré-processador’ em arquivos de cabeçalho para impedir que o código

utilize esse código diretamente no pré-processador ou no código-fonte de outras definições. Visto que uma classe pode ser definida apenas uma vez, isso evita que o código-cliente aceda ao código-fonte de outras definições.

Em seguida, discutimos o escopo de classe e os relacionamentos entre membros de uma classe. Demonstramos também como o código-cliente pode acessar a memória de uma classe por meio de três tipos de ‘handles’ — o nome de um objeto, uma referência a um objeto ou um ponteiro para um objeto. Como você verá, os nomes e as referências do objeto podem ser utilizados com o operador de seleção de membro para acessar um membro; os ponteiros podem ser utilizados com o operador de seleção de membro seta (→).

Discutimos funções de acesso que podem ler ou exibir dados em um objeto. Uma utilização comum de funções de acesso é testar a verdade ou falsidade de condições — essas funções são conhecidas como funções predicado. Também demonstramos a noção de função utilitária (também chamada função auxiliar) — uma função que não opera diretamente sobre o objeto-membro da classe, mas não é projetada para ser utilizada por clientes da classe.

No segundo exemplo do estudo de caso, discutimos como passar argumentos para construtores e de que maneira argumentos-padrão podem ser utilizados no construtor para permitir que o código-cliente inicialize objetos de uma classe utilizando uma variedade de argumentos. Em seguida, discutimos uma função-membro especial chamada destrutor que faz parte de cada classe e é utilizada para realizar uma ‘faxina de terminação’ em um objeto antes de ele ser destruído. Então demonstramos a ordem em que construtores e destruidores são chamados, porque a corretude dos seus programas depende do uso adequado de objetos inicializados ainda não foram destruídos.

Nosso último exemplo do estudo de caso neste capítulo mostra uma prática de programação perigosa em que uma

função atribui diretamente a memória de um objeto. Essa é uma operação perigosa de que os objetos pertencentes à mesma classe podem ser destruídos uns aos outros utilizando a atribuição-padrão de membro a membro, à qual copia os membros de dados no objeto de atribuição para os membros de dados correspondentes do objeto à esquerda da atribuição. O capítulo conclui com uma discussão da reusabilidade de software.

9.2 Estudo de caso da classe Time

Nosso primeiro exemplo (figuras 9.1–9.3) é um passo-a-passo de teste que testa a classe. Você já criou várias classes neste livro. Nesta seção, revisamos muitos dos conceitos tratados no Capítulo 3 e demonstramos um importante conceito da engenharia de software.

de software do C++ — utilizar um ‘empacotador de pré-processador’ em arquivos de cabeçalho para impedir que o código no cabeçalho seja incluído no mesmo arquivo de código-fonte mais de uma vez. Visto que uma classe só pode ser definida uma única vez, usar essas diretivas de pré-processador impede erros de múltiplas definições.

Definição da classe Time

A definição de classe (Figura 9.1) contém protótipos (linhas 13–16) para as funções-membro printStandard. A classe inclui os membros hora, minute e second (linhas 18–20). Os membros privados da classe Time podem ser acessados somente pelas suas quatro funções-membro. O Capítulo 12 introduz um terceiro especificador de acesso protected, já que estudamos herança e o papel que ela desempenha na programação orientada a objetos.



Boa prática de programação 9.1

Por questão de clareza e legibilidade, utilize cada especificador de acesso uma única vez em uma definição de classe. Coloque o especificador public primeiro, onde eles são fáceis de encontrar.



Observação de engenharia de software 9.1

Todo elemento de uma classe deve ter visibilidade que possa ser comprovado que o elemento precise de visibilidade public. Esse é outro exemplo do princípio do menor privilégio.

Na Figura 9.1, observe que a definição de classe está incluída no seguinte(a) (linhas 5–7 e 23):

```
// impede múltiplas inclusões de arquivo de cabeçalho
#ifndef TIME_H
#define TIME_H
...
#endif
```

Quando construirmos programas maiores, outras definições e declarações também serão colocadas em arquivos de cabeçalho. O empacotador de pré-processador anterior impede que o código dentro de #ifndef TIME_H seja incluído se o nome TIME_H tiver sido definido. Se o cabeçalho não foi incluído anteriormente, a diretiva #define e as instruções de arquivo de cabeçalho serão incluídas. Se o cabeçalho estiver já incluído, o arquivo de cabeçalho não será novamente incluído. Tentativas de incluir um arquivo de cabeçalho múltiplas vezes (inadvertidamente)

```
1 // Figura 9.1: Time.h
2 // Declaração da classe Time.
3 // Funções-membro são definidas em Time.cpp
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef TIME_H
7 #define TIME_H
8
9 // definição da classe Time
10 class Time
11 {
12 public :
13     Time();    // construtor
14     void setTime( int , int , int ); // configura hora, minuto e segundo
15     void printUniversal(); // imprime a hora no formato de data/hora universal
16
17     private :printStandard(); // imprime a hora no formato-padrão de data/hora
18     int hour; // 0 - 23 (formato de relógio de 24 horas)
19     int minute; // 0 - 59
20     int second; // 0 - 59
21 }; // fim da classe Time
22
23 #endif
```

Figura 9.1 Definição da classe Time

geral ocorrem em programas grandes com muitos arquivos de cabeçalho que podem eles mesmos incluir outros arquivos de cabeçalho. [Nota: A convenção comumente utilizada para o nome da constante simbólica nas diretivas de pré-processador é simplesmente o nome do arquivo de cabeçalho em maiúscula com o caractere sublinhado substituindo o ponto.]



Dica de prevenção de erro 9.1

Utilize diretivas de pré-processador `#define` e `#ifndef` para formar um empacotador de pré-processador que impede que os arquivos de cabeçalho sejam incluídos mais de uma vez em um programa.



Boa prática de programação 9.2

Utilize o nome do arquivo do cabeçalho em caixa alta com o ponto substituído por um sublinhado nas diretivas de pré-processador: `#ifndef _FILE_H_` define um arquivo de cabeçalho.

Funções-membro da classe

Na Figura 9.2, o construtor (linhas 14–17) inicializa os membros de dados como 0 (isto é, o equivalente de data/hora universal de 12 AM). Isso assegura que o objeto inicia em um estado consistente. Os valores inválidos não podem ser armazenados nos membros de dados de um objeto porque o construtor é chamado quando o objeto é criado; todas as tentativas subsequentes de um cliente de modificar os membros de dados serão escritas diretamente sobre o membro original (o que é correto). Por fim, é importante observar que o programador pode definir vários construtores sobrecarregados para uma classe.

Os membros de dados de uma classe não podem ser inicializados onde são declarados, no corpo da classe. É altamente recomendável que esses membros de dados sejam inicializados pelo construtor da classe (como não há inicialização-padrão para membros de tipo fundamental). Os membros de dados também podem receber valores. [Nota: Esse exemplo demonstra que apenas os membros de dados de uma classe dos tipos int e double podem ser inicializados no corpo da classe.]



Erro comum de programação 9.1

Tentar inicializar explicitamente um membro de classe dentro da classe na definição de classe é um erro de sintaxe.

A função `setTime` (linhas 21–26) é uma função que declara três parâmetros para configurar a hora. Uma expressão condicional testa cada argumento para determinar se o valor está dentro de um intervalo especificado. Por exemplo, `hour` (linha 23) deve ser maior ou igual a 0 e menor que 24, porque o formato de data/hora universal representa horas como inteiro de 0 a 23 (por exemplo, 1 PM é a hora 13, 12 PM é a hora 12, etc.). De forma similar, os outros intervalos são configurados como zero para assegurar sempre que os dados consistentes — isto é, os valores dos dados do objeto sempre são mantidos em um intervalo, mesmo se os valores fornecidos como argumentos para a função estiverem incorretos. Nesse exemplo, zero é um valor consistente para

Um valor passado para é um valor correto se estiver no intervalo permitido para o membro que ele está inicializando. Portanto, qualquer número no intervalo de 0 a 23 seria um valor correto para sempre é um valor consistente. Entretanto, um valor consistente não necessariamente é um valor correto porque o argumento recebido estava fora do intervalo correto somente se a hora atual for, coincidentemente, meia-noite.

A função `printUniversal` (linhas 29–33 da Figura 9.2) não aceita argumentos e gera saída da data no formato de data/hora universal, consistindo em três pares de dígitos separados por dois-pontos — para a hora, minuto e segundo, respectivamente. Por exemplo, se a data/hora fosse 1:30:07 PM, a função retornaria 01:30:07. Observe que a linha 31 utiliza o manipulador de fluxo `setw` para especificar o caractere de preenchimento que é exibido quando um inteiro é enviado para a saída em um campo maior do que o número de dígitos no valor. Por padrão, os caracteres de preenchimento aparecem à esquerda dos dígitos. Nesse exemplo, se o valor 02, ele será exibido como 02, porque o caractere de preenchimento está configurado com zero ('0'). Se o número cuja saída está sendo gerada preencher o campo especificado, o caractere de preenchimento não será exigido (é este o caso para todos os valores subsequentes que são exibidos em campos maiores do que o valor especificado). Isso contrasta com

que se aplica somente ao próximo caractere exibido (configuração ‘não aderente’).



Dica de prevenção de erro 9.2

Toda configuração aderente (como um caractere de preenchimento ou a precisão de ponto flutuante) deve ser restaurada à sua configuração anterior quando não for mais necessária. A falha em fazer isso pode resultar em uma saída formatada incorretamente mais tarde em um programa. O Capítulo 15, “Entrada/saída de fluxo”, discute como redefinir o caractere de preenchimento e a precisão.

A função `printStandard` (linhas 36–41) não recebe argumentos e gera saída de uma data no formato-padrão de data/hora, que consiste nos valores dia, mês, ano, hora, minuto e segundo separados por dois-pontos e seguidos por um indicador AM ou PM (por exemplo,

```

1 // Figura 9.2: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
11
12 // O construtor de Time inicializa cada membro de dados como zero.
13 // Assegura que todos os objetos Time iniciem em um estado consistente.
14 Time::Time()
15 {
16     hour = minute = second = 0;
17 } // fim do construtor de Time
18
19 // configura novo valor de Time utilizando a hora universal; assegura que
20 // os dados permaneçam consistentes configurando valores inválidos como zero
21 void Time::setTime( int h, int m, int s )
22 {
23     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
24     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
25     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
26 } // fim da função setTime
27
28 // imprime a hora no formato de data/hora universal (HH:MM:SS)
29 void Time::printUniversal()
30 {
31     cout << setfill('0') << setw( 2 ) << hour << ":"
32         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
33 } // fim da função printUniversal
34
35 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
36 void Time::printStandard()
37 {
38     cout << ( ( hour == 0 || hour == 12 ) ? 12: hour % 12 ) << ":"
39         << setfill('0') << setw( 2 ) << minute << ":" << setw( 2 )
40         << second << ( hour < 12 ? "AM": "PM" );
41 } // fim da função printStandard

```

Figura 9.2 Definições de função-membro da classe Time

Semelhantemente à função `printUniversal`, a função `printStandard` utiliza `setfill('0')` para formatação de hora e segundos como dois valores de dígito com zeros à esquerda se necessário. A linha 38 utiliza o operador condicional (`? 12: hour % 12`) para determinar se a hora a ser exibido é 0 ou 12 (AM ou PM), ela aparece como 12; caso contrário, como um valor de 1 a 11. O operador condicional na linha 40 determina se AM ou PM será exibido.

Definindo funções-membro fora da definição de classe; escopo de classe

Embora uma função-membro declarada em uma definição de classe possa ser definida fora dessa definição de classe (e associada via operador de resolução de escopo binário), essa função-membro ainda está dentro da classe. Isso é conhecido apenas por outros membros da classe a menos que referenciado via um objeto da classe, uma referência a um objeto da classe ou um ponteiro para um objeto da classe ou o operador binário de resolução de escopo. Em breve, falaremos mais sobre o escopo de classe.

Se uma função-membro é definida no corpo de uma definição de classe, o compilador C++ tenta colocar inline as chamadas para essa função-membro. As funções-membro definidas fora de uma definição de classe podem ser colocadas inline utilizando a palavra-chave `inline` explicitamente. Lembre-se de que o compilador reserva o direito de não colocar inline nenhuma função.



Dica de desempenho 9.1

Definir uma função-membro dentro da definição de classe coloca a função-membro inline (se o compilador escolher fazer isso). Isso pode melhorar o desempenho.



Observação de engenharia de software 9.2

Definir uma pequena função-membro dentro da definição de classe não promove a melhor engenharia de software, porque os clientes da classe serão capazes de ver a implementação da função, e o código-cliente deve ser recompilado se a definição de função mudar.



Observação de engenharia de software 9.3

Apenas as funções-membro mais simples e mais estáveis (isto é, cujas implementações provavelmente não mudarão) devem ser definidas no cabeçalho de classe.

Funções-membro versus funções globais

É interessante que as funções-membro `eprintStandard` não aceitem nenhum argumento. Isso ocorre porque essas funções-membro sabem implicitamente que devem imprimir os membros pertencentes ao objeto. Elas são invocadas. Isso pode tornar as chamadas de função-membro mais concisas do que as chamadas de função convencionais na programação procedural.



Observação de engenharia de software 9.4

Utilizar freqüentemente uma abordagem de programação orientada a objetos pode simplificar chamadas de função reduzindo o número de parâmetros que será passado. Esse benefício da programação orientada a objetos deriva do fato de que encapsular o membro de dados e as funções-membro dentro de um objeto dá às funções-membro o direito de acessar os membros de dados.



Observação de engenharia de software 9.5

Em geral, as funções-membro são mais curtas do que funções em programas não orientados a objeto, porque os dados armazenados em membros de dados foram idealmente validados por um construtor ou por funções-membro que armazenam novos dados. Com os dados já estando no objeto, as chamadas de função-membro freqüentemente não têm argumentos ou pelo menos têm menos argumentos que as típicas chamadas de função em linguagens não orientadas a objeto. Portanto, as chamadas, as definições de função e os protótipos de função são menores. Isso facilita muitos aspectos do desenvolvimento de programa.



Dica de prevenção de erro 9.3

O fato de que as chamadas de função-membro geralmente não aceitam argumentos ou aceitam substancialmente menos argumentos que as chamadas de função convencionais em linguagens não orientadas a objeto reduz a probabilidade de passar os argumentos errados, os tipos de argumentos errados ou o número errado de argumentos.

Utilizando a classe Time

Uma vez que a classe Time definida, ela pode ser utilizada como um tipo em declarações de objeto, array, ponteiro e referência como mostrado a seguir:

```
Time sunset; // objeto do tipo Time
Time arrayOfTimes[5]; // array de 5 objetos Time
Time &dinnerTime = sunset; // referência a um objeto Time
Time *timePtr = &dinnerTime; // ponteiro para um objeto Time
```

A Figura 9.3 utiliza a classe Time definida na Adinha 12 para instanciar um único objeto Time. Quando o objeto é instanciado, o construtor é chamado para inicializar cada membro de dados. As linhas 16 e 18 imprimem a hora nos formatos universal e padrão para confirmar que os membros foram inicializados adequadamente. A linha 20 configura uma nova hora chamando a função-membro setTime. As linhas 24 e 26 imprimem novamente a hora em ambos os formatos. A linha 28 tenta utilizar setTime para configurar os membros de dados como valores inválidos e configura os valores inválidos como 0 para manter o objeto em um estado consistente. Por fim, as linhas 33 e 35 imprimem novamente a hora em ambos os formatos.

```

1 // Figura 9.3: fig09_03.cpp
2 // Programa para testar a classe Time.
3 // NOTA: Esse arquivo deve ser compilado com Time.cpp.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
9
10 int main()
11 {
12     Time t; // instancia o objeto t da classe Time
13
14     // gera saída de valores iniciais do objeto Time t
15     cout << "The initial universal time is " ;
16     t.printUniversal(); // 00:00:00
17     cout << "\nThe initial standard time is " ;
18     t.printStandard(); // 12:00:00 AM
19
20     t.setTime( 13, 27, 6 ); // muda a hora
21
22     // gera saída de novos valores do objeto Time t
23     cout << "\n\nUniversal time after setTime is " ;
24     t.printUniversal(); // 13:27:06
25     cout << "\nStandard time after setTime is " ;
26     t.printStandard(); // 1:27:06 PM
27
28     t.setTime( 99, 99, 99 ); // tenta configurações inválidas
29
30     // gera saída de valores de t depois de especificar valores inválidos
31     cout << "\n\nAfter attempting invalid settings:" ;
32
33     cout << "\nUniversal time: " ;
34     t.printUniversal(); // 00:00:00
35     cout << "\nStandard time: " ;
36     t.printStandard(); // 12:00:00 AM
37     cout << endl;
38 } // fim de main

```

```

The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM

```

Figura 9.3 Programa para testar a classe Time

Planejando-se para composição e herança

Freqüentemente, as classes não precisam ser criadas ‘a partir do zero’. Em vez disso, elas podem incluir objetos de outras classes ou podem ~~em si~~^{em si} usar de outras classes que fornecem atributos e comportamentos que as novas classes podem utilizar. Essa reutilização de software pode aprimorar significativamente a produtividade do programador e simplificar a manutenção de código. A inclusão de objetos de uma classe como membros de outras classes é chamada de ~~classe~~^{classe} composição. A herança é discutida no Capítulo 10. A derivação de novas classes de classes existentes é discutida no Capítulo 12.

Tamanho de objeto

As pessoas sem experiência em programação orientada a objetos freqüentemente supõem que os objetos devem ser bem grande eis contêm membros de dados e funções-membro. Logicamente, isso é verdade — o programador pode pensar nos objetos contendo dados e funções (e nossa discussão certamente encorajou essa visão); fisicamente, porém, isso não é verdade.



Dica de desempenho 9.2

Os objetos contêm apenas dados, portanto, os objetos são muito menores do que se também contivessem funções-membro. Aplicando-se o escopo a um nome de classe ou a um objeto dessa classe informará somente o tamanho dos membros de dados da classe. O compilador cria uma (única) cópia das funções-membro separada de todos os objetos da classe. Todos os objetos da classe compartilham esta única cópia. Cada objeto precisa, naturalmente, de sua própria cópia dos dados da classe, porque os dados podem variar entre os objetos. O código de função é não modificável (também chamado de puro), daí, pode ser compartilhado entre todos os objetos de uma classe.

9.3 Escopo de classe e acesso a membros de classe

Os membros de dados de uma classe (variáveis declaradas na definição de classe) e as funções-membro (funções declaradas na definição de classe) pertencem ao escopo dessa classe. As funções não-membro são definidas no escopo externo.

Dentro do escopo de uma classe, os membros de classe são imediatamente acessíveis por todas as funções-membro dessa classe. Eles podem ser referenciados por nome. Fora do escopo de uma classe, são referidos como membros de escopo limitado em um objeto — um nome de objeto, uma referência a um objeto ou um ponteiro para um objeto. O tipo do objeto, referência ou ponteiro especifica a interface (isto é, as funções-membro) acessível(is) ao cliente. [Veremos no Capítulo 10 que um handle implícito é inserido pelo compilador em cada referência a um membro de dados ou função-membro a partir de dentro de um objeto.]

As funções-membro de uma classe podem ser sobrecarregadas, mas apenas por outras funções-membro dessa classe. Para sobrecarregar uma função-membro, simplesmente forneça na definição de classe um protótipo para cada versão da função sobrepõendo uma definição de função separada para cada versão da função.

As variáveis declaradas em uma função-membro têm escopo de bloco e são conhecidas somente por essa função. Se uma função-membro definir uma variável com o mesmo nome de uma variável com escopo de classe, a variável de escopo de classe é oculta. Variáveis de escopo de bloco no escopo de bloco. Essa variável oculta pode ser acessada colocando-se o nome da classe seguido do operador de resolução de escopo (isto é, o escopo) do nome da variável. As variáveis globais ocultas podem ser acessadas com o operador de resolução de escopo (ver o Capítulo 6).

~~membro do objeto. O operador de seleção de membro (objeto) indica que o objeto é o ponto de partida para a referência ao membro do objeto.~~

A Figura 9.4 utiliza uma classe simples (chamada `Counter`) com o membro de dados `tipat` (linha 24), a função-membro `setX` (linhas 12–15) e a função-membro `print` (linhas 18–21), para ilustrar o acesso aos membros de uma classe com os operadores de seleção de membro. Para simplificar, incluímos essa pequena classe no mesmo arquivo da main que a utiliza. As linhas 29–31 criam três variáveis relacionadas com um objeto: `counter`, `counterPtr` (um ponteiro para um objeto `Counter`), `counterRef` (uma referência a um objeto variável `Counter`). A variável `counter` aponta para `counter`. Nas linhas 34–35 e 38–39, observe que o programa pode invocar funções-membro `print` utilizando o operador de seleção de membro direto pelo nome do objeto (`counter`) por uma referência ao objeto (`counterRef`, que é um alias para `counter`). De maneira semelhante, as linhas 42–43 demonstram que o programa pode invocar funções-membro `print` utilizando um ponteiro (`counterPtr`) e o operador de seleção de membro seta (`*`).

9.4 Separando a interface da implementação

No Capítulo 3 começamos incluindo a definição de uma classe e definições de função-membro em um arquivo. Em seguida, dermos a separação desse código em dois arquivos — um arquivo de cabeçalho para a definição de classe (isto é, a interface da classe) e um arquivo de código-fonte para as definições de função-membro da classe (isto é, a implementação da classe). Lembre-se de que facilita a modificação de programas — contanto que os clientes de uma classe estejam envolvidos, as alterações na implementação da classe não afetam o cliente desde que a interface da classe permaneça inalterada.



Observação de engenharia de software 9.6

Os clientes de uma classe não precisam de acesso ao código-fonte da classe para utilizá-la. Entretanto, os clientes precisam, de fato, ser capazes de se vincular ao código-objeto da classe (isto é, a versão compilada da classe). Isso encoraja os fornecedores de softwares independentes (ISVs) a vender pacotes para serializar ou licenciar bibliotecas de classes. Os ISVs fornecem em seus produtos somente os arquivos de cabeçalho e os módulos dos objetos. Informações ‘não-proprietárias’ (isto é, não-patenteadas) são reveladas — como seria o caso se o código-fonte fosse fornecido. A comunidade de usuários do C++ se beneficia tendo mais bibliotecas de classes produzidas e disponibilizadas por ISVs.

```

1 // Figura 9.4: fig09_04.cpp
2 // Demonstrando os operadores de acesso ao membro de classe com . e -
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Count
8 class Count
9 {
10 public : // dados public são perigosos
11     // configura o valor do membro de dados private x
12     void setX( int value )
13     {
14         x = value;
15     } // fim da função setX
16
17     // imprime o valor do membro de dados private x
18     void print()
19     {
20         cout << x << endl;
21     } // fim da função print
22
23 private :
24     int x;
25 }; // fim da classe Count
26
27 int main()
28 {
29     Count counter; // cria objeto counter
30     Count *counterPtr = &counter; // cria ponteiro para counter
31     Count &counterRef = counter; // criar referência para counter
32
33     cout << "Set x to 1 and print using the object's name: " ;
34     counter.setX( 1 ); // configura membro de dados x como 1
35     counter.print(); // chama função-membro print
36
37     cout << "Set x to 2 and print using a reference to an object: " ;
38     counterRef.setX( 2 ); // configura membro de dados x como 2
39     counterRef.print(); // chama função-membro print
40
41     cout << "Set x to 3 and print using a pointer to an object: " ;
42     counterPtr->setX( 3 ); // configura membro de dados x como 3
43     counterPtr->print(); // chama função-membro print
44
45 } // fim de main

```

Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3

Figura 9.4 Acessando funções-membro de um objeto por meio de cada tipo de handle de objeto — o nome do objeto, uma referência ao objeto e um ponteiro para o objeto.

Na realidade, as coisas não são essa maravilha toda. Os arquivos de cabeçalho contêm algumas partes da implementação e sobre outras. As funções-membro inline, por exemplo, precisam estar em um arquivo de cabeçalho para que, quando o compilador compilar um cliente, este possa incluir adequadamente a definição da classe. Os membros que não aparecem na definição de classe no arquivo de cabeçalho, portanto esses membros são visíveis aos clientes mesmo que os clientes possam acessar os membros. No Capítulo 10, mostramos como utilizar uma 'classe proxy' para ocultar até mesmo os dados de uma classe a partir de clientes da classe.



Observação de engenharia de software 9.7

As informações importantes para a interface para uma classe devem ser incluídas no arquivo de cabeçalho. As informações que serão utilizadas internamente na classe e não serão necessárias aos clientes da classe devem ser incluídas no arquivo-fonte não publicado. Esse é ainda outro exemplo do princípio do menor privilégio.

9.5 Funções de acesso e funções utilitárias

As funções de acesso podem ler ou exibir dados. Outra utilização comum das funções de acesso é testar a verdade ou falsidade de condições — essas são frequentemente chamadas de predicados. Um exemplo de uma função predicado seria uma função isEmpty para qualquer classe contêiner — uma classe capaz de armazenar muitos objetos — como uma lista vinculada, uma pilha ou uma fila. Um programa talvez precise de tentar ler outro item do objeto contêiner. Uma função predicado pode testar um objeto de classe contêiner para determinar se ela tem ou não espaço adicional. As funções predicado úteis para nossa TimepodriamissePM

O programa das figuras 9.5–9.7 demonstram a noção de uma função utilitária. A função utilitária printAnnualSales é uma função membro da classe SalesPerson. Ela faz parte da interface da classe; em vez disso, é uma função membro que importa a operação das funções-membro da classe. As funções utilitárias não são projetadas para ser utilizadas por clientes de uma classe (mas podem ser utilizadas dentro de uma classe, como veremos no Capítulo 10).

A classe SalesPerson (Figura 9.5) declara um array de 12 estimativas de vendas mensais (linha 16) e os protótipos para o construtor e as funções-membro da classe que manipulam o array.

Na Figura 9.6, o construtor SalesPerson (linhas 15–19) inicializa sales[12] no zero. A função-membro getSales (linhas 36–43) configura a estimativa de vendas da classe. A função-membro printAnnualSales (linhas 46–51) imprime o total de vendas dos últimos 12 meses. A função utilitária printAnnualSales (linhas 54–62) soma as 12 estimativas mensais de vendas em monetário. A função-membro printAnnualSales edita as estimativas de vendas em formato monetário.

Na Figura 9.7 note que o método aplicativo inclui uma única sequência simples de chamadas de função-membro — não há nenhuma instrução de controle. A lógica de saída é totalmente encapsulada nas funções-membro da classe SalesPerson.

```

1 // Figura 9.5: SalesPerson.h
2 // Definição da classe SalesPerson.
3 // Funções-membro definidas em SalesPerson.cpp.
4 #ifndef SALES_P_H
5 #define SALES_P_H
6
7 class SalesPerson
8 {
9 public :
10 SalesPerson(); // construtor
11 void getSalesFromUser(); // insere as vendas a partir do teclado
12 void printAnnualSales(); // resulta em imprimir as vendas um mês específico
13 private :
14 double totalAnnualSales(); // protótipo para função utilitária
15 double sales[ 12]; // 12 estimativas de vendas mensais
16
17 }; // fim da classe SalesPerson
18
19 #endif

```

Figura 9.5 Definição da classe SalesPerson

```

1 // Figura 9.6: SalesPerson.cpp
2 // Funções-membro para a classe SalesPerson.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include "SalesPerson.h" // inclui definição da classe SalesPerson
13
14 // inicializa elementos do array sales como 0.0
15 SalesPerson::SalesPerson()
16 {
17     for ( int i = 0; i < 12; i++ )
18         sales[ i ] = 0.0;
19 } // fim do construtor SalesPerson
20
21 // obtém 12 estimativas de vendas do usuário no teclado
22 void SalesPerson::getSalesFromUser()
23 {
24     double salesFigure;
25
26     for ( int i = 1; i <= 12; i++ )
27     {
28         cout << "Enter sales amount for month " << i << ": ";
29         cin >> salesFigure;
30         setSales( i, salesFigure );
31     } // fim do for
32 } // fim da função getSalesFromUser
33
34 // configura uma das 12 estimativas de vendas mensais; a função subtrai
35 // um do valor mensal para o subscrito adequado no array sales
36 void SalesPerson::setSales( int month, double amount )
37 {
38     // testa a validade do mês e do valor
39     if ( month >= 1 && month <= 12 && amount > 0 )
40         sales[ month - 1 ] = amount; // ajusta para subscritos 0-11
41     else // mês ou valor inválido
42         cout << "Invalid month or sales figure" << endl;
43 } // fim da função setSales
44
45 // imprime o total das vendas anuais (com a ajuda da função utilitária)
46 void SalesPerson::printAnnualSales()
47 {
48     cout << setprecision( 2 ) << fixed
49     << "\nThe total annual sales are: $"
50 } // fim da função printAnnualSales() // chama a função utilitária
51
52 // função utilitária privada para somar vendas anuais
53 double SalesPerson::totalAnnualSales()
54 {
55     double total = 0.0; // inicializa o total
56
57

```

Figura 9.6 Definições de função-membro da classe SalesPerson

(continua)

```

58     for ( int i = 0; i < 12; i++ ) // resume resultados de vendas
59         total += sales[ i ];      // adiciona vendas do mês i ao total
60
61     return total;
62 } // fim da função totalAnnualSales

```

Figura 9.6 Definições de função-membro da classe SalesPerson

(continuação)

```

1 // Figura 9.7: fig09_07.cpp
2 // Demonstrando uma função utilitária.
3
4 // Compile esse programa com SalesPerson.cpp
5 // inclui a definição da classe SalesPerson a partir de SalesPerson.h
6 #include "SalesPerson.h"
7
8 int main()
9 {
10    SalesPerson s; // cria o objeto SalesPerson s
11
12    s.getSalesFromUser(); // anota código seqüencial simples;
13    s.printAnnualSales(); // nenhuma instrução de controle em main
14    return 0;
15 } // fim de main

```

```

Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03

Enter sales amount for month 5: 5838.45
Enter sales amount for month 6: 4439.22
Enter sales amount for month 7: 5893.57
Enter sales amount for month 8: 4909.67
Enter sales amount for month 9: 5123.45
Enter sales amount for month 10: 4024.97
Enter sales amount for month 11: 5923.92
Enter sales amount for month 12: 58120.59

The total annual sales are: $60120.59

```

Figura 9.7 Demonstração da função utilitária.



Observação de engenharia de software 9.8

Um fenômeno da programação orientada a objetos é que, uma vez que uma classe é definida, a criação e a manipulação de objetos dessa classe envolvem freqüentemente emitir apenas uma seqüência simples de chamadas de função-membro — poucas instruções de controle, se houver, são necessárias. Por contraste, é comum ter instruções de controle na implementação de funções-membro de uma classe.

9.6 Estudo de caso da classe Time: construtores com argumentos-padrão

O programa das figuras 9.8–9.10 apresenta e demonstra como os argumentos são implicitamente passados para um construtor. O construtor definido na Figura 9.8 iniciaizou o horário (isto é, meia-noite no horário universal). Como outras funções, os construtores podem especificar argumentos-padrão. A linha 13 da Figura 9.8 declara o construtor de incluir argumentos-padrão, especificando um valor-padrão de zero para cada argumento passado para o construtor. Na Figura 9.9, linhas 14–17 definem a nova versão do construtor de valores dos parâmetros que serão utilizados para

inicializar os membros de dados hour, minute e second respectivamente. Observa que essas funções set para cada membro de dados. O construtor chama as funções setHour, setMinute e setSecond para validar e atribuir valores aos membros de dados. Os argumentos-padrão para o construtor asseguram que, mesmo se nenhum valor for fornecido em uma chamada de construtor, o construtor ainda inicializará os membros de dados para manter o objeto consistente. Um construtor que assume os padrões para todos os seus argumentos também é um construtor-padrão — isto é, que pode ser invocado sem argumentos. Pode haver um máximo de um construtor-padrão por classe.

Na Figura 9.9, a linha 16 do construtor chama as funções-membro passados para o construtor (ou os valores-padrão). A função setHour para assegurar que o valor fornecido esteja no intervalo 0–23, em seguida chama setMinute e setSecond para assegurar que os valores de minute e second estejam cada um no intervalo 0–59. Se um valor estiver fora do intervalo, esse valor é configurado como zero (para assegurar que cada membro de dados permaneça em um estado consistente). Um construtor que assume os padrões para todos os seus argumentos também é um construtor-padrão — isto é, que pode ser invocado sem argumentos. Pode haver um máximo de um construtor-padrão por classe.

Observe que o construtor poderia ser escrito para incluir as mesmas instruções para cada função-membro. No entanto, isso seria redundante, pois cada função-membro chama a mesma função interna. Devido ao overhead de chamada, copiar o código das linhas 31, 37 e 43 no construtor eliminaria esse overhead de chamada. Codificar o construtor dessa forma ou a função-membro como uma cópia do código nas linhas 31, 37 e 43 tornaria a manutenção dessa classe mais difícil. Se as implementações de setHour, setMinute e setSecond precisarem ser alteradas, a implementação de qualquer função-membro que duplica as linhas 31, 37 e 43 teria de ser alterada de maneira correspondente. Fazer o construtor de fazer a chamada a setHour, setMinute e setSecond permite limitar as alterações no código que envolvem o construtor.

```

1 // Figura 9.8: Time.h
2 // Declaração da classe Time.
3 // Funções-membro definidas em Time.cpp.
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef TIME_H
7 #define TIME_H
8
9 // Definição de tipo de dados abstrato Time
10 class Time
11 {
12 public :
13 Time( int = 0, int = 0, int = 0); // construtor-padrão
14
15 // funções set
16 void setTime( int , int , int ); // configura hour, minute, second
17 void setHour( int ); // configura hour (depois da validação)
18 void setMinute( int ); // configura minute (depois da validação)
19 void setSecond( int ); // configura second (depois da validação)
20
21 // funções get
22 int getHour(); // retorna hour
23 int getMinute(); // retorna minute
24 int getSecond(); // retorna second
25
26 void printUniversal(); // gera saída da hora no formato universal de data/hora
27 private :printStandard(); // gera saída da hora no formato-padrão de data/hora
28 int hour; // 0 - 23 (formato de relógio de 24 horas)
29 int minute; // 0 - 59
30 int second; // 0 - 59
31
32 }; // fim da classe Time
33
34 #endif

```

Figura 9.8 A classe Time contendo um construtor com argumentos-padrão.

com a função correspondente. Isso reduz a probabilidade de erros ao alterar a implementação da classe. Além disso, o desempenho do construtor de `Time` pode ser aprimorado declarando-os explicitamente e removendo-os na definição de classe (que torna a definição de função implicitamente inline).



Observação de engenharia de software 9.9

Se a função-membro de uma classe já fornecer toda ou parte da funcionalidade requerida por um construtor (ou por outra função-membro) da classe, chame essa função-membro a partir do construtor (ou de outra função-membro). Isso simplifica a manutenção do código e reduz a probabilidade de um erro se a implementação do código for modificada. Como regra geral, evite a repetição de código.



Observação de engenharia de software 9.10

Quando gerar código para valores de argumento? - padrão de uma função exige que o código-cliente seja recompilado (para assegurar

```

1 // Figura 9.9: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setw;
8 using std::setw;
9
10 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
11
12 // Construtor de Time inicializa cada membro de dados como zero;
13 // assegura que os objetos Time iniciem em um estado consistente
14 Time::Time( int hr, int min, int sec )
15 {
16     setTime( hr, min, sec ); // valida e configura time
17 } // fim do construtor de Time
18
19 // configura novo valor de Time utilizando a hora universal; assegura que
20 // os dados permaneçam consistentes configurando valores inválidos como zero
21 void Time::setTime( int h, int m, int s )
22 {
23     setHour( h ); // configura campo private hour
24     setMinute( m ); // configura campo private minute
25     setSecond( s ); // configura campo private second
26 } // fim da função setTime
27
28 // configura valor de hour
29 void Time::setHour( int h )
30 {
31     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
32 }
33 } // fim da função setHour
34 // configura valor de minute
35 void Time::setMinute( int m )
36 {
37     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
38 } // fim da função setMinute
39

```

Figura 9.9 Definições de função-membro da classe Time, incluindo um construtor que aceita argumentos.

(continua)

```

40 // configura valor de second
41 void Time::setSecond( int s )
42 {
43     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
44 } // fim da função setSecond
45
46 // retorna valor de hour
47 int Time::getHour()
48 {
49     return hour;
50 } // fim da função getHour
51
52 // retorna valor de minute
53 int Time::getMinute()
54 {
55     return minute;
56 } // fim da função getMinute
57
58 // retorna valor de second
59 int Time::getSecond()
60 {
61     return second;
62 } // fim da função getSecond
63
64 // imprime a hora no formato universal de data/hora (HH:MM:SS)
65 void Time::printUniversal()
66 {
67     cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
68         << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
69 } // fim da função printUniversal
70
71 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
72 void Time::printStandard()
73 {
74     cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
76         << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
77 } // fim da função printStandard

```

Figura 9.9 Definições de função-membro da classe Time, incluindo um construtor que aceita argumentos.

(continuação)

A função `main` na Figura 9.10 inicializa cinco **objetos** com os três argumentos convertidos para sua configuração-padrão na chamada de construtor implícita (linha 11), um com um argumento especificado (linha 12), um com dois argumentos especificados (linha 13), um com três argumentos especificados (linha 14) e um com três argumentos inválidos especificados (linha 15). Em seguida, o programa exibe cada objeto nos formatos de data/hora universal e data/hora padrão.

Notas relacionadas com as funções `set` e o construtor da classe `Time`

As funções `set` e `setHour`, `setMinute` e `setSecond` da classe `Time` são chamadas por todo o corpo da classe. Em particular, nas linhas 21–26 da Figura 9.9, chama as funções `setHour`, `setMinute` e `setSecond` e as funções `printUniversal` e `printStandard` chamam as funções `getHour`, `getMinute` e `getSecond` nas linhas 67–68 e 74–76, respectivamente. Em cada caso, essas funções poderiam ter acessado diretamente os dados internos da classe sem chamar as funções `set`. Considere a possibilidade de alterar a representação da hora de três valores (requerendo 12 bytes de memória) para uma única variável que represente o número total de segundos que se passou desde a meia-noite (requerendo quatro bytes de memória). Se fizéssemos essa alteração, somente o corpo das funções que accedem diretamente precisaria mudar — em particular, `setHour`, `setMinute` e `setSecond`. Não há nenhuma necessidade de modificar o corpo das funções `printUniversal` ou `printStandard`, porque elas não acessam os dados diretamente. Projetar a classe dessa maneira reduz a probabilidade de erros de programação ao alterar a implementação da classe.

```

1 // Figura 9.10: fig09_10.cpp
2 // Demonstrando um construtor-padrão para a classe Time.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
8
9 int main()
10 {
11     Time t1;    // todos os argumentos convertidos para sua configuração-padrão
12     Time t2( 2); // hour especificada; minute e second convertidos para o padrão
13     Time t3( 21, 34); // hour e minute especificados; second convertido para o padrão
14     Time t4( 12, 25, 42); // hour, minute e second especificados
15     Time t5( 27, 74, 99 ); // valores inválidos especificados
16
17     cout << "Constructed with:\n\tt1: all arguments defaulted\n\t\t";
18     t1.printUniversal();           // 00:00:00
19     cout << "\n\tt2: hour specified; minute and second defaulted\n\t\t";
20     t1.printStandard();          // 12:00:00 AM
21
22     cout << "\n\tt3: hour and minute specified; second defaulted\n\t\t";
23     t2.printUniversal();          // 02:00:00
24     cout << "\n\tt4: hour, minute and second specified\n\t\t";
25     t2.printStandard();          // 2:00:00 AM
26
27     cout << "\n\tt5: all invalid values specified\n\t\t";
28     t3.printUniversal();          // 21:34:00
29     cout << "\n\tt6: ";
30     t3.printStandard();          // 9:34:00 PM
31
32     cout << "\n\tt7: ";
33     t4.printUniversal();          // 12:25:42
34     cout << "\n\tt8: ";
35     t4.printStandard();          // 12:25:42 PM
36
37     cout << "\n\tt9: ";
38     t5.printUniversal();          // 00:00:00
39     cout << "\n\tt10: ";
40     t5.printStandard();          // 12:00:00 AM
41     cout << endl;
42     return 0;
43 } // fim de main

```

Constructed with:

t1: all arguments defaulted
00:00:00
12:00:00 AM

t2: hour specified; minute and second defaulted
02:00:00
2:00:00 AM

Construtor com argumentos-padrão.

(continua)

t3: hour and minute specified; second defaulted

21:34:00

9:34:00 PM

t4: hour, minute and second specified

12:25:42

12:25:42 PM

t5: all invalid values specified

00:00:00

12:00:00 AM

[Construtor com argumentos-padrão.](#)

(continuação)

De maneira semelhante, o construtor pode ser escrito para incluir uma cópia das instruções apropriadas a partir da função setTime. Fazer isso talvez seja um pouco mais eficiente, porque a chamada extra para o construtor é evitada. Entretanto, duplicar instruções em múltiplas funções ou construtores dificulta a alteração da representação interna de dados da classe. Fazer o construtor chamar a função diretamente requer que qualquer alteração na implementação de setTime seja feita somente uma vez.



Erro comum de programação 9.2

Um construtor pode chamar outras funções-membro da classe, mas só pode fazê-lo quando está inicializando o objeto, os membros de dados ainda podem não estar em um estado consistente. Utilizar os membros de dados antes de eles serem adequadamente inicializados pode causar erros de lógica.

9.7 Destrutores

Um destrutor é outro tipo de função-membro especial. O nome do destrutor é determinado pelo nome de

o construtor correspondente, com o sufixo ~. Por exemplo, se a classe Time tem um construtor com a abreviação 'ctor' na literatura. Preferimos não utilizar essa abreviação.

O destrutor de uma classe é chamado implicitamente quando um objeto é destruído. Isso ocorre, por exemplo, logo que um objeto é destruído quando a execução do programa deixa o escopo em que ele foi criado. O destrutor é chamado quando a memória do objeto faz a sua de terminação antes de o sistema reivindicar a memória do objeto, assim a memória pode ser reutilizada para armazenar novos objetos.

Um destrutor não recebe parâmetros nem retorna um valor. Um destrutor pode não especificar um tipo de retorno — nem mesmo void. Uma classe pode ter somente um único destrutor — a sobrecarga de destrutor não é permitida.



Erro comum de programação 9.3

É um erro de sintaxe tentar passar argumentos para um destrutor, especificar um tipo de retorno para um destrutor (mesmo void não pode ser especificado), retornar valores de um destrutor ou sobrepor um destrutor.

Apesar de não fornecermos destrutores para as classes apresentadas até agora, toda classe tem um destrutor. Se o programador não fornecer um destrutor explicitamente, o compilador cria um destrutor vazio se o destrutor criado implicitamente, de fato, realiza operações importantes em objetos criados por meio de composição (Capítulo 10) e herança (Capítulo 12). No Capítulo 11, construiremos destrutores apropriados para classes cujos objetos contêm memória alocada dinamicamente (por exemplo, em arquivos).

atrigos e de estabeleceram diretrizes para sistemas dinâmicos do sistema (Capítulo 17), que estudaremos no Capítulo 17. Discutimos



Observação de engenharia de software 9.11

Como veremos no restante do livro, construtores e destrutores têm uma proeminência muito maior em C++ e na programação orientada a objetos do que a ensinada aqui apenas com a nossa breve introdução.

9.8 Quando construtores e destrutores são chamados

Os construtores e destrutores são chamados implicitamente pelo compilador. A ordem em que essas chamadas de função ocorre depende da ordem em que a execução entra e sai dos escopos em que os objetos são instanciados. Geralmente, as chamadas de construtor são feitas na ordem inversa das chamadas de construtor correspondentes, mas, como veremos nas figuras 9.11–9.13, as classes de armazenamento de objetos podem alterar a ordem em que destrutores são chamados.

Os construtores são chamados para objetos definidos no escopo global antes de quaisquer outras funções (incluindo começar a execução (embora a ordem de execução de construtores de objeto global entre arquivos não seja garantida). Os destrutores correspondentes serão chamados na ordem inversa. A função `exit()` força um programa a terminar imediatamente e não executa os destrutores de objetos automáticos. A função é frequentemente utilizada para terminar um programa quando um erro é detectado na entrada ou quando não é possível abrir um arquivo a ser processado. A função `atexit()` é semelhante à função `exit()`, mas força o programa a terminar imediatamente, sem permitir que os destrutores de quaisquer objetos sejam chamados. A função `abort()` é normalmente utilizada para indicar uma terminação anormal do programa. (Ver o Capítulo 24, “Outros tópicos”, para obter informações adicionais sobre as funções)

O construtor de um objeto local automático é chamado quando a execução alcança o ponto em que o objeto é definido — o construtor correspondente é chamado quando a execução deixa o escopo do objeto (isto é, quando o bloco em que esse objeto é definido termina a execução). Os construtores e destrutores para objetos automáticos são chamados toda vez que a execução entra e sai do escopo do objeto. Os destrutores não serão chamados para objetos automáticos se o programa terminar com uma chamada à função `exit()`.

O construtor de um objeto global é chamado uma única vez, logo que a execução alcança o ponto em que o objeto foi definido — o destrutor correspondente é chamado quando o programa termina. Os objetos globais e locais são destruídos na ordem inversa de sua criação. Os destrutores não são chamados para os objetos que são destruídos na ordem inversa de sua criação.

O programa das figuras 9.11–9.13 demonstra a ordem em que os construtores e destrutores são indicados para os objetos da classe `CreateAndDestroy` (figuras 9.11 e 9.12) de várias classes de armazenamento em diversos escopos. Cada objeto da classe `CreateAndDestroy` contém (linhas 16–17) um membro `objectID` (message) que são utilizados na saída do programa para identificar o objeto. Esse exemplo mecânico é puramente para propósitos pedagógicos. Por essa razão, a linha 23 do destrutor na Figura 9.13 determina se o objeto que está sendo destruído é o último a ser criado, gera saída de um caractere de nova linha. Essa linha ajuda a tornar a saída do programa mais fácil de seguir.

A Figura 9.13 define o objeto (linha 12) no escopo global. Seu construtor na realidade é chamado antes que qualquer instrução em seu escopo seja executada e seu destrutor é chamado na terminação do programa depois de os destrutores de todos os outros objetos serem executarem.

A função `main` (linhas 14–26) declara três objetos da classe `CreateAndDestroy`. Nesse escopo, o construtor é chamado quando a execução alcança o ponto em que esse objeto é declarado. Os destrutores são chamados (isto é, o inverso da ordem em que seus construtores foram chamados) quando a execução deixa o escopo. No entanto, ele existe até a terminação do programa. O destrutor é chamado antes do destrutor para os objetos declarados em todos os outros escopos serem destruídos.

```

1 // Figura 9.11: CreateAndDestroy.h
2 // Definição da classe CreateAndDestroy.
3 // Funções-membro definidas em CreateAndDestroy.cpp.
4 #include <string>
5 using std::string;
6
7 #ifndef CREATE_H
8 #define CREATE_H
9
10 class CreateAndDestroy
11 public :
12
13 CreateAndDestroy( int , string ); // construtor
14 ~CreateAndDestroy(); // destrutor
15 private :
16     int objectID; // Número de ID do objeto

```

Figura 9.11 Definição da classe `CreateAndDestroy`

(continua)

```

17     string message; // mensagem descrevendo o objeto
18 } // fim da classe CreateAndDestroy
19
20 #endif

```

Figura 9.11 Definição da classe CreateAndDestroy

(continuação)

```

1 // Figura 9.12: CreateAndDestroy.cpp
2 // Definições de função-membro da classe CreateAndDestroy.
3 #include <iostream>
4 using std::cout;
5
6
7 #include "CreateAndDestroy.h" // inclui a definição da classe CreateAndDestroy
8
9 // construtor
10 CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
11 {
12     objectID = ID; // configura o número de ID do objeto
13     message = messageString // configura mensagem descritiva do objeto
14
15     cout << "Object " << objectID << " constructor runs "
16     << message << endl;
17 } // fim do construtor CreateAndDestroy
18
19 // destrutor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22     // para saída de uma linha para os objetos ajuda a legibilidade
23
24     cout << "Object " << objectID << " destructor runs "
25     << message << endl;
26 } // fim do destrutor ~CreateAndDestroy

```

Figura 9.12 Definições de função-membro da classe CreateAndDestroy

```

1 // Figura 9.13: fig09_13.cpp
2 // Demonstrando a ordem em que construtores e
3 // destrutores são chamados.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "CreateAndDestroy.h" // inclui a definição da classe CreateAndDestroy
9 void create( void ); // protótipo
10
11 CreateAndDestroy first( 1, "(global before main)" ); // objeto global
12
13 int main()
14 {

```

Figura 9.13 A ordem em que construtores e destrutores são chamados.

(continua)

```

16    cout << "\nMAIN FUNCTION: EXECUTION BEGINS";
17    CreateAndDestroy second( 2, "(local automatic in main)" );
18    static CreateAndDestroy third( 3, "(local static in main)" );
19
20    create(); // chama função para criar objetos
21
22    cout << "\nMAIN FUNCTION: EXECUTION RESUMES";
23    CreateAndDestroy fourth( 4, "(local automatic in main)" );
24    cout << "\nMAIN FUNCTION: EXECUTION ENDS";
25    return 0;
26 } // fim de main
27
28 // função para criar objetos
29 void create( void )
30 {
31    cout << "\nCREATE FUNCTION: EXECUTION BEGINS";
32    CreateAndDestroy fifth( 5, "(local automatic in create)" );
33    static CreateAndDestroy sixth( 6, "(local static in create)" );
34    CreateAndDestroy seventh( 7, "(local automatic in create)" );
35    cout << "\nCREATE FUNCTION: EXECUTION ENDS";
36 } // fim da função create

```

Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS

Object 2 constructor runs (local automatic in main)
Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS

Object 5 constructor runs (local automatic in create)
Object 6 constructor runs (local static in create)

Object 7 destructor runs (local automatic in create)
Object 5 destructor runs (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES

Object 4 constructor runs (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS

Object 4 destructor runs (local automatic in main)
Object 2 destructor runs (local automatic in main)

Object 6 destructor runs (local static in create)
Object 3 destructor runs (local static in main)

Object 1 destructor runs (global before main)

Figura 9.1 A ordem em que construtores e destrutores são chamados.

(continuação)

A função `create` (linhas 29–36) declara três objetos (`fifth` (linha 32), `sixth` (linha 33) e `seventh` (linha 34) como objetos automáticos locais e um objeto `static` (`sixth`). Os destrutores desses objetos, em seguida, são chamados (isto é, a ordem inversa em que seus construtores foram chamados). Quando o bloco `static`, ele existe até a terminação do programa. O destrutor `seventh` é chamado antes dos destrutores `sixth` e `fifth`, assim, depois de todos os outros serem destruídos.

9.9 Estudo de caso da classe Time: uma armadilha sutil — retornar uma referência a um membro de dados `private`

Uma referência a um objeto é um alias para o nome do objeto e, portanto, pode ser utilizado à esquerda de uma instrução de atribuição. Nesse contexto, a referência é perfeitamente aceitável que pode receber um valor. Uma maneira de utilizar essa capacidade (infelizmente!) é fazer uma função-membro classe retornar uma referência a um `dados de classe`. Observe que, se uma função retornar uma `referência`, pode ser utilizada `abuso` modifável.

O programa das figuras 9.14–9.16 utiliza a classe Time (figuras 9.14 e 9.15) para demonstrar o retorno de uma referência a um membro de dados a função-membro `hour` (declarada na Figura 9.14 na linha 15 e definida na Figura 9.15 nas linhas 29–33). Esse retorno de referência na realidade torna `Time` uma `chamada à função-membro` o membro de dados `hour`! A chamada de função pode ser utilizada de todas as maneiras que o membro de dados pode ser utilizado, inclusive `atualizar` uma instrução de atribuição, permitindo, assim, que os clientes da classe sobrescrevam accidentalmente os dados da classe à vontade! Observe que o mesmo problema ocorria `seu` ponteiro para os dados.

Figura 9.14: O código de Time.h (linhas 12) e a referência (linha 15), que é inicializada com a referência retornada pela chamada `badSetHour(20)`. A linha 17 exibe o valor do alias, que mostra como esse quebra o encapsulamento da classe — as instruções não devem ter acesso aos dados da classe. Em seguida, a linha 18 utiliza o alias para configurar o valor `hour` como 30 (um valor inválido) e a linha 19 exibe o valor `retornado` para mostrar que atribuir um valor a `hour` realmente modifica os dados do objeto `t`. Por fim, a linha 23 utiliza a própria chamada de função `badSetHour` como `value` atribui 74 (outro valor inválido) à referência retornada pela função. A linha 28 exibe novamente o valor retornado pela função para mostrar que atribuir um valor ao resultado da chamada de função na linha 23 modifica os dados `private` no objeto `t`.

```

1 // Figura 9.14: Time.h
2 // Declaração da classe Time.
3 // Funções-membro definidas em Time.cpp
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef TIME_H
7 #define
8
9 class Time
10 {
11 public :
12     Time( int = 0, int = 0, int = 0 );
13     void setTime( int , int , int );
14     int getHour();
15     int &badSetHour( int ); // retorno de referência PERIGOSO
16 private :
17     int hour;
18     int minute;
19     int second;
20 }; // fim da classe Time
21
22 #endif

```

Figura 9.14 Retorno uma referência a um membro de `private`.

```

1 // Figura 9.15: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include "Time.h" // inclui definição da classe Time
4
5 // função construtor para inicializar dados private:

```

Figura 9.15 Retorno uma referência a um membro de `private`.

(continua)

```

6 // chama a função-membro setTime para configurar variáveis;
7 // valores-padrão são 0 (ver definição de classe)
8 Time::Time( int hr, int min, int sec )
9 {
10    setTime( hr, min, sec );
11 } // fim do construtor de Time
12
13 // configura valores de hour, minute e second
14 void Time::setTime( int h, int m, int s )
15 {
16    hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
17    minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
18    second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
19 } // fim da função setTime
20
21 // retorna valor de hour
22 int Time::getHour()
23 {
24    return hour;
25 } // fim da função getHour
26
27 // PRÁTICA DE PROGRAMAÇÃO RUIM:
28 // Retornar uma referência a um membro de dados private.
29 int &Time::badSetHour( int hh )
30 {
31    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
32    return hour; // retorno de referência PÉRIGOSO
33 } // fim da função badSetHour

```

Figura 9.15 Retornando uma referência a um membro de `private`.

(continuação)

```

1 // Figura 9.16: fig09_16.cpp
2 // Demonstrando uma função-membro public que
3 // retorna uma referência a um membro de dados private.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // inclui definição da classe Time
9
10 int main()
11 {
12    Time t; // cria o objeto Time
13
14    // inicializa hourRef com a referência retornada por badSetHour
15    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
16
17    cout << "Valid hour before modification: " << hourRef;
18    hourRef = 30; // use hourRef to set invalid value in Time object t
19    cout << "\nInvalid hour after modification: " << t.getHour();
20
21    // Dangerous: Function call that returns
22    // a reference can be used as an lvalue!
23    t.badSetHour( 12 ) = 74; // assign another invalid value to hour

```

Figura 9.16 Retornando uma referência a um membro de `private`.

(continua)

```

24
25     cout << "\n*****\n"
26     << "POOR PROGRAMMING PRACTICE!!!!!!\n"
27     << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
28     << t.getHour()
29     << "\n*****"                                << endl;
30     return 0;
31 } // fim de main

```

Valid hour before modification: 20
 Invalid hour after modification: 30

```

*****PROGRAMMING PRACTICE!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*****

```

Figura 9.16 Retornando uma referência a um membro de `private`.

(continuação)

Dica de prevenção de erro 9.4

Retornar uma referência ou um ponteiro para um membro de dados encapsulado da classe é considerado uma prática perigosa que deve ser evitada.

9.10 Atribuição-padrão de membro a membro

O operador de atribuição (`=`) é utilizado para atribuir um objeto a outro objeto do mesmo tipo. Por padrão, tal atribuição é realizada pela [atribuição de membro a membro](#) para cada membro de dados do objeto à direita do operador de atribuição é atribuído individualmente ao membro correspondente do lado esquerdo. Figura 9.17 mostra a implementação de Date para o padrão de atribuição de membro a membro.

```

1 // Figura 9.17: Date.h
2 // Declaração da classe Date.
3 // Funções-membro são definidas em Date.cpp
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef DATE_H
7 #define DATE_H
8
9 // definição da classe Date
10 class Date
11 {
12 public :
13     Date( int = 1, int = 1, int = 2000); // construtor-padrão
14     void :print();
15     int month;
16     int day;
17     int year;
18 };
19 } // fim da classe Date
20
21 #endif

```

Figura 9.17 Arquivo de cabeçalho da classe Date

```
1 // Figura 9.18: Date.cpp
2 // Definições de função-membro da classe Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // inclui a definição da classe Date a partir de Date.h
8
9 // construtor Date (deve fazer verificação de intervalo)
10 Date::Date( int m, int d, int y )
11 {
12     month = m;
13     day = d;
14     year = y;
15 } // fim do construtor Date
16
17 // imprime Date no formato mm/dd/aaaa
18 void Date::print()
19 {
20     cout << month << '/' << day << '/' << year;
21 } // fim da função print
```

Figura 9.18 Definições de função-membro da classe Date

do objeto `date1` aos membros de dados correspondentes de `date2`. Neste caso, o membro `date1` é atribuído ao membro `date2`, o membro `date1` é atribuído ao membro `date2` e o membro `date1` é atribuído ao membro `date2`. [Atenção: atribuição de membro a membro pode causar problemas sérios quando utilizada com uma classe cujos membros de dados contêm ponteiros para a memória alocada dinamicamente; discutimos esses problemas no Capítulo 11 trazemos como lidar com eles.] Note que não existem nenhuma verificação de erros; deixamos isso para os exercícios.

São criados poderosos passos que permitem o uso de argumentos e pode ser usado para definir passos de função. Essas passagens permitem que o novo objeto e utilizam `um construtor de cópia` para copiar os valores do objeto original para o novo objeto. Para cada classe, o compilador fornece um construtor de cópia-padrão que copia cada membro do objeto original para o membro correspondente do novo objeto. A atribuição de membro a membro, os construtores de cópia podem causar sérios problemas quando utilizados com uma classe cujos membros de dados contêm ponteiros para memória dinamicamente alocada. O Capítulo 11 discute como programadores podem definir um construtor de cópia personalizado que copia de maneira adequada os objetos contendo ponteiros para a memória dinamicamente alocada.

```
1 // Figura 9.19: fig09_19.cpp
2 // Demonstrando que os objetos de classe podem ser atribuídos
3 // um ao outro utilizando atribuição-padrão de membro a membro.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Date.h" // inclui a definição da classe Date a partir de Date.h
9
10 int main()
11 {
12     Date date1( 7, 4, 2004);
13     Date date2; // date2 assume padrão de 1/1/2000
14
15     cout << "date1 = " ;
```

FiguAtribuição-padrão de membro a membro.

(continua)

```

16     date1.print();
17     cout << "\ndate2 = ";
18     date2.print();
19
20     date2 = date1; // atribuição-padrão de membro a membro
21
22     cout << "\n\nAfter default memberwise assignment, date2 = " ;
23     date2.print();
24     cout << endl;
25     return 0;
26 } // fim de main

```

date1 = 7/4/2004

After default memberwise assignment, date2 = 7/4/2004

Figura 9.10 Atribuição-padrão de membro a membro.

(continuação)



Dica de desempenho 9.3

A passagem de um objeto por valor é boa de um ponto de vista de segurança, porque a função chamada não tem acesso ao original no chamador, mas pode degradar o desempenho ao fazer uma cópia de um objeto grande. Um objeto pode ser passado por referência passando um ponteiro ou uma referência para o objeto. A passagem por referência oferece bom desempenho, mas é mais fraca de um ponto de vista de segurança, porque a função chamada recebe acesso ao objeto original. A passagem por referência consta é uma alternativa segura de bom desempenho (essa pode ser implementada `const` ou `const` parâmetro de referência) um parâmetro de ponteiro `const` ou `const`

As pessoas que escrevem programas orientados a objetos se concentram na implementação de classes úteis. Há uma enorme rede de capturar e catalogar classes para que possam ser acessadas por grandes segmentos da comunidade de programação. Existem bibliotecas de classe substanciais e outras estão sendo desenvolvidas em todo o mundo. Os softwares estão sendo construídos, mais, a partir de componentes existentes, bem definidos, cuidadosamente testados, bem documentados, portáveis, de alto desempenho e amplamente disponíveis. Esse tipo de reusabilidade de software acelera o desenvolvimento de software poderoso e de alta qualidade. O desenvolvimento rápido de aplicativos (*rapid applications development*, RAD) por meio dos mecanismos de componentes reutilizáveis tornou-se um campo importante.

Entretanto, problemas significativos devem ser resolvidos antes de o potencial de reusabilidade de software ser realizado. Precisamos de esquemas de catalogação e licenciamento, mecanismos de proteção para assegurar que cópias-mestre de classes não são rompidas, esquemas de descrição para que designers de novos sistemas possam determinar facilmente se objetos existentes atendem suas necessidades, mecanismos de navegação para determinar quais classes estão disponíveis e o rigor com que atendem aos requisitos do desenvolvedor de software etc. Muitos problemas interessantes de pesquisa e desenvolvimento precisam ser resolvidos. Há uma grande motivação em resolver esses problemas, porque o valor potencial de suas soluções é enorme.

9.12 Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)

Neste capítulo, estudaremos o projeto de engenharia de software do sistema ATM. Anteriormente, no Capítulo 3, introduzimos o conceito de herança. Neste capítulo, vamos implementar o projeto de software com classes C++. Agora, iniciaremos a implementação do nosso projeto orientado a objetos em C++. No final desta seção, mostraremos como converter diagramas de classes em arquivos de cabeçalho C++. No final da seção “Estudo de caso de engenharia de software” (Seção 13.10), modificaremos os arquivos de cabeçalho para incorporar o conceito orientado a objetos de herança. Apresentaremos a completa implementação de código C++ no Apêndice G.

Visibilidade

Agora, aplicamos especificadores de acesso aos membros das nossas classes. No Capítulo 3, introduzimos os especificadores `public` e `private`. Os modificadores de acesso determinam a visibilidade dos atributos e operações de um objeto a

outros objetos. Antes de iniciarmos a implementação do nosso projeto, devemos considerar quais atributos e operações das nossas classes devem ser públicas e quais devem ser privadas.

No Capítulo 3, observamos que membros de dados devem ser privados e membros invocadas por clientes de uma dada classe devem ser públicos. Entretanto, as funções-membro chamadas apenas por outras funções-membro da classe como ‘funções utilitárias’ devem ser privadas. A UML emprega marcadores de visibilidade para modelar a visibilidade dos atributos e operações. Visibilidade pública é indicada colocando-se um sinal de adição (+) antes de uma operação ou atributo; um sinal de subtração (-) indica visibilidade privada. A Figura 9.20 mostra nosso diagrama de classes atualizado com marcadores de visibilidade incluídos. [Nota: Não incluímos parâmetros de operação na Figura 9.20. Isso é perfeitamente normal. Adicionar marcadores de visibilidade não afeta os parâmetros já modelados nos diagramas de classes das figuras 6.22–6.25.]

Navegabilidade

Antes de começarmos a implementar nosso projeto em C++, apresentaremos uma notação da UML adicional. O diagrama de classes na Figura 9.21 refina ainda mais os relacionamentos entre as classes no sistema ATM adicionando setas de naveabilidade às descrições.

UML pode ser percorrido de várias maneiras, mas a melhor é a navegação sequencial (Figura 7.12). Ao implementar um sistema projetado utilizando a UML, os programadores utilizam setas de naveabilidade para ajudar a terminar os objetos que precisam de referências ou ponteiros para outros objetos. Por exemplo, a seta de naveabilidade que aponta de `BankDatabase` para a classe `Account` indica que podemos navegar da primeira à última, permitindo que o programador que a classe `BankDatabase` para a classe `Account` saiba que as associações em um diagrama de classes que têm setas de naveabilidade nas duas extremidades ou que simplesmente não têm setas de naveabilidade indicam bidirecionalidade — a navegação pode acontecer em qualquer direção pela associação.

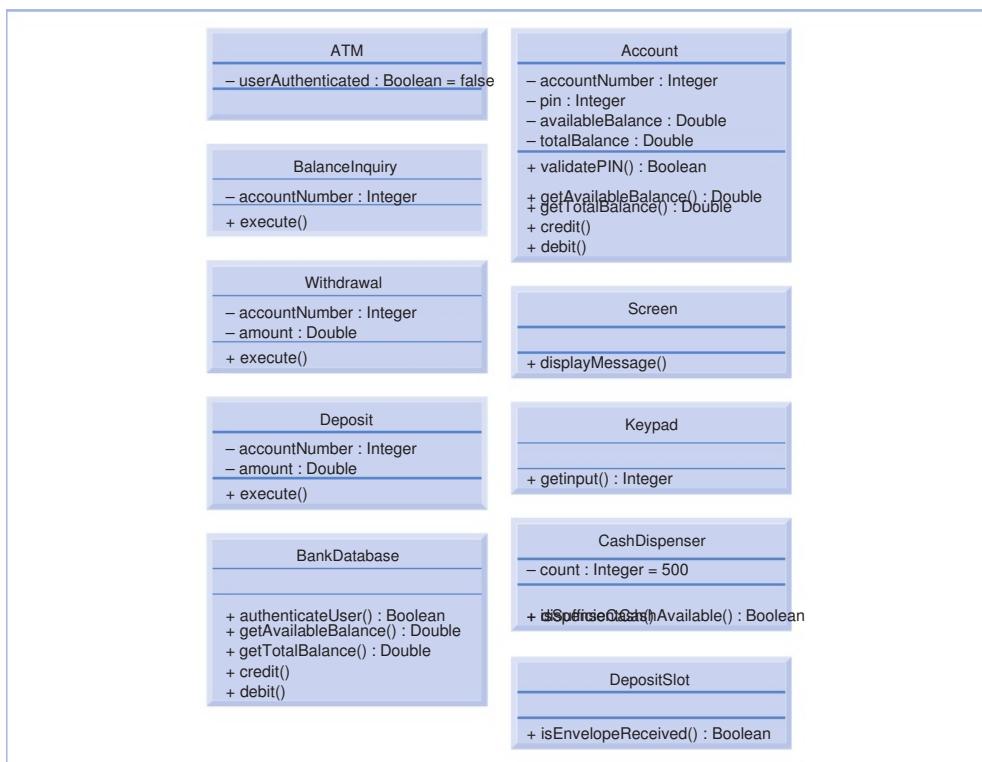


Figura 9.20 Diagrama de classes com marcadores de visibilidade.

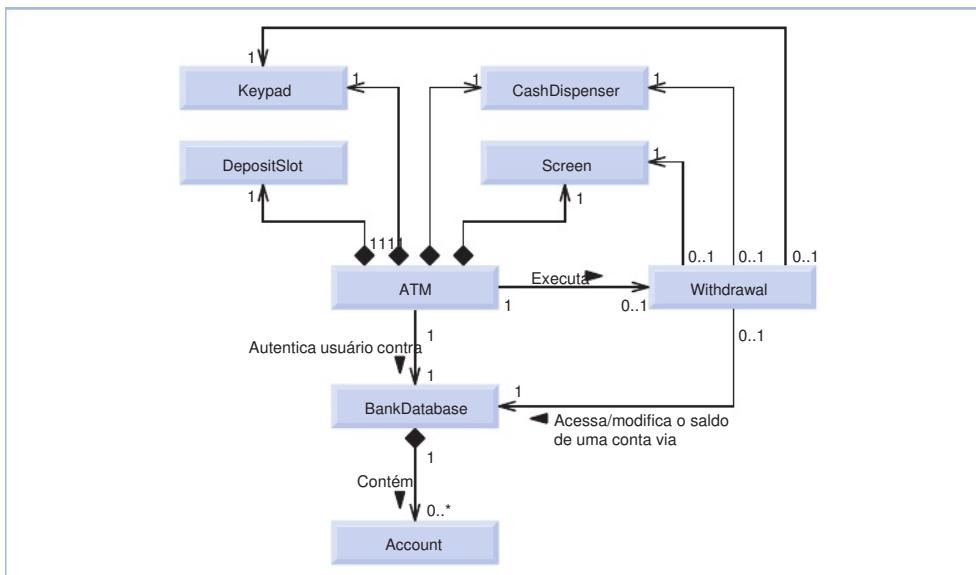


Figura 9.21 Diagrama de classes com setas de navegabilidade.

Como ocorre com o diagrama de classes da Figura 3.23, o diagrama de classes da Figura 9.21 omite as classes **Deposit** e **Withdrawal** para manter o diagrama simples. A navegabilidade das associações de que essas classes participam é bem parecida com a do diagrama de classes da Figura 3.23. [Nota: Modelamos essas classes e associações adicionais no nosso diagrama de classes final na Seção 13.10, depois de simplificarmos a estrutura do nosso sistema incorporando o conceito de herança ao projeto orientado a objetos.]

Implementando o sistema ATM a partir de seu projeto em UML

Agora, estamos prontos para começar a implementar o sistema ATM. Primeiro convertemos as classes nos diagramas das figuras 9.21 em arquivos de cabeçalho C++. Esse código representará o 'esqueleto' do sistema. No Capítulo 13, modificamos os arquivos de cabeçalho para incorporar o conceito de herança orientado a objetos. No Apêndice G, "Código para o estudo de caso do ATM", sentarmos o código C++ funcional completo para nosso modelo.

Como um exemplo, começamos a desenvolver o arquivo **Withdrawal.h** da Figura 9.20. Utilizamos essa figura para determinar os atributos e operações da classe. Utilizamos o modelo da UML na Figura 9.21 para determinar as associações entre as classes. Seguimos as cinco diretrizes a seguir para cada classe:

- Utilize o nome localizado no primeiro compartimento de uma classe em um diagrama de classes para definir a classe em um arquivo de cabeçalho (Figura 9.22). Isso evita que você defina a classe para impedir que o compilador quebre o projeto.
- Utilize os atributos localizados no segundo compartimento da classe para declarar os membros de dados. Por exemplo, atributos `private accountNumber` e `amount` da classe `Withdrawal` produzem o código na Figura 9.23.
- Utilize as associações descritas no diagrama de classes para declarar referências (ou ponteiros, onde apropriado) a outras classes. Por exemplo, de acordo com a Figura 9.20, pode acessar um objeto `ATM` a partir de um objeto da classe `Withdrawal`. A classe `Withdrawal` deve manter handles nesses objetos para enviar-lhes mensagens, portanto, as linhas 19–22 da Figura 9.24 declaram quatro referências como membros `dados` `private`. Na implementação da classe no Apêndice G, um construtor inicializa esses membros de dados.

```

1 // Figura 9.22: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Withdrawal
7 {
8 };
9 // fim da classe Withdrawal
10#endif // WITHDRAWAL_H

```

Figura 9.22 Definição da classe `Withdrawal` incluída em empacotadores de pré-processador.

```

1 // Figura 9.23: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Withdrawal
7 {
8 private :
9     // atributos
10    int accountNumber;/ conta a sacar fundos
11    double amount; // quantia a sacar
12 };
13 // fim da classe Withdrawal
14#endif // WITHDRAWAL_H

```

Figura 9.23 Adicionando atributos ao arquivo de cabeçalho de `Withdrawal`

com referência a objetos reais. Observe que as linhas `#include <Keypad.h>`, `<BankDatabase.h>` e `<CashDispense.h>` (de cabeçalho contendo as definições das classes `Keypad`, `CashDispense` e `BankDatabase`) foram omitidas para podermos declarar as referências aos objetos dessas classes nas linhas 19–22.

4. Acabamos descobrindo que incluir os arquivos de cabeçalho às classes `Keypad`, `CashDispense` e `BankDatabase` na definição da classe `Withdrawal` faz mais do que é necessário. A classe `Withdrawal` não tem referências a objetos dessas classes — ela não contém objetos reais — e a quantidade de informações requeridas pelo compilador para criar uma referência difere daquela que requerida para criar um objeto. Lembre-se de que criar um objeto requer fornecer ao compilador uma definição da classe com o nome da classe como um novo tipo definido pelo usuário e que indique os membros de dados que determinam a quantidade de memória necessária para armazenar referência (ou ponteiro) a um objeto, porém, requer somente que o compilador saiba que a classe do objeto existe — ele não precisa saber o tamanho do objeto. Qualquer referência (ou ponteiro), independentemente da classe do objeto à qual ela se refere, contém apenas o endereço de memória do objeto real. A quantidade de memória requerida para o armazenamento de um endereço é uma característica física do hardware computador. Assim o compilador sabe o tamanho de qualquer referência (ou ponteiro). Como resultado, é desnecessário incluir o arquivo de cabeçalho completo de uma classe ao declarar apenas uma referência a um objeto dessa classe — precisa declarar a classe e suas referências. Onde só precisamos informar o tamanho de dados de memória que o compilador já sabe o tamanho de um arquivo de cabeçalho contendo referências ou ponteiros para uma classe, mas que a definição da classe reside fora do arquivo de cabeçalho. Podemos substituir a definição da classe `Withdrawal` (linhas 6–9 na Figura 9.24 pelas declarações antecipadas das classes `Keypad`, `CashDispense` e `BankDatabase` (linhas 6–9 na Figura 9.25). Em vez de incluir (`#include`) o arquivo de cabeçalho inteiro em cada uma dessas classes, colocamos apenas uma declaração antecipada de cada classe no arquivo de cabeçalho `main.h`. Observe que, se a classe `Withdrawal` contivesse objetos reais em vez de referências (isto é, se os sinais de & nas linhas 19–22 fossem omitidos), então, de fato, precisaríamos incluir os arquivos de cabeçalho completos.

```

1 // Figura 9.24: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Screen.h" // inclui a definição da classe Screen
7 #include "Keypad.h" // inclui a definição da classe Keypad
8 #include "CashDispenser.h" // inclui a definição da classe CashDispenser
9 #include "BankDatabase.h" // inclui a definição da classe BankDatabase
10
11 class Withdrawal
12 {
13 private :
14     // atributos
15     int accountNumber; // conta a sacar fundos
16     double amount; // quantia a sacar
17
18     // referências a objetos associados
19     Screen &screen; // referência à tela do ATM
20     Keypad &keypad; // referência ao teclado do ATM
21     CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
22     BankDatabase &bankDatabase; // referência ao banco de dados de info de conta
23 };
24
25 #endif // WITHDRAWAL_H

```

Figura 9.24 Declarando referências aos objetos associados com `#include`

```

1 // Figura 9.25: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Screen; // declaração antecipada da classe Screen
7 class Keypad; // declaração antecipada da classe Keypad
8 class CashDispenser; // declaração antecipada da classe CashDispenser
9 class BankDatabase; // declaração antecipada da classe BankDatabase
10
11 class Withdrawal
12 {
13 private :
14     // atributos
15     int accountNumber; // conta a sacar fundos
16     double amount; // quantia a sacar
17
18     // referências a objetos associados
19     Screen &screen; // referência à tela do ATM
20     Keypad &keypad; // referência ao teclado do ATM
21     CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
22     BankDatabase &bankDatabase; // referência ao banco de dados de informações de conta
23 };
24
25 #endif // WITHDRAWAL_H

```

Figura 9.25 Utilizando declarações antecipadas em vez de `#include`.

Observe que utilizar uma declaração antecipada (onde possível) em vez de incluir um arquivo de cabeçalho inteiro ajuda evitar um problema de pré-processador chamado **#include loop**. Esse problema ocorre quando o arquivo de cabeçalho de uma classe inclui #includes do arquivo de cabeçalho em outras classes. Alguns pré-processadores não são capazes de resolver essas dependências usando um erro de compilação. Por exemplo, utiliza somente uma referência a um objeto dentro de uma classe no arquivo de cabeçalho da classe e substituído por uma declaração antecipada da classe para impedir a inclusão circular.

5. Utilize as operações localizadas no terceiro compartimento da Figura 9.20 para escrever os protótipos de função das funções-membro da classe. Se ainda não especificamos um tipo de retorno para uma operação, declaramos a função-membro com tipo de retorno. Consulte os diagramas de classes das figuras 6.22–6.25 para declarar qualquer parâmetro necessário. Por exemplo, adicionar **operação** na classe **Withdrawal**, que tem uma lista de parâmetros vazia, resulta no protótipo da linha 15 da Figura 9.26. [Adifcamos as definições de funções-membro para simplificarmos o sistema ATM completo no Apêndice G.]



Observação de engenharia de software 9.12

Várias ferramentas de modelagem UML podem converter projetos baseados em UML em código C++, aumentando consideravelmente a velocidade do processo de implementação. Para obter informações adicionais sobre esses geradores de códigos ‘automáticos’, consulte os recursos na Internet e na Web listados no final da Seção 2.8.

Isso conclui nossa discussão sobre os princípios básicos da geração de arquivos de cabeçalho de classe a partir de diagramas. Na seção ‘Estudo de caso de engenharia de software’ final (Seção 3.11), demonstramos como modificar os arquivos de cabeçalho para incorporar o conceito de herança orientado a objetos.

Exercícios de revisão do estudo de caso de engenharia de software

- 9.1 Determine se a seguinte sentença é verdadeira ou falsa, e explique por quê: Se um atributo de uma classe estiver marcado com um sinal de subtração (um diagrama de classes, o atributo não estará diretamente acessível fora da classe.

```

1 // Figura 9.26: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Screen; // declaração antecipada da classe Screen
7 class Keypad; // declaração antecipada da classe Keypad
8 class CashDispenser; // declaração antecipada da classe CashDispenser
9 class BankDatabase; // declaração antecipada da classe BankDatabase
10
11 class Withdrawal
12 {
13 public :
14     // operações
15     void execute(); // realiza a transação
16 private :
17     // atributos
18     int accountNumber;/ conta a sacar fundos
19     double amount; // quantia a sacar
20
22// referências a objetos associados
23 Keypad &keypad; // referência ao teclado do ATM
24 CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
25 BankDatabase &bankDatabase; // referência ao banco de dados de informações de conta
26 }; // fim da classe Withdrawal
27
28 #endif // WITHDRAWAL_H

```

Figura 9.26 Adicionando operações ao arquivo de cabeçalho da **Withdrawal**

- 9.2 Na Figura 9.21, a associação `Customer` para `ATM` indica que:
- podemos navegar para ATM
 - podemos navegar para Customer
 - Tanto a como b; a associação é bidirecional
 - Nenhuma acima
- 9.3 Escreva um código C++ para começar a implementação do projeto da classe

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 9.1 Verdadeira. O sinal de `privado` indica visibilidade privada. Mencionamos `friend` como uma exceção à visibilidade privada. `friend` é discutida no Capítulo 10.
- 9.2 b.
- 9.3 O projeto para `Customer` produz o arquivo de cabeçalho na Figura 9.27.

9.13 Síntese

Este capítulo aprofundou nossa discussão sobre classes, utilizando um projeto para tratar das classes. Você viu que as funções-membro são normalmente menores que as funções globais porque podem acessar os membros de dados de um objeto; portanto, as funções-membro podem receber menos argumentos que as funções em linguagem procedural. Você aprendeu a utilizar o operador seta para acessar os membros de um objeto por meio de um ponteiro da classe do objeto.

Você aprendeu que as funções-membro têm escopo de classe — isto é, o nome da função-membro só é conhecido por outros membros da classe a menos que referenciado por meio de um objeto da classe, uma referência a um objeto da classe, um ponteiro da classe ou pelo operador de resolução de escopo binário. Também discutimos as funções de acesso (comumente utilizadas para recuperar os valores de membros de dados ou testar a verdade ou falsidade de condições) e as funções utilitárias (funções-membro que suportam a operação de funções da classe).

Você aprendeu que um construtor pode especificar argumentos-padrão que permitem que ele seja chamado de várias maneiras. Também aprendeu que qualquer construtor que pode ser chamado sem argumentos é um construtor-padrão e que há, no máximo, um construtor-padrão por classe. Discutimos os destrutores e seus propósitos de realizar uma faxina de terminação em um objeto de um tipo antes de esse objeto ser destruído. Demonstramos também a ordem em que os construtores e destrutores de um objeto são chamados.

Demonstramos os problemas que podem ocorrer quando uma função-membro retorna uma referência a um membro de uma classe.

`private`, que quebra o encapsulamento da classe. Mostramos também que os objetos do mesmo tipo podem ser atribuídos uns aos outros.

```

1 // Figura 9.27: Account.h
2 // Definição da classe Account. Representa uma conta bancária
3 #ifndef ACCOUNT_H
4 #define ACCOUNT_H
5
6 class Account
7 {
8 public :
9     bool validatePIN( int ); // o PIN especificado pelo usuário é correto?
10    double getAvailableBalance(); // retorna o saldo disponível
11    double getTotalBalance(); // retorna saldo total
12    void credit( double ); // adiciona um valor a Account
13    void debit( double ); // subtrai um valor de Account
14 private :
15     int accountNumber// número da conta
16     int pin; // PIN para autenticação
17     double availableBalance; // fundos disponíveis para saque
18     double totalBalance; // fundos disponíveis + fundos esperando compensação
19 }; // fim da classe Account
20
21 #endif // ACCOUNT_H

```

Figura 9.27 O arquivo de cabeçalho da classe `Account`, baseado nas figuras 9.20 e 9.21.

utilizando atribuição-padrão de membro a membro. Por fim, discutimos o benefício de utilizar a biblioteca de classes para aprimorar a velocidade com que o código pode ser criado e para aumentar a qualidade do software.

O Capítulo 10 apresenta recursos de classe adicionais. **Demais membros** para indicar que uma função-membro não modifica o objeto de uma classe. Você aprenderá a construir classes com composição — isto é, classes que contêm de outras classes como membros. Mostraremos como uma classe pode permitir que as chamadas funções ‘friend’ (amigas) acessem membros **públicos** da classe. Também mostraremos como funções **de membro** podem utilizar um ponteiro especial **chamada** para acessar os membros de um objeto. Em seguida, você aprenderá a utilizar os operadores C++ que permitem aos programadores obter e liberar memória conforme necessário durante a execução de um programa.

Resumo

- As direitas de saída por ~~setores~~ é que essas direitas são utilizadas para impedir que o cliente use um código que pode ser utilizado para impedir inclusões futuras, e o código é incluído no arquivo de código-fonte.
 - Os membros de dados de uma classe não podem ser inicializados no local em que são declarados no corpo de classe (exceto pelos membros de dados `const` de uma classe do tipo inteiro, que você verá no Capítulo 10). É altamente recomendado que esses membros de dados sejam inicializados pelo construtor da classe (uma vez que não há inicialização-padrão de membros de tipos fundamentais).
 - O manipulador `setw` especifica o caractere de preenchimento que é exibido quando um inteiro é enviado para a saída em um campo que é maior do que o número de dígitos no valor.
 - Por padrão, os caracteres de preenchimento aparecem antes dos dígitos no número.
 - O manipulador `setfill` é uma configuração ‘aderente’, o quer dizer que, uma vez que o caractere de preenchimento é configurado, ele se aplica a todos os campos subsequentes sendo impressos.
 - Embora uma função-membro declarada em uma definição de classe possa ser definida fora dessa definição de classe (e associada à classe operador binário de resolução de escopo), essa função-membro ainda está dentro do escopo dessa classe; por exemplo, seu nome é conhecido somente pelos outros membros da classe a menos que referenciado via um objeto da classe, uma referência a um objeto da classe ou um ponteiro para um objeto da classe.
 - Se uma função-membro é definida no corpo de uma definição de classe, o compilador C++ tenta colocar inline as chamadas para função-membro.
 - As classes não precisam ser criadas ‘a partir do zero’. Em vez disso, elas podem incluir objetos de outras classes como membros ou podem ser derivadas de outras classes que fornecem atributos e comportamentos que podem ser utilizados pelas novas classes. Incluir objetos de classe como membros de outras classes é chamado de composição.
 - Os membros de dados e as funções-membro de uma classe pertencem ao escopo dessa classe.
 - As funções não-membro são definidas no escopo de arquivo.
 - Dentro do escopo de uma classe, os membros da classe são imediatamente acessíveis por todas as funções-membro dessa classe e podem ser referenciados pelo nome.
 - Fora do escopo de uma classe, os membros de classe são referenciados por um dos handles em um objeto — um nome de objeto, uma referência a um objeto ou um ponteiro para um objeto.
 - As funções-membro de uma classe podem ser sobreescritas, mas apenas por outras funções-membro dessa classe.
 - Para sobreescrita de uma função-membro, forneça na definição da classe um protótipo para cada versão da função sobreescrita e uma definição separada para cada versão da função.
 - As variáveis declaradas em uma função-membro têm escopo de bloco e são conhecidas somente por essa função.
 - Se uma função-membro definir uma variável com o mesmo nome de uma variável com escopo de classe, a variável de escopo de classe é obscurecida pela variável de escopo de bloco no escopo de bloco.
 - O operador de seleção de membro `>>` é precedido pelo nome de um objeto ou por uma referência a um objeto para acessar membros public do objeto.
 - O operador de seleção de membro `<<` é precedido por um ponteiro para um objeto para alterar esse objeto.
 - Os arquivos de cabeçalho contêm algumas partes da implementação e dicas sobre outras. As funções-membro inline, por exemplo, precisam estar em um arquivo de cabeçalho, para que, quando o compilador compilar um cliente, este possa incluir adequadamente a definição de função inline.
 - Os membros privados de uma classe que são listados na definição de classe no arquivo de cabeçalho são visíveis aos clientes, mesmo que estes possam não acessar os membros.

- Uma função utilitária (também chamada defunção auxiliar) é uma que não opera sobre o membro da classe. As funções utilitárias não foram projetadas para ser utilizadas por clientes de uma classe, mas podem ser utilizadas por uma classe.
- Como outras funções, os construtores podem especificar argumentos-padrão.
- O destrutor de uma classe é chamado implicitamente quando um objeto da classe é destruído.
- O nome do destrutor de uma classe é **destrutor** (name da classe).
- Na realidade, um destrutor não libera o armazenamento de um objeto — ele realiza a faxina de terminação antes de o sistema reivindicar memória de um objeto, assim a memória pode ser reutilizada para armazenar novos objetos.
- Um destrutor não recebe parâmetros nem retorna um valor. Uma classe pode ter somente um destrutor.
- Se o programador não fornecer um destrutor explicitamente, o compilador cria um destrutor ‘vazio’, então cada classe tem exatamente um destrutor.
- A ordem em que os construtores e destrutores são chamados depende da ordem em que a execução entra e sai dos escopos onde os objetos são instanciados.
- Geralmente, as chamadas de destrutor são feitas na ordem inversa das chamadas de construtor correspondentes, mas as classes de armazenamento de objetos podem alterar a ordem em que os destrutores são chamados.
- Uma referência a um objeto é um alias para o nome do objeto e, portanto, pode ser utilizada à esquerda de uma instrução de atribuição. Nesse contexto, a referência é **referência** perfeitamente aceitável que pode receber um valor. Uma maneira de utilizar essa capacidade (infelizmente!) é fazer uma função-membro de uma classe retornar uma referência a um membro dessa classe. Se a função retorna uma referência, então a referência não pode ser utilizada **diretamente**.
- O operador de atribuição (**=**) é usado para atribuir um objeto a outro objeto do mesmo tipo. Por padrão, essa atribuição é realizada pela atribuição de membro a membro — cada membro do objeto à direita do operador de atribuição é atribuído individualmente ao mesmo membro do objeto à esquerda do operador de atribuição.
- Os objetos podem ser passados como funções argumentos e podem ser retornados a partir de funções. Essa passagem e esse retorno são realizados utilizando a passagem por valor por padrão — uma cópia do objeto é passada ou retornada. Nessas casos, o C++ cria um novo objeto e usa um construtor de cópia para copiar os valores do objeto original para o novo objeto. Explicamos esses casos detalhadamente no Capítulo 1 ‘Sobrecarga de operadores; objetos string e array’.
- Para cada classe, o compilador fornece um construtor de cópia-padrão que copia cada membro do objeto original para o membro correspondente do novo objeto.
- Existem muitas bibliotecas de classe, e outras estão sendo desenvolvidas em todo o mundo.
- A reusabilidade de software acelera o desenvolvimento de softwares de alta qualidade e poderosos. O desenvolvimento rápido de aplicativos (rapid applications development, RAD) por meio dos mecanismos de componentes reutilizáveis tornou-se um importante campo.

Terminologia

#define, diretiva de pré-processador	definir	handle de referência em um objeto
#endif, diretiva de pré-processador	terminar	handle em um objeto
#ifndef, diretiva de pré-processador	desenvolvimento rápido de aplicativos (RAD)	handle implícito em um objeto
abort, função	(rapid application development, RAD)	herança
agregação	destrutor	initializador
argumentos-padrão com construtor	despacotador de pré-processador	objeto sai do escopo
ativo de software	escopo de arquivo	operador de seleção de membro seta (→)
atribuição de membro a membro	escopo de classe	ordem em que construtores e destrutores são chamados
atribuição-padrão de membro a membro	embalhamento	passar um objeto por valor
atribuindo objetos de classe	função auxiliar	preparação de terminação
bibliotecas de classe	função de acesso	
caractere de preenchimento	função preditora sobrecarregada	procedure principal de fluxo parametrizado
componentes reutilizáveis	handle de nome em um objeto	til (~), caractere, em um nome de destrutor
composição	handle de objeto	
construtor de cópia	handle de ponteiro em um objeto	

Exercícios de revisão

- 9.1 Preencha as lacunas em cada uma das seguintes sentenças:
- Os membros de classe são acessados via operador em conjunto com o nome de um objeto (ou referência a um objeto) da classe ou via operador em conjunto com um ponteiro para um objeto da classe.
 - Os membros de classe especificados como são acessíveis apenas dentro do escopo da classe e
 - Os membros de classe especificados como são acessíveis em qualquer lugar que um objeto da classe esteja no escopo.
 - A pode ser utilizada para atribuir um objeto de uma classe a outro objeto da mesma classe.
- 9.2 Localize o(s) erro(s) em cada uma das seguintes seqüências e explique como corrigi-lo(s):
- Suponha que o seguinte protótipo é declarado na classe

```
void ~Time( int );
```
 - A seguinte definição é uma definição Time da classe

```
class Time
{
public :
    // protótipos de função
private :
    int hour = 0;
    int minute = 0;
    int second = 0;
}; // fim da classe Time
```
 - Suponha que o seguinte protótipo é declarado na classe

```
int Employee(const char *, const char *);
```

Respostas dos exercícios de revisão

- 9.1 a) ponto)(seta). b)private . c)public . d) atribuição-padrão de membro a membro (realizada pelo operador de atribuição).
- 9.2 a) Erro: Os destrutores não têm permissão de retornar valores (nem mesmo de especificar um tipo de retorno) nem de aceitar argumentos.
 Correção: Remova o tipo ~~de retorno~~ parâmetro da declaração.
- b) Erro: Os membros não podem ser explicitamente inicializados na definição de classe.
 Correção: Remova a inicialização explícita da definição de classe e inicialize os membros de dados em um construtor.
- c) Erro: Os construtores não têm permissão de retornar valores.
 Correção: Remova o tipo ~~de retorno~~ declaração.

Exercícios

- 9.3 Qual é o propósito do operador de resolução de escopo?
- 9.4 (Aprimorando a ~~classe~~) Forneça um construtor que seja capaz de utilizar ~~até hora atulda fadão~~ o cabeçalho `<ctime>` da C++ Standard Library — para inicializar um ~~objeto~~ da classe
- 9.5 (Classe Complex) Crie uma classe ~~Complex~~ para realizar aritmética com números complexos. Escreva um programa para testar sua classe.

Os números complexos têm a forma

$$\text{parteReal} + \text{parteImaginária} \cdot i$$

onde $i = \sqrt{-1}$

Utilize as variáveis ~~double~~ para representar os dados da classe. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve conter valores-padrão no caso de nenhum inicializador ser fornecido. Forneça funções-membro que realizam as tarefas a seguir:

- Somar dois números complexos partes reais são somadas de um lado e as partes imaginárias são somadas de outro.
- Subtrair dois números complexos a parte real do operando direito é subtraída da parte real do operando esquerdo e a parte imaginária do operando direito é subtraída da parte imaginária do operando esquerdo.
- Imprimir os números complexos na forma $a + bi$, onde a é a parte real e b é a parte imaginária.

9.6 (Classificacional) Crie uma classe `Fraction` para realizar aritmética com frações. Escreva um programa para testar sua classe.

Utilize variáveis do tipo inteiro para representar o numerador e o denominador. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve conter valores-padrão no caso de nenhum inicializador ser fornecido e deve armazenar a fração na forma reduzida. Por exemplo, a fração $\frac{2}{4}$

$\frac{2}{4}$

seria armazenada no objeto como `numerator = 2` e `denominator = 4`. Forneça funções-membro que realizam cada uma das tarefas a seguir:

- Somar dois números racionais. O resultado deve ser armazenado na forma reduzida.
- Subtrair dois números racionais. O resultado deve ser armazenado na forma reduzida.
- Multiplicar dois números racionais. O resultado deve ser armazenado na forma reduzida.
- Dividir dois números racionais. O resultado deve ser armazenado na forma reduzida.
- Imprimir os números racionais na forma `número/denominador`, onde `número` é o numerador.
- Imprimir os números racionais em formato de ponto flutuante.

9.7 (Aprimorando a classe) Modifique a classe das figuras 9.8–9.9 para incluir uma função que incrementa a hora armazenada em um segundo. O objeto sempre deve permanecer em um estado consistente. Escreva um programa que testa a função-membro loop que imprime a hora no formato-padrão durante cada iteração do loop para ilustrar que a função-membro funciona corretamente. Certifique-se de testar os seguintes casos:

- Incrementar para o próximo minuto.
- Incrementar para a próxima hora.
- Incrementar para o próximo dia (isto é, 11:59:59 PM para 12:00:00 AM).

9.8 (Aprimorando a classe) Modifique a classe das figuras 9.17–9.18 para realizar a verificação de erros nos valores inicializadores para membros de `Day` e `Year`. Além disso, forneça uma função para incrementar o dia por um. O objeto `Date` sempre deve permanecer em um estado consistente. Escreva um programa que teste que a função `incrementa` a data durante cada iteração para funcionar corretamente. Certifique-se de testar os seguintes casos:

- Incrementar para o próximo mês.
- Incrementar para o próximo ano.

9.9 (Combinando a classe) Combine a classe modificada do Exercício 9.7 com a classe do Exercício

9.8 em uma classe `DateAndTime`. No Capítulo 12, discutiremos a herança, que permitirá realizar essa tarefa rapidamente sem modificar as definições de classe existentes. Modifique a função `formatDate` para que a hora for incrementada para o dia seguinte. Modifique a função `printUniversal` para gerar saída da data e da hora. Escreva um programa para testar a nova classe. Especificamente, teste incrementar a hora para o dia seguinte.

9.10 (Retornando indicadores de erros das funções) Modifique as classes das figuras 9.8–9.9 para retornar valores de erros apropriados se a função `incrementa` receber de dados de um objeto da classe `Date` um valor inválido for feita. Escreva um programa que testa sua função `incrementa` e teste se ela gera o erro quando tenta incrementar valores de erros.

9.11 (Classe Rectangle) Crie uma classe `Rectangle` com atributos `width` e `height`, cada um dos quais assume o padrão de 1. Forneça funções-membro que calculam `perimeter` e `area` do retângulo. Além disso, forneça funções-membro `isSquare` e `isLandscape` que determinam se `width` é igual a `height` e se `width` é maior que `height`, respectivamente. As funções devem verificar se `width` e `height` são números de ponto flutuante maiores que 0,0 e menores que 20,0.

9.12 (Classe Rectangle) Modifique a classe `Rectangle` do Exercício 9.11. Essa classe armazena somente as coordenadas cartesianas dos quatro vértices do retângulo. O construtor chama uma função que recebe quatro conjuntos de coordenadas e verifica se cada um deles está no primeiro quadrante. Se estiver, a classe armazena essas coordenadas. A função também verifica se as coordenadas fornecidas especificam de fato um retângulo. Forneça funções-membro que calculam `length`, `width`, `perimeter` e `area`. O comprimento é o maior das duas dimensões. Inclua uma função predicable

que determina se o retângulo é um quadrado.

9.13 (Aprimorando a classe) Modifique a classe `Rectangle` do Exercício 9.12 para incluir uma função que determine se o retângulo está dentro de uma caixa de 25 por 25 para incluir a parte do primeiro quadrante em que o retângulo reside. Inclua uma função `character` para especificar o caractere a partir do qual o corpo do retângulo será desenhado. Inclua uma função `border` para especificar o caractere que será utilizado para desenhar a borda do retângulo. Se você se sentir ambicioso, talvez queira incluir uma função para dimensionar o tamanho do retângulo, rotacioná-lo e movê-lo dentro da parte designada do primeiro quadrante.

9.14 (Classe BigInteger) Crie uma classe `BigInteger` que utiliza um array de 40 elementos de dígitos para armazenar inteiros com até 40 dígitos cada. Forneça as funções-membro `add`, `subtract`, `multiply` e `divide`. Para comparar objetos `BigInteger` forneça funções `isEqual`, `isNotEqual`, `isGreater`, `isLess`, `isGreaterOrEqual` e `isLessOrEqual` para cada uma

dessas é uma função 'predicado' que simplesmente retorna `True` se o relacionamento se mantém entre os dois membros e `false` se o relacionamento não se mantém. Além disso, forneça `isZero`, `isOdd` e `isEven`, que, juntas com `isPredicable`, forneca as funções-membro `multiply`, `divide` e `modulus`.

9.15 (Classic Tic Tac Toe) Crie uma classe `TicTacToe` que permitirá escrever um programa completo para jogar o jogo-da-velha (Tic-Tac-Toe). A classe contém campo de dados um array bidimensional 3 por 3 de inteiros. O construtor deve inicializar a grade vazia com todos como zero. Permita dois jogadores humanos. Para onde quer que o primeiro jogador se move, coloque um 1 no quadrado especificado. Coloque 2 para onde quer que o segundo jogador se move. Todo movimento deve ocorrer em um quadrado vazio. Depois de cada movimento, determine se houve uma derrota ou um empate. Se você se sentir motivado, modifique seu programa de modo que o computador faça o movimento para um dos jogadores. Além disso, permita que o jogador especifique se quer ser o primeiro ou o segundo. Se você se sentir excepcionalmente motivado, desenvolva um programa que jogue um jogo-da-velha tridimensional em uma grade 4 por 4 por 4. [Atenção! Esse é um projeto extremamente desafiador que pode exigir muitas semanas de esforço!]



Mas o que, para servir nossos fins pessoais,
Nos proíbe de enganar nossos amigos?

Charles Churchill
Em vez dessa divisão absurda em sexos, deveríamos classificar as pessoas como estáticas e dinâmicas.
Evelyn Waugh

Não tenha amigos iguais a você
Confúcio

Classes: um exame mais profundo, parte 2

OBJETIVOS

Neste capítulo, você aprenderá:

- A especificar objetos `const` (constante) e funções-membro `const`.
 - A criar objetos compostos de outros objetos.
 - A utilizar funções `friend` e classes `friend`.
 - A utilizar o ponteiro `this`.
 - A criar e destruir dinamicamente objetos com os operadores `new` e `delete`, respectivamente.
 - A utilizar membros de dados e funções-membro `static`.
 - O conceito de uma classe contêiner.
- Anotação:** Classes iteradoras que percorrem os elementos de classes contêineres.
- A utilizar classes proxy para ocultar detalhes de implementação dos clientes de uma classe.

10.1	Introdução
10.2	Objetos const (constante) e funções-membro const
10.3	Composição: objetos como membros de classes
10.4	Funções friend e classes friend
10.5	Utilizando o ponteiro this
10.6	Gerenciamento de memória dinâmico com os operadores new e delete
10.7	Membros de classe static
10.8	Abstração de dados e ocultamento de informações
10.8.1	Exemplo: tipo de dados abstrato array
10.8.2	Exemplo: tipo de dados abstrato string
10.8.3	Exemplo: tipo de dados abstrato queue
10.9	Classes contêineres e iteradores
10.10	Classes proxy
10.11	Síntese

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

10.1 Introdução

Neste capítulo, continuamos nosso estudo de classes e abstração de dados com vários tópicos mais avançados. Utilizamos objetos funções-membro para impedir modificações de objetos e impor o princípio do menor privilégio. Discutimos a composição — uma forma de reutilização em que uma classe pode ter objetos de outras classes como membros. Em seguida, introduzimos a amizade (friendship) que permite ao designer de classes especificar funções não-membro que podem acessar membros não-técnicos que é freqüentemente utilizada na sobrecarga de operadores (Capítulo 11) por razões de desempenho. Discutimos um par especial (chamado) que é um argumento implícito para cada uma das funções-membro que permite que essas funções-membro acessem os membros de dados corretamente. Então discutimos o gerenciamento de memória dinâmico e mostramos como criar e destruir objetos dinamicamente com os operadores

seguidos, motivamos a classe std::vector, mostrando como essa classe utiliza strings membro de dados de implementação de classe (incluindo seus membros de dados) a partir de clientes da classe.

Lembre-se de que o Capítulo 3 introduziu classes da Standard Library para representar strings como objetos de classe completos. Neste capítulo, porém, utilizamos as strings baseadas em ponteiro introduzidas no Capítulo 8 para ajudar o leitor a dominar os ponteiros e a se preparar para o mundo profissional em que verá uma grande quantidade de código C legado implementado nas últimas décadas. Assim, o leitor se familiarizará com os dois métodos mais predominantes de criar e manipular strings em C++.

10.2 Objetos const (constante) e funções-membro const

Enfatizamos o princípio do menor privilégio como um dos princípios mais fundamentais da boa engenharia de software. Vejamos como esse princípio se aplica aos objetos.

Alguns objetos precisam ser modificáveis e alguns não. O programador pode utilizar o tipo const para garantir que o objeto não é modificável e que qualquer tentativa de modificá-lo deve resultar em um erro de compilação. A instrução

```
const Time noon(12, 0, 0);
```

declara um objeto noon da classe Time e o inicializa como 12:00 AM (meio-dia).



Observação de engenharia de software 10.1

Declarar um objeto const ajuda a impor o princípio do menor privilégio. As tentativas de modificar o objeto são capturadas em tempo de compilação em vez de causar erros em tempo de execução. Isso é crucial para o design de classe, o design de programa e a codificação adequados.



Dica de desempenho 10.1

Declarar variáveis e objetos de melhorar o desempenho — os sofisticados compiladores de otimização atuais podem realizar certas otimizações em constantes que não podem ser executadas em variáveis.

Os compiladores C++ não permitem chamadas de função-membro para objetos próprias funções-membro também sejam declarados. Isso é verdadeiro mesmo para funções-membro que modificam o objeto. Além disso, o compilador não permite que funções-membro declarem o próprio objeto.

Uma função é especificada como const em seu protótipo (Figura 10.1; linhas 19–24) definição (Figura 10.2; linhas 47, 53, 59 e 65) inserindo a palavra-chave const na lista de parâmetros da função e, no caso da definição da função, antes da chave esquerda que inicia o corpo da função.



Erro comum de programação 10.1

Definir como const uma função-membro que modifica um membro de dados de um objeto é um erro de compilação.



Erro comum de programação 10.2

const
Definir como const uma função-membro que chama uma função-membro de uma mesma instância da classe é um erro de compilação.



Erro comum de programação 10.3

Invocar uma função-membro const em um objeto é um erro de compilação.



Observação de engenharia de software 10.2

Uma função-membro pode ser sobreescrita com uma versão const. O compilador escolhe que função-membro sobreescrita utilizar com base no objeto em que a função é invocada. Se o objeto não for const, o compilador utilizará a versão não-const.

Um problema interessante surge para construtores e destrutores, cada um dos quais em geral modifica objetos. A declaração não é permitida para construtores e destrutores. Um construtor deve ter permissão de modificar um objeto para que o objeto seja inicializado adequadamente. Um destrutor deve ser capaz de realizar suas tarefas de faxina de terminação antes de a memória do objeto ser reivindicada pelo sistema.



Erro comum de programação 10.4

Tentar declarar um construtor ou destrutor const gera um erro de compilação.

Definindo e utilizando funções-membro const

O programa das figuras 10.1–10.3 modifica as figuras 9.9–9.10 fazendo algumas alterações. As funções const. No arquivo de cabeçalho (Figura 10.1), as linhas 19–21 e 24 agora incluem a palavra-chave const na lista de parâmetros de cada função. A definição correspondente de cada função na Figura 10.2 (linhas 47, 53, 59 e 65, respectivamente) especifica a palavra-chave const depois da lista de parâmetros de cada função.

A Figura 10.3 instancia dois objetos — o objeto wakeUp (linha 7) e o objeto noon (linha 8). O programa tenta invocar as funções-membro const (linha 13) e non-const (linha 14) no objeto noon. Em cada caso, o compilador gera uma mensagem de erro. O programa também ilustra as três outras combinações de chamada de função-membro — uma função-membro const em um objeto não-const (linha 11), uma função-membro non-const em um objeto const (linha 15) e uma função-membro non-const em um objeto non-const (linhas 17–18). As mensagens de erro geradas para chamadas de função-membro const em um objeto não-const são mostradas na janela de saída. Note que, embora alguns compiladores atuais emitam somente mensagens de advertência para as linhas 13 e 20 (permitindo, assim, que esse programa seja executado), consideram avisos como erros — o padrão ANSI/ISO C++ não permite a invocação de uma função const não-

Note que, mesmo que um construtor precise ser uma função const (Figura 10.2, linhas 15–18), ele ainda pode ser utilizado para inicializar um objeto (Figura 10.3, linha 8). A definição const (Figura 10.2, linhas 15–18) mostra

que o construtor pode inicializar um objeto const (linhas 21 e 22) — copiando os valores de uma constante const para o objeto const. A constância de um objeto impõe a partir do momento em que o construtor completa a inicialização do objeto até o destrutor desse objeto ser chamado.

Também note que a linha 20 na Figura 10.3 gera um erro de compilação mesmo sendo a classe const. A constância de um objeto não modifica o objeto em que ela é invocada. O fato de uma função-membro não modificar um objeto em que ela é invocada não é suficiente para indicar que a função é uma função constante — a função deve ser explicitamente declarada.

```

1 // Figura 10.1: Time.h
2 // Definição da classe Time.
3 // Funções-membro definidas em Time.cpp.
4 #ifndef TIME_H
5 #define TIME_H
6
7 class Time
8 {
9 public :
10 Time( int = 0, int = 0, int = 0); // construtor-padrão
11
12 // funções set
13 void setTime( int , int , int ); // configura time
14 void setHour( int ); // configura hour
15 void setMinute( int ); // configura minute
16 void setSecond( int ); // configura second
17
18 // funções get (normalmente declaradas const)
19 int getHour() const; // retorna hour
20 int getMinute() const; // retorna minute
21 int getSecond() const; // retorna second
22
23 // funções print (normalmente declaradas const)
24 void printUniversal() const ; // imprime hora universal
25 void printStandard(); // imprime hora-padrão (deve ser const)
26 private :
27 int hour; // 0 - 23 (formato de relógio de 24 horas)
28 int minute; // 0 - 59
29 int second; // 0 - 59
30 }; // fim da classe Time
31
32 #endif

```

Figura 10.1 A definição da classe Time com funções-membro const.

```

1 // Figura 10.2: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // inclui definição da classe Time
11
12 // função constructor para inicializar dados private;
13 // chama a função-membro setTime para configurar variáveis;
14
15 Time::Time( int hour, int minute, int second )
16 {
17     setTime( hour, minute, second );
18 } // fim do construtor de Time
19
20 // configura valores de hour, minute e second

```

Figura 10.2 Definições de função-membro da classe Time incluindo as funções-membro const.

(continua)

```

21 void Time::setTime( int hour, int minute, int second )
22 {
23     setHour( hour );
24     setMinute( minute );
25     setSecond( second );
26 } // fim da função setTime
27
28 // configura valor de hour
29 void Time::setHour( int h )
30 {
31     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
32 } // fim da função setHour
33
34 // configura valor de minute
35 void Time::setMinute( int m )
36 {
37     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
38 } // fim da função setMinute
39
40 // configura valor de second
41 void Time::setSecond( int s )
42 {
43     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
44 } // fim da função setSecond
45
46 // retorna valor de hour
47 int Time::getHour() const // funções get devem ser const
48 {
49     return hour;
50 } // fim da função getHour
51
52 // retorna valor de minute
53 int Time::getMinute() const
54 {
55     return minute;
56 } // fim da função getMinute
57
58 // retorna valor de second
59 int Time::getSecond() const
60 {
61     return second;
62 } // fim da função getSecond
63
64 // imprime a hora no formato universal de data/hora (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68     << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // fim da função printUniversal
70
71 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
72 void Time::printStandard() // nota a falta da declaração const
73 {
74     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75     << ":" << setfill( '0' ) << setw( 2 ) << minute
76     << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM");
77 } // fim da função printStandard

```

Figura 10.2 Definições de função-membro da classe Time incluindo as funções-membroconst.

(continuação)

```

1 // Figura 10.3: fig10_03.cpp
2 // Tentando acessar um objeto const com funções-membro não-const.
3 #include "Time.h" // inclui definição da classe Time
4
5 int main()
6 {
7     Time wakeUp( 6, 45, 0); // objeto não-constante
8     const Time noon( 12, 0, 0); // objeto constante
9
10    // OBJETO      FUNÇÃO-MEMBRO
11    wakeUp.setHour( 18); // não-const  não-const
12
13    noon.setHour( 12); // const      non-const
14
15    wakeUp.getHour(); // não-const  const
16
17    noon.getMinute(); // const      const
18    noon.printUniversal(); // const      const
19
20    noon.printStandard(); // const      non-const
21
22 } // fim de main

```

Mensagens de erro do compilador de linha de comando Borland C++:

```

Warning W8037 fig10_03.cpp 13: Non-const function Time::setHour(int)
called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Time::printStandard()
called for const object in function main()

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
'Time &'
    Conversion loses qualifiers
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
'Time &'
    Conversion loses qualifiers

```

Mensagens de erro do compilador GNU C++:

```

fig10_03.cpp:13: error: passing `const Time` as `this` argument of
`void Time::setHour(int)` discards qualifiers
fig10_03.cpp:20: error: passing `const Time` as `this` argument of
`void Time::printStandard()` discards qualifiers

```

Figura 10.3 Objetos const e funções-membro const.

Inicializando um membro de dados const com um inicializador de membro

O programa das figuras 10.4–10.6 introduz a utilização de inicializador de membro. Todos os membros de dados inicializados utilizando a sintaxe de inicializador de membro, mas esse método de inicialização é só para os membros de dados que são referências de objetos. Mais adiante neste capítulo, veremos que os objetos-membro também devem ser inicializados dessa maneira. Quando estudarmos herança no Capítulo 12, veremos que partes de classes básicas de classes também devem ser inicializadas dessa maneira.

A definição de construtor (Figura 10.5, linhas 11–16) utiliza uma lista de membros para inicializar os membros de dados da classe — o inteiro `count` e o inteiro `increment` (declarados nas linhas 19–20 da Figura 10.4). Os inicializadores de membro aparecem entre a lista de parâmetros de um construtor e a chave esquerda que inicia o corpo do construtor. A lista de inicializadores de membro (Figura 10.5, linhas 12–13) é separada da lista de parâmetros por dois-pontos (`:`) e consiste no nome do membro de dados seguido por parênteses contendo o valor inicial do membro. Nesse exemplo, `count` é inicializado com o valor de parâmetro `c`, `increment` é inicializado com o valor de parâmetro `i`. Observe que múltiplos inicializadores de membro são separados por vírgulas. Além disso, observe que a lista de inicializadores de membro é executada antes de o corpo do construtor executar.



Observação de engenharia de software 10.3

Um objeto `const` não pode ser modificado por atribuição, portanto ele deve ser inicializado. Quando um membro de dados de uma classe é declarado, um inicializador de membro deve ser utilizado para fornecer ao construtor o valor inicial do membro de dados para um objeto da classe. O mesmo é verdadeiro para referências.

```

1 // Figura 10.4: Increment.h
2 // Definição da classe Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public :
9     Increment( int c = 0, int i = 1 ); // construtor-padrão
10
11    // definição da função addIncrement
12    void addIncrement()
13    {
14        count += increment;
15    } // fim da função addIncrement
16
17    void print() const; // imprime count e increment
18 private :
19     int count;
20     const int increment; // membro de dados const
21 }; // fim da classe Increment
22
23 #endif

```

Figura 10.4 A definição da classe `Increment` contendo o membro de dados `const` `count` e o membro de dado `const` `increment`.

```

1 // Figura 10.5: Increment.cpp
2 // Definições de função-membro para a classe Increment demonstram o uso do
3 // inicializador de membro para inicializar uma constante de um tipo de dados predefinido.
4 #include <iostream>
5 using std::cout;
6
7 using std::endl;
8 #include "Increment.h" // inclui a definição da classe Increment
9
10 // construtor
11 Increment::Increment( int c, int i )
12     : count( c ), // inicializador para membro não-const
13       increment( i ) // inicializador requerido para membro const

```

Figura 10.5 O inicializador de membro utilizado para inicializar uma constante de um tipo de dados predefinido.

(continua)

```

14 {
15     // corpo vazio
16 } // fim do construtor Increment
17
18 // imprime valores de count e increment
19 void Increment::print() const
20 {
21     cout << "count = " << count << ", increment = " << increment << endl;
22 } // fim da função print

```

Figura 10.5 O inicializador de membro utilizado para inicializar uma constante de um tipo de dados predefinido.

(continuação)

```

1 // Figura 10.6: fig10_06.cpp
2 // Programa para testar a classe Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // inclui a definição da classe Increment
7
8 int main()
9 {
10     Increment value( 10, 5 );
11
12     cout << "Before incrementing: " ;
13     value.print();
14
15     for ( int j = 1; j <= 3; j++ )
16     {
17         value.addIncrement();
18         cout << "increment " << j << ":" ;
19         value.print();
20     } // fim do for
21
22     return 0;
23 } // fim de main

```

Before incrementing: count = 10, increment = 5

After increment 1: count = 15, increment = 5

After increment 2: count = 20, increment = 5

After increment 3: count = 25, increment = 5

Figura 10.6 Invocando funções-membro `print` e `addIncrement` de um objeto `Increment`Tentando inicializar erroneamente um membro de `dados` com uma atribuição

O erro mais comum de programação 10.5 Figura 10.8 errou ao tentar inicializar um membro de dados de membro. Observe que a linha 13 da Figura 10.8 não gera um erro de compilador. Observe também que os erros de compilação produzidos pelo Microsoft Visual C++ .NET referem-se a um 'membro de dados 'objeto`const`'. O padrão ANSI/ISO C++ define um 'objeto' como qualquer 'região de armazenamento'. Como as instâncias de classes as variáveis do tipo fundamental também ocupam espaço na memória, portanto são freqüentemente referidas como 'objetos'.

**Erro comum de programação 10.5**

Não fornecer um inicializador de membro para um membro de dados de compilação.



Observação de engenharia de software 10.4

Membros de dados constantes (`objetos-const`) e membros de dados declarados como referências devem ser inicializados com a sintaxe de inicializador de membro; não são permitidas atribuições para esses tipos de dados no corpo do construtor.

Observe que a função (Figura 10.8, linhas 18–21) é declarada parecer estranho rotular essa função que, provavelmente, um programa nunca terá. No entanto, é possível que um programa venha a ter uma referência a um objeto que aponta para um objeto. Em geral, isso ocorre quando objetos da classe são passados para funções ou retornados de funções. Nesses casos, somente as funções-membro da classe podem ser chamadas por meio da referência ou do ponteiro. Portanto, é razoável declarar a função — fazer isso impede erros nas situações em que é tratado como um objeto.

```

2 // Figura 10.7: Increment.h
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public :
9     Increment(      int c = 0, int i = 1); // construtor-padrão
10
11    // definição da função addIncrement
12    void addIncrement()
13    {
14        count += increment;
15    } // fim da função addIncrement
16
17    void print() const ; // imprime count e increment
18 private :
19     const int count;increment; // membro de dados const
20 };
21 // fim da classe Increment
22
23 #endif

```

Figura 10.7 A definição da classe `Increment` contendo o membro de dados `const` `count` e o membro de dado `const` `increment`.

```

1 // Figura 10.8: Increment.cpp
2 // Tentando inicializar uma constante de
3 // um tipo de dados predefinido com uma atribuição.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // inclui a definição da classe Increment
9
10 // construtor; membro constante 'increment' não é inicializado
11 Increment::Increment( int c, int i )
12 {
13     count = c; // permitida porque count não é constante
14     increment = i; // ERRO: Não é possível modificar um objeto const
15 } // fim do construtor Increment
16

```

Figura 10.8 Tentativa errônea de inicializar uma constante de um tipo de dados predefinido por atribuição.

(continua)

```

17 // imprime valores de count e increment
18 void Increment::print() const
19 {
20     cout << "count = " << count << ", increment = " << increment << endl;
21 } // fim da função print

```

Figura 10.8 Tentativa errônea de inicializar uma constante de um tipo de dados predefinido por atribuição.

(continuação)

```

1 // Figura 10.9: fig10_09.cpp
2 // Programa para testar a classe Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // inclui a definição da classe Increment
7
8 int main()
9 {
10     Increment value( 10, 5);
11
12     cout << "Before incrementing: " ;
13     value.print();
14
15     for ( int j = 1; j <= 3; j++ )
16     {
17         value.addIncrement();
18         cout << "After increment " << j << ":" ;
19         value.print();
20     } // fim do for
21
22     return 0;
23 } // fim de main

```

Mensagem de erro do compilador de linha de comando da Borland C++:

Error E2024 Increment.cpp 14: Cannot modify a const object in function
 Increment::Increment(int,int)

Mensagens de erro do compilador Microsoft Visual C++.NET:

C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
 'Increment::increment' : must be initialized in constructor
 base/member initializer list
 C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.h(20) :
 see declaration of 'Increment::increment'
 C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(14) : error C2166:
 l-value specifies const object

Mensagens de erro do compilador GNU C++:

Increment.cpp:12: error: uninitialized member 'Increment::increment' with
 'const' type 'const int'
 Increment.cpp:14: error: assignment of read-only data-member
 'Increment::increment'

Figura 10.9 O programa para testar a classe Increment gera erros de compilação.



Dica de prevenção de erro 10.1

Declare **const** todas as funções-membro de uma classe que não modificam o objeto em que elas operam. Ocasionalmente isso pode parecer inadequado, porque você não terá nenhuma intenção de deixar os objetos dessa classe por meio de referências para. Apesar disso, declarar essas funções **const** oferece um benefício. Se a função-membro for inadvertidamente escrita para modificar o objeto, o compilador emitirá uma mensagem de erro.

10.3 Composição: objetos como membros de classes

Um objeto `alarmClock` precisa saber quando deve soar seu alarme, então por que não incluir um objeto da classe `alarmClock`? Essa capacidade é chamada de **composição**. Uma classe pode ter objetos de outras classes como membros.



Observação de engenharia de software 10.5

Uma forma comum de reusabilidade de software é a composição, em que uma classe tem objetos de outras classes como membros.

Quando um objeto é criado, seu construtor é chamado automaticamente. Anteriormente, vimos como passar argumentos para o construtor de um objeto que **chama este construtor**. Esta seção mostra como o construtor de um objeto pode passar argumentos para construtores de objeto-membro, o que é realizado via inicializadores de membro. Os objetos-membro são construídos na ordem em que são declarados na definição de classe (não na ordem em que são listados na lista de inicializadores de membro do construtor) e antes dos objetos da sua classe contêiner (às vezes **chamados** de construídos).

O programa das figuras 10.10–10.14 **utiliza** (Figura 10.10–10.11) e **explora** (Figuras 10.12–10.13) para demonstrar objetos como membros de outros objetos. A definição da classe `Date` contém membros que armazenam dados `firstName, lastName, birthDate` e `hireDate`. Os membros `birthDate` e `hireDate` são objetos da classe `Date`, que contêm os membros de dados `month`, `day` e `year`. O cabeçalho do construtor (Figura 10.13, linhas 18–21) especifica que o construtor recebe quatro parâmetros (`dateOfBirth` e `dateOfHire`). Os primeiros dois parâmetros são utilizados no corpo do construtor para inicializar os arrays de membros. Os dois últimos parâmetros são passados via inicializadores de membro para o construtor da classe `Date`. Dessa maneira, o cabeçalho separa os inicializadores de membro da lista de parâmetros. Os inicializadores de membro especificam os parâmetros **const** destinados para os construtores de objetos membro. O parâmetro `dateOfBirth` é passado para o construtor da classe `Date` (Figura 10.13, linha 20), e o parâmetro `dateOfHire` é passado para o construtor da classe `Date` (Figura 10.13, linha 21). Novamente, os inicializadores de membro são separados por vírgulas. Ao estudar `Date` (Figura 10.10), note que ela não fornece um construtor que recebe **const** parâmetros do tipo

```

1 // Figura 10.10: Date.h
2 // Definição da classe Date; funções-membro definidas em Date.cpp
3 #ifndef DATE_H
4 #define DATE_H
5
6 class Date
7 {
8 public :
9     Date( int month = 1, int day = 1, int year = 1900); // construtor-padrão
10    void print() const; // imprime data no formato mês/dia/ano
11    ~Date(); // fornecida para confirmar a ordem de destruição
12 private :
13    int month; // 1-12 (janeiro-dezembro)
14    int day; // 1-31 dependendo o mês
15
16    // função utilitária para verificar se o dia é adequado para o mês e ano
17    int checkDay(int ) const;
18 };
19 }; // fim da classe Date
20
21 #endif

```

Figura 10.10 Definição da classe `Date`

como a lista de inicializadores de membro no construtor da classe inicializar os objetos `hireDate` passando os parâmetros `date` e `hireDate`. Como mencionamos no Capítulo 9, o compilador fornece a cada classe um construtor de cópia padrão que copia cada membro do objeto de argumento do construtor para o membro correspondente sendo inicializado. O Capítulo 11 discute como os programadores podem definir construtores de cópia personalizados.

A Figura 10.14 cria dois objetos (`date1` e `date2`) e os passa como argumentos para o construtor `addObjeto` definido nas linhas 11–28 da Figura 10.11. Exibe uma linha de saída para mostrar que o construtor foi chamado (ver as duas primeiras linhas da saída de exemplo). Linha 13 da Figura 10.14 produz duas chamadas adicionais para o construtor `addObjeto` na saída do programa. Quando todos os objetos `date` são inicializados na lista de inicializadores de membro do construtor `Employee`, o construtor de cópia padrão é chamado. Esse construtor é definido implicitamente pelo compilador e não contém nenhuma instrução de saída para demonstrar quando ele é chamado. Discutimos os construtores de cópia e os construtores de cópia padrão em detalhes no Capítulo 11.]

A classe `Date` e `Employee` incluem, cada uma, um destrutor (linhas 37–42 da Figura 10.11 e linhas 51–55 da Figura 10.13, respec-

tivamente) que consiste em duas chamadas para o destrutor da classe `vector` para liberar a memória ocupada por os destruidos depois de objetos que os contêm. Note as últimas quatro linhas na saída da Figura 10.14. As duas últimas linhas são as saídas do destrutor executando nos objetos `date1` (linha 12) e `date2` (linha 11), respectivamente. Essas saídas confirmam que os três objetos criados foram destruídos na ordem inversa daquela em que foram criados. (A saída do destrutor é a quinta linha contando de cima para baixo.) A terceira e a quarta linhas contando de cima para baixo da janela de saída mostram os destrutores que executam para objetos `Employee` (Figura 10.12, linha 20) e `Date` (Figura 10.12, linha 19).

```

1 // Figura 10.11: Date.cpp
2 // Definições de função-membro da classe Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // inclui definição da classe Date
8
9 // construtor confirma valor adequado para month; chama
10 // função utilitária checkDay para confirmar o valor adequado para day
11 Date::Date( int mn, int dy, int yr )
12 {
13     if ( mn > 0 && mn <=12 ) // valida month
14         month = mn;
15     else
16     {
17         month = 1; // mês inválido configurado como 1
18         cout << "Invalid month (" << mn << ") set to 1.\n" ;
19     } // fim de else
20
21     year = yr; // poderia validar yr
22     day = checkDay( dy ); // valida day
23
24     // gera saída do objeto Date para mostrar quando seu construtor é chamado
25     cout << "Date object constructor for date " ;
26     print();
27 }
28 // construtor Date
29
30 // imprime objeto Date na forma de mês/dia/ano
31 void Date::print() const
32 {
33     cout << month << '/' << day << '/' << year;
34 } // fim da função print

```

Figura 10.11 Definições de função-membro da classe Date

(continua)

```

35
36 // gera saída do objeto Date para mostrar quando seu destrutor é chamado
37 Date::~Date()
38 {
39     cout << "Date object destructor for date " ;
40     print();
41     cout << endl;
42 } // fim do destrutor ~Date
43
44 // função utilitária para confirmar valor adequado de day
45 // com base em month e year; também trata anos bissextos
46 int Date::checkDay( int testDay ) const
47 {
48     static const int daysPerMonth[13] =
49     { 0, 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31 };
50
51     // determina se testDay é válido durante mês especificado
52     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53         return testDay;
54
55     // verificação 29 de fevereiro para ano bissexto
56     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57         ( year % 4 == 0 && year % 100 != 0 ) ) )
58         return testDay;
59
60     cout << "Invalid day (" << testDay << ") set to 1.\n" ;
61     return 1; // deixa o objeto em estado consistente se valor ruim
62 } // fim da função checkDay

```

Figura 10.11 Definições de função-membro da classe Date

(continuação)

```

1 // Figura 10.12: Employee.h
2 // Definição da classe Employee.
3 // Funções-membro definidas em Employee.cpp.
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include "Date.h" // inclui definição da classe Date
8
9 class Employee
10 {
11 public :
12     Employee( const char * const , const char * const ,
13                const Date &, const Date & );
14     void print() const ;
15     ~Employee(); // fornecida para confirmar a ordem de destruição
16
17     private:
18     char firstName[ 25];
19     char lastName[ 25];
20     const Date birthDate; // composição: objeto-membro
21     const Date hireDate; // composição: objeto-membro
22 };
23 #endif

```

Figura 10.12 Definição da classe Employee para mostrar a composição.

```

1 // Figura 10.13: Employee.cpp
2 // Definições de função-membro da classe Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos strlen e strcpy
8 using std::strlen;
9 using std::strcpy;
10
11 #include "Employee.h" // definição da classe Employee
12 #include "Date.h" // definição da classe Date
13
14 // construtor usa lista de inicializadores de membro para passar valores de inicializadores
15 // para construtores dos objetos-membro birthDate e hireDate
16 // [Nota: Isso invoca o chamado 'construtor de cópia padrão' que o
17 // compilador C++ fornece implicitamente.]
18 Employee::Employee( const char * const first, const char * const last,
19   const Date &dateOfBirth, const Date &dateOfHire )
20 : birthDate( dateOfBirth ), // inicializa birthDate
21   hireDate( dateOfHire ) // inicializa hireDate
22 {
23   // copia primeiro para firstName e certifica-se de que ele se ajusta
24   int length = strlen( first );
25   length = ( length < 25 ? length : 24 );
26   strcpy( firstName, first, length );
27   firstName[ length ] = '\0';
28
29   // copia por último para lastName e certifica-se de que ele se ajusta
30   length = strlen( last );
31   length = ( length < 25 ? length : 24 );
32   strcpy( lastName, last, length );
33   lastName[ length ] = '\0';
34
35   // gera saída do objeto Employee para mostrar quando o construtor é chamado
36   cout << "Employee object constructor: "
37     << firstName << " " << lastName << endl;
38 } // fim do construtor Employee
39
40 // imprime objeto Employee
41 void Employee::print() const
42 {
43   cout << lastName << ", " << firstName << " Hired: " ;
44   hireDate.print();
45   cout << " Birthday: " ;
46   birthDate.print();
47   cout << endl;
48 } // fim da função print
49
50 // gera saída do objeto Employee para mostrar quando seu destrutor é chamado
51 {
52
53   cout << "Employee object destructor: "
54     << lastName << ", " << firstName << endl;
55 } // fim do destrutor ~Employee

```

Figura 10.13 Definições de função-membro da classe Employee incluindo o construtor com uma lista de inicializadores de membro.

Essas saídas confirmam que o objeto destruído de fora para dentro — isto é, o destrutor é executado primeiro (saída mostrada na quinta linha contando de cima para baixo da janela de saída), então os objetos-membro são destruídos na ordem daquela em que foram construídos. Novamente, as saídas na Figura 10.14 não mostraram os construtores que executam para o porque esses eram os construtores de cópia padrão fornecidos pelo compilador C++.

Um objeto-membro não precisa ser explicitamente inicializado por um inicializador de membro. Se um inicializador de membro não for fornecido, o construtor-padrão do objeto-membro será chamado implicitamente. Os valores (se houver) estabelecidos pelo construtor-padrão podem ser sobreescritos se forem fornecidos para a inicialização complexa, essa abordagem pode exigir trabalho e tempo adicional significativos.



Erro comum de programação 10.6

Ocorre um erro de compilação se um objeto-membro não é inicializado com um inicializador de membro e se a classe do objeto-membro não fornece um construtor-padrão (isto é, a classe do objeto-membro define um ou mais construtores, mas nenhum deles é um construtor-padrão).

```

1 // Figura 10.14: fig10_14.cpp
2 // Demonstrando composição -- um objeto com objetos-membro.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // definição da classe Employee
8
9 int main()
10 {
11     Date birth( 7, 24, 1949);
12     Date hire( 3, 12, 1988);
13     Employee manager( "Bob", "Blue", birth, hire );
14
15     manager.print();
16
17     cout << "\nTest Date constructor with invalid values:\n" ;
18     Date lastDayOff( 14, 35, 1994); // mês e dia inválidos
19     cout << endl;
20     return 0;
21 }
22 } // fim de main

```

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

```

```
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949
```

```
Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
```

```
Date object constructor for date 1/1/1994
Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

Figura 10.14 Inicializadores de objeto-membro.



Dica de desempenho 10.2

Incialize explicitamente objetos-membro por meio de inicializadores de membro. Isso elimina o overhead de ‘inicializar duplamente’ objetos-membro — uma vez quando o construtor-padrão do objeto-membro é chamado, e a outra quando as funções set são chamadas no corpo do construtor (ou posteriormente) para inicializar o objeto-membro.



Observação de engenharia de software 10.6

Se um membro de classe for um objeto de outra classe, torna-se esse objeto encapsulado e a ocultação de membros desse objeto-membro. Entretanto, isso viola o encapsulamento e a ocultação da implementação da classe contêiner, portanto os objetos-membro dos tipos de classe ainda podem ser outros membros de dados.

Na linha 26 da Figura 10.11, note a chamada para a função `Employee::print()`. Muitas funções-membro de classes em C++ não requerem argumentos. Isso porque cada função-membro contém um handle implícito (na forma de um ponteiro) para o objeto em

ela. Apenas Discutimos o ponteiro implícito de classe na Seção 10.7.8 para representar o nome e sobrenome do employee. Esses arrays podem desperdiçar espaço com nomes com menos de 24 caracteres. (Lembre-se, um caractere em cada array para o caractere de terminação `\0`.) Além disso, nomes longos com mais de 24 caracteres devem ser truncados para se ajustarem nesses arrays de caracteres de tamanho fixo. A Seção 10.7 apresenta uma versão alternativa que a quantidade exata de espaço necessária para armazenar o nome e o sobrenome.

Observe que a maneira mais simples de representar o nome e sobrenome é utilizar dois `std::string`s. C++ Standard Library foi introduzida no Capítulo 3). Se fizéssemos isso, o construtor `Employee` apareceria da seguinte maneira:

```
Employee::Employee(const string &first, const string &last,
                   const Date &dateOfBirth, const Date &dateOfHire)
    : firstName(first), // inicializa firstName
      lastName(last), // inicializa lastName
      birthDate(dateOfBirth), // inicializa birthDate
      hireDate(dateOfHire) // inicializa hireDate
{
    // gera saída do objeto Employee para mostrar quando o construtor é chamado
    cout << "Employee object constructor: "
    // gerar saída de firstName e lastName
} // fim do construtor Employee
```

Note que os membros `firstName` e `lastName` (agora objetos) são inicializados por inicializadores de membro. As classes `Employee` apresentadas nos capítulos 12–13 utilizam desestruturação. Neste capítulo, utilizamos strings baseadas em ponteiro para fornecer ao leitor contato adicional com a manipulação de ponteiro.

10.4 Funções friend e classes friend

Uma `funcão friend` de uma classe é definida fora do escopo dessa classe, ainda que tenha `public` (pode acessar membros não-public) da classe. Funções independentes ou classes inteiras podem ser declaradas como amigas [

Utilizar funções `friend` pode aprimorar o desempenho. Esta seção apresenta um exemplo `friend` genérico de como uma função opera. Mais adiante no livro, veremos `friend` utilizadas para sobrepor operadores no uso com objetos de classe (Capítulo 11) e para criar as classes iteradoras (Capítulo 21). Os objetos de uma classe iteradora podem selecionar itens sucessivamente ou uma operação sobre itens em um objeto da classe contêiner (ver Seção 10.9). Os objetos de classes contêineres podem armazenar Utilizar funções `amigas` é frequentemente apropriado quando uma função-membro não pode ser utilizada para certas operações, como veremos no Capítulo 11.

Para declarar uma função como amiga de uma classe, preceda o protótipo de função na definição de classe com a palavra-chave `friend`. Para declarar todas as funções-membro de uma classe como amigas da classe, coloque uma declaração na

```
forma friend class ClassTwo;
na definição da classe
```



Observação de engenharia de software 10.7

Mesmo que os protótipos para funções sejam na definição de classe, as funções amigas não são funções-membro.



Observação de engenharia de software 10.8

As noções de acesso a `private`, `protected` e `public` não são relevantes às declarações `friend` — tanto as declarações `friend` podem ser colocadas em qualquer lugar de uma definição de classe.



Boa prática de programação 10.1

Coloque todas as declarações de amizade em primeiro lugar dentro do corpo da definição de classe e não as preceda com nenhum especificador de acesso.

A amizade é concedida, não aceita — isto é, para a classe B ser amiga da classe A, esta deve declarar explicitamente que a classe é sua amiga. Além disso, a relação de amizade não é nem simétrica nem transitiva; isto é, se a classe A é amiga da classe B e esta da classe C, você não pode inferir que a classe B seja amiga da classe A (novamente, a amizade não é simétrica), que a classe A seja amiga da classe B (também porque a amizade não é simétrica), ou que a classe A seja amiga da classe C (a amizade não é transitiva).



Observação de engenharia de software 10.9

Algumas pessoas na comunidade OOP acreditam que a ‘amizade’ corrompe o ocultamento de informações e enfraquece o valor da abordagem do projeto orientado a objetos. Neste texto, identificamos vários exemplos de uso responsável da amizade.

Modificando os dados `private` de uma classe com uma função amiga

A Figura 10.15 é um exemplo mecânico em que definimos a função `setX` para configurar o membro de dados `Count`. Observe que a declaração (linhas 10) aparece em primeiro lugar (por convenção) na definição de classe, mesmo antes das funções-membro serem declaradas. Novamente, essa declaração aparecer em qualquer lugar na classe.

A função `setX` (linhas 30–33) é uma função independente no estilo C — ela não é uma função-membro da classe. A razão é que é invocada para o objeto `a` (linhas 42–43) — passar como um argumento `a` para a função é mais conveniente do que utilizar um handle (como o nome do objeto) para chamar a função, como em

```
counter.setX( 8 );
```

Como mencionamos, a Figura 10.15 é um exemplo mecânico de uso da amizade. Normalmente também seria apropriado definir a função `setX` como uma função-membro da classe. Normalmente também seria apropriado separar o programa da Figura 10.15 em três arquivos:

1. Um arquivo de cabeçalho (por exemplo, contendo a definição da classe) por sua vez contém o protótipo de função `void setX`
2. Um arquivo de implementação (por exemplo, contendo as definições das funções-membro da classe) com a definição da função `setX`
3. Um programa de teste (por exemplo) com a função

Tentando modificar erroneamente um membro `private` com uma função não-`friend`

O programa da Figura 10.16 demonstra as mensagens de erro produzidas pelo compilador quando a função não-`friend` (linhas 29–32) é chamada para modificar o membro de dados.

É possível especificar funções sobrecarregadas como amigas de uma classe. Cada função sobrecarregada projetada para ser amiga deve ser explicitamente declarada na definição de classe como uma amiga da classe.

```

1 // Figura 10.15: fig10_15.cpp
2 // Friends podem acessar membros private de uma classe.
3 #include <iostream>

5 using std::endl;
6
7 // definição da classe Count
8 class Count
9 {
10     friend void setX( Count &, int ); // declaração friend
11 public :
```

Figura 10.15 Friends podem acessar os membros `private`.

(continua)

```

12 // construtor
13 Count()
14     : x( 0 ) // inicializa x como 0
15 {
16     // corpo vazio
17 } // fim do construtor de Count
18
19 // gera saída de x
20 void print() const
21 {
22     cout << x << endl;
23 } // fim da função print
24 private :
25     int x; // membro de dados
26 }; // fim da classe Count
27
28 // a função setX pode modificar os dados private de Count
29 // porque setX é declarada como uma amiga de Count (linha 10)
30 void setX( Count &c, int val )
31 {
32     c.x = val; // permitido pois setX Count é uma amiga de Count
33 } // fim da função setX
34
35 int main()
36 {
37     Count counter; // cria o objeto Count
38
39     cout << "counter.x after instantiation: " ;
40     counter.print();
41
42     setX( counter, 8 ); // configura x utilizando uma função friend
43     cout << "counter.x after call to setX friend function: " ;
44     counter.print();
45     return 0;
46 } // fim de main

```

counter.x after instantiation: 0
counter.x after call to setX friend function: 8

Figura 10.15 Friends podem acessar os membros privados.

(continuação)

```

1 // Figura 10.16: fig10_16.cpp
2 // Funções não-friend/não-membro não podem acessar dados private de uma classe.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Count (observe que não há declaração de amizade)
8 class Count
9 {
10 public :
11     // construtor
12     Count()
13     : x( 0 ) // inicializa x como 0

```

Figura 10.16 As funções nãofriend /não-membro não podem acessar membros private .

(continua)

```

14     {
15         // corpo vazio
16     } // fim do construtor de Count
17
18     // gera saída de x
19     void print() const
20     {
21         cout << x << endl;
22     } // fim da função print
23 private :
24     int x; // membro de dados
25 }; // fim da classe Count
26
27 // função cannotSetX tenta modificar dados private de Count,
28 // mas não pode porque a função não é amiga de Count
29 void cannotSetX( Count &c, int val )
30 {
31     c.x = val; // ERROR: não é possível acessar o membro private member em Count
32 } // fim da função cannotSetX
33
34 int main()
35 {
36     Count counter; // cria o objeto Count
37
38     cannotSetX( counter, 3 ); // cannotSetX não é friend
39     return 0;
40 } // fim de main

```

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2247 Fig10_16/fig10_16.cpp 31: 'Count::x' is not accessible in
function cannotSetX(Count &,int)
```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Count::x'
: cannot access private member declared in class 'Count'
    C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(24) : see declaration
of 'Count::x'
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(9) : see declaration
of 'Count'
```

Mensagens de erro do compilador GNU C++:

```
fig10_16.cpp:24: error: `int Count::x' is private
fig10_16.cpp:31: error: within this context
```

Figura 10.16 As funções não-friend / não-membro não podem acessar membros privados.

(continuação)

10.5 Utilizando o ponteiro this

Vimos que as funções-membro de um objeto podem manipular os dados do objeto. Como as funções-membro sabem quais membros do objeto devem manipular? Cada objeto tem acesso ao seu próprio endereço de memória chamado **this**. O ponteiro **this** é um objeto que não faz parte do objeto em si — isto é, o tamanho da memória ocupado pelo ponteiro **this** não é refletido no resultado de usar o operador **sizeof** para o objeto. Em vez disso, o ponteiro é passado (pelo compilador) como um argumento implícito para cada uma das funções-membro do objeto. Seção 10.7 introduz os membros de classe e explica por que o ponteiro **this** é passado implicitamente para funções-membro.

Os objetos utilizam o ponteiro implicitamente (como fizemos até agora) ou explicitamente para referenciar seus membros de dados e funções-membro. O tipo do ~~ponteiro~~ depende do tipo do objeto e do fato de a função-membro ser constante ou não declarada como const. Por exemplo, em uma função-membro não constante, o ponteiro tem o tipo Employee * const (um ponteiro constante para um objeto constante). Em uma função-membro constante da classe Employee, o ponteiro tem o tipo de dado Employee * const (um ponteiro constante para um objeto constante).

Nosso primeiro exemplo nesta seção mostra a utilização implícita e explícita de ponteiros no código do Capítulo 10 e no Capítulo 11, mostramos alguns exemplos substanciais resultantes da utilização de

Utilizando o ponteiro `this` implicitamente para acessar membros de dados de um objeto

A Figura 10.17 demonstra a utilização implícita e explícita de ponteiros que uma função-membro da classe imprime dados de um objeto.

Para fins de ilustração, a função `printData` (linhas 25–37) imprime ~~utilizando~~ implicitamente o ponteiro implicitamente (linha 28) — apenas o nome do membro de dados `Employee::printData` indica as variações diferentes para os diferentes tipos de ponteiros.

~~Observa-se~~ Operador ponto (`.`) e operador parêntese (`()`) devem ser usados juntos (operador ponto precede o nome do membro) (Figura 10.18) — o uso de operador ponto (`.`) sozinho é errado (Figura 10.19). Os operadores ponto e parêntese são necessários porque o operador ponto tem precedência maior do que o operador parêntese, o que é um erro de compilação, porque o operador ponto não pode ser utilizado com um ponteiro.

Uma utilização interessante do `ponteiro` é permitir que um objeto seja atribuído a si mesmo. Como veremos no Capítulo 11, a auto-atribuição pode causar sérios erros quando o objeto contiver ponteiros para armazenamento dinamicamente alocado.



Erro comum de programação 10.7

Tentar utilizar o operador de seleção (`.`) de membro para um objeto é um erro de compilação — o operador ponto de seleção de membro pode ser utilizado para o nome de um objeto, uma referência para um objeto ou um ponteiro desreferenciado para um objeto.

Utilizando o ponteiro `this` para permitir chamadas de função em cascata

Outra utilização do ponteiro `this` para permitir chamadas de função-membro em cascata é nas quais múltiplas funções são invocadas na mesma instrução (como na linha 14 da Figura 10.20). O programa das figuras 10.18–10.20 modifica as funções `setMinute` e `setSecond` da classe `Time` de modo que cada uma retorna uma referência para o próprio objeto chamadas de

`funções 26 e 26` (Figura 10.20). Note na Figura 10.20 que a última instrução no corpo de cadastro dessas funções-membro retorna o próprio objeto (`Time t`). Por que a técnica de retorno de uma referência funciona? O operador de seleção à esquerda para a direita, então a linha 14 primeiro avalia `t.setMinute(18)` e, em seguida, retorna uma referência para o próprio objeto dessa chamada de função. A expressão restante é então interpretada como

`t.setSecond(22);`

A chamada `t.setSecond(30)` executa e retorna uma referência ao próprio objeto restante é interpretada como

`t.setSecond(22);`

A linha 26 também utiliza cascataamento. As chamadas devem aparecer na ordem mostrada na linha 26, porque como definido na classe, não retorna uma referência à memória standard antes da chamada na linha 26 resulta em erro de compilação. O Capítulo 11 apresenta vários exemplos práticos do uso de chamadas de função em cascata. Um exemplo assim utiliza múltiplos operadores para gerar saída de múltiplos valores em uma única instrução.

10.6 Gerenciamento de memória dinâmico com os operadores new e delete

O C++ permite aos programadores controlar a alocação e desalocação de memória em um programa de qualquer tipo predefinido pelo usuário. Isso é conhecido como gerenciamento de memória dinâmico e é realizado com operadores `new` e `delete`.

Memorize-se daqui: Entendendo as figuras 10.12 e 10.13, utilize dois arrays de 25 caracteres para representar o nome e sobrenome ao declará-los como membros de dados, porque o tamanho destes dita a quantidade de memória requerida para armazenar um Employee. Como discutimos anteriormente, esses arrays podem desperdiçar espaço em nomes com menos de 24 caracteres. Além disso, os nomes com mais de 24 caracteres devem ser truncados para se ajustarem nesses arrays de tamanho fixo.

Não seria interessante se pudéssemos utilizar arrays com o número exato de elementos necessários para armazenar o nome e o sobrenome de um Employee? O gerenciamento de memória dinâmico permite fazer exatamente isso. Como veremos no exemplo da Seção 10.7, se substituirmos os membros de dados de array por ponteiros para alocar dinamicamente (isto é, reservar) a quantidade exata de memória necessária para armazenar todos os nomes em tempo de execução. Alocar memória dinamicamente desse modo faz com que um array (ou qualquer outro tipo predefinido ou definido

usuário) seja criado no **armazenamento livre** (às vezes chamado — uma região da memória atribuída a cada programa para armazenar objetos criados em tempo de execução. Uma vez que a memória de um array é alocada no armazenamento livre, podemos ganhar acesso a ela apontando um ponteiro para o primeiro elemento do array. Quando não precisamos mais dos arrays, podemos tornar a memória ao armazenamento livre utilizando o operador **delete** (isto é, liberar) a memória, que então pode ser reutilizada por futuras operações.

```

1 // Figura 10.17: fig10_17.cpp
2 // Utilizando o ponteiro this para referenciar membros de objeto.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 class Test
8 {
9 public :
10    Test( int = 0); // construtor-padrão
11    void print() const ;
12 private :
13    int x;
14 }; // fim da classe Test
15
16 // construtor
17 Test::Test( int value )
18 : x( value ) // inicializa x como value
19 {
20     // corpo vazio
21 } // fim do construtor Test
22
23 // imprime x utilizando ponteiros this implícito e explícito;
24 void Test::print() const
25 {
26     // utiliza implicitamente o ponteiro this para acessar o membro x
27     cout << "      x = " << x;
28
29     // utiliza explicitamente o ponteiro this e o operador seta
30     // para acessar o membro x
31     cout << "\n this->x = " << this ->x;
32
33     // utiliza explicitamente o ponteiro this desreferenciado e
34     // o operador ponto para acessar o membro x
35     cout << "\n(*this).x = " << (* this ).x << endl;
36
37 } // fim da função print
38
39 int main()
40 {
41     Test testObject( 12); // instancia e inicializa testObject
42
43     testObject.print();
44     return 0;
45 } // fim de main

```

```

x = 12
this->x = 12
(*this).x = 12

```

Figura 10.17 Ponteiro `this` acessando implicitamente e explicitamente os membros de um objeto.

```

1 // Figura 10.18: Time.h
2 // Colocando chamadas de função-membro em cascata.
3
4 // Definição da classe Time.
5 // Funções-membro definidas em Time.cpp.
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public :
12 Time( int = 0, int = 0, int = 0); // construtor-padrão
13
14 // funções set (os tipos de retorno Time & que permitem cascateamento)
15 Time &setTime(int , int , int ); // configura hour, minute, second
16 Time &setHour(int ); // configura hour
17 Time &setMinute(int ); // configura minute
18 Time &setSecond(int ); // configura second
19
20 // funções get (normalmente declaradas const)
21 int getHour() const ; // retorna hour
22 int getMinute() const ; // retorna minute
23 int getSecond() const ; // retorna second
24
25 // funções print (normalmente declaradas const)
26 void printUniversal() const ; // imprime hora universal
27 void printStandard() const ; // imprime a hora-padrão
28 private :
29 int hour; // 0 - 23 (formato de relógio de 24 horas)
30 int minute; // 0 - 59
31 int second; // 0 - 59
32 }; // fim da classe Time
33
34 #endif

```

Figura 10.18 Definição da classe Time modificada para permitir chamadas de função-membro em cascata.

```

1 // Figura 10.19: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // definição da classe Time
11 // função constructor para inicializar dados private;
12 // chama a função-membro setTime para configurar variáveis;
13 // valores-padrão são 0 (ver definição de classe)
14 // valores-padrão são 0 (ver definição de classe)
15 Time::Time( int hr, int min, int sec )
16 {
17     setTime( hr, min, sec );

```

Figura 10.19 Definições de função-membro da classe Time modificadas para permitir chamadas de função-membro em cascata.

(continua)

```

18 } // fim do construtor de Time
19
20 // configura valores de hour, minute e second
21 Time &Time::setTime( int h, int m, int s ) // observe o retorno Time &
22 {
23     setHour( h );
24     setMinute( m );
25     setSecond( s );
26     return * this ; // permite cascataemento
27 } // fim da função setTime
28
29 // configura valor de hour
30 Time &Time::setHour( int h ) // observe o retorno Time &
31 {
32     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
33     return * this ; // permite cascataemento
34 } // fim da função setHour
35
36 // configura valor de minute
37 Time &Time::setMinute( int m ) // observe o retorno Time &
38 {
39     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
40     return * this ; // permite cascataemento
41 } // fim da função setMinute
42
43 // configura valor de second
44 Time &Time::setSecond( int s ) // observe o retorno Time &
45 {
46     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
47     return * this ; // permite cascataemento
48 } // fim da função setSecond
49
50 // obtém valor da hora
51 int Time::getHour() const
52 {
53     return hour;
54 } // fim da função getHour
55
56 // obtém valor dos minutos
57 int Time::getMinute() const
58 {
59     return minute;
60 } // fim da função getMinute
61
62 // obtém valor dos segundos
63 int Time::getSecond() const
64 {
65     return second;
66 } // fim da função getSecond
67
68 // imprime a hora no formato universal de data/hora (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // fim da função printUniversal
74

```

Figura 10.19 Definições de função-membro da classe Time modificadas para permitir chamadas de função-membro em cascata.

(continua)

```

75 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
76 void Time::printStandard() const
77 {
78     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79     << ":" << setfill( '0' ) << setw( 2 ) << minute
80     << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // fim da função printStandard

```

Figura 10.19 Definições de função-membro da classe Time modificadas para permitir chamadas de função-membro em cascata. (continuação)

```

1 // Figura 10.20: fig10_20.cpp
2 // Colocando chamadas de função-membro em cascata com o ponteiro this.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // definição da classe Time
8
9 int main()
10 {
11     Time t; // cria o objeto Time
12
13     // chamadas de função em cascata
14     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
15
16     // gera saída da hora nos formatos universal e padrão
17     cout << "Universal time: " ;
18     t.printUniversal();
19
20     cout << "\nStandard time: " ;
21     t.printStandard();
22
23     cout << "\n\nNew standard time: " ;
24
25     // chamadas de função em cascata
26     t.setTime( 20, 20, 20 ).printStandard();
27     cout << endl;
28     return 0;
29 } // fim de main

```

Universal time: 18:30:22
 Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

Figura 10.20 Colocando função-membro em cascata.

Novamente, apresentamos a classe Time modificada como descrita aqui no exemplo da Seção 10.7. Primeiro, apresentamos os detalhes da utilização dos operadores para alocação dinâmica de memória para armazenamento de objetos, tipos fundamentais e arrays.

Considere a seguinte declaração e instrução:

```
Time *timePtr;
timePtr = new Time;
```

O operador `new` aloca armazenamento do tamanho adequado para o **objeto do tipo** padrão para inicializar o objeto e retorna um ponteiro do tipo especificado **adivinha é o mesmo**. Observe que pode ser utilizado para alocar dinamicamente qualquer tipo fundamental (o tipo de classe) se incapaz de localizar espaço suficiente na memória para o objeto, ele indica que ocorreu um erro 'lançando uma exceção'. O Capítulo 16, "Tratamento de exceção" discute como lidar com as exceções de texto do padrão ANSI/ISO C++. Em particular, mostraremos como 'capturar' a exceção lançada pelo tratá-la. Quando um programa não 'captura' uma exceção, ele é o operador `create`. [um ponteiro versões do C++ anteriores ao padrão ANSI/ISO. Utilizamos a versão padrão do operador `delete`]

Para destruir um objeto alocado dinamicamente e liberar o espaço para o objeto seguinte o operador

```
delete timePtr;
```

Essa instrução chama primeiro o destrutor do **objeto apontado** e, em seguida, desaloca a memória associada com o objeto. Depois da instrução precedente, a memória pode ser reutilizada pelo sistema para alocar outros objetos.



Erro comum de programação 10.8
Não liberar memória alocada dinamicamente quando ela não é mais necessária pode fazer com que o sistema fique sem memória prematuramente. Isso às vezes é chamado de **memória**.

O C++ permite fornecer **argumentos** para uma variável do tipo fundamental recém-criada, como em

```
double *ptr = new double ( 3.14159 );
```

que inicializa o **objeto** recém-criado com esse atribui o ponteiro resultante. A mesma sintaxe pode ser utilizada para especificar uma lista de argumentos separados por vírgulas para o construtor de um objeto. Por exemplo,

```
Time *timePtr = new Time( 12, 45, 0 );
```

inicializa um **objeto** recém-criado como 12:45 PM e atribui o ponteiro resultante a

Como discutido anteriormente, o operador `new` pode ser utilizado para alocar arrays dinamicamente. Por exemplo, um array de inteiros de 10 elementos pode ser alocado e atribuído como mostrado a seguir:

```
int *gradesArray = new int [ 10 ];
```

o que declara o **array** gradesArray atribui a ele um ponteiro para o primeiro elemento de um array de 10 elementos alocado dinamicamente de inteiros. Lembre-se de que o tamanho de um array criado em tempo de compilação deve ser especificado utilizando expressão integral constante. Entretanto, o tamanho de um array alocado dinamicamente deve ser especificado com expressão integral que seja avaliada em tempo de execução. Observe também que, ao alocar dinamicamente um array de objetos, o progr

Agora pode passar os argumentos para o array criado para o destrutor. Fazendo isso, cada objeto no array é inicializado por seu construtor

```
delete [] gradesArray;
```

A instrução precedente desaloca o array para **paparica**. Se o ponteiro na instrução precedente apontar para um array de objetos, a instrução primeiramente chamará o destrutor para cada objeto no array e, depois, desalocará a memória. Se a instrução `delete` não incluisse os colchetes, gradesArray apontasse para um array de objetos, somente o primeiro objeto no array receberia uma chamada de destrutor.



Erro comum de programação 10.9

Utilizar `delete` em vez de `delete []` para arrays de objetos pode levar a erros de lógica em tempo de execução. Para assegurar que cada objeto no array recebe uma chamada de destrutor, sempre exclua a memória **alocada** como array com o operador `[]`. De modo semelhante, sempre exclua a memória alocada como um elemento individual com o operador

10.7 Membros de classe static

Há uma exceção importante à regra que diz que cada objeto de uma classe tem sua própria cópia de todos os membros de dados. Em certos casos, apenas uma cópia de uma variável deve ser compartilhada por **todos os objetos** de uma classe. Um

compartilhado entre classes instâncias. Essa projeto de elemento é chamado de **classe** e é apropriada para a criação de classes que representam constantes que representam o número de criaturas alienígenas.

Vamos motivar ainda mais a necessidade de dados compartilhados entre classes. Suponha que tivéssemos um video-game `Martians` [marcianos] e outras criaturas do espaço. Gostaria de ser corajoso e a atacar outras criaturas espaciais quando `Martian` está ciente de que há pelo menos `Martians` presentes. Se menos de cinco estiverem presentes, cada individualmente torna-se covarde. Assim, precisaria saber a `MartianCount`. Poderíamos fornecer a cada instância da classe `Martian` um `MartianCount` como um membro de dados. Nesse caso, teríamos uma cópia separada do membro de dados. Toda vez que criarmos um `Martian`, terímos de atualizar o membro de dados de todos os objetos. Isso exigiria

que cada objeto tivesse, ou acessasse, handles para todos os outros objetos. Isso desperdiça espaço, com as cópias redundantes, e tempo, atualizando as cópias separadas. Enquanto os membros declarados fazem dados de escopo de classe. Todo `MartianCount` pode acessar `MartianCount` como se fosse um membro de dados do escopo da classe. Todo `MartianCount` é mantida pelo C++. Isso economiza espaço. Economizamos tempo fazendo o construtor `MartianCount` incrementar a variável `MartianCount` e o destrutor `MartianCount` decrementar. Como há somente uma cópia, não temos de incrementar ou decrementar cópias separadas do objeto.



Dica de desempenho 10.3

Utilize os membros de dados para poupar armazenamento quando uma única cópia dos dados de todos os objetos de uma classe for suficiente.

Embora possam parecer variáveis globais, os membros de dados escopos de classe. Além disso, os membros `static` podem ser declarados `private` ou `protected`. Um membro de dados tipo fundamental é inicializado `public` quando a classe é definida, mas pode ser inicializado de forma diferente quando a classe é instanciada. Os membros `static` são inicializados quando a classe é definida, mas podem ser redefinidos quando a classe é instanciada. Entretanto, todos os outros membros declarados são definidos no escopo de arquivo (isto é, fora do corpo da definição de classe) e inicializados somente nessas definições. Observe que o escopo de classe é, objetos-membro `static`) que têm construtores-padrão não precisam ser inicializados porque seus construtores-padrão serão chamados.

Os membros `protected static` de uma classe são normalmente acessados por `public` membros `friend` da classe. (No Capítulo 12, veremos que os membros `static` de uma classe também podem ser acessados por funções-membro da classe.) Os membros de uma classe existem até mesmo quando não existe nenhum objeto dessa classe. Para acessar um membro de classe quando não existe nenhum objeto da classe, simplesmente prefixe o nome de classe e o operador de resolução de escopo binário para o membro de dados. Por exemplo, se nossa variável `MartianCount` precedente, ela pode ser acessada com a expressão `MartianCount::MartianCount` quando não houver nenhum objeto `Martian`. (Naturalmente, o uso de `protected` é encorajado.)

Os membros de classe `static` de uma classe também podem ser acessados por qualquer objeto dessa classe utilizando o nome do objeto, o operador ponto e o nome do membro (por exemplo). Para acessar um membro de classe `private` ou `protected static` quando não houver nenhum objeto da classe, forneça uma função-chamador a função prefixando seu nome com o nome de classe e o operador de resolução de escopo binário. (Como veremos no Capítulo 12, o uso de `protected static` pode também servir a esse propósito.) Uma função-membro da classe, não de um objeto específico da classe,



Observação de engenharia de software 10.10

Os membros de classe funções-membro de uma classe existem e podem ser utilizados mesmo quando nenhum objeto dessa classe tiver sido instanciado.

O programa das figuras 10.21–10.23 demonstra um exemplo de `lastName` (Figura 10.21, linha 21) e uma função-membro `static` chamada `lastNameCount` (Figura 10.21, linha 15). Na Figura 10.22, a linha 14 define e inicializa o membro de dados `lastName` como zero no escopo de arquivo e as linhas 18–21 definem a função `lastName` que nem a linha 14 nem a linha 18 incluem a palavra-chave `static`; ambas as linhas referenciam membros de classe `lastName`. Quando `static` é aplicado a um item no escopo de arquivo, esse item torna-se conhecido apenas desse arquivo. Os membros de classe precisam estar disponíveis a partir de qualquer código-cliente que acesse o arquivo, portanto não podemos declará-los no arquivo — eles são declarados apenas no arquivo membro de dados, mantém uma contagem do número de objetos da classe que foram instanciados. Quando há objetos de classe, a função `lastName` pode ser referenciado por qualquer função-membro de um objeto. Figura 10.22 é referenciado tanto pela linha 33 no construtor como pela linha 48 no destrutor. Além disso, observe que esse poderia ter sido inicializado no arquivo de cabeçalho na linha 21 da Figura 10.21.



Erro comum de programação 10.10

É um erro de compilação incluir a palavra-chave `static` na definição de membros de dados no escopo de arquivo.

Na Figura 10.22, note o uso do operador `new` (linhas 27 e 30) no construtor para alocar dinamicamente a quantidade correta de memória para os membros `lastName`. Se o operador `new` incapaz de atender ao pedido de memória para um ou para esses dois arrays de caracteres, o programa terminará imediatamente. No Capítulo 16, forneceremos um mecanismo melhor para casos em que não consegue alocar memória.

Note também na Figura 10.22 que as implementações das funções `getLastName` (linhas 61–67) retornam ponteiros para os dados de classe. Nessa implementação, se o cliente quiser reter uma cópia do nome ou do sobrenome, ele será responsável por copiar a memória alocada dinamicamente depois de obter o ponteiro para os dados de caractere. Tanto é possível implementar `getLastName`, portanto o cliente é solicitado a passar um

array de caracteres e o tamanho dele para cada função. Então as funções poderiam copiar o nome ou o sobrenome no array de c fornecido pelo cliente. Mais uma vez, observe que poderíamos ~~utilizar~~ ~~retornar~~ uma cópia de um objeto string para o chamador em vez de retornar um ponteiro para os dados.

```

1 // Figura 10.21: Employee.h
2 // Definição da classe Employee.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 class Employee
7 {
8 public:
9     Employee( const char * const , const char * const ); // construtor
10    ~Employee(); // destrutor
11    const char *getFirstName() const; // retorna o nome
12    const char *getLastName() const; // retorna o sobrenome
13
14    // função-membro static
15    static int getCount(); // retorna número de objetos instanciados
16 private:
17    char *firstName;
18    char *lastName;
19
20    // dados static
21    static int count; // número de objetos instanciados
22}; // fim da classe Employee
23
24 #endif

```

Figura 10.21 Definição da classe Employee com um membro de dado static para monitorar o número de objetos Employee na memória.

```

1 // Figura 10.22: Employee.cpp
2 // Definições de função-membro da classe Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos de strlen e strcpy
8 using std::strlen;
9 using std::strcpy;
10
11 #include "Employee.h" // definição da classe Employee
12
13 // define e inicializa o membro de dados static no escopo de arquivo
14 int Employee::count = 0;
15
16 // define a função-membro static que retorna o número de
17 // objetos Employee instanciados (static declarado em Employee.h)
18 int Employee::getCount()
19 {
20     return count;
21 } // fim da função static getCount
22

```

Figura 10.22 Definições de função-membro da classe Employee

(continua)

```

23 // o construtor aloca dinamicamente espaço para o nome e o sobrenome e
24 // usa strcpy para copiar o nome e o sobrenome para o objeto
25 Employee::Employee( const char * const first, const char * const last )
26 {
27     firstName = new char [ strlen( first ) + 1 ];
28     strcpy( firstName, first );
29
30     lastName = new char [ strlen( last ) + 1 ];
31     strcpy( lastName, last );
32
33     count++; // incrementa contagem estática de empregados
34
35     cout << "Employee constructor for " << firstName
36     << " " << lastName << " called." << endl;
37 } // fim do construtor Employee
38
39 // o destrutor desaloca memória dinamicamente alocada
40 Employee::~Employee()
41 {
42     cout << "~Employee() called for " << firstName
43     << " " << lastName << endl;
44
45     delete [] firstName; // libera memória
46     delete [] lastName; // libera memória
47
48     count--; // decrementa contagem estática de empregados
49 } // fim do destrutor ~Employee
50
51 // retorna o nome do empregado
52 const char *Employee::getFirstName() const
53 {
54     // const antes do tipo de retorno impede que o cliente modifique
55     // dados private; o cliente deve copiar a string retornada antes de
56     // o destrutor excluir o armazenamento para impedir um ponteiro indefinido
57     return firstName;
58 } // fim da função getFirstName
59
60 // retorna sobrenome do empregado
61 const char *Employee::getLastName() const
62 {
63     // const antes do tipo de retorno impede que o cliente modifique
64     // dados private; o cliente deve copiar a string retornada antes de
65     // o destrutor excluir o armazenamento para impedir um ponteiro indefinido
66     return lastName;
67 } // fim da função getLastname

```

Figura 10.22 Definições de função-membro da classe Employee

(continuação)

A Figura 10.23 utiliza a função `static int getCount()` para determinar o número de objetos `Employee` realmente instanciados. Observe que, quando nenhum objeto é instanciado no programa, a função é chamada de `tempo` (linhas 14 e 38). Entretanto, quando objetos são instanciados, pode-se chamar por qualquer um dos objetos, como mostrado na instrução das linhas 22–23 que utiliza `tempo` para invocar a função. Observe que `tempo.getCount()` OLEEmployee::getCount() na linha 23 produziria o mesmo resultado, já que `tempo` sempre acessa o mesmo membro.



Observação de engenharia de software 10.11

Algumas organizações especificam em seus padrões de engenharia de software que todas as chamadas a funções-membro são feitas utilizando o nome da classe, e não um handle de objeto.

```

1 // Figura 10.23: fig10_23.cpp
2 // Driver para testar a classe Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // Definição da classe Employee
8
9 int main()
10 {
11     // utiliza o nome da classe e o operador de resolução de escopo binário para
12     // acessar a função static number getCount
13     cout << "Number of employees before instantiation of any objects is "
14         << Employee::getCount() << endl; // utiliza o nome da classe
15
16     // utiliza new para criar dinamicamente dois novos Employees
17     // operador new também chama o construtor do objeto
18     Employee *e1Ptr = new Employee("Susan", "Baker");
19     Employee *e2Ptr = new Employee("Robert", "Jones");
20
21     // chama getCount no primeiro objeto Employee
22     cout << "Number of employees after objects are instantiated is "
23         << e1Ptr->getCount();
24
25     cout << "\n\nEmployee 1: "
26         << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
27         << "\nEmployee 2: "
28         << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n" ;
29
30     delete e1Ptr; // desaloca memória
31     e1Ptr = 0; // desconecta o ponteiro do espaço de armazenamento livre
32     delete e2Ptr; // desaloca memória
33     e2Ptr = 0; // desconecta o ponteiro do espaço de armazenamento livre
34
35     // não existe nenhum objeto, portanto chama a função-membro static getCount
36     // utilizando o nome da classe e o operador de resolução de escopo binário
37     cout << "Number of employees after objects are deleted is "
38         << Employee::getCount() << endl;
39     return 0;
40 } // fim de main

```

Number of employees before instantiation of any objects is 0

Employee constructor for Susan Baker called.

Employee constructor for Robert Jones called.

Number of employees after objects are instantiated is 2

Employee 1: Susan Baker

Employee 2: Robert Jones

~~~Employee()~~ called for Susan Baker

~~~Employee()~~ called for Robert Jones

Number of employees after objects are deleted is 0

Figura 10.23 Membro de dados static monitorando o número de objetos de uma classe.

Uma função-membro deve ser declarada para acessar membros de dados da classe. Ao contrário das funções-membro, uma função-membro não tem um ponteiro porque os membros de dados existem independentemente de qualquer objeto de uma classe. O ponteiro referenciar um objeto específico da classe e, quando uma função-membro não haja nenhum objeto de sua classe na memória.



Erro comum de programação 10.11

Utilizar o ponteiro em uma função-membro é um erro de compilação.



Erro comum de programação 10.12

Declarar uma função-membro é um erro de compilação. O compilador que uma função não pode ser declarada dentro do objeto em que ela opera, mas as funções-membro são independentemente de qualquer classe.

As linhas 18–19 da Figura 10.23 utilizam o `new` para alocar dinamicamente dois objetos.

O programa terminará imediatamente se não conseguir alocar um ou ambos os objetos. Quando cada objeto construtor é chamado. Quando utilizado nas linhas 30 e 32 para desalocar os objetos, o destrutor de cada objeto é chamado.



Dica de prevenção de erro 10.2

Depois de excluir a memória alocada dinamicamente, o código que referencia essa memória. Isso desconecta o ponteiro do espaço anteriormente alocado no armazenamento livre. Esse espaço na memória ainda poderia conter informações apesar de ter sido excluído. Configurando o código para perder qualquer acesso a esse espaço de armazenamento livre, o qual, de fato, já poderia ter sido realocado para um propósito diferente. Se você não configurasse o código como poderia acessar inadvertidamente essas novas informações, causando erros de lógica extremamente sutis e não repetíveis.

10.8 Abstração de dados e ocultamento de informações

Em geral, uma classe oculta de seus clientes os detalhes da sua implementação. Isso se chama ocultamento de informações. C

As pilhas ocultam implementações concretas de estruturas de dados, pilhas e pilhas vinculadas. Discutimos as pilhas e vinculadas no Capítulo 14, "Templates", e no Capítulo 21, "Estruturas de dados".) Um cliente de uma classe na pilha não precisa preocupar com a implementação da pilha. O cliente sabe apenas que, quando itens de dados forem colocados na pilha, eles serão novamente na ordem do último a entrar, primeiro a sair. O cliente pode usar a pilha oferecendo com essa funcionalidade é implementada. Esse conceito é referido. Embora os programadores talvez conheçam os detalhes da implementação de uma classe, eles não devem escrever código que dependa desses detalhes. Isso permite a um particular (como uma que implementa uma pilha) ser substituída por uma outra versão sem afetar o restante do sistema. Contanto que os serviços não mudem (isto é, cada função-membro ainda tem o mesmo protótipo na nova definição de classe), o restante do sistema não é afetado.

Muitas linguagens de programação enfatizam as ações. Nessas linguagens, os dados existem para suportar as ações que os podem tomar. Os dados são 'menos interessantes' que as ações. Os dados são 'brutos'. Há apenas alguns tipos de dados predefinidos para os programadores criarem seus próprios tipos. O C++ e o estilo de programação orientado a objetos elevam a importância dos dados. As principais atividades da programação em C++ orientada a objetos são a criação de tipos (isto é, classes) e a expressão de interações entre os objetos desses tipos. Para criar linguagens que enfatizam dados, a comunidade das linguagens de programação precisou formalizar algumas noções sobre os dados. A formalização que impõe considerações à noção de (*abstract data types ADT*), que melhora o processo de desenvolvimento de programas.

O que é um tipo de dado abstrato? Considere o tipo de dado que as pessoas associaria com um inteiro em matemática. Em vez disso, uma representação abstrata de um inteiro. Diferentemente dos inteiros matemáticos, computador têm tamanho fixo. Por exemplo, máquinas populares de 32 bits atuais é, em geral, limitado ao intervalo de -2.147.483.648 a +2.147.483.647. Se o resultado de um cálculo cai fora desse intervalo, ocorre um erro de estouro e o computador responde de alguma maneira dependente da máquina. Por exemplo, ele poderia produzir um resultado incorreto 'silenciosamente', um valor muito grande para caber em um inteiro matemático chamado aritmético. Inteiros matemáticos não têm esse problema. Portanto, a noção de computador é somente uma aproximação da noção de um inteiro do mundo real. O mesmo se aplica a浮点数.

Mesmo que é uma aproximação; os valores normalmente padrões de oito bits de uns e zeros; esses padrões não se parecem em nada com os caracteres que representam maiúsculas e minúsculas, um sinal de círculo grego e assim por diante. Os valores da maioria dos computadores são limitados comparados ao intervalo de caracteres do mundo real.

real. O conjunto de caracteres ASCII de sete bits (Apêndice B) oferece 128 valores de caracteres diferentes. Isso é inadequado para sentir línguas como japonês e chinês, que requerem milhares de caracteres. Como o uso da Internet e da World Wide Web torna vez mais disseminado, a popularidade do mais novo conjunto de caracteres Unicode está crescendo rapidamente, devido à sua capacidade de representar os caracteres da maioria das linguagens. Para informações adicionais sobre o Unicode, visite

A questão é que mesmo os tipos de dados predefinidos fornecidos pelas linguagens de programação como o C++ são realmente apenas aproximações ou modelos imperfeitos de conceitos e comportamentos do mundo real. Até aqui, assumimos garantido, mas agora temos uma nova perspectiva a considerar. Tipos abstratos são todos exemplos de tipos de dados abstratos. Eles são essencialmente maneiras de representar noções do mundo real em algum nível satisfatório de precisão de um sistema de computador.

Um tipo de dados abstrato na realidade captura duas noções: duas operações que podem ser realizadas nesses dados. Por exemplo, `operator<` contém valor do tipo inteiro (dados) e fornece as operações adição, subtração, multiplicação, divisão e módulo (entre outras) — a divisão por zero é indefinida. Essas operações permitidas são realizadas de uma forma sensível a parâmetros de máquina, como o tamanho fixo de palavras do sistema de computador subjacente. Outro exemplo é a

~~é feito de string. Quando se programa, é sempre útil ter classes para lidar com strings. Por exemplo, para implementar um ADT de pilha, criamos nossas próprias classes de pilha no Capítulo 14, “Templates”, e no Capítulo 21, “Estruturas de dados”, e estudamos a biblioteca-padrão do Capítulo 23, “Standard Template Library (STL)”.~~

10.8.1 Exemplo: tipo de dados abstrato array

Discutimos os arrays no Capítulo 7. Como descrito, um array não é nada mais do que um ponteiro e um espaço na memória. Essa simplicidade primitiva é aceitável para realizar operações de array se o programador for atencioso e flexível. Há muitas operações que são elegantes se realizadas com arrays, mas que não são construídas em C++. Com as classes C++, o programador pode desenvolver um ADT de array que é preferível aos arrays ‘brutos’. A classe de array pode fornecer muitas novas capacidades úteis como

- verificação de intervalo de subscripto;
- um intervalo arbitrário de subscritos em vez de ter de iniciar com 0;
- atribuição de array;
- comparação de array;
- entrada/saída de array;

- arrays que sabem seus tamanhos;
- arrays que se expandem dinamicamente para acomodar mais elementos;
- arrays que podem imprimir a si mesmos em um formato tabular organizado.

Criamos nossas próprias classes de array com muitas dessas capacidades no Capítulo 11, “Sobrecarga dos operadores; `string` e `array`”. Lembre-se de que o template `class` Standard Library (introduzido no Capítulo 7) também fornece muitas dessas capacidades. O Capítulo 23 explica o template `class`. C++ tem um pequeno conjunto de tipos predefinidos. As classes estendem a linguagem de programação básica com novos tipos.



Observação de engenharia de software 10.12

O programador é capaz de criar novos tipos pelo mecanismo de classe. Esses novos tipos podem ser projetados para ser utilizados convenientemente quanto os tipos predefinidos. Portanto, o C++ é uma linguagem extensível. Embora a linguagem seja fácil de estender com esses novos tipos, a linguagem básica em si não pode ser alterada.

Novas classes criadas em ambientes C++ podem ser ‘proprietárias’ (isto é, patenteadas) de um indivíduo, pequenos grupos ou empresas. As classes também podem ser colocadas em bibliotecas de classes padrão destinadas à ampla distribuição. O ANSI (American National Standards Institute) e a ISO (International Organization for Standardization) desenvolveram uma versão-padrão de C++. Inclui uma biblioteca de classes padrão. O leitor que aprende C++ e a programação orientada a objetos estará pronto para tirar vantagens desses tipos de desenvolvimento rápido de softwares orientado a componentes possibilitados por bibliotecas cada vez mais

10.8.2 Exemplo: tipo de dados abstrato string

O C++ é uma linguagem intencionalmente esparsa que fornece aos programadores somente as capacidades elementares necessárias para criar uma ampla série de sistemas (considere-o uma ferramenta para fazer ferramentas). A linguagem é projetada para minimizar o fardo dos problemas de desempenho. O C++ é uma linguagem adequada para a programação tanto de aplicativos como de sistemas — esta última impõe uma extraordinária demanda de desempenho sobre os programas. Certamente, teria sido possível incluir uma classe `string` entre os tipos de dados predefinidos do C++. Em vez disso, a linguagem foi projetada incluindo mecanismos para

e implementar tipos de dados abstratos string por meio de classes. Introduz Standard Library no Capítulo 3 e desenvolveremos nossa própria ADT no Capítulo 11. Discutiremos a classe detalhes no Capítulo 18.

10.8.3 Exemplo: tipo de dados abstrato queue

Todo mundo uma vez ou outra precisa pegar uma fila. Em inglês uma fila. Esperamos na fila de caixa do supermercado, esperamos na fila do posto de gasolina, esperamos na fila do ônibus, esperamos na fila do pedágio e todos os est conhecem muito bem a fila de espera para fazer matrícula. Os sistemas de computador utilizam muitas filas de espera internam então precisamos escrever programas que simulam o que são filas e o que elas fazem.

Uma fila é um bom exemplo de um tipo de dados abstrato. As filas oferecem comportamentos bem conhecidos para seus clientes. Os clientes colocam uma coisa por vez em uma fila — invocando a operação de fila — e recuperam essas coisas individualmente por demanda — invocando a operação de fila. Conceitualmente, uma fila pode tornar-se infinitamente longa. Uma fila real é naturalmente finita. Os itens são retornados de uma fila na ordem primeiro a sair(first-in, first-out—FIFO) — o primeiro item inserido na fila é o primeiro item removido dela.

A fila oculta uma representação interna dos dados que de algum modo monitora os itens que atualmente esperam na fila e oferece um conjunto de operações para seus clientes. Os clientes não estão preocupados com a implementação da fila. Eles simplesmente querem que a fila opere ‘como anunciado’. Quando um cliente enfileira um novo item, a fila deve aceitar esse item e colocá-lo internamente em alguma estrutura de dados do tipo primeiro a entrar, primeiro a sair. Qual cliente quiser o próximo item na frente da fila, esta deve remover o item de sua representação interna e entregá-lo para o mundo e (isto é, o cliente da fila) na ordem FIFO (isto é, o item que esteve na fila por mais tempo deve ser o próximo retornado pela operação de desenfileiramento).

A fila ADT garante a integridade de sua estrutura de dados interna. Os clientes não podem manipular diretamente essa estrutura de dados. Apenas as funções-membro de fila têm acesso aos seus dados internos. Os clientes podem fazer com que somente as operações admissíveis sejam realizadas na representação de dados; operações não fornecidas na interface pública do ADT são rejeitadas de maneira apropriada. Isso poderia significar emitir uma mensagem de erro, lançar uma exceção (ver Capítulo 16), terminar a execução ou simplesmente ignorar a solicitação de operação.

Criamos nossa própria classe de fila no Capítulo 21, “Estruturas de dados: Standard Library”, e no Capítulo 23, “Standard Template Library (STL)”.

10.9 Classes contêineres e iteradores

Entre os tipos de classes mais populares estão as classes contêineres (também chamadas de coleções), isto é, classes projetadas para armazenar coleções de objetos. As classes contêineres fornecem comumente serviços como inserção, exclusão, pesquisa e iteração de um item para determinar se ele está ou não em memória da coleção. Os arrays, pilhas, filas, árvores e listas vinculadas são exemplos de classes contêineres; estudamos arrays no Capítulo 7 e estudaremos cada uma dessas outras estruturas de dados no Capítulo 21, “Estruturas de dados”, e no Capítulo 23, “Standard Template Library (STL)”.

É comum associar iteradores — ou mais simplesmente — com classes contêineres. O iterador é um objeto que ‘percorre’ uma coleção, retornando o próximo item (ou realizando alguma ação nele). Uma vez que um iterador de uma classe é escrito, obter o próximo elemento da classe pode ser expressado de maneira simples. Exatamente como um livro compartilhado por várias pessoas poderia ter vários marcadores de uma vez, uma classe contêiner pode ter vários iteradores operacionais de uma vez. Um iterador mantém suas próprias informações de ‘posição’. Discutiremos contêineres e iteradores em detalhes no Capítulo 23.

10.10 Classes proxy

Lembre-se de que dois dos princípios fundamentais da boa engenharia de software são separar a interface da implementação e detalhes da implementação. Esforçamo-nos para alcançar esses objetivos definindo uma classe em um arquivo de cabeçalho e mantendo suas funções-membro em um arquivo de implementação separado. Entretanto, como indicamos no Capítulo 9, os arquivos de cabeçalho também têm alguma parte da implementação de uma classe e dicas sobre outras. Por exemplo, os membros de uma classe são listados na definição de classe em um arquivo de cabeçalho, então esses membros são visíveis aos clientes, que os clientes não possam acessar diretamente. Informações de uma classe dessa maneira potencialmente expõe

informações que não precisam ser acessadas. Agora é o momento de que entendemos que é importante que o cliente que conhece somente a interface pública para sua classe permita aos clientes utilizar os serviços da sua classe sem dar acesso de cliente aos detalhes de implementação de sua classe.

Implementar uma classe proxy exige vários passos, que demonstramos nas figuras 10.24–10.27. Primeiro, criamos a definição de classe para a classe que contém a implementação proprietária que gostaríamos de ocultar. Nossa classe de exemplo, chamada `implementation`, é mostrada na Figura 10.24. A classe proxy, mostrada nas figuras 10.25–10.26. O programa de teste e a saída de exemplo são mostrados na Figura 10.27.

A classe `implementation` (Figura 10.24) fornece um único membro de dados, `value` (os dados que gostaríamos de ocultar do cliente), um construtor para inicializar `value` e um método `getValue`.

```

1 // Figura 10.24: Implementation.h
2 // Arquivo de cabeçalho para a classe Implementation
3
4 class Implementation
5 {
6 public :
7     // construtor
8     Implementation( int v )
9         : value( v ) // inicializa value como v
10    {
11        // corpo vazio
12    } // fim do construtor de Implementation
13
14    // configura valor como v
15    void setValue( int v )
16    {
17        value = v; // deve validar v
18    } // fim da função setValue
19
20    // retorna value
21    int getValue() const
22    {
23        return value;
24    } // fim da função getValue
25 private :
26     int value; // dados que gostaríamos de ocultar do cliente
27 }; // fim da classe Implementation

```

Figura 10.24 Definição da classe `Implementation`

```

1 // Figura 10.25: Interface.h
2 // Arquivo de cabeçalho da classe Interface
3 // O cliente vê esse código-fonte, mas o código-fonte não revela
4 // o layout de dados da classe Implementation.
5
6 class Implementation; // declaração de classe antecipada requerida pela linha 17
7
8 class Interface
9 {
10 public :
11     Interface( int ); // construtor
12     void setValue( int ); // mesma interface public que
13     int getValue() const ; // a classe Implementation tem
14     ~Interface(); // destrutor
15 private :
16     // requer a declaração antecipada anterior (linha 6)
17     Implementation *ptr;
18 }; // fim da classe Interface

```

Figura 10.25 Definição da classe `Interface`.

Definimos uma classe proxy ([Figura 10.25](#)) com uma interface (exceto pelos nomes do construtor e do destrutor) àquela classe. O único membro da classe proxy é um ponteiro para um objeto da classe Implementation. Utilizar um ponteiro dessa maneira permite ocultar do cliente os detalhes de implementação da classe. Note que as únicas menções à classe proxy na implementação proprietária estão na declaração de ponteiro (linha 17) e na linha 6 ([Declaração de classe antecipa Edward class declarati](#)). Quando uma definição de classe (como a classe Interface) utiliza somente um ponteiro para ou referência a um objeto de outra classe, o cliente não precisa incluir esse arquivo de cabeçalho de classe para essa outra classe (que pode ser usada por outras classes). Você pode simplesmente declarar essa outra classe como um tipo de dados com uma declaração de classe antecipa (linha 6) antes de o tipo ser utilizado no arquivo.

O arquivo de implementação de função-membro da classe proxy ([Figura 10.26](#)) é o único arquivo que inclui o arquivo de cabeçalho Implementation.h (linha 5) contendo a classe Implementation. O arquivo Interface.cpp ([Figura 10.26](#)) é fornecido para o cliente como um arquivo de código-objeto pré-compilado junto com o arquivo de cabeçalho. Os tipos de função dos serviços fornecidos pela classe proxy são compilados e disponibilizados para o cliente somente como código-

objeto proxy também é separado da classe proxy e o cliente não vê o código proxy (consulte as páginas 9, 17, 23 e 29). Considerando a velocidade dos computadores atuais e o fato de que muitos compiladores podem colocar inline chama função simples automaticamente, o efeito dessas chamadas de função extra no desempenho é freqüentemente insignificante.

A Figura 10.27 testa a classe. Note que apenas o arquivo de cabeçalho é usado no código-cliente (linha 7) — não há nenhuma menção da existência de uma classe separada. Quando o cliente nunca vê os dados private da classe Implementation nem o código-cliente pode tornar-se dependente do código.



Observação de engenharia de software 10.13

Uma classe proxy isola o código-cliente das alterações na implementação.

```

1 // Figura 10.26: Interface.cpp
2 // Implementação da classe Interface – o cliente recebe somente esse arquivo
3 // como código-objeto pré-compilado, mantendo a implementação oculta.
4 #include "Interface.h" // Definição da classe Interface
5 #include "Implementation.h" // Definição da classe Implementation
6
7 // construtor
8 Interface::Interface( int v )
9 : ptr ( new Implementation( v ) ) // inicializa ptr para apontar para
10 { // corpo vazio
11 // um novo objeto Implementation
12 } // fim do construtor Interface
13
14 // chama a função setValue de Implementation
15 void Interface::setValue( int v )
16 {
17     ptr->setValue( v );
18 } // fim da função setValue
19
20 // chama a função getValue de Implementation
21 int Interface::getValue() const
22 {
23     return ptr->getValue();
24 } // fim da função getValue
25
26 // destrutor
27 Interface::~Interface()
28 {
29     delete ptr;
30 } // fim do destrutor ~Interface

```

Figura 10.26 Definições de função-membro da classe Interface.

```

1 // Figura 10.27: fig10_27.cpp
2 // Ocultando dados privados de uma classe com a classe proxy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Interface.h" // Definição da classe Interface
8
9 int main()
10 {
11     Interface i( 5 ); // cria objeto Interface
12
13     cout << "Interface contains: " << i.getValue()
14     << endl;
15
16     i.setValue( 10 );
17
18     cout << "Interface contains: " << i.getValue()
19     << endl;
20     return 0;
21 } // fim de main

```

Interface contains: 5 before setValue
 Interface contains: 10 after setValue

Figura 10.27 Implementando uma classe proxy.

10.11 Síntese

Neste capítulo, você aprendeu sobre vários tipos de modificadores de acesso e sobre classes abstratas. Você também aprendeu a usar a palavra-chave friend para implementar funções que operam diretamente sobre membros privados de uma classe. Introduzimos o tópico de friendship e apresentamos exemplos que demonstram como utilizar funções.

Você aprendeu que o operador this é passado como um argumento implícito para cada uma das funções-membro não-constante de uma classe, permitindo às funções acessar membros de dados do objeto correto. Você também aprendeu não-o uso explícito do ponteiro para acessar os membros da classe e permitir chamadas de função-membro em cascata.

O capítulo introduziu o conceito de gerenciamento de memória dinâmico. Você aprendeu que programadores em C++ podem e destruir objetos dinamicamente com o operador new. Vamos a necessidade dos membros de dados declarados como declarar e utilizar membros efêmeros de classes-membro em suas próprias classes.

Você aprendeu a abstração de dados e o ocultamento de informações — dois dos conceitos fundamentais da programação orientada a objetos. Discutimos os tipos de dados abstratos — as maneiras de representar noções do mundo real ou noções conceituais em satisfatório dentro de um sistema de computador. Então você aprendeu sobre os três tipos de dados abstratos de C++ — arrays, strings e filas. Introduzimos o conceito de uma classe contêiner que armazena uma coleção de objetos, bem como a criação de uma classe iteradora que percorre os elementos de uma classe contêiner. Por fim, você aprendeu a criar uma classe proxy para os detalhes de implementação (incluindo os membros de classe privados) e a ocultar esses detalhes dos clientes da classe.

No Capítulo 11, continuamos nosso estudo de classes e objetos mostrando como permitir que operadores do C++ funcionem com objetos — um processo chamado sobrecarga dos operadores. Por exemplo, você verá como o operador [] pode ser utilizado para gerar saída de um array completo sem usar uma instrução de repetição explicitamente.

Resumo

- A palavra-chave const pode ser utilizada para especificar que um objeto não é modificável e que qualquer tentativa de modificá-lo deve resultar em um erro de compilação.
- Os compiladores C++ não permitem chamadas de função-membro constante — uma tentativa de modificar um objeto de sua classe gerará um erro de compilação.

- Uma função é especificada tanto em seu protótipo como em sua definição.
- Um objeto deve ser inicializado, não atribuído.
- Os construtores e destrutores não podem ser declarados
- O membro de dados os membros de dados que são referências devem ser inicializados com inicializadores de membro.
- Uma classe pode ter objetos de outras classes como membros — esse conceito é chamado composição.
- Os objetos-membro são construídos na ordem em que são declarados na definição de classe e antes de seus objetos da classe contêiner : construídos.
- Se um objeto-membro não receber um inicializador de membro, o construtor-padrão do objeto-membro será chamado implicitamente.
- Uma função de uma classe é definida fora do escopo dessa classe, ainda que ela tenha opção de acessar membros não-public da classe. Funções independentes ou classes inteiras podem ser declaradas amigas de outra classe.
- Uma declaração amiga pode aparecer em qualquer lugar da classe. Uma amiga é especialmente essa parte da interface
- A relação de amizade não é simétrica nem transitiva.
- Cada objeto tem acesso ao seu próprio endereço pelo ponteiro
- O ponteiro de um objeto não faz parte do objeto em si — isto é, o tamanho da memória ocupada pelo ponteiro resultado de uma operação de memória.
- O ponteiro é passado (pelo compilador) como um argumento implícito para cada membro de função que não
- Os objetos utilizam o ponteiro implicitamente (como fizemos até agora) ou explicitamente para referenciar seus membros de dados e funções-membro.
- O ponteiro permite chamadas de função-membro em cascata em que múltiplas funções são invocadas na mesma instrução.
- O gerenciamento de memória dinâmico permite aos programadores controlar a alocação e a desalocação de memória em um programa p: qualquer tipo predefinido ou definido pelo usuário.
- O armazenamento livre (às vezes chamado de heap) é uma região da memória atribuída a cada programa para armazenar objetos dinamicamente alocados em tempo de execução.
- O operador `new` aloca o armazenamento do tamanho adequado para um objeto, executa o construtor do objeto e retorna um ponteiro do tipo correto. O operador `new` pode ser utilizado para alocar dinamicamente qualquer tipo fundamental ou tipo de classe. Se não conseguir localizar o espaço na memória para o objeto, ele indica que ocorreu um erro 'lançando' uma 'exceção'. Isso normalmente faz com que o programa termine imediatamente.
- Para destruir um objeto dinamicamente alocado e liberar o espaço para o objeto, utilize o operador
- Um array de objetos pode ser alocado dinamicamente com

```
int *ptr = new int [ 100];
```

que aloca um array de 100 inteiros e atribui a localização. O array anterior é excluído com a instrução

```
delete [] ptr;
```

- Um membro de dados representa informações 'no nível de classe' (isto é, uma propriedade da classe compartilhada por todas as instâncias, não uma propriedade de um objeto específico da classe).
- Membros de dados têm escopo de classe e podem ser declarados protected.
- Os membros de classe de uma classe existem até mesmo quando não existe nenhum objeto dessa classe.
- Para acessar um membro de classe quando não existe nenhum objeto da classe, simplesmente prefixe o nome de classe e o operador de resolução de escopo com o nome do membro de dados.
- Os membros de classe static de uma classe podem ser acessados por qualquer objeto dessa classe.
- Uma função-membro deve ser declarado para acessar membros de dados ou funções-membro da classe. Ao contrário das funções-membro, nenhuma função-membro não tem um ponteiro porque os membros de dados e as funções-membro existem independentemente de qualquer objeto de uma classe.
- Os tipos de dados abstratos são formas de representar noções do mundo real e noções conceituais em um nível satisfatório de precisão de um sistema de computador.
- Um tipo de dados abstrato captura duas noções: uma representação de dados e as operações que podem ser realizadas nesses dados.
- O C++ é uma linguagem intencionalmente esparsa que fornece aos programadores as capacidades elementares necessárias para criar uma série de sistemas. O C++ é projetado para minimizar cargas de desempenho.
- Os itens são retornados de uma fila na ordem primeiro inserido, último sair. O primeiro item inserido na fila é o primeiro item removido dela.

- As classes contêineres (também chamadas classes de coleção) são projetadas para armazenar coleções de objetos. As classes contêm nem comumente serviços como inserção, exclusão, pesquisa, classificação e teste de um item para determinar se ele é ou não um membro da coleção.
- É comum associar iteradores com classes contêineres. O iterador é um objeto que 'percorre' uma coleção, retornando o próximo item (ou liberando alguma ação nele).
- Fornecer aos clientes de sua classe uma classe proxy que ~~que implementa a interface~~ permite que eles utilizem os serviços da sua classe sem ter acesso aos detalhes de implementação, como seus dados.
- Quando uma definição de classe utiliza apenas um ponteiro ou referência para um objeto de outra classe, o arquivo de cabeçalho de classe dessa outra classe (que comumente ~~revelaria detalhes~~) não precisa ser incluído. Ele pode simplesmente declarar essa outra classe como um tipo de dados com uma declaração de classe antecipada antes de o tipo ser utilizado no arquivo.
- O arquivo de implementação que contém funções-membro de uma classe proxy é o único arquivo que inclui o arquivo de cabeçalho para a classe cujos dados gostaríamos de ocultar.
- O arquivo de implementação contendo as funções-membro para a classe proxy é fornecido ao cliente como um arquivo de código-objeto compilado junto com o arquivo de cabeçalho que inclui os protótipos de função dos serviços fornecidos pela classe proxy.

Terminologia

| | | |
|--------------------------------------|--------------------------------------|---|
| abstração de dados | desalocar memória | objeto host |
| alocar memória | desenfileiramento (operação de fila) | objetos dinâmicos |
| armazenamento livre | enfileiramento (operação de fila) | ocultamento de informações |
| chamadas de função-membro em classes | memoramento de tipo de dados | operações em um ADT |
| classe contêiner | estouro aritmético | primeiro a entrar, primeiro a sair (first-out) |
| classe de coleção | friend , classe | – FIFO) |
| classe proxy | friend , função | representação de dados |
| composição | gerenciamento de memória dinâmica | static , função-membro |
| const, função-membro | heap | static , membro de dados |
| const, objeto | inicializador de membro | tem um, relacionamento |
| construtor de objeto-membro | iterador | this , ponteiro |
| controle de acesso de membro | lista de inicializadores de membro | tipo de dados abstrato (ADT) |
| declaração de classe antecipada | mémoria vazamento | último a entrar, primeiro a sair (last-in, first-out) |
| delete, operador new | new | LIFO) |
| delete[], operador new[], Operador | new[] | – LIFO) |

Exercícios de revisão

- 10.1 Preencha as lacunas em cada uma das seguintes sentenças:
- Os (As) _____ devem ser utilizados(as) para inicializar membros constantes de uma classe.
 - Uma função não-membro deve ser declarada como _____ de uma classe para ter acesso a membros de dados da classe.
 - O operador _____ aloca memória dinamicamente para um objeto de um tipo _____ especificado e retorna um _____ para esse tipo.
 - Um objeto constante _____ deve ser _____; depois de criado, ele não pode ser modificado.
 - Um membro de dados _____ representa informações no nível da classe.
 - As funções-membro _____ de um objeto têm acesso a um 'autoponteiro' para o objeto-chamado ponteiro _____.
 - A palavra-chave _____ especifica que um objeto ou variável não é modificável depois de inicializado.
 - Se um inicializador de membro não receber um objeto-membro de uma classe, o objeto _____ é chamado.
 - Uma função-membro deve ser _____ para acessar os membros de classe _____.
 - Os objetos de membro _____ são os membros de memória anteriormente declarados por contêiner.
- 10.2 Localize os erros na seguinte classe e explique como corrigi-los:

```
class Example
{
public :
    Example( int y = 10 )
        : data( y )
    { }
```

```

        // corpo vazio
    } // fim do construtor Example

    int getIncrementedData() const
    {
        return data++;
    } // fim da função getIncrementedData
    static int getCount()
    {
        cout << "Data is " << data << endl;
        return count;
    } // fim da função getCount
private :
    int data;
    static int count;
}; // fim da classe Example

```

Respostas dos exercícios de revisão

- 10.1 a) inicializadores de membro. b) new ponteiro. d) inicializadora. e) this. f) const. h) construtor-padrão. i) são. j) antes. do.

- 10.2 Erro: A definição de classe ~~Example~~ tem dois erros. O primeiro ocorre na função ~~getIncrementedData~~. A função é declarada const, mas modifica o objeto.

Correção: Para corrigir o primeiro erro, remova a palavra-chave const.

Erro: O segundo erro ocorre na função ~~getCount~~. Essa função é declarada const, portanto não tem permissão de acessar nenhum membro da classe ~~Example~~.

Correção: Para corrigir o segundo erro, remova a linha ~~const~~ da definição.

Exercícios

- 10.3 Compare e contraste os operadores de alocação e de desalocação com o uso de memória.

- 10.4 Explique a noção de amizade em C++. Explique os aspectos negativos de amizade como descritos no texto. Uma definição da classe ~~ref~~ pode incluir os dois construtores a seguir? Se não, explique por que.

```

Time( int h = 0, int m = 0, int s = 0 );
Time();

```

- 10.6 O que acontece quando um tipo de retorno é especificado para um construtor ou destrutor?

- 10.7 Modifique a classe ~~Time~~ na Figura 10.10 para obter as seguintes capacidades:

- a) Dar saída para a data em múltiplos formatos como

```

DDD YYYY
MM/DD/YY
June 14, 1992

```

- b) Utilizar construtores sobrecarregados para manipular com datas dos formatos na parte (a).

- c) Criar um construtor que lê a data de sistema utilizando as funções-padrão de ~~library e configuração~~ membro ~~ctime~~ (Consulte a documentação de referência do [seu compilador](#) ou [ref/ctime/index.html](#) para obter informações sobre as funções ~~ctime~~ e ~~localtime~~).

No Capítulo 11, seremos capazes de criar operadores para testar a igualdade de duas datas e compará-las para determinar se uma veio antes ou depois da outra.

- 10.8 Crie uma classe ~~SavingsAccount~~. Utilize um membro de classe ~~annualInterestRate~~ para armazenar a taxa de juros anual para cada um dos correntistas. Cada membro da classe ~~correntista~~ deve ter um membro para indicar a quantia que os correntistas têm atualmente em depósito. Forneça a função ~~monthlyInterest~~ que calcula os juros mensais multiplicando ~~balance[saldo]~~ pelo ~~annualInterestRate~~ dividido por 12; esses juros devem ser adicionados ao ~~balance~~. Forneça uma função ~~setInterestRate~~ que configura ~~annualInterestRate~~ com um novo valor. Escreva um programa de driver para a classe ~~SavingsAccount~~. Instancie dois objetos diferentes da classe ~~saver1~~ e ~~saver2~~ com saldos de \$ 2.000,00 e \$ 3.000,00, respectivamente. Configure ~~saver1~~ para 3%. Em seguida, calcule os juros mensais e imprima os novos saldos de cada um dos correntistas. Em seguida, configure o ~~saver2~~ para 4%, calcule os juros do próximo mês e imprima os novos saldos para cada um dos poupanças.

- 10.9 Crie a classe `IntegerSet` pela qual cada objeto pode armazenar inteiros no intervalo 0 a 100. Um conjunto é representado internamente como um array de uns e zeros. O elemento `at(i)` é `true` se o inteiro `i` estiver no conjunto. O elemento `at(i)` é `false` se o inteiro `i` não estiver no conjunto. O construtor-padrão inicializa um conjunto para o chamado 'conjunto vazio', isto é, um conjunto cuja representação de array só contém zeros.

Forneça funções-membro para as operações comuns de conjuntos. Por exemplo, forneça uma função-membro `intersectionOfSets` que cria um terceiro conjunto que seja a união teórica de dois conjuntos existentes (isto é, um elemento do array do terceiro conjunto é configurado como 1 se esse elemento for 1 em qualquer um dos conjuntos existentes, ou em ambos, e um elemento do array do terceiro conjunto configurado como 0 se esse elemento for 0 em cada um dos conjuntos existentes).

Forneça uma função-membro `unionOfSets` que cria um terceiro conjunto que seja a intersecção teórica de dois conjuntos existentes (isto é, um elemento do array do terceiro conjunto é configurado como 0 se esse elemento for 0 em qualquer um ou ambos os conjuntos existentes, e um elemento do array do terceiro conjunto é configurado como 1 se esse elemento for 1 em cada um dos conjuntos existentes).

Forneça uma função-membro `insertElement` que insere um novo elemento no conjunto (configurando-o como 1). Forneça uma função-membro `removeElement` que exclui o elemento (configurando-o como 0).

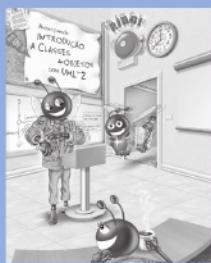
Forneça uma função-membro `printSet` que imprime um conjunto como uma lista de números separados por espaços. Imprima somente aqueles elementos que estiverem presentes no conjunto (isto é, sua posição no array `at(i)` é `true`). Imprima vazio.

Forneça uma função-membro `isEqual` que determina se dois conjuntos são iguais.

Forneça um construtor adicional que recebe um array de inteiros e o tamanho desse array e utiliza o array para inicializar um objeto configurado.

Agora escreva um programa de driver para testar suas classes e diversos objetos `Set`. Teste se todas as suas funções-membro funcionam adequadamente.

- 10.10 Seria perfeitamente razoável para as figuras 10.18–10.19 representar a hora internamente como o número de segundos desde a meia-noite em vez de representá-la com os três valores `second`, `minute` e `hour`. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados. Modifique a Figura 10.18 para implementar a hora como o número de segundos desde a meia-noite e mostrar que não há alteração visível na funcionalidade esperada pelos clientes. [Este exercício demonstra com precisão as virtudes do ocultamento da implementação.]



Toda a diferença entre construção e criação é exatamente esta: que uma coisa construída só pode ser amada

depois de ser feita, não antes de existir.

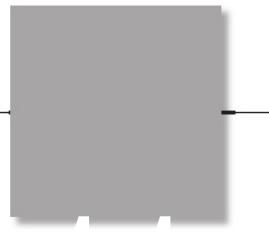
Gilbert Keith Chesterton

Os dados foram lançados.

Júlio César

Nosso médico realmente nunca iria nos operar a menos que necessário. Ele era assim. Se não precisasse do dinheiro, nem tocaria em você.

Herb Shriner



Sobrecarga de operadores; objetos string e array

OBJETIVOS

Neste capítulo, você aprenderá:

O que é sobre carga de operadores e como ela torna os programas mais legíveis e a programação mais conveniente.

Como redefinir (sobre carregar) operadores para trabalhar com objetos de classes definidas pelo usuário.

As diferenças entre sobre carregar operadores unários e binários.

Como converter objetos de uma classe em outra classe.

Quando sobre carregar e quando não sobre carregar operadores.

Como criar classes `PhoneNumberArray`, `String` e `Date` que

demonstram a sobre carga de operadores.

Como utilizar operadores sobre carregados e outras funções-membro da biblioteca de classe `padding`.

Como utilizar a palavra-chave `explicit` para impedir o compilador de utilizar construtores de um único argumento para realizar conversões implícitas.

- 11.1 Introdução
- 11.2 Fundamentos de sobrecarga de operadores
- 11.3 Restrições à sobrecarga de operadores
- 11.4 Funções operadoras como membros de classe versus funções globais
- 11.5 Sobrecarregando operadores de inserção e extração de fluxo
- 11.6 Sobrecarregando operadores unários
- 11.7 Sobrecarregando operadores binários
- 11.8 Estudo de caso: classe `ArrayList`
- 11.9 Convertendo entre tipos
- 11.10 Estudo de caso: classe `String`
- 11.11 Sobrecarregando `++` e `--`
- 11.12 Estudo de caso: uma classe `Date`
- 11.13 Classe `String` da biblioteca-padrão
- 11.14 Construtores explicit
- 11.15 Síntese

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

11.1 Introdução

Os capítulos 9–10 introduziram os princípios básicos de classes em C++. Os serviços eram obtidos de objetos enviando mens forma de chamadas de função-membro) para os objetos. Essa notação de chamada de função é incômoda para certos tipos (como classes matemáticas). Além disso, muitas manipulações comuns são realizadas com operadores (por exemplo, entrad: Podemos utilizar o rico conjunto de operadores predefinidos do C++ para especificar manipulações comuns de objeto. Este c mostra como permitir que os operadores C++ funcionem com objetos *stream processor*. Esse é um exemplo natural estender o C++ com essas novas capacidades, mas isso deve ser feito cautelosamente.

Um exemplo de um operador sobrecarregado comumente usado é como operador de inserção de fluxo e como operador de bits de deslocamento para a esquerda (que é discutido no Capítulo 25). Bits, manipuladores, strings e semelhantes também é sobre carregado; é utilizado como operador de extração de fluxo e como operador de bits de deslocamento para a direita. Os operadores de bits de deslocamento para a esquerda e para a direita são discutidos em detalhes no Capítulo 25. Esses dois operadores são sobre carregados na C++ Standard Library.

Embora a sobrecarga de operadores pareça uma capacidade exótica, a maioria dos programadores utiliza operadores sobreescritos regularmente. Por exemplo, a própria linguagem C++ sobreescrita o operador de adição (`+operator`) para executar a adição de inteiros, na aritmética de ponto flutuante e de ponteiros.

O C++ permite ao programador sobrecarregar a maioria dos operadores para que eles se tornem sensíveis ao contexto em utilizados — o compilador gera o código apropriado com base no contexto (em particular, os tipos dos operandos). Alguns operadores são sobrecarregados freqüentemente, em especial o operador de atribuição e vários operadores aritméticos como realizados por operadores sobrecarregados também podem ser realizados por chamadas de função explícita, mas a notação é muitas vezes, mais clara e mais familiar aos programadores.

Discussimos quando utilizar e quando não utilizar sobrecarga de operadores. Implementamos as classes definidas pelo `PhoneNumber`, `String` e `Date` para demonstrar como sobrepor operadores, incluindo os operadores de inserção de fluxo, de extração de fluxo, de atribuição, de igualdade, de subscrito, relacionais, de negação lógica, parênteses e os operadores de inserção. O capítulo termina com um exemplo da biblioteca-padrão do C++, que fornece muitos operadores sobrepor.

que são comumente usadas para implementar operações matemáticas. Os operadores de atribuição (`=`, `+=`, `-=`, etc.) e os operadores de comparação (`<`, `>`, `==`, etc.) já foram mencionados anteriormente. No entanto, é possível implementar classes que possam ser utilizadas para implementar várias operações matemáticas adicionais.

11.2 Fundamentos de sobrecarga de operadores

A programação em C++ é um processo sensível ao tipo e focalizado no tipo. Os programadores podem utilizar tipos fundamentais para definir novos tipos. Os tipos fundamentais podem ser utilizados com a rica coleção de operadores do C++. Os operadores fornecem aos programadores uma notação concisa para expressar manipulações de objetos de tipos fundamentais.

Os programadores também podem utilizar operadores com tipos definidos pelo usuário. Embora não permita que novos operadores sejam criados, o C++ permite que a maioria dos operadores existentes seja sobre carregada de modo que, quando forem utilizados objetos, eles tenham um significado apropriado para esses objetos. Essa é uma capacidade poderosa.



Observação de engenharia de software 11.1

A sobre carga de operadores contribui para a extensibilidade do C++ — um dos atributos mais atraentes da linguagem.



Boa prática de programação 11.1

Utilize a sobre carga de operadores quando ela torna um programa mais claro do que realizar as mesmas operações com chamadas de função.



Boa prática de programação 11.2

Os operadores sobre carregados devem simular a funcionalidade de suas contrapartes predefinidas — por exemplo, o operador deve ser sobre carregado para realizar adição, não subtração. Evite o uso excessivo ou inconsistente de sobre carga de operadores, já que isso pode tornar um programa obscuro e difícil de ler.

Um operador é sobre carregado escrevendo uma definição de função sobrecarregada — por exemplo, o operador de atribuição (`=`) deve ser sobre carregado para realizar adição, não subtração. Evite o uso excessivo ou inconsistente de sobre carga de operadores, já que isso pode tornar um programa obscuro e difícil de ler.

Para utilizar um operador em objetos de classe, esse operador deve ser sobre carregado — com três exceções. O operador de atribuição (`=`) pode ser utilizado com toda a classe para realizar atribuição de membro a membro dos membros de dados da classe — membro de dados é atribuído a partir do objeto ‘srcem’ para o objeto ‘alvo’ da atribuição. Logo veremos que essa atribuição de membro padrão é perigosa para classes com membros ponteiro; sobre carregaremos explicitamente o operador de atribuição para classes. Os operadores de endereço (`&` e `*`) também podem ser utilizados com objetos de qualquer classe sem a sobre carga. O operador de endereço retorna o endereço do objeto na memória. O operador vírgula avalia a expressão à sua esquerda e, depois, à direita. Esses dois operadores também podem ser sobre carregados.

Sobre carregar é especialmente apropriado para classes matemáticas. Essas classes muitas vezes requerem que um conjunto de operadores seja sobre carregado para consistir com a maneira como estas classes implementam operações aritméticas também são comumente utilizados com números complexos.

A sobre carga de operadores fornece as mesmas expressões concisas e familiares para tipos definidos pelo usuário que o C++ — com sua rica coleção de operadores para tipos fundamentais. A sobre carga de operadores não é automática — você deve escrever de sobre carga de operadores para realizar as operações desejadas. As vezes, essas funções são melhores quando tornadas funções globais. Discutimos essas questões por todo o capítulo.

11.3 Restrições à sobre carga de operadores

A maioria dos operadores do C++ pode ser sobre carregada. Esses operadores são mostrados na Figura 11.1. A Figura 11.2 mostra os operadores que não podem ser sobre carregados.



Erro comum de programação 11.1

Tentar sobre carregar um operador não sobre carregável é um erro de sintaxe.

Precedência, associatividade e número de operandos

A precedência de um operador não pode ser alterada pela sobre carga. Isso pode levar a situações incômodas em que um operador sobre carregado de uma maneira pela qual sua precedência fixa é inadequada. Entretanto, os parênteses podem ser utilizados para alterar a ordem de avaliação de operadores sobre carregados em uma expressão.

A associatividade de um operador (isto é, se o operador é aplicado da direita para a esquerda ou da esquerda para a direita) pode ser alterada pela sobre carga.

Não é possível alterar a ‘aridade’ de um operador (isto é, o número de operandos que um operador aceita): operadores unários sobre carregados permanecem operadores unários; operadores binários sobre carregados permanecem operadores binários. O único operador ternário (`? :`) do C++ não pode ser sobre carregado. Os operadores `operator <<` e `operator >>` a versão unária como a binária; essas versões unária e binária podem ser sobre carregadas.

| Operadores que podem ser sobre carregados | | | | | | | | |
|---|-----------|----|----|----|----|-----|--------|--|
| +*/%^& | | | | | | | | |
| ~ | ! | = | < | > | += | -= | *= | |
| /= | %= | ^= | &= | = | << | >> | >>= | |
| <<= | == | != | <= | >= | && | | ++ | |
| -- | ->* | , | -> | [] | () | new | delete | |
| new[] | delete [] | | | | | | | |

Figura 11.1 Operadores que podem ser sobre carregados.

| Operadores que não podem ser sobre carregados | | | |
|---|---|----|---|
| . | * | :: | : |

Figura 11.2 Operadores que não podem ser sobre carregados.

Erro comum de programação 11.2

Tentar alterar a ‘aridade’ de um operador via sobre carga de operadores é um erro de compilação.

Criando novos operadores

Não é possível criar novos operadores; apenas os operadores existentes podem ser sobre carregados. Infelizmente, isso impede o programador de utilizar notações populares ~~como as operadoras~~ em algumas outras linguagens de programação para exponenciação. [Nota: Você poderia sobre carregar ~~para~~ operador exponenciação — como ele faz em algumas outras linguagens.]

Erro comum de programação 11.3

Tentar criar novos operadores via sobre carga de operadores é um erro de sintaxe.

Operadores para tipos fundamentais

O significado de como um operador funciona em objetos de tipos fundamentais não pode ser alterado pela sobre carga de operadores. O programador não pode, por exemplo, alterar o significado de + entre tipos fundamentais. A sobre carga de operadores funciona somente com objetos de tipos definidos pelo usuário ou com uma mistura de um objeto de um tipo definido pelo usuário e um objeto de tipo fundamental.

Observação de engenharia de software 11.2

Pelo menos um argumento de uma função operadora deve ser um objeto ou referência de um tipo definido pelo usuário. Isso impõe que os programadores alterem a maneira como os operadores funcionam sobre tipos fundamentais.

Erro comum de programação 11.4

Tentar modificar a maneira como um operador funciona com objetos de tipos fundamentais é um erro de compilação.

Operadores relacionados

Sobre carregando um operador de atribuição e um operador de adição para permitir instruções como

```
object2 = object2 + object1;
```

não implica que o operador também seja sobre carregado para permitir instruções como

```
object2 += object1;
```

Esse comportamento só pode ser alcançado sobre carregando ~~desse~~ o operador



Erro comum de programação 11.5

Supor que sobre carregar um operador ~~compara~~ ~~me ga~~ operadores relacionados ~~sobre carregar~~ ~~comparar~~ é errado. Operadores só podem ser sobre carregados explicitamente; não há sobre carga implícita.

11.4 Funções operadoras como membros de classe *versus* funções globais

As funções operadoras podem ser funções-membros ou funções globais; as funções globais são freqüentemente feitas razões de desempenho. As funções-membros ~~utilizam implicitamente~~ para obter um de seus argumentos de objeto de classe (o operando esquerdo para operadores binários). Os argumentos para ambos os operandos de um operador binário devem ser explicitamente listados em uma chamada de função global.

Operadores que devem ser sobre carregados como funções-membro

Ao sobre carregar `<=` ou qualquer um dos operadores de atribuição, a função de sobre carga de operadores devem ser declaradas como um membro de classe. Para os outros operadores, as funções de sobre carga de operadores podem ser membros de classes globais.

Operadores como funções-membro e funções globais

Se uma função operadora é implementada como uma função-membro ou como uma função global, o operador ainda é utilizada mesma maneira em expressões. Então qual implementação é melhor?

Quando uma função operadora é implementada como uma função-membro, o operando na extrema esquerda (ou único) deve ser um objeto (ou uma referência a um objeto) da classe do operador. Se o operando esquerdo precisar ser um objeto de uma classe ou um tipo fundamental, essa função operadora deve ser implementada como uma função global (como faremos na Seção 11.5 sobre carregamento). Como os operadores de inserção de fluxo e extração de fluxo, respectivamente). Uma função operadora global pode se tornar parte de uma classe se essa função precisar acessar membros dessa classe diretamente.

As funções-membro do operador de uma classe específica são chamadas (implicitamente pelo compilador) somente quando o operando esquerdo de um operador binário for especificamente um objeto dessa classe ou quando o único operando de um operador for um objeto dessa classe.

Por que os operadores sobre carregados de inserção e de extração de fluxo são sobre carregados como funções globais?

O operador de inserção `<operator>` é sobre carregado é utilizado em uma expressão em que o operando esquerdo tem o tipo

`ostream &`. Para utilizar o operador dessa maneira, o tipo do operando direito é um objeto de uma classe definida da classe. Isso não é possível para classes definidas pelo usuário, visto que não temos permissão de modificar as classes C++ Standard Library. De modo semelhante, o operador `>>operator>` é sobre carregado é utilizado em uma expressão em que o operando esquerdo tem o tipo `const &`, como `em >> classObject`, e o operando direito é um objeto de uma classe definida pelo usuário, portanto ele também deve ser uma função global. Além disso, cada uma dessas funções operadoras sobre carregadas requerer acesso aos membros de um objeto de classe cuja saída ou entrada está sendo feita, portanto podemos transformar essas funções operadoras sobre carregadas ~~sem razões de desempenho~~ por razões de desempenho.



Dica de desempenho 11.1

É possível sobre carregar um operador como uma função global mas essa função que requer acesso aos dados ~~protected~~ de uma classe precisaria utilizar funções decididas na interface dessa classe. O overhead de chamar essas funções poderia causar um desempenho ruim, desse modo essas funções podem ser colocadas inline para melhora desempenho.

Operadores comutativos

Outra razão pela qual poderíamos preferir uma função global a sobre carregar um operador é permitir que o operador seja comutativo.

~~Por exemplo, se o operador de adição é sobre carregado para um tipo de classe, é necessário que o resultado seja de máquina do hardware suportado por esse tipo de classe. Isso significa que o resultado deve ser de tipo long int (que é o tipo de máquina do hardware suportado por esse tipo de classe). O operador de adição é temporário, como a soma de um long int e um HugelInteger (como na expressão `number + BigInteger1`). Portanto, queremos que o operador de adição seja comutativo (exatamente como ele é com dois operandos do tipo fundamental). O problema é que o objeto de classe deve aparecer à esquerda do operador de adição se este precisar ser sobre carregado como uma função-membro. Então, sobre carregamos o operador como uma função para permitir que o operador de adição apareça à direita da adição. A função lida como long int à esquerda, ainda pode ser uma função-membro.~~

11.5 Sobrecrevendo operadores de inserção e extração de fluxo

O C++ é capaz de realizar a entrada e a saída dos tipos fundamentais utilizando o operador de extração de fluxo inserção de fluxo das bibliotecas de classes fornecidas com os compiladores C++ sobrecrevem esses operadores para processar todos os tipos fundamentais, inclusive os ~~ponteiros~~ estáticos. Os operadores de inserção de fluxo e de extração de fluxo também podem ser sobrecregados para realizar entrada e saída de tipos definidos pelo usuário. O programa das figuras demonstra como sobrecrever esses operadores para tratar dados de uma classe de número de telefone definida pelo usuário.

`PhoneNumbe` Esse programa supõe que os números de telefone são inseridos corretamente.

A função do operador de extração de fluxo (Figura 11.4, linhas 22–31) aceita a referência de um argumento `PhoneNumber` como argumentos e retorna uma referência. A função operadora >> realiza a entrada de números de telefone na forma

(800) 555-1212

em objetos da classe `PhoneNumber`. Quando o compilador vê a expressão

`cin >> phone`

na linha 19 da Figura 11.5, ele gera a chamada de função global

`operator >>(cin, phone);`

Quando essa chamada executa, o parâmetro (Figura 11.4, linha 22) se torna um aliado de referência, que torna um aliado de referência para a função operadora. As três partes do número de telefone nos membros `areaCode` (linha 25), `change` (linha 27) e `line` (linha 29) do objeto referenciado pelo parâmetro. O manipulador de fluxo `>>` limita o número de caracteres lidos em cada array de caracteres. Quando utilizados com o número de caracteres lidos ao número de caracteres especificados por `n`, os caracteres restantes não são lidos. Os caracteres de parênteses, espaço e traço são ignorados. Chama-se `chamada de membro` (linhas 24, 26 e 28), que descarta o número especificado de caracteres no fluxo de entrada (um caractere por padrão). A função retorna um stream à referência `cin` (isto é, `cin`). Isso permite que as operações de entrada em objetos sejam colocadas em cascata com operações de entrada sobre outros objetos de outros tipos de dados. Por exemplo, um programa pode inserir dois objetos em uma instrução como esta:

`cin >> phone1 >> phone2;`

Primeiro, a expressão `phone1` executa fazendo a chamada de função global

`operator >>(cin, phone1);`

```

1 // Figura 11.3: PhoneNumber.h
2 // definição da classe PhoneNumber
3 #ifndef PHONENUMBER_H
4 #define PHONENUMBER_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15     friend ostream &operator<<( ostream &, const PhoneNumber & );
16
17     friend istream &operator>>( istream &, PhoneNumber & );
18     string areaCode; // código de área (de cidade) de 3 algarismos
19     string exchange; // prefixo (de bairro/região) de 3 algarismos
20     string line; // linha de 4 algarismos
21 }; // fim da classe PhoneNumber
22
23 #endif

```

Figura 11.3 Classe `PhoneNumber` com operadores de inserção de fluxo e de extração de fluxo sobrecregados definidos

```

1 // Figura 11.4: PhoneNumber.cpp
2 // Operadores de inserção de fluxo e de extração de fluxo sobre carregados
3 // para a classe PhoneNumber.
4 #include <iomanip>
5 using std::setw;
6
7 #include "PhoneNumber.h"
8
9 // operador de inserção de fluxo sobre carregado; não pode ser
10 // uma função-membro se quiséssemos invocá-lo com
11 // cout << somePhoneNumber;
12 ostream &operator <<( ostream &output, const PhoneNumber &number )
13 {
14     output << "(" << number.areaCode << "
15         << number.exchange << " " << number.line;
16     return output; // permite cout << a << b << c;
17 } // fim da função operator<<
18
19 // operador de extração de fluxo sobre carregado; não pode ser
20 // uma função-membro se quiséssemos invocá-lo com
21 // cin >> somePhoneNumber;
22 istream &operator >>( istream &input, PhoneNumber &number )
23 {
24     input.ignore(); // pula (
25     input >> setw( 3 ) >> number.areaCode; // entrada do código de área
26     input.ignore( 2 ); // pula ) e espaço
27     input >> setw( 3 ) >> number.exchange; // entrada do prefixo (exchange)
28     input.ignore(); // pula traço (-)
29     input >> setw( 4 ) >> number.line; // entrada de linha
30     return input; // permite cin >> a >> b >> c;
31 } // fim da função operator>>

```

Figura 11.4 Os operadores de inserção e de extração de fluxo sobre carregados para a classe PhoneNumber.

```

1 // Figura 11.5: fig11_05.cpp
2 // Demonstrando os operadores de inserção
3 // e extração de fluxo sobre carregados da classe PhoneNumber.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "PhoneNumber.h"
10
11 int main()
12 {
13|4     PhoneNumber phone; // cria objeto phone
14     cout << "Enter phone number in the form (123) 456-7890." << endl;
15
16
17 // cin >> phone invoca operator>> emitindo implicitamente
18 // a chamada da função global operator>>( cin, phone )
19     cin >> phone;
20

```

Figura 11.5 Operadores de inserção e extração de fluxo sobre carregados.

(continua)

```

21 cout << "The phone number entered was:";
22 // cout << phone invoca operator<< emitindo implicitamente
23 // chamada da função global operator<<( cout, telefone )
24 cout << phone << endl;
25 return 0;
26 } // fim de main

```

Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212

Figura 11 Operadores de inserção e extração de fluxo sobrecarregados.

(continuação)

Essa chamada então retorna uma referência para o operador `<<` do tipo `operator <<(cout, phone)`, de modo que a parte restante da expressão é interpretada simplesmente como `phone`. Isso executa fazendo a chamada de função global

`operator >>(cin, phone2);`

A função operadora de inserção de fluxo (Figura 11.4, linhas 12–17) ~~recebe~~ uma referência para `PhoneNum& number` como argumentos e retorna uma referência para `operator <<` que exibe objetos do tipo `PhoneNumber`. Quando o compilador vê a expressão

`cout << phone`

na linha 25 da Figura 11.5, ele gera a chamada de função global

`operator <<(cout, phone);`

A função `operator <<` exibe as partes do número de telefone porque elas são armazenadas como objetos



Dica de prevenção de erro 11.1

Retornar uma referência a partir de uma função operadora sobrecarregada é normalmente bem-sucedido, porque

os membros dos objetos de tipo global ou de membro de longa duração. Retornar uma referência a uma variável automática:

Observe que as funções `<<` e `>>` são declaradas como `operator <<(ostream& os, const PhoneNumber& number)` e `operator >>(istream& is, PhoneNumber& number)` (Figura 11.3, linhas 15–16). Elas são funções globais porque os objetos parecem serem cada caso como o operando direito do operador. Lembre-se de que as funções operadoras sobrecarregadas para operadores binários só podem ser funções-membro se o operando esquerdo for um objeto da classe em que a função é membro. Os operadores de entrada e saída sobrecarregados só podem ser declarados como membros da classe diretamente por razões de desempenho ou porque a classe não pode oferecer as funcionalidades necessárias para implementar os operadores. Observe também que a lista de parâmetros da função `operator <<` (Figura 11.4, linha 17), porque `PhoneNumber` é simplesmente enviado para a saída, não inclui os parâmetros da função `<<` (linha 22) é ótima, porque o objeto `PhoneNumber` deve ser modificado para armazenar a entrada do número de telefone no objeto.



Observação de engenharia de software 11.3

Novas capacidades de entrada/saída para tipos definidos pelo usuário são adicionadas ao C++ sem modificar as classes de entrada/saída da biblioteca-padrão do C++. Esse é outro exemplo da extensibilidade da linguagem de programação C++.

Um operador unário de uma classe pode ser sobrecarregado como uma função global com um argumento; esse argumento deve ser um objeto da classe ou uma referência a um objeto da classe. As funções que implementam os operadores sobrecarregados devem ter permissão de acessar os dados da classe da classe. Lembre-se de que as funções podem acessar apenas os membros da classe.

Mais adiante neste capítulo, sobrearemos para este tipo de operador que veríamos (Seção 11.10) está vazio e retornar um resultado para a expressão que é um objeto da classe. Quando um operador unário é sobrecarregado como uma função-membro sem argumentos e é o compilador que expressão chama de `operator()`. O operador de objeto de classe para o qual a função-membro está sendo invocada. A função é declarada na definição de classe como mostrado a seguir:

```

class String
{
public :
    bool operator !() const;
    ...
}; // fim da classe String

```

Um operador unário `pode` ser sobre carregado como uma função global com um argumento de duas maneiras diferentes — com um argumento que seja um objeto (isso requer uma cópia do objeto, para que os efeitos colaterais da função não sejam ao objeto `srcinal`), ou com um argumento que seja uma referência a um objeto (nenhuma cópia do objeto `srcinal` é feita, desse modo todos os efeitos colaterais dessa função são aplicados ao `objeto original` (ou uma referência a um objeto da classe), então é tratado como se a `chamada` tivesse sido escrita, invocando a função global que é declarada como segue:

```
bool operator !( const String & );
```

11.7 Sobre carregando operadores binários

Um operador binário é sobre carregado como uma função-membro ou como uma função global com dois argumentos (um desses argumentos deve ser um objeto de classe ou uma referência a um objeto de classe).

Mais adiante neste capítulo, sobre carregando dois objetos. A sobre carregar o operador binário uma função-membro de uma classe com um argumento, se os objetos das passagens z será tratado como `operator<(z)` tivesse sido escrita, invocando a função-membro

```

class String

public :
    bool operator<( const String & ) const;
    ...
}; // fim da classe String

```

Se o operador binário `pode` ser sobre carregado como uma função global, ele deve aceitar dois argumentos — um dos quais deve ser um objeto de classe ou uma referência a um objeto de classe. Se as referências a objetos da classe `String`, então z é tratado como se a `chamada` `(y, z)` tivesse sido escrita no programa, invocando a função global `operator<` declarada da seguinte maneira:

```
bool operator <( const String &, const String & );
```

11.8 Estudo de caso: classe Array

Arrays baseados em ponteiro apresentam diversos problemas. Por exemplo, um programa pode facilmente ‘ultrapassar’ qualquer das extremidades de um array, porque o C++ não verifica se os subscritos saem fora do intervalo de um array (embora o programador possa fazer isso explicitamente). Os arrays podem ter seus elementos inteiros alternativos de subscrito não são permitidos. A entrada ou saída de um array pode ser realizada de uma vez; cada elemento do array deve ser lido ou gravado individualmente. Dois arrays não podem ser comparados significativamente com operadores de igualdade ou operadores relacionais (porque os nomes de array são simplesmente ponteiros para o local em que os arrays iniciam na memória, naturalmente, dois arrays sempre estarão em posições da memória diferentes). Quando um array é passado para uma função geral projetada para lidar com arrays de qualquer tamanho, o tamanho do array deve ser passado como um argumento adicional. Um array não pode ser atribuído a outro com o(s) operador(es) de atribuição (porque os nomes de array são ponteiros constantes e não podem ser utilizados à esquerda do operador de atribuição). Essas e outras capacidades certamente parecem ‘naturais’ lidar com arrays, mas os arrays baseados em ponteiro não fornecem essas capacidades. Entretanto, o C++ realmente fornece de implementar essas capacidades de array pelo uso de classes e de sobre carga de operadores.

Nesse exemplo, criamos uma poderosa classe de array que realiza a verificação de intervalo para assegurar que os subscritos permanecem dentro dos limites. Isso permite que um objeto de array seja atribuído a outro com o operador de atribuição.

Os objetos da classe `Array` conhecem seu tamanho, portanto o tamanho não precisa ser passado separadamente como um argumento durante a passagem para uma função. A entrada e saída pode ser feita com os operadores de extração e inserção de fluxo, respectivamente. As comparações feitas com os operadores de igualdade

Esse exemplo afiará seu conhecimento sobre abstração de dados. Provavelmente você vai querer sugerir outros aprimoramentos para essa classe. O desenvolvimento de classe é uma atividade interessante, criativa e intelectualmente desafiadora — sempre com o objetivo de ‘fabricar classes valiosas’.

O programa das figuras 11.6–11.8 demonstra classes sobre carregados. Primeiramente, consideraremos a definição de classe (Figura 11.6) e cada uma das funções-membro e definições de função classe (Figura 11.7).

```

1 // Figura 11.6: Array.h
2 // Classe Array para armazenar arrays de inteiros.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public :
15     Array( int = 10 ); // construtor-padrão
16     Array( const Array & ); // construtor de cópia
17     ~Array(); // destrutor
18     int getSize() const; // retorna tamanho
19
20     const Array &operator =( const Array & ); // operador de atribuição
21     bool operator ==( const Array & ) const; // operador de igualdade
22
23     // operador de desigualdade; retorna o oposto do operador ==
24     bool operator !=( const Array &right ) const
25     {
26         return !(* this == right ); // invoca Array::operator==
27     } // fim da função operator!=
28
29     // operador subscrito de objetos não-const retorna lvalue modificável
30     int &operator []( int );
31
32     // operador de subscrito de objetos const retorna rvalue
33     int operator [] ( int ) const;
34 private :
35     int size; // tamanho do array baseado em ponteiro
36     int *ptr; // ponteiro para o primeiro elemento do array baseado em ponteiro
37 }; // fim da classe Array
38
39 #endif

```

Figura 11.6 Definição da class~~Array~~ com operadores sobrecarregados.

```

1 // Fig 11.7: Array.cpp
2 // Definições de função-membro para a classe Array
3 #include <iostream>
4 using std::cerr;
5
6 using std::cout;
7 using std::endl;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // sai do protótipo de função

```

Figura 11.7 Definições de função-membro e função ~~main~~ da class~~Array~~.

(continua)

```

13  using std::exit;
14
15 #include "Array.h" // definição da classe Array
16
17 // construtor-padrão para a classe Array (tamanho padrão 10)
18 Array::Array( int arraySize )
19 {
20     size = (arraySize > 0 ? arraySize : 10); // valida arraySize
21     ptr = new int [size]; // cria espaço para array baseado em ponteiro
22
23     for ( int i = 0; i < size; i++ )
24         ptr[ i ] = 0; // configura elemento do array baseado em ponteiro
25 } // fim do construtor-padrão de Array
26
27 // copia o construtor da classe Array;
28 // deve receber uma referência para impedir a recursão infinita
29 Array::Array( const Array &arrayToCopy )
30     : size( arrayToCopy.size )
31 {
32     ptr = new int [size]; // cria espaço para array baseado em ponteiro
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = arrayToCopy.ptr[ i ]; // copia para o objeto
36 } // fim do construtor de cópia do Array
37
38 // destrutor para a classe Array
39 Array::~Array()
40 {
41     delete [] ptr; // libera espaço do array baseado em ponteiro
42 } // fim do destrutor
43
44 // retorna o número de elementos do Array
45 int Array::getSize() const
46 {
47     return size; // número de elementos em Array
48 } // fim da função getSize
49
50 // operador de atribuição sobrecarregado;
51 // retorno const evita: (a1 = a2) = a3
52 const Array &Array::operator =( const Array &right )
53 {
54     if ( &right != this ) // evita auto-atribuição:
55     {
56         // para Arrays de tamanhos diferentes, desaloca array do lado esquerdo
57         // sinal, então aloca o novo array à esquerda
58         if ( size != right.size )
59         {
60             delete [] ptr; // libera espaço
61             size = right.size; // redimensiona esse objeto
62         }
63     }
64     ptr = new int [size]; // cria espaço para a cópia do array
65     for ( int i = 0; i < size; i++ )
66         ptr[ i ] = right.ptr[ i ]; // copia o array para o objeto
67 } // fim do if externo
68

```

Figura 11.7 Definições de função-membro e função `main` da classe `Array`.

(continua)

```

69     return *this; // permite x = y = z, por exemplo
70 } // fim da função operator=
71
72 // determina se dois Arrays são iguais e
73 // retorna true, caso contrário retorna false
74 bool Array:: operator ==( const Array &right ) const
75 {
76     if ( size != right.size )
77         return false ; // arrays com diferentes números de elementos
78
79     for ( int i = 0; i < size; i++ )
80         if ( ptr[ i ] != right.ptr[ i ] )
81             return false ; // o conteúdo do Array não é igual
82
83     return true ; // Arrays são iguais
84 } // fim da função operator==
85
86 // operador de subscrito sobreescarregado para Arrays não-const;
87 // retorno de referência cria um lvalue modificável
88 int &Array:: operator []( int subscript )
89 {
90     // verifica erro de subscrito fora do intervalo
91     if ( subscript < 0 || subscript >= size )
92     {
93         cerr << "\nError: Subscript " << subscript
94         << " out of range" << endl;
95         exit( 1 ); // termina o programa; subscrito fora do intervalo
96     } // fim do if
97
98     return ptr[ subscript ]; // retorno da referência
99 } // fim da função operator[]
100
101 // operador de subscrito sobreescarregado para Arrays const
102 // retorno de referência const cria um rvalue
103 int Array:: operator [] ( int subscript ) const
104 {
105     // verifica erro de subscrito fora do intervalo
106     if ( subscript < 0 || subscript >= size )
107     {
108         cerr << "\nError: Subscript " << subscript
109         << " out of range" << endl;
110         exit( 1 ); // termina o programa; subscrito fora do intervalo
111     } // fim do if
112
113     return ptr[ subscript ]; // retorna cópia desse elemento
114 } // fim da função operator[]
115
116 // operador de entrada sobreescarregado para a classe Array;
117 // entrada de valores para o Array inteiro
118 istream & operator >>( istream &input, Array &a )
119 {
120     for ( int i = 0; i < a.size; i++ )
121         input >> a.ptr[ i ];
122
123     return input; // permite cin >> x >> y;
124 } // fim da função

```

Figura 11.7 Definições de função-membro e função-função da classArray.

(continua)

```

125 // operador de saída sobre carregado para classe Array
126 ostream &operator <<( ostream &output, const Array & a )
127 {
128     int i;
129
130     // gera saída do array baseado em ptr private
131     for ( i = 0; i < a.size; i++ )
132     {
133         output << setw( 12 ) << a.ptr[ i ];
134
135         if ( ( i + 1 ) % 4 == 0 ) // 4 números por linha de saída
136             output << endl;
137     } // fim do for
138
139     if ( i % 4 != 0 ) // termina a última linha de saída
140         output << endl;
141
142     return output; // permite cout << x << y;
143 } // fim da função operator<<
144 
```

Figura 11.7 Definições de função-membro e função `operator<<` da classe `Array`.

(continuação)

```

1 // Figura 11.8: fig11_08.cpp
2 // Programa de teste da classe Array.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using
7     std::endl;
8 #include "Array.h"
9
10 int main()
11 {
12     Array integers1( 7 ); // Array de sete elementos
13     Array integers2; // Array de 10 elementos por padrão
14
15     // imprime o tamanho e o conteúdo de integers1
16     cout << "Size of Array integers1 is "
17     << integers1.getSize()
18     << "\nArray after initialization:\n" << integers1;
19
20     // imprime o tamanho e o conteúdo de integers2
21     cout << "Size of Array integers2 is "
22     << integers2.getSize()
23     << "\nArray after initialization:\n" << integers2;
24
25     // insere e imprime integers1 e integers2
26     cout << "\nEnter 17 integers:" << endl;
27     cin >> integers1 >> integers2;
28
29     cout << "\nAfter input, the Arrays contain:\n"
30     << "integers1:\n" << integers1
31     << "integers2:\n" << integers2;

```

Figura 11.8 Programa de teste da classe `Array`.

(continua)

```

32
33 // utiliza o operador de desigualdade (!=) sobrecarregado
34 cout << "\nEvaluating: integers1 != integers2" << endl;
35
36 if ( integers1 != integers2 )
37 cout << "integers1 and integers2 are not equal" << endl;
38
39 // cria Array integers3 utilizando integers1 como um
40 // inicializador; imprime tamanho e conteúdo
41 Array integers3(integers1); // invoca o construtor de cópia
42
43 cout << "\nSize of Array integers3 is "
44 << integers3.getSize()
45 << "\nArray after initialization:\n" << integers3;
46
47 // utiliza operador atribuição (=) sobrecarregado
48 cout << "\nAssigning integers2 to integers1:" << endl;
49 integers1 = integers2; // note que o Array alvo é menor
50
51 cout << "integers1:\n" << integers1
52 << "integers2:\n" << integers2;
53
54 // utiliza operador de igualdade (==) sobrecarregado
55 cout << "\nEvaluating: integers1 == integers2" << endl;
56
57 if ( integers1 == integers2 )
58 cout << "integers1 and integers2 are equal" << endl;
59
60 // utiliza operador de subscrito sobrecarregado para criar rvalue
61 cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63 // utiliza operador de subscrito sobrecarregado para criar lvalue
64 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65 integers1[ 5 ] = 1000
66 cout << "integers1:\n" << integers1;
67
68 // tentativa de utilizar subscrito fora do intervalo
69 cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70 integers1[ 15 ] = 1000; // ERRO: fora do intervalo
71 return 0;
72 } // fim de main

```

Size of Array integers1 is 7

Array after initialization:

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | |

Size of Array integers2 is 10

Array after initialization:

| | | | |
|---|---|---|---|
| 8 | 8 | 8 | 8 |
| 0 | 0 | | |

Enter 17 integers:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Figura 11.8 Programa de teste da classe `Array`.

(continua)

```

After input, the Arrays contain:
integers1:
    1      2      3      4
    5      6      7
integers2:
    8      9      10     11      15
    12     13     14
    16     17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
    1      2      3      4
    5      6      7

Assigning integers2 to integers1:
integers1:
    8      9      10     11      15
    12     13     14
    16     17
integers2:
    8      9      10     11      15
    12     13     14
    16     17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
    8      9      10     11      15
    12     1000    14
    16     17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range

```

Figura 11.8 Programa de teste da classe `array`.

(continuação)

Criando `Array*`, gerando saída de seu tamanho e exibindo seu conteúdo

O programa começa instanciando dois objetos da classe `Array` (Figura 11.8, linha 12) com sete elementos, e (Figura 11.8, linha 13) com o **tamapadrão** — 10 elementos (especificado pelo protótipo do construtor-padrão

Figura 11.8 Utilizando o operador de inserção de fluxo para preencher arrays. No exemplo, os primeiros elementos do Array foram configurados corretamente como zeros pelo construtor. Em seguida, as linhas 21–23 geram saída do tamanho do `integers2` e geram saída de `integers2`, utilizando o operador de inserção de fluxo sobrecarregado

Utilizando o operador de inserção de fluxo sobrecarregado para preencher `Array`

A linha 26 pede para o usuário inserir 17 inteiros. A linha 27 utiliza o operador de extracção de fluxo sobrecarregado valores para ambos os arrays. Os sete primeiros valores são armazenados em `integers1`; os restantes são armazenados em `integers2`. As linhas 29–31 geram saída dos dois arrays com o operador de inserção de fluxo sobrecarregado que a entrada foi realizada corretamente.

Utilizando o operador de desigualdade sobrecarregado

A linha 36 testa o operador de desigualdade sobrecarregado avaliando a condição

```
integers1 != integers2
```

A saída do programa mostra que os dois arrays não são realmente iguais.

Inicializando um novo `Array` com uma cópia do conteúdo de `Array` existente

A linha 41 instancia um `Array` chamado `integers3` e o inicializa com uma `cópia` de `integers1`. Isso invoca o construtor

de cópia de `Array` para copiar os elementos de `integers3`. Discutimos os detalhes do construtor de cópia em breve.

Observe que o construtor de cópia também pode ser invocado escrevendo a linha 41 da seguinte maneira:

```
Array integers3 = integers1;
```

O sinal de igual na instrução indica ao compilador que é necessário invocar o operador de atribuição. Quando um sinal de igual aparece na declaração de um objeto, ele invoca um construtor para esse objeto. Essa forma pode ser utilizada para passar somente um único argumento para um construtor.

As linhas 43–45 geram a saída desejada do código `integers3`, utilizando o operador de inserção de fluxo sobre-

carregado para confirmar que os arrays foram configurados corretamente pelo construtor de cópia.

Utilizando o operador de atribuição sobrecarregado

Em seguida, a linha 49 testa o operador de atribuição sobrecarregado atribuindo `integers3` para `integers1`. As linhas 51–52 imprimem ambos os objetos para confirmar que a atribuição foi bem-sucedida. Observe que originalmente 7 inteiros e foi redimensionado para armazenar uma cópia dos 10 elementos. Portanto, o operador de atribuição sobrecarregado realiza essa operação de redimensionamento de uma maneira que é transparente ao código-cliente.

Utilizando o operador de igualdade sobrecarregado

Em seguida, a linha 57 utiliza o operador de igualdade sobrecarregado para confirmar que os objetos `integers1` e `integers2` são de fato idênticos depois da atribuição.

Utilizando o operador de subscrito sobrecarregado

A linha 61 utiliza o operador de subscrito sobrecarregado para referenciar o elemento no intervalo de `integers1`. Esse nome subscrito é utilizado para imprimir o valor armazenado em `integers1[5]`. A linha 65 utiliza `integers1[5]` como um valor modifíável no lado esquerdo de uma instrução de atribuição para atribuir o novo valor, `integers1`. Veremos que `integers1[5]` retorna uma referência para `integers1[5]` modifíável depois que o operador confirma que é um subscrito válido para.

A linha 70 tenta atribuir 000 para `integers1[15]` — um elemento fora do intervalo. Neste ponto, o programa termina, porque o subscrito está fora do intervalo, imprime uma mensagem e termina o programa. Observe que destacamos em itálico a lirada para enfatizar que é um erro acessar um elemento que está fora do intervalo. Esse é um erro de lógica de tempo de não é um erro de compilação.

Curiosamente, o operador de subscrito de `array` é restrito somente ao uso de arrays; ele também pode ser utilizado, por exemplo, para selecionar elementos de outros tipos de classes contêineres, como listas vinculadas, strings e dicionários. Além disso, quando as funções `<<` e `>>` são definidas, os subscritos não mais precisam ser inteiros — caracteres, strings, números de ponto flutuante ou até objetos de classes definidas pelo usuário também poderiam ser utilizados. No Capítulo 23, “Standard Template (STL)”, discutimos a classe `vector` que permite subscritos do tipo não inteiro.

Definição da classArray

Agora que vimos como esse programa opera, vamos percorrer o cabeçalho de classe (Figura 11.6). À medida que nos referirmos à função-membro no cabeçalho, discutimos a implementação dessa função na Figura 11.7. Na Figura 11.6, as linhas 35–36 representam os membros de dados da classe `array`. Cada objeto `array` consiste em um `memória` — que armazena o número de elementos no array e um `ponteiro` — que aponta para o array de inteiros baseado em ponteiro dinamicamente alocado gerenciado pelo objeto `array`.

Sobrecrevendo operadores de inserção e extração de fluxo `friend`

As linhas 120–133 da Figura 11.6 definem os operadores de inserção e extração de fluxo sobrecarregados como

```
operator <<( cout, arrayObject )
```

Quando o compilador vê uma expressão `cout << arrayObject`, ele invoca a função `<<` global com a chamada

```
operator >>( cin, arrayObject )
```

Notamos novamente que essas funções operadoras de inserção e de extração de fluxo `array` não podem ser membros da classe `array`, porque o objeto `array` é sempre mencionado à direita do operador de inserção de fluxo e do operador de extração de fluxo. Se essas funções operadoras precisassem ser membros da classe `array`, as instruções mal construídas teriam de ser utilizadas para gerar a saída e a entrada de `array`.

```
arrayObject << cout;
arrayObject >> cin;
```

Essas instruções seriam confusas para a maioria dos programadores em C++, que esperam operadores com os operandos esquerdos e respectivamente.

A função `operator<<` (definida na Figura 11.7, linhas 127–144) imprime o número de elementos indicados por array de inteiro para `cout`. A função `operator>>` (definida na Figura 11.7, linhas 118–124) realiza a entrada direamente no array para `cin`. Cada uma dessas funções operadoras retorna uma referência apropriada para permitir instruções de saída e entrada em cascata, respectivamente. Observe que cada uma dessas funções tem tempo de execução associado ao processamento dos dados. Além disso, observe que as funções `operator[]` da classe `array` poderiam ser utilizadas por `<< operator>>`, caso em que essas funções operadoras não precisariam ser Array. Entretanto, as chamadas de função adicionais poderiam aumentar o overhead de tempo de execução.

Construtor-padrãoArray

Ado hão el 5 da Figura 11.6 declara o construtor padrãoArray (definido na Figura 11.7, linhas 1–10) para a classe `array` com capacidade de 10 elementos. Quando padrão nesse exemplo realmente recebe um único argumento (o construtor-padrão de 10). O construtor-padrão (definido na Figura 11.7, linhas 18–25) valida e atribui o argumento ao ponteiro para o array que aponta para a memória para a representação interna baseada em ponteiro desse array e atribui o ponteiro interno do array. Neste ponto, o construtor usa uma instrução para configurar todos os elementos do array como zero. É possível inicializar os membros se, por exemplo, esses membros precisarem ser lidos em algum momento posterior; mas essa é considerada uma prática de programação errada. Os membros devem ser adequadamente inicializados e mantidos em um estado consistente.

Construtor de cópiaArray

A linha 16 da Figura 11.6 declara um `operator=` (definido na Figura 11.7, linhas 29–36) que inicializa uma cópia de um objeto existente. Essa cópia deve ser feita cuidadosamente para evitar a armadilha de deixar os objetos apontando para a mesma memória dinamicamente alocada. Esse é exatamente o problema que ocorreria com a cópia de um membro padrão, se o compilador tivesse permissão de definir um construtor de cópia-padrão para essa classe. Os construtores são invocados sempre que a cópia de um objeto for necessária, como ao passar um objeto por valor para uma função, ao retornar um objeto por valor a partir de uma função ou ao inicializar um objeto com uma cópia de outro objeto da mesma classe. O construtor de cópia é chamado em uma declaração quando um objeto é inicializado com outro objeto da classe, como na declaração da linha 41 da Figura 11.8.



Observação de engenharia de software 11.4

O argumento para um construtor de cópia deve ser uma referência para que o objeto seja copiado.



Erro comum de programação 11.6

Observe que um construtor de cópia deve receber seu argumento por referência, não por valor. Caso contrário, a chamada do construtor de cópia resulta em recursão infinita (um erro de lógica fatal) porque receber um objeto por valor requer que o construtor de cópia faça uma cópia do objeto de argumento. Lembre-se de que a qualquer hora em que a cópia de um objeto for requerida, o construtor de cópia da classe será chamado. Se recebesse seu argumento por valor, o construtor de cópia chamaria a si mesmo recursivamente para fazer uma cópia de seu argumento!

O construtor de cópia ~~partilha~~¹ um inicializador de membro (Figura 11.7, linha 30) para obter a memória para a representação interna baseada em ponteiro dentro do membro de classe `new` (linha 32) para obter a memória para a representação interna baseada em ponteiro desse array e atribui o ponteiro retornado ~~membrão~~² de `new`. Em seguida, o construtor de cópia utiliza uma instrução `for` para copiar todos os elementos do array para o novo array. Observe que um objeto de uma classe pode ver os dados de qualquer outro objeto dessa classe (utilizando um handle que indica que objeto acessar).



Erro comum de programação 11.7

Se o construtor de cópia simplesmente copiasse o ponteiro no objeto de source para o ponteiro do objeto-alvo, então ambos os objetos apontariam para a mesma memória dinamicamente alocada. O primeiro destrutor a executar então excluiria a memória dinamicamente alocada. Outro objeto seria indefinido, uma situação chamada de ~~garantida~~³ que provavelmente resultaria em um grave erro de tempo de execução (como a terminação precoce do programa) quando o ponteiro fosse utilizado.

¹ Observe que poderia não conseguir obter a memória necessária. [Bônus Capítulo 11: tratamento de exceções](#).

Destrutor Array

A linha 17 da Figura 11.6 declara o destrutor para a classe (definido na Figura 11.7, linhas 39–42). O destrutor é invocado quando da classe sai de escopo. O destrutor utiliza para liberar a memória alocada dinamicamente pelo array.

Função-membro getSize

A linha 18 da Figura 11.6 declara a função (**getsize**) definida na Figura 11.7, linhas 45–48) que retorna o número de elementos no Array.

Operador de atribuição sobrecarregado

A linha 20 da Figura 11.6 declara a função operadora de atribuição sobrecarregada para a classe. Ao ver a expressão integers2 na linha 49 da Figura 11.8, o compilador invoca a função comumente chamada

```
integers1. operator =( integers2 )
```

A implementação da função membro (Figura 11.7, linhas 52–70) testa ação (linha 54) em que um objeto da classe Array está sendo atribuído a si mesmo. Quando o endereço do operando auto-atribuição está sendo tentada, então a atribuição é ignorada (isto é, o objeto já é ele mesmo; logo veremos por que a auto-atribuição é perigosa). Se uma auto-atribuição, então a função-membro determina se os tamanhos dos dois arrays são idênticos (linha 58); nesse caso, inteiros srcinal no lado esquerdo não é realocado. Caso contrário, (linha 60) para liberar a memória alocada srcinalmente para o array-alvo e coloca o ponteiro retornado por **operator new** (linhas 61–62) para alocar memória para o array-alvo e coloca o ponteiro retornado por **operator new** (linhas 65–66) para copia os elementos do array de srcem para o array-alvo. Independentemente de essa ser ou não uma auto-atribuição, a função retorna o objeto atual (linha 69) como uma referência constante; isso permite que o código x = y = z . Se ocorresse a auto-atribuição, as funções associadas com objetos que a atribuição estivesse completa, isto é, daria para a memória que foi desalocada, o que poderia levar a erros fatais de tempo de execução.

**Observação de engenharia de software 11.5**

Um construtor de cópia, um destrutor e um operador de atribuição sobrecarregado normalmente são fornecidos como um grupo para qualquer classe que utiliza memória dinamicamente alocada.

**Erro comum de programação 11.8**

Não fornecer um operador de atribuição sobrecarregado e um construtor de cópia para uma classe quando os objetos dessa classe contêm ponteiros para memória dinamicamente alocada é um erro de lógica.

**Observação de engenharia de software 11.6**

É possível impedir que um objeto de uma classe seja atribuído a outra. Isso é feito declarando o operador de atribuição como um membro private da classe.

**Observação de engenharia de software 11.7**

É possível impedir que objetos de classe sejam copiados; para fazer isso, simplesmente torne ambos, o operador de atribuição sobrecarregado e o construtor de cópia, privados dessa classe.

Operadores de igualdade e desigualdade sobrecarregados

A linha 21 da Figura 11.6 declara o operador de igualdade para a classe. Ao ver a expressão

```
integers1. operator ==( integers2 )
```

A função-membro == (definida na Figura 11.7, linhas 74–84) verifica automaticamente se os membros arrays não forem iguais. Caso contrário, compara cada par de elementos. Se todos eles forem iguais, a função retorna primeiro par de elementos que diferir faz com que a função retorne.

As linhas 24–27 do arquivo de cabeçalho definem o operador de desigualdade para a classe. A função-membro operator!= usa a função == sobrecarregada para determinar se a classe é igual a outro, então retorna o oposto desse resultado.

² Mais uma vez poderia falhar. Discutimos isso no Capítulo 16.

Escrever operador= dessa maneira permite ao programador reduzir a quantidade de código que deve ser escrita na classe. Além disso, observe que a definição de função está sempre perto da definição de operador[], o que permite ao compilador colocar inline a definição para eliminar o overhead da chamada de função extra.

Operadores de subscrito sobre carregados

As linhas 30 e 33 da Figura 11.6 declaram dois operadores de subscrito sobre carregados (definidos na Figura 11.7 nas linhas 103–114, respectivamente). Ao ver a expressão `integers1[5]` (Figura 11.8, linha 61), o compilador invoca a função-membro sobre-carregada `operator[]` apropriada gerando a chamada

```
integers1.operator []( 5)
```

O compilador cria uma chamada `operator[](5)` (Figura 11.7, linhas 103–114) quando o operador de subscrito é utilizado em um objeto `array`. Por exemplo, se o objeto instanciado com a instrução

```
const Array z( 5);
```

então a versão `operator[]` é requerida para executar uma instrução como

Lembre-se, um programa pode invocar somente as funções sobrecarregadas que existem no escopo.

Cada definição `operator[]` determina se o subscrito que ele recebe como um argumento está no intervalo. Se não estiver, cada função imprime uma mensagem de erro e termina o programa com uma catena de erros.³ Se o argumento do subscrito estiver no intervalo, `operator[]` retorna o elemento do array apropriado como uma referência para que ele possa ser utilizado diretamente (por exemplo, no lado esquerdo de uma instrução de atribuição). Se o subscrito estiver no intervalo, a versão `operator[]` retorna uma cópia do elemento apropriado do array. O resultado retornado é um

11.9 Convertendo entre tipos

A maioria dos programas processa informações de muitos tipos. Às vezes, todas as operações ‘permanecem dentro de um tipo’ — por exemplo, adicionar `array1` produz `array` (contanto que o resultado não seja muito grande para ser representado como um `int`). Entretanto, freqüentemente é necessário converter dados de um tipo em dados de outro tipo. Isso pode acontecer em atribuições, em cálculos, na passagem de valores para funções e no retorno de valores a partir de funções. O compilador sabe executar conversões entre tipos fundamentais (como discutimos no Capítulo 6). Os programadores podem usar operadores de coerção para fazer conversões entre tipos fundamentais.

Mas o que dizer dos tipos definidos pelo usuário? O compilador não pode saber antecipadamente como converter entre tipos definidos pelo usuário?

Um operador de conversão (também chamado de operador de coerção) pode ser usado para converter um objeto de uma classe em um objeto de outra classe ou em um objeto de um tipo fundamental. Esse operador de conversão deve ser uma função-não-`static`. O protótipo de função

```
A::operator char *() const;
```

declara uma função operadora de coerção sobre carregada para converter um objeto de tipo definido pelo usuário

`char *` temporário. A função operadora `char *` não modifica o objeto principal. Um operador de coerção sobre carregada não especifica um tipo de retorno — o tipo de retorno é o tipo no qual o objeto está sendo convertido. Se de classe, ao ver a expressão `char *(<s>)`, o compilador gera a chamada

```
s.operator char ()
```

O operador de objeto de classe é o qual a função membro `char *` está sendo invocada.

As funções operadoras de coerção sobre carregadas podem ser definidas para converter objetos de tipos definidos pelo usuário para tipos fundamentais ou em objetos de outros tipos definidos pelo usuário. Os protótipos

```
A::operator int () const;
```

declaram as funções operadoras de coerção sobre carregadas que podem converter um objeto do tipo definido pelo usuário inteiro ou em um objeto do tipo definido pelo usuário respectivamente.

Uma das características interessantes dos operadores de coerção e dos construtores de conversão é que, quando necessário, o compilador pode chamar essas funções implicitamente para criar objetos temporários. Por exemplo, se um objeto `String` definida pelo usuário aparecer em um programa em um local em que é esperado, como

³ Observe que é mais apropriado quando um subscrito está fora de intervalo ‘lançar uma exceção’ indicando o subscrito fora do intervalo. Então o programa pode ‘capturar’ essa exceção, processá-la e, possivelmente, continuar a execução. Consulte o Capítulo 16 para obter informações adicionais sobre exceções.

```
cout << s;
o compilador pode chamar a função operadora de coerção sobreparagada para converter o objeto em um
utilizando o char * resultante na expressão. Com esse operador de coerção fornecido para o uso da inserção de
fluxo não precisa ser sobrecarregado para gerar saída para uma
```

11.10 Estudo de caso: classe String

Como um exercício fundamental para nosso estudo sobre sobreulação, constituímos a classe String para a manipulação de strings (figuras 11.9–11.11). A biblioteca C++ padrão também tem uma classe.

Apresentamos um exemplo de classe na Seção 11.13 e a estudamos em detalhes no Capítulo 18. Por enquanto, utilizaremos extensamente a sobreulação de operadores para string, nossa própria classe.

Primeiro, apresentamos o arquivo de cabeçalho `classString.h`, que define os dados privados utilizados para representar o objeto string. Então percorremos a interface da classe, discutindo cada um dos serviços que a classe fornece. Discutimos as

definições das funções-membro da classe String e suas respectivas implementações sobrecarregadas, mostrando o código no editor.

Definição da classe String

Agora vamos percorrer o arquivo de cabeçalho da classe String (Figura 11.9). Começamos com a representação interna baseada em ponteiro de string. As linhas 55–56 declaram os membros da classe. Nossa classe tem um campo `length`, que representa o número de caracteres na string, não incluindo o caractere nulo. O campo `data` tem um ponteiro para a memória dinamicamente alocada que representa a string de caracteres.

Sobrecarregando operadores de inserção e extração de fluxo

As linhas 12–13 (Figura 11.9) declaram a função operadora de inserção de fluxo (`<<`) (Figura 11.10, linhas 170–174) e a função operadora de extração de fluxo (`>>`) (Figura 11.10, linhas 177–183) como membros da classe. A implementação é simples e direta. Observe que a restrição que restringe o número total de caracteres que podem ser lidos é de 100 (linha 180); a posição 100 é reservada para o caractere nulo de terminação da string. [Nota: Não tivemos essa restrição na classe string (Figuras 11.6–11.7), porque o tipo dessa classe leu um elemento por vez e parou de ler os valores quando o fim do array foi alcançado.] Sobrepõe-se ao padrão para a entrada de arrays de caracteres. Além disso, a linha 180 serve para atribuir o estilo C ao objeto String que referencia. Essa instrução invoca o construtor de conversão para o tipo `char` que tem a string no estilo do C; o objeto temporário é, então, atribuído a `s`. Poderíamos eliminar o overhead de criação do objeto

aqui fornecendo outro operador de atribuição sobrecarregado que recebe um parâmetro do tipo `Construtor de conversão`.

A linha 15 (Figura 11.9) declara um construtor de conversão. Esse construtor (definido na Figura 11.10, linhas 22–27) aceia argumentos `const char *` (que assume o padrão da string vazia; Figura 11.9, linha 15) e `const string &` (que indica que o objeto é uma string de caractere). É possível imaginar qualquer argumento como um construtor de conversão. Como veremos, esses construtores são úteis quando estamos fazendo conversões entre tipos. O construtor de conversão pode converter uma string para um objeto, que pode ser então atribuído ao objeto. A disponibilidade desse construtor de conversão significa que não é necessário fornecer um operador de atribuição sobrecarregado para strings de caracteres. O operador invoca o construtor de conversão para criar um objeto temporário contendo a string de caracteres; então o operador de atribuição sobrecarregado é invocado para atribuir o objeto temporário a outro objeto.



Observação de engenharia de software 11.8

Quando um construtor de conversão é utilizado para realizar uma conversão implícita, o C++ pode aplicar apenas uma chamada de construtor implícito (isto é, uma única conversão definida pelo usuário) para tentar atender às necessidades de outro operador sobrecarregado. O compilador não atenderá às necessidades de um operador sobrecarregado para realizar uma série de conversões implícitas definidas pelo usuário.

```
1 // Figura 11.9: String.h
2 // Definição da classe String.
3 #ifndef STRING_H
4 #define STRING_H
```

Figura 11.9 Definição da classe String sobrecarga de operadores.

(continua)

```

5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class String
11 {
12     friend ostream &operator<<( ostream &, const String & );
13     friend istream &operator>>( istream &, String & );
14 public :
15     String( const char * = "" ); // construtor de conversão/padrão
16     String( const String & ); // construtor de cópia
17     ~String(); // destrutor
18
19     const String & operator =( const String & ); // operador de atribuição
20     const String & operator +=( const String & ); // operador de concatenação
21
22     bool operator !=() const; // a String está vazia?
23     bool operator ==( const String & ) const; // testa s1 == s2
24     bool operator <( const String & ) const; // testa s1 < s2
25
26     // testa s1 != s2
27     bool operator !=( const String &right ) const
28     {
29         return !( * this == right );
30     } // fim da função operator!=
31
32     // testa s1 > s2
33     bool operator >( const String &right ) const
34     {
35         return right < * this ;
36     } // fim da função operator>
37
38     // testa s1 <= s2
39     bool operator <=( const String &right ) const
40     {
41         return !( right < * this );
42     } // fim da função operator<=
43
44     // testa s1 >= s2
45     bool operator >=( const String &right ) const
46     {
47         return !( * this < right );
48     } // fim da função operator>=
49
50     char &operator []( int ); // operador de subscrito (lvalue modificável)
51     char operator [] ( int ) const; // operador de subscrito (rvalue)
52     String operator ()( int , int = 0 ) const; // retorna uma substring
53     int getLength() const; // retorna o comprimento da string
54
55 private:
56     int length; // comprimento de string (sem contar terminador nulo)
57     char *sPtr; // ponteiro para iniciar string baseada em ponteiro
58
59     void setString( const char * ); // função utilitária
60
61 }; // fim da classe String
62
63 #endif

```

Figura 11.9 Definição da classe String sobrecarga de operadores.

(continuação)

```

1 // Figura 11.10: String.cpp
2 // Definições de função-membro para a classe String.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstring> // protótipos de strcpy e strcat
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // sai do protótipo
17 using std::exit;
18
19 #include "String.h" // definição da classe String
20
21 // construtor de conversão (e padrão) converte char * em String
22 String::String( const char *s )
23 : length( ( s != 0 ) ? strlen( s ) : 0 )
24 {
25     cout << "Conversion (and default) constructor: " << s << endl;
26     setString( s ); // chama função utilitária
27 } // fim do construtor de conversão String
28
29 // construtor de cópia
30 String::String( const String &copy )
31 : length( copy.length )
32 {
33     cout << "Copy constructor: " << copy.sPtr << endl;
34     setString( copy.sPtr ); // chama função utilitária
35 } // fim do construtor de cópia String
36
37 // Destruitor
38 String::~String()
39 {
40     cout << "Destruitor: " << sPtr << endl;
41     delete [] sPtr; // libera memória de string baseada em ponteiro
42 } // fim do destrutor ~String
43
44 // operador = sobrecarregado; evita auto-atribuição
45 const String &String::operator=( const String &right )
46 {
47     cout << "operator= called" << endl;
48
49     if ( &right != this ) // evita auto-atribuição
50     {
51         delete [] sPtr; // impede vazamento de memória
52         length = right.length; // novo comprimento de String
53         setString( right.sPtr ); // chama função utilitária
54     } // fim do if
55     else
56     cout << "Attempted assignment of a String to itself" << endl;
57 }
```

Figura 11.10 Definições de função-membro e função `operator=` da classe String.

(continua)

```

58     return * this ; // permite atribuições em cascata
59 } // fim da função operator=
60
61 // concatena operando direito com esse objeto e armazena nesse objeto
62 const String &String::operator+=( const String &right )
63 {
64     size_t newLength = length + right.length;      // novo comprimento
65     char *tempPtr = new char[ newLength + 1 ]; // cria memória
66
67     strcpy( tempPtr, sPtr ); // copia sPtr
68     strcpy( tempPtr + length, right.sPtr ); // copia right.sPtr
69
70     delete [] sPtr; // reivindica espaço antigo
71     sPtr = tempPtr; // atribui novo array a sPtr
72     length = newLength; // atribui novo comprimento a length
73     return * this ; // permite chamadas em cascata
74 } // fim da função operator+=
75
76 // esta String está vazia?
77 bool String::operator!() const
78 {
79     return length == 0;
80 } // fim da função operator!
81
82 // esta String é igual à String direita?
83 bool String::operator==( const String &right ) const
84 {
85     return strcmp( sPtr, right.sPtr ) == 0;
86 } // fim da função operator==
87
88 // esta String é menor que a String direita?
89 bool String::operator<( const String &right ) const
90 {
91     return strcmp( sPtr, right.sPtr ) < 0;
92 } // fim da função operator<
93
94 // retorna a referência ao caractere na String como um lvalue modificável
95 char &String:: operator[]( int subscript )
96 {
97     // testa se o subscrito está fora do intervalo
98     if ( subscript < 0 || subscript >= length )
99     {
100         cerr << "Error: Subscript " << subscript
101         << " out of range" << endl;
102         exit( 1 ); // termina o programa
103     } // fim do if
104
105     return sPtr[ subscript ]; // retorno não-const; lvalue modificável
106 } // fim da função operator[]

107 // retorna referência a caractere em String como rvalue
108 char String::operator[]( int subscript ) const
109 {
110     // testa se o subscrito está fora de intervalo
111     if ( subscript < 0 || subscript >= length )
112     {

```

Figura 11.10 Definições de função-membro e função `friend` da classe `String`.

(continua)

```

114     cerr << "Error: Subscript "
115         << " out of range" << endl;
116     exit( 1 ); // termina o programa
117 } // fim do if
118
119 return sPtr[ subscript ]; // retorna cópia desse elemento
120 } // fim da função operator[]
121
122 // retorna uma substring que começa em index e tem comprimento de subLength
123 String String::operator()( int index, int subLength ) const
124 {
125     // se o índice estiver fora do intervalo ou o comprimento de substring < 0,
126     // retorna um objeto String vazio
127     if ( index < 0 || index >= length || subLength < 0 )
128         return "" ; // convertido em um objeto String automaticamente
129
130     // determina o comprimento da substring
131     int len;
132
133     if ( ( subLength == 0 ) || ( index + subLength > length ) )
134         len = length - index;
135     else
136         len = subLength;
137
138     // aloca array temporário para a substring e
139     // caractere de terminação nulo
140     char *tempPtr = new char[ len + 1 ];
141
142     // copia a substring para o array char e termina a string
143     strncpy( tempPtr, &sPtr[ index ], len );
144     tempPtr[ len ] = '\0' ;
145
146     // cria objeto String temporário contendo a substring
147     String tempString( tempPtr );
148     delete [] tempPtr; // exclui o array temporário
149     return tempString; // retorna cópia da String temporária
150 } // fim da função operator()
151
152 // retorna o comprimento da string
153 int String::getLength() const
154 {
155     return length;
156 } // fim da função getLength
157
158 // função utilitária chamada por construtores e operator=
159 void String::setString( const char *string2 )
160 {
161     sPtr = new char[ length + 1 ]; // aloca memória
162
163     if ( string2 != string2 ) // se string2 não for um ponteiro nulo, copia o conteúdo
164         strcpy( sPtr, string2 ); // copia literal para objeto
165     else // se string2 é um ponteiro nulo, torna essa string uma string vazia
166         sPtr[ 0 ] = '\0' ; // string vazia
167 } // fim da função setString
168
169 // operador de saída sobreescarregado

```

Figura 11.10 Definições de função-membro e função `main` da classe `String`.

(continua)

```

170 ostream &operator<<( ostream &output, const String &s )
171 {
172     output << s.sPtr;
173     return output; // permite cascataamento
174 } // fim da função operator<<
175
176 // operador de entrada sobrecarregado
177 istream &operator>>(istream &input, String &s)
178 {
179     char temp[ 100]; // buffer para armazenar entrada
180     input >> setw( 100) >> temp;
181     s = temp; // utiliza operador de atribuição da classe String
182     return input; // permite cascataamento
183 } // fim da função operator>>

```

Figura 11.10 Definições de função-membro e função-função da classe String.

(continuação)

```

1 // Figura 11.11: fig11_11.cpp
2 // Programa de teste da classe String.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha;
7
8 #include "String.h"
9
10 int main()
11 {
12     String s2( "happy day" );
13     String s3;
14
15     // testa operadores igualdade e relacionais sobrecarregados
16     cout << "s1 is '"
17         << s1 << "'"; s2 is '"
18         << s2 << "'"
19         << endl; s3 is '"
20         << s3 << "'"
21         << endl; The results of comparing s2 and s1:"
22         << endl; s2 == s1 yields "
23         << endl; s2 != s1 yields "
24         << endl; s2 > s1 yields "
25         << endl; s2 < s1 yields "
26         << endl; s2 >= s1 yields "
27         << endl; s2 <= s1 yields "
28
29     // testa operador de String sobrecarregado vazio (!)
30     cout << endl; Testing !s3: << endl;
31
32     if ( !s3 )
33     {
34         cout << "s3 is empty; assigning s1 to s3;" << endl;
35         s3 = s1; // testa a atribuição sobrecarregada
36     } // fim do if
37

```

Figura 11.11 Programa de teste da classe String.

(continua)

```

38     // testa o operador de concatenação de String sobrecarregado
39     cout << "\nns1 += s2 yields s1 = " ;
40     s1 += s2; // testa concatenação sobrecarregada
41     cout << s1;
42
43     // testa o construtor de conversão
44     cout << "\nns1 += \" to you\" yields"    << endl;
45     s1 += " to you"; // testa o construtor de conversão
46     cout << "s1 = " << s1 << "\n\n" ;
47
48     // testa o operador de chamada de função sobrecarregado ()para a substring
49     cout << "The substring of s1 starting at\n"
50     << "location 0 for 14 characters, s1(0, 14), is:\n"
51     << s1( 0, 14) << "\n\n" ;
52
53     // testa a opção de substring "to-end-of-String"
54     cout << "The substring of s1 starting at\n"
55     << "location 15, s1(15), is: "
56     << s1( 15) << "\n\n" ;
57
58     // testa o construtor de cópia
59     String *s4Ptr = newString( s1 );
60     cout << "\n*s4Ptr = " << *s4Ptr << "\n\n" ;
61
62     // testa o operador de atribuição (=) com a auto-atribuição
63     cout << "assigning *s4Ptr to *s4Ptr" << endl;
64     *s4Ptr = *s4Ptr; // testa a atribuição sobrecarregada
65     cout << "*s4Ptr = " << *s4Ptr << endl;
66
67     // testa o destrutor
68     delete s4Ptr;
69
70     // testa o uso do operador de subscrito para criar um lvalue modificável
71     s1[ 0 ] = 'H';
72     s1[ 6 ] = 'B';
73     cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
74     << s1 << "\n\n" ;
75
76     // testa o subscrito fora do intervalo
77     cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
78     s1[ 30 ] = 'd'; // ERRO: subscrito fora do intervalo
79     return 0;
80 } // fim de main

```

Conversion (and default) constructor: happy
 Conversion (and default) constructor: birthday
 Conversion (and default) constructor:
 s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
 s2 == s1 yields false
 s2 != s1 yields true
 s2 > s1 yields false
 s2 < s1 yields true
 s2 >= s1 yields false
 s2 <= s1 yields true

Figura 11.11 Programa de teste da classString .

(continua)

```

Testing ls3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (anddefault) constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range

```

Figura 11.11 Programa de teste da classe String .

(continuação)

O construtor de conversão poderia ser invocado nessa declaração⁴ (não é). O construtor de conversão calcula o comprimento de seu argumento de string de caracteres e ~~atribui diretamente os dados de membro~~⁵. Então, a linha 26 chama a função ~~definida~~⁶ na Figura 11.10, linhas 159–167, que aloca uma quantidade suficiente de memória para os ~~dados~~⁷ a fim de copiar a string de caracteres para a memória para a aposta.

⁴ Há uma questão sutil na implementação desse construtor de conversão. Quando implementado, se o parâmetro é um ponteiro nulo, o programa falhará. A maneira adequada de implementar esse construtor seria detectar se o argumento de construtor é um ponteiro nulo e, então, uma exceção⁸. O Capítulo 16 discute como podemos tornar as classes mais robustas dessa maneira. Além disso, observe que um ponteiro nulo (o mesmo que uma string vazia) é um ponteiro que não aponta para nada. Uma string vazia é uma string real que contém um único caractere nulo().

Construtor de cópiaString

A linha 16 na Figura 11.9 declara um construtor de cópia (definido na Figura 11.10, linhas 30–35) que inicializa um objeto fazendo a cópia de ~~um objeto~~ o objeto. Como com a ~~nova~~ (linhas 11.6–11.7), essa cópia deve ser feita cuidadosamente para evitar a armadilha em que ~~aponta~~ aponta para a mesma memória dinamicamente alocada. O construtor de cópia opera de maneira semelhante ao construtor de conversão, exceto pelo fato de que simplesmente copia o membro `objetoString` de `srcem` para o `objetoAlvo`. Observe que o construtor de cópia ~~criaria~~ criaria o novo espaço para a string de caracteres interna do objeto-alvo. Se ~~simplesmente~~ simplesmente apontaria o novo espaço para o objeto-alvo, então ambos os objetos apontariam para mesma memória dinamicamente alocada. O primeiro destrutor a executar então excluiria a dinamicamente alocada de outro objeto seria indefinido (`string` é um ponteiro oscilante), uma situação potencial para causar um erro sério de tempo de execução.

DestrutorString

A linha 17 da Figura 11.9 declara o destrutor na Figura 11.10, linhas 38–42). O destrutor libera

a memória dinâmica para ~~aponta~~ aponta.

Operador de atribuição sobrecarregado

A linha 19 (Figura 11.9) declara a função operadora de atribuição (`operator=`) definida na Figura 11.10, linhas 45–59).

Ao ver a expressão `s1 = s2`, o compilador gera a chamada de função

```
string1. operator =( string2 );
```

A função operadora de atribuição ~~sobrecreta~~ sobrecarta a auto-atribuição. Se essa for uma auto-atribuição, a função não precisará mudar o objeto. Se esse teste fosse omitido, a função excluiria imediatamente o espaço no objeto-alvo e, assim, perder de caracteres, de modo que o ponteiro não apontaria para dados válidos — um exemplo clássico de um ponteiro oscilante. Se r auto-atribuição, a função exclui a memória ~~aponta~~ aponta para o objeto-alvo. Em seguida, chama `String` para criar o novo espaço para o objeto-alvo e copia a string de caracteres do objeto de `srcem` para o objeto-alvo. Essa uma auto-atribuição ~~operador~~ torna `operator=` para permitir atribuições em cascata.

Operador de atribuição de adição sobrecarregado

A linha 20 da Figura 11.9 declara o operador de concatenação (`operator+=`), definido na Figura 11.10, linhas 62–74). Ao ver a expressão `s1 += s2` (linha 40 da Figura 11.11), o compilador gera a chamada de função-membro

```
s1.operator +=( s2 );
```

~~Anuparando temporários~~ A função de operador de concatenação sobrecarregado é matizada para tratar a string concatenada. Em seguida, `operator+=` utiliza `strcpy` para copiar as strings de caracteres da string `srcinal` para a string `temp`. Observe que a localização ~~aponta~~ aponta para o primeiro caractere, devido ao cálculo aritmético de `ponteiro + length`. Esse cálculo indica que o primeiro caractere deve ser colocado na localização array para o qual aponta. Em seguida, `operator+=` utiliza `delete []` para liberar o espaço ocupado pela string de caracteres `srcinal` desse objeto `temp`, de modo que esse objeto aponte para a nova string de caracteres. `operator+=` atribui `length` de modo que esse objeto contenha o novo comprimento de string `newLength`. `String &` para permitir a cascata de operadores.

Precisamos de um segundo operador de concatenação sobrecarregado para permitir a concatenação de uma string para o tipo `char`. O construtor de conversão converte uma string no estilo C ~~para~~ para o tipo `char`, que então corresponde ao operador de concatenação sobrecarregado existente. Isso é exatamente o que o compilador faz quando encontra a linha 4–11. Novamente, o C++ pode realizar essas conversões somente em um nível profundo para facilitar uma correspondência também pode realizar uma conversão definida por compilador implícita entre tipos fundamentais antes de realizar a conversão um tipo fundamental e uma classe. Observe que, quando o operador é criado nesse caso, o construtor de conversão e o destrutor são chamados (veja a saída resultante da linha 45 da Figura 11.11). Esse é um exemplo de overhead de chamada de função que fica oculto do cliente da classe quando objetos de classe temporários são criados e destruídos conversões implícitas. Overhead semelhante é gerado por construtores de cópia em passagem de parâmetros de chamada no retorno de objetos de classe por valor.



Dica de desempenho 11.2

Sobrecarregar o operador de concatenação versão adicional que aceita um único argumento do tipo `char` torna a execução mais eficiente do que ter apenas uma versão que aceita um argumento do tipo `String`. O argumento `char` seria primeiramente convertido em `String` o construtor de conversão da classe `String`, então o operador `+=` receberia um argumento que seria chamado para realizar a concatenação.



Observação de engenharia de software 11.9

Utilizar conversões implícitas com operadores sobre carregados, em vez de sobre carregar operadores para muitos tipos diferentes de operandos, freqüentemente exige menos código, o que torna uma classe mais fácil de modificar, manter e depurar.

Operador de negação sobre carregado

A linha 22 da Figura 11.9 declara o operador de negação sobre carregado (definido na Figura 11.10, linhas 77–80). Esse operador termina se um objeto de `nossa classe`. Por exemplo, ao ver a expressão `!string1`, o compilador gera a chamada de função

```
string1. operator !()
```

Essa função simplesmente retorna o resultado `de guarda zero`.

Operadores de igualdade e relacionais sobre carregados

As linhas 23–24 da Figura 11.9 declaram o operador de igualdade sobre carregado (definido na Figura 11.10, linhas 83–86) e o menor que sobre carregado (definido na Figura 11.10, linhas 87–90). Para os operadores `==` e `<`, entretanto, vamos discutir somente um exemplo, a saber, sobre carregar o operador `==`. Se `string1 == string2`, o compilador gera a chamada de função-membro

```
string1. operator ==( string2 )
```

que retorna `se string1 for igual a string2`. Cada um desses operadores utiliza `(funcção)` para comparar as strings de caractere no objeto. Muitos programadores em C++ defendem o uso de algumas funções operadoras sobre carregadas para implementar outras. Então, os operadores implementados (Figura 11.9, linhas 27–48) em termos de `operator<`. Por exemplo, a função sobre carregada (implementada nas linhas 45–48 no arquivo de cabeçalho) utiliza o operador sobre carregado para determinar se um objeto é menor ou igual ao outro. Observe que as funções operadoras para `<`, `>`, `<=` e `=` são definidas no arquivo de cabeçalho. O compilador coloca essas definições inline para eliminar o overhead da chamadas de função extras.



Observação de engenharia de software 11.10

Implementando funções-membro com funções-membro anteriormente definidas, o programador reutiliza o código para reduzir a quantidade de código que deve ser escrita e mantida.

Operadores de subscrito sobre carregados

As linhas 50–51 no arquivo de cabeçalho declaram dois operadores de subscrito sobre carregados (definidos na Figura 11.10, linhas 106 e 109–120, respectivamente). Um para `string` e um para `Sconst`. Ao ver uma expressão como `string[0]`, o compilador gera a chamada de função-membro

```
string1. operator []( 0 )
```

(utilizando a versão apropriada com base no fato de ser ou não⁵). Cada implementação valida primeiro o subscrito para assegurar que ele está no intervalo. Se o subscrito estiver fora do intervalo, cada função imprime mensagem de erro e termina o programa com `exit(1)`. Se o subscrito estiver no intervalo, a função `operator[]` retorna `char &` para o caractere apropriado do `string` e pode ser utilizado `char[]` para modificar o caractere designado. No caso de `Sconst`, a versão de `operator[]` retorna o caractere apropriado, isso pode ser utilizado somente `const` para ler o valor do caractere.



Dica de prevenção de erro 11.2

É perigoso retornar uma referência para um operador de subscrito sobre carregado de uma classe. Por exemplo, o cliente poderia utilizar essa referência para `inserir qualquer lugar` da string.

Adendo 52 da Figura 11.9 declara a função sobre carregada de função sobre carregada (definido na Figura 11.10, linhas 123–150). Sobrecarregamos esse operador para selecionar uma substring. Subscritos inteiros do tipo inteiro especificam a localização inicial e o comprimento da substring sendo selecionada. Se a localização inicial estiver fora do intervalo ou o comprimento de substring for negativo, o operador simplesmente ignora. Se o comprimento da substring for 0, então a substring é selecionada até o final. Por exemplo, suponha-seja um objeto `contendo a string`. Para a expressão `contendo(2, 2)`, o compilador gera a chamada de função-membro

⁵ Novamente, é mais apropriado quando um subscrito está fora de intervalo ‘lançar uma exceção’ indicando o subscrito fora do intervalo.

```
string1. operator ()( 2, 2 )
```

Quando essa chamada executa, ela produzirá uma string completa.

Sobrecregar o operador de chamada é um pouco mais poderoso, porque as funções podem aceitar listas de parâmetros arbitrariamente longas e complexas. Então podemos utilizar essa capacidade para muitos propósitos interessantes. Esse tipo de operador de chamada de função é uma notação de subscrito de array alternativa: em vez de utilizar a complicada notação com índices duplos para arrays bidimensionais baseados em ponteiros, alguns programadores preferem sobrecregar o operador de chamada de função para permitir a chamada de array. O operador de chamada de função sobrecregido deve ser uma função-membro da classe. Esse operador é utilizado somente quando o 'nome da função' é um objeto da classe.

Função-membro `getLength` da string

A linha 53 na Figura 11.9 declara a função `getLength` (definida na Figura 11.10, linhas 153–156), que retorna o comprimento de uma String.

`String`

Neste ponto, o compilador deve passar pelo código para janela de saída e verificar cada utilização de um operador sobrecregido. Ao estudar a saída, preste atenção especial às chamadas de construtor implícito que são geradas para criar objetos por todo o programa. Muitas dessas chamadas introduzem um overhead adicional no programa que podem ser evitados se a necessidade operadores sobrecregidos que aceitam argumentos. Entretanto, essas funções operadoras adicionais podem tornar a classe mais difícil de manter, modificar e depurar.

11.11 Sobrecregando ++ e --

As versões prefixada e pós-fixada dos operadores de incremento e decremento podem ser inteiramente sobrecregadas. Verifique o compilador distingue entre a versão prefixada e a pós-fixada de um operador de incremento ou decremento.

Para sobrecregar o operador de incremento para permitir tanto o uso de incremento prefixado como de pós-fixado, toda operadora sobrecregida deve ter uma assinatura distinta para que o compilador possa distinguir que versão de versões prefixadas são sobrecregidas exatamente como qualquer outro operador unário prefixado seria.

Sobrecregendo o operador de incremento prefixado

Suponha, por exemplo, que quiséssemos adicionar a expressão de pré-incremento ao compilador gera a chamada de função-membro

```
d1.operator ++()
```

O protótipo dessa função operadora seria

```
Date &operator ++();
```

Se o operador de incremento prefixado é implementado como uma função global, então o compilador gera a expressão de função

```
operator ++( d1 )
```

O protótipo dessa função operadora seria declarado na classe

```
Date &operator ++( Date & );
```

Sobrecregendo o operador de incremento pós-fixado

Sobrecregar o operador de incremento pós-fixado apresenta um desafio, porque o compilador deve ser capaz de distinguir assinaturas das funções operadoras sobrecregidas de incremento prefixado e pós-fixado. A convenção que foi adotada em é ver a expressão de pós-incremento compilador gera a chamada de função-membro

```
d1.operator ++( 0 )
```

O protótipo dessa função é

```
Date operator ++( int )
```

Fixando o problema: é um 'valor fictício' que permite ao compilador distinguir entre as funções operadoras de incremento pós-fixado.

Se o incremento pós-fixado é implementado como uma função global, então, quando o compilador vê a expressão de função

```
operator ++( d1, 0 )
```

O protótipo dessa função seria

```
Date operator ++( Date &, int );
```

Mais uma vez, o argumento é utilizada pelo compilador para distinguir entre os operadores de incremento prefixado e pós-fixado implementados como funções globais. Observe que o operador de incremento pós-fixado gera uma chamada de função

operador de incremento prefixado ~~retorna um objeto temporário~~, porque o operador de incremento pós-fixado em geral retorna um objeto temporário que contém o valor original do objeto antes de o incremento ocorrer. ~~O resultado é que~~ Esses objetos como não podem ser utilizados no lado esquerdo de uma atribuição. O operador de incremento prefixado retorna o objeto real incrementado com seu novo valor. Esse objeto pode ser utilizado em uma expressão contínua.



Dica de desempenho 11.3

O objeto extra que é criado pelo operador de incremento (ou decremento) pós-fixado pode resultar em um problema de desempenho significativo — especialmente quando o operador é utilizado em um loop. Por essa razão, você só deve utilizar o operador de incremento (ou decremento) pós-fixado quando a lógica do programa exigir pós-incremento (ou pós-decremento).

Tudo declarado nesta seção sobre a sobre carga de operadores de incremento prefixado e pós-fixado se aplica à sobre carga de operadores de pré-decremento e pós-decremento. Em seguida, ~~examinaremos~~ ~~os~~ sobre carregamento prefixado e pós-fixado sobrecarregados.

11.12 Estudo de caso: uma classe Date

O programa das figuras 11.12–11.14 demonstra a classe Date com operadores de incremento prefixado e pós-fixado sobrecarregados para adicionar 1 ao dia, mês e ano. Se necessário, o arquivo de cabeçalho (Figura 11.12) especifica que o projeto produz incrementos apropriados ao mês e ano, se necessário. O arquivo de cabeçalho (Figura 11.12) especifica que o projeto inclui um operador de inserção de fluxo sobrecarregado (linha 11), um construtor-padrão (linha 13), um operador de incremento prefixado sobrecarregado (linha 15), um operador de incremento pós-fixado sobrecarregado (linha 16), um operador de negação (linha 17), uma função para testar anos bissexto (linha 18) e uma para determinar se um dia é ou não o último dia do mês (linha 19).

A função main (Figura 11.14) cria três objetos (linhas 11–13) e inicializado por padrão para o dia 1 de janeiro de 2000; é inicializado como 27 de dezembro de 1992 como uma data inválida (definida na Figura 11.13, linhas 11–14) apenas para validar o mês, dia e ano especificados. Um mês inválido é configurado como 1, um ano inválido é configurado como 1900 e um dia inválido é configurado como 1.

```

1 // Figura 11.12: Date.h
2 // Definição da classe Date.
3 #ifndef DATE_H
4 #define DATE_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class Date
10 {
11     friend ostream &operator<<( ostream &, const Date & );
12 public :
13     Date( int m = 1, int d = 1, int y = 1900); // construtor-padrão
14     void setDate( int , int , int ); // configura month, day, year
15     Date &operator++(); // operador de incremento prefixado
16     Date operator++( int ); // operador de incremento pós-fixado
17     const Date &operator+=( int ); // adiciona dias, modifica objeto
18     bool leapYear( int ) const; // a data está em um ano bissexto?
19     bool endOfMonth( int ) const; // a data está no fim do mês?
20 private :
21     int month;
22     int day;
23
24     static const int days[]; // array de dias por mês
25     void helpIncrement(); // função utilitária para incrementar data
26 };
27 // fim da classe Date
28
29 #endif

```

Figura 11.12 A definição da classe Date com operadores de incremento sobrecarregados.

As linhas 15–16 geraram saída de cada um dos objetos criados, utilizando o operador de inserção de fluxo sobre-carregado (definido na Figura 11.13, linhas 96–103). A linha 17 é uma operação sobrecarregada para adicionar sete dias a `ad2`. A linha 18 utiliza a função `setDate()` para configurá-lo como 28 de fevereiro de 1992, que é um ano bissexto. Então, a linha 20 é uma operação de pré-incremento para mostrar que a data é incrementada adequadamente como 29 de fevereiro. Em seguida, a linha 22 cria um objeto `Date d4`, que é inicializado com a data 13 de julho de 2002. Depois, a linha 24 adiciona 20 incrementos a `d4`, prefixado sobrecarregado. As linhas 24–27 geram saída da operação de pré-incremento, para confirmar que ela funcionou corretamente. Por fim, a linha 29 adiciona 10 incrementos pós-fixado sobrecarregado. As linhas 29–32 geram saída de `dates` e depois da operação de pós-incremento, para confirmar que ela funcionou corretamente.

```

1 // Figura 11.13: Date.cpp
2 // Definições de função-membro da classe Date.
3 #include <iostream>
4 #include "Date.h"
5
6 // inicializa membro static no escopo de arquivo; uma cópia no nível da classe
7 const int Date::days[] =
8 { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
10 // construtor Date
11 Date::Date( int m, int d, int y )
12 {
13     setDate( m, d, y );
14 } // fim do construtor Date
15
16 // configura month, day e year
17 void Date::setDate( int mm, int dd, int yy )
18 {
19     month = ( mm >= 1 && mm <=12 ) ? mm : 1;
20     year = ( yy >= 1900&& yy <= 2100 ) ? yy : 1900
21
22     // testa se é um ano bissexto
23     if ( month == 2 && leapYear( year ) )
24         day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25     else
26         day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27 } // fim da função setDate
28
29 // operador de incremento prefixado sobrecarregado
30 Date &Date::operator++()
31 {
32     helpIncrement(); // incrementa data
33     return * this ; // retorno de referência para criar um lvalue
34 } // fim da função operator++
35
36 // operador de incremento pós-fixado sobrecarregado; observe que
37 // parâmetro fictício do tipo inteiro não tem um nome de parâmetro
38 Date Date::operator++( int )
39 {
40     Date temp = *this ; // armazena o estado atual do objeto
41     helpIncrement();
42
43     // retorna o objeto temporário, salvo, não incrementado
44     return temp; // retorno de valor; não um retorno de referência
45 } // fim da função operator++
46
46

```

Figura 11.13 Definições de função-membro e função `operator++` da classe Date

(continua)

```

47 // adiciona número especificado de dias a date
48 const Date &Date::operator+=( int additionalDays )
49 {
50     for ( int i = 0; i < additionalDays; i++ )
51         helpIncrement();
52     return * this ; // permite cascamenteamento
53 } // fim da função operator+=
54
55 // se o ano for um ano bissexto, retorna true; caso contrário, retorna false
56 bool Date::leapYear( int testYear ) const
57 {
58     if ( testYear % 400 == 0 ||
59         ( testYear % 100 != 0 && testYear % 4 == 0 ) )
60         return true ; // um ano bissexto
61     else
62         return false ; // não um ano bissexto
63 } // fim da função leapYear
64
65 // determina se o dia é o último dia do mês
66 bool Date::endOfMonth(int testDay) const
67 {
68     if ( month == 2 && leapYear( year ) )
69         return testDay == 29; // último dia de fevereiro em ano bissexto
70     else
71         return testDay == days[ month ];
72 } // fim da função endOfMonth
73
74 // função para ajudar a incrementar a data
75 void Date::helpIncrement()
76 {
77
78     // dia não é o fim do mês
79     if ( !endOfMonth( day ) )
80         day++; // incrementa dia
81     else
82         if ( month < 12 ) // dia é o fim do mês e mês < 12
83         {
84             month++; // incrementa mês
85             day = 1; // primeiro dia do novo mês
86         } // fim do if
87         else // último dia do ano
88         {
89             year++; // incrementa ano
90             month = 1; // primeiro mês do novo ano
91             day = 1; // primeiro dia do novo mês
92         } // fim do else
93 } // fim da função helpIncrement
94
95 // operador de saída sobrecarregado
96 ostream &operator<<( ostream &output, const Date &d )
97 {
98     static char *monthName[ 3 ] = { "", "January", "February",
99                                 "March", "April", "May", "June", "July", "August",
100                                "September", "October", "November", "December" };
101     output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
102     return output; // permite cascamenteamento
103 } // fim da função operator<<

```

Figura 11.13 Definições de função-membro e função end da classe Date

(continuação)

```

1 // Figura 11.14: fig11_14.cpp
2 // Programa de teste da classe Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // definição da classe Date
8
9 int main()
10 {
11     Date d1; // assume o padrão de 1º de janeiro de 1900
12     Date d2( 12, 27, 1992); // 27 de dezembro de 1992
13     Date d3( 0, 99, 8045); // data inválida
14
15     cout << "d1 is " << d1 << endl << d2 << endl << d3;
16     cout << endl << d2 += 7 is " << ( d2 += 7);
17
18     d3.setDate( 2, 28, 1992);
19     cout << endl << d3 is " << d3;
20     cout << endl << ++d3 is " << ++d3 << "(leap year allows 29th)" ;
21
22     Date d4( 7, 13, 2002);
23
24     cout << endl << "Testing the prefix increment operator:\n"
25         << " d4 is " << d4 << endl;
26     cout << "++d4 is " << ++d4 << endl;
27     cout << " d4 is " << d4;
28
29     cout << endl << "Testing the postfix increment operator:\n"
30         << " d4 is " << d4 << endl;
31     cout << "d4++ is " << d4++ << endl;
32     cout << " d4 is " << d4 << endl;
33     return 0;
34 } // fim de main

```

d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
d4 is July 13, 2002
++d4 is July 14, 2002
d4 is July 14, 2002

Testing the postfix increment operator:
d4 is July 14, 2002
d4++ is July 14, 2002
d4 is July 15, 2002

Figura 11.14 Programa de teste da classe Date

Sobrecarregar o operador de incremento prefixado é simples e direto. O operador de incremento prefixado (definido na Figura 11.13, linhas 30–34) chama a função `operator++(int)` (definida na Figura 11.13, linhas 76–93) para incrementar a data. Essa função lida com as ‘voltas’ ou ‘reinícios’ que ocorrem quando incrementamos o último dia do mês. Essas voltas requerem incrementar Se o mês já for 12, então o ano também deve ser incrementado e o mês deve ser configurado como 1. A função `adicionarMeses` para incrementar o dia corretamente.

O operador de incremento prefixado sobrecarregado retorna `const Date&` (uma referência à data que acabou de ser incrementado). Isso ocorre porque o operador é tratado como um `lvalue` que permite que um objeto incrementado seja utilizado como `rvalue`, que é como o operador de incremento prefixado predefinido funciona para tipos fundamentais.

Sobrecarregar o operador de incremento pós-fixado (definido na Figura 11.13, linhas 38–45) é mais difícil. Para emular o efeito de pós-incremento, devemos retornar uma `const Date&` (uma referência ao objeto). Por exemplo, essa é a nova instrução:

```
cout << x++ << endl;
```

gera saída do valor `srcinal da Entrada`. Entendemos que nosso operador de incremento pós-fixado operasse da mesma maneira que um `operator++`. Na entrada `operator++`, salvamos o objeto `this` (`temp`) (linha 40). Em seguida, definimos `temp` para incrementar `temp`. Então, a linha 44 retorna a cópia não incrementada do objeto previamente armazenado em `temp`. Observe que essa função não pode retornar uma referência local ao objeto. A variável local é destruída quando a função em que ela é declarada termina. Portanto, declarar o tipo de retorno para essa função é errônea: um objeto que não existe mais. Retornar uma referência (ou um ponteiro) a uma variável local é um erro comum para o qual a maioria dos compiladores emitirá um aviso.

11.13 Classe string da biblioteca-padrão

Neste capítulo, você aprendeu que é possível ~~constituir uma classe~~ (Figura 11.11) que é melhor que os strings estilo C que C++ absorveu de C. Também aprendeu que ~~é melhor que arrays baseados em ponteiro~~ (Figura 11.12) que é melhor que os arrays baseados em ponteiro no estilo C que C++ absorveu de C.

Construir classes úteis reutilizáveis ~~não exige trabalho~~. Entretanto, depois que essas classes foram testadas e depuradas, elas podem ser reutilizadas por você, seus colegas, sua empresa, muitas empresas, uma indústria inteira ou até mesmo indústrias (se forem colocadas em bibliotecas públicas ou comerciais). Os designers de C++ fizeram exatamente isso, construindo a classe `string` (que temos utilizado desde o Capítulo 3) e o template `std::vector` (discutido no Capítulo 7) no padrão C++. Essas classes estão disponíveis a qualquer pessoa construindo aplicativos com C++. Como você verá no Capítulo 23, “Standard Library (STL)”, a C++ Standard Library fornece vários templates de classes predefinidos para você utilizar em programas.

~~Na seção anterior, vimos que a classe string é útil para manipular strings. Vamos detalhar as três funções-membro da classe string que não faziam parte da classe string no exemplo. A função `empty` determina se uma string está vazia, a função `at` retorna uma `char` que representa uma parte de uma string e a função `operator[]` retorna o caractere em um índice específico (depois de verificar se o índice está no intervalo). O Capítulo 18 apresenta as classes detalhes.~~

Classe `string` da biblioteca-padrão

O programa da Figura 11.15 reimplementa o programa da Figura 11.14 para a classe `string`. Verá neste exemplo, a classe `string` fornece toda a funcionalidade de nossas classes definidas nas figuras 11.9–11.10. A classe `string` é definida no cabeçalho (linha 7) e pertence ao namespace.

As linhas 12–14 criam três objetos inicializado com `literal` e `s3` utiliza o construtor de `string` para criar. As linhas 17–18 geram saída desses três objetos, utilizando o operador que os projetistas da classe sobrecarregaram para tratar objetos, as linhas 19–25 mostram os resultados da comparação utilizando os operadores de igualdade e relacionais sobre carregados da classe.

```
2 // Programa de teste da classe string da biblioteca-padrão.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
```

Figura 11.15 Programa de teste da classe `string` da biblioteca-padrão.

(continua)

```

9
10 int main()
11 {
12     string s1( "happy");
13     string s2( " birthday" );
14     string s3;
15
16     // testa operadores de igualdade e relacionais sobrecarregados
17     cout << "s1 is \\" << s1 << "\", s2 is \\" << s2
18     << "\"; s3 is \\" << s3 << \""
19     << "\n\nThe results of comparing s2 and s1:"
20     << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
21     << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
22     << "\ns2 > s1 yields " << ( s2 > s1 ? "true" : "false" )
23     << "\ns2 < s1 yields " << ( s2 < s1 ? "true" : "false" )
24     << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
25     << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
26
27     // testa a função-membro string vazia
28     cout << "\n\nTesting s3.empty():" << endl;
29
30     if ( s3.empty() )
31     {
32         cout << "s3 is empty; assigning s1 to s3;" << endl;
33         s3 = s1; // atribui s1 a s3
34         cout << "s3 is \\" << s3 << \"";
35     } // fim do if
36
37     // testa operador de concatenação de string sobrecarregado
38     cout << "\n\ns1 += s2 yields s1 = \" ";
39     s1 += s2; // testa a concatenação sobrecarregada
40     cout << s1;
41
42     // testa operador de concatenação de string sobrecarregado com string no estilo C
43     cout << "\n\ns1 += \' to you\' yields" << endl;
44     s1 += " to you";
45     cout << "s1 = " << s1 << "\n\n";
46
47     // testa função-membro string substr
48     cout << "The substring of s1 starting at location 0 for\n"
49     << "14 characters, s1.substr(0, 14), is:\n"
50     << s1.substr( 0, 14) << "\n\n";
51
52     // testa a opção de substr "to-end-of-string"
53     cout << "The substring of s1 starting at\n"
54     << "location 15, s1.substr(15), is:\n"
55     << s1.substr( 15) << endl;
56
57     // testa o construtor de cópia
58     string *s4Ptr = new string(s1);
59     cout << s4Ptr << endl;
60
61     // testa o operador de atribuição (=) com a auto-atribuição
62     cout << "assigning *s4Ptr to *s4Ptr" << endl;
63     *s4Ptr = *s4Ptr;
64     cout << "*s4Ptr = " << *s4Ptr << endl;

```

Figura 11.13 Biblioteca C++: classe string

(continua)

```

65
66 // testa o destrutor
67 delete s4Ptr;
68
69 // testa o uso do operador de subscripto para criar lvalue
70 s1[ 0 ] = 'H';
71 s1[ 6 ] = 'B';
72 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73     << s1 << "\n\n";
74
75 // testa o subscripto fora do intervalo com a função-membro string 'at'
76 cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77 s1.at( 30 ) = 'd'; // ERRO: subscripto fora do intervalo
78 return 0;
79 } // fim de main

```

s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:

```

s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

```

Testing s3.empty():

```

s3 is empty; assigning s1 to s3;
s3 is "happy"

```

```

s1 += s2 yields s1 = happy birthday
s1 += " to you" yields
s1 = happy birthday to you

```

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at(30) yields:

abnormal program termination

Figura 11.14. Biblioteca `string`

(continuação)

Nossa classe `String` (figuras 11.9–11.10) fornece um sobrecarregado que testa para determinar se ela estava vazia ou não. A classe padrão não fornece essa funcionalidade como um operador sobrecarregado; em vez disso, fornece a função-membro `isEmpty`, que demonstramos na linha 30. A função-membro `String` estiver vazia; caso contrário, retorna `false`.

A linha 33 demonstra o operador de atribuição sobrecarregado da classe `String`. Nessa linha 33 gera saída para demonstrar que a atribuição funcionou corretamente.

A linha 39 demonstra o operador sobrecarregado da classe concatenação de `String`. Nesse caso o conteúdo de acrescentado. Então a linha 40 gera a saída da string resultante, que é a string original com a string que um literal string no estilo C pode ser acrescentado. O resultado da linha 45 exibe o resultado.

Nossa classe `String` (figuras 11.9–11.10) fornece um sobrecarregado para obter substrings. A classe padrão não fornece essa funcionalidade como um operador sobrecarregado; em vez disso, define a função-membro `substring` na linha 50 obtém uma substring de 14 caracteres (especificada pelo segundo argumento) de 0 (especificada pelo primeiro argumento). A chamada obtém uma substring que inicia a posição 15 de

o argumento, não iniciando com a localização especificada. A classe `String` fornece uma chamada para o construtor de cópia da classe. A linha 63 utiliza o operador sobrecarregado da classe para demonstrar que ela trata a auto-atribuição adequadamente.

As linhas 70–71 utilizaram o operador sobrecarregado da classe criado que permitem que novos caracteres substituam os caracteres existentes. A linha 78 gera saída do novo valor da classe (figuras 11.9–11.10), o operador sobrecarregado realizou verificação de limites para determinar se o subscrito que ele recebeu como um argumento é válido na string. Se o subscrito fosse inválido, o operador imprimiria uma mensagem de erro e terminaria o programa. O operador sobrecarregado da classe padrão não realiza nenhuma verificação de limites. Portanto, o programador deve assegurar que as operações que utilizam o operador sobrecarregado da classe padrão não manipulam accidentalmente os elementos fora dos limites da string. A classe `String` fornece verificação de limites em sua função-membro exceção se seu argumento for um subscrito inválido. Por padrão, isso teria sido subprograma. No entanto, o caractere na localização especificada pode ser modificado (isto é, uma referência dependendo do contexto em que a chamada aparece. A linha 77 demonstra uma substring inválida)

11.14 Construtores explicit

Nas seções 11.8 e 11.9, discutimos que qualquer construtor de um único argumento pode ser utilizado pelo compilador para uma conversão implícita — o tipo recebido pelo construtor é convertido em um objeto da classe em que o construtor é definido. Essa conversão é automática e o programador não precisa utilizar um operador de coerção. Em algumas situações, as conversões implícitas podem ser desejáveis ou propensas a erros. Por exemplo, a figura 11.6 define um construtor que aceita um único argumento `int`. A intenção desse construtor é criar um array com o número de elementos especificado. Entretanto, esse construtor pode ser mal-empregado pelo compilador para realizar uma conversão implícita.



Erro comum de programação 11.9

Infelizmente, o compilador poderia utilizar conversões implícitas em casos que você não espera, resultando em expressões ambíguas que geram erros de compilação ou resultam em erros de lógica em tempo de execução.

Utilizando accidentalmente um construtor de um único argumento como um construtor de conversão

O programa (Figura 11.16) utiliza a figura 11.6–11.7 para demonstrar uma conversão implícita inadequada.

A linha 13 cria uma instância do objeto `integers1` e chama o construtor de um único argumento para o valor `3`. A intenção é especificar o número de elementos. No entanto, a partir do que foi visto na Figura 11.6, que o construtor `integers1` inicializa todos os elementos do array como 0. A linha 14 chama `length` (definida nas linhas 20–24), que recebe como seu argumento `integers1` para uma. A função gera saída do número de elementos em seu argumento `Array` e do conteúdo do array. Nesse caso, o tamanho dentro de `length` é enviado para a saída.

A linha 15 chama a função `outputArray` com o valor `3` como um argumento. Entretanto, esse programa não contém uma função chamada `outputArray` que aceite um argumento. Como é que o compilador determina que o argumento é considerado um construtor de conversão? O compilador assume que o construtor é o construtor de conversão e o utiliza para converter o argumento `3` para um objeto temporário que contém três elementos. Entretanto, o compilador passa o objeto temporário à função `outputArray` para gerar a saída do conteúdo. Portanto, mesmo que não fornecemos explicitamente uma função `outputArray` que recebe um argumento, o compilador consegue compilar a linha 15. A saída mostra o conteúdo do array de três elementos contendo

⁶ Novamente, o Capítulo 16, “Tratamento de exceções” demonstra como ‘capturar’ essas exceções.

```

1 // Figura 11.16: fig11_16.cpp
2 // Driver para a classe Array simples.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray( const Array & ); // protótipo
10
11 int main()
12 {
13     Array integers1( 7 ); // array de 7 elementos
14     outputArray( integers1 ); // gera saída do Array integers1
15     outputArray( 3 ); // converte 3 em um Array e gera saída do conteúdo de Array
16     return 0;
17 } // fim de main
18
19 // imprime conteúdo de Array
20 void outputArray( const Array &arrayToOutput )
21 {
22     cout << "The Array received has " << arrayToOutput.getSize()
23     << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // fim de outputArray

```

The Array received has 7 elements. The contents are:

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | |

The Array received has 3 elements. The contents are:

| | | |
|---|---|---|
| 0 | 0 | 0 |
|---|---|---|

Figura 11.16 Construtores de um único argumento e conversões implícitas.

Impedindo a utilização acidental de um construtor de um único argumento como um construtor de conversão. O C++ fornece a palavra-chave `explicit` para suprimir as conversões implícitas via construtores de conversão quando essas conversões não devem ser permitidas. Um construtor que é declarado com `explicit` não pode ser utilizado em uma conversão implícita. A Figura 11.17 declara um construtor `explicit` na classe `Array`. A única modificação feita no código-fonte foi a adição da palavra-chave `explicit` à declaração do construtor de um único argumento na linha 15. Nenhuma modificação é requerida no arquivo de código-fonte contendo de função-membro da classe.

A Figura 11.18 apresenta uma versão ligeiramente modificada do programa na Figura 11.16. Quando esse programa é compilado, o compilador produz uma mensagem de erro que indica que o valor `3` do tipo `int` não pode ser convertido para o tipo `const Array &`. A mensagem de erro do compilador é mostrada na janela de saída. A linha 16 demonstra como o construtor explícito pode ser utilizado para fornecer os elementos e passá-lo para a função.



Erro comum de programação 11.10

Tentar invocar um construtor para uma conversão implícita é um erro de compilação.



Erro comum de programação 11.11

Utilizar a palavra-chave `explicit` em membros de dados ou funções-membro diferentes de um construtor de um único argumento é um erro de compilação.

```

1 // Figura 11.17: Array.h
2 // Classe Array para armazenar arrays de inteiros.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public :
15     explicit Array( int = 10 ); // construtor-padrão
16     Array( const Array & ); // construtor de cópia
17     ~Array(); // destrutor
18     int getSize() const; // retorna tamanho
19
20     const Array &operator=( const Array & ); // operador de atribuição
21     bool operator==( const Array & ) const; // operador de igualdade
22
23     // operador de desigualdade; retorna o oposto do operador ==
24     bool operator!=( const Array &right ) const
25     {
26         return !(*this == right); // invoca Array::operator==
27     } // fim da função operator!=
28
29     // operador de subscrito para objetos não-const retorna lvalue
30     int &operator[]( int );
31
32     // operador de subscrito de objetos const retorna rvalue
33     const int &operator[]( int ) const;
34 private :
35     int size; // tamanho do array baseado em ponteiro
36     int *ptr; // ponteiro para o primeiro elemento do array baseado em ponteiro
37 }; // fim da classe Array
38
39 #endif

```

Figura 11.17 Definição da classe `Array` com o construtor `explicit`.

```

1 // Figura 11.18: fig11_18.cpp
2 // Driver para a classe Array simples.
3 #include <iostream>
4 using std::cout;
5
6 using std::endl;
7 #include "Array.h"
8
9 void outputArray( const Array & ); // protótipo
10
11 int main()
12 {

```

Figura 11.18 Demonstrando um construtor `explicit`.

(continua)

```

13     Array integers1(    7); // array de 7 elementos
14     outputArray(integers1); // gera saída do Array integers1
15     outputArray( 3 ); // converte 3 em um Array e gera saída do conteúdo do Array
16     outputArray( Array(    3 ) ); // chamada do construtor explicit de um único argumento
17     return 0;
18 } // fim de main
19
20 // imprime o conteúdo do array
21 void outputArray( const Array &arrayToOutput )
22 {
23     cout << "The Array received has " << arrayToOutput.getSize()
24     << " elements. The contents are:\n" << arrayToOutput << endl;
25 } // fim de outputArray

```

```
c:\cpphtp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
Reason: cannot convert from 'int' to 'const Array'
Constructor for class 'Array' is declared 'explicit'
```

Figura 11.18 Demonstrando um construtor explicit.

(continuação)

**Dica de prevenção de erro 11.3**

Utilize a palavra-chave `explicit` em construtores de um único argumento que não devem ser utilizados pelo compilador para realizar conversões implícitas.

11.15 Síntese

Neste capítulo, você aprendeu a construir classes mais robustas definindo operadores sobre carregados que permitem aos programadores usar operadores comuns para manipular os tipos de dados implementados. Sobrecarregamos os operadores básicos da classe `String` para permitir a manipulação de strings baseadas em ponteiros. Vimos como é possível implementar operadores sobre carregados como funções-membro ou funções globais. Discutimos as diferenças entre sobre carregar o unários e binários como funções-membro e funções globais. Com as funções globais, mostramos como permitir entrada e saída de nossas classes utilizando os operadores de extração e de inserção de fluxo sobre carregados, respectivamente. Mostramos uma classe especial que é requerida para diferenciar entre as versões prefixada e pós-fixada de operadores de sobre carregamento (por exemplo, `>>>`). Aprendemos a utilizar strings baseadas em ponteiros no estilo C. Por fim, você aprendeu a sobre carregar operadores para lidar com strings baseadas em ponteiros no estilo C++. No próximo capítulo, continuamos a discussão sobre classes introduzindo uma forma de reutilização de software chamada herança. Veremos que as classes freqüentemente compartilham atributos e comportamentos comuns. Nesses casos, é possível definir os atributos e comportamentos em uma classe comum e ‘herdar’ essas capacidades em novas definições de classe.

Resumo

- O C++ permite ao programador sobre carregar a maioria dos operadores para se tornarem sensíveis ao contexto em que são utilizados. O compilador gera o código apropriado com base no contexto (em particular, os tipos dos operandos).
- Muitos operadores do C++ podem ser sobre carregados para trabalhar com tipos definidos pelo usuário.
- Um exemplo de um operador sobre carregado construído que é utilizado para o operador de inserção de fluxo e como o operador de deslocamento de bits para a esquerda. De maneira similar, é sobre carregado; é utilizado como o operador de extração de fluxo e como o operador de bits de deslocamento para a direita. Esses dois operadores são sobre carregados na C++ Standard Library.
- A própria linguagem C++ sobre carrega operadores executam de modo diferente, dependendo de seu contexto na aritmética de inteiros, na aritmética de ponto flutuante e na aritmética de ponteiros.
- Os trabalhos realizados por operadores sobre carregados também podem ser realizados por chamadas de função, mas a notação de operadores é freqüentemente mais clara e mais familiar para programadores.

- Um operador é sobre carregado escrevendo uma definição de função global em que o nome da função é a palavra-chave `operator` seguida pelo símbolo do operador sendo sobre carregado.
- Quando operadores são sobre carregados como funções-membro, podem ser chamados sobre um objeto da classe e operar sobre esse objeto.
- Para utilizar um operador em objetos de classe, esse operador deve ser sobre carregado — com três exceções: o operador de atribuição (`=`) e o operador vírgula (`,`).
- Você não pode mudar a precedência e a associatividade de um operador sobre carregando-o.
- Você não pode mudar a ‘aridade’ de um operador (isto é, o número de operandos que um operador aceita).
- Você não pode criar novos operadores; somente os operadores existentes podem ser sobre carregados.
- Você não pode mudar o significado de como um operador funciona em objetos de tipos fundamentais.
- Sobre carregar um operador de atribuição e um operador de adição para uma classe não é permitido.
- Esse comportamento só pode ser alcançado sobre carregando explicitamente o operador.
- As funções-operadoras podem ser funções-membros ou globais; as funções-membro devem ter prefixo `operator`. As funções-membro utilizam implicitamente para obter um de seus argumentos de objeto de classe (o operando esquerdo para operadores binários). Os argumentos para ambos os operandos de um operador binário devem ser explicitamente listados em uma função global.
- Ao sobre carregar `>` ou qualquer um dos operadores de atribuição, a função de sobre carga de operadores deve ser declarada como membros de classe. Para os outros operadores, as funções de sobre carga de operadores podem ser membros de classe ou funções globais.
- Quando uma função operadora é implementada como uma função-membro, o operando na extrema esquerda (ou único) deve ser uma referência a um objeto da classe do operador.
- Se o operando esquerdo precisar ser um objeto de uma classe diferente ou um tipo fundamental, essa função operadora deve ser implementada como uma função global.
- Uma função operadora global pode ser tornada membro de uma classe se essa função precisar acessar membros dessa classe diretamente.
- O operador de inserção `<<` sobre carregado é utilizado em uma expressão em que o operando esquerdo tem o tipo `ostream`; essa razão, ele deve ser sobre carregado como uma função global. Para ser um `operator` de membro da classe `ostream` mas isso não é possível, uma vez que não temos permissão para modificar classes da C++ Standard Library. De maneira semelhante, o operador de extração `>>` sobre carregado deve ser uma função global.
- Outra razão para escolher uma função global para sobre carregar um operador é permitir que o operador seja comutativo.
- Quando utilizado com `string s.setw`, restringe o número de caracteres lidos ao número de caracteres especificados por seu argumento.
- A função-membro ignore descarta o número especificado de caracteres no fluxo de entrada (um caractere por padrão).
- Os operadores de entrada e saída sobre carregados só devem conter membros de classe diretamente por razões de desempenho.
- Um operador unário de uma classe pode ser sobre carregado como uma função-membro ou como uma função global com um argumento; esse argumento deve ser um objeto da classe ou uma referência a um objeto da classe.
- As funções-membro que implementam os operadores sobre carregados que possam acessar os dados não-em cada objeto da classe.
- Um operador binário é sobre carregado como uma função-membro com dois argumentos (um desses argumentos deve ser um objeto de classe ou uma referência a um objeto de classe).
- Um construtor de cópia inicializa um novo objeto de uma classe copiando os membros de um objeto existente dessa classe. Quando os objetos da classe contêm memória dinamicamente alocada, a classe deve fornecer um construtor de cópia para assegurar que cada cópia de um objeto tem sua própria cópia separada da memória dinamicamente alocada. Em geral, essa classe também forneceria um destrutor e um operador de atribuição sobre carregado.
- A implementação da função `operator=` deve testar a auto-atribuição, em que um objeto está sendo atribuído a ele mesmo.
- O compilador chama `operator[]` quando o operador de subscrito é utilizado em uma classe que não é const do operador quando ele é utilizado em `const`.
- O operador de subscrita (`[]`) está restrito ao uso com arrays. Ele pode ser utilizado para selecionar elementos de outros tipos de classes contêineres. Além disso, com a sobre carga, os valores de índice não precisam mais ser do tipo inteiro; os caracteres ou strings podem ser utilizados, por exemplo.
- O compilador não pode saber antecipadamente como converter entre tipos definidos pelo usuário, e entre tipos fundamentais, portanto o compilador deve especificar como fazer isso. Essas conversões podem ser realizadas com construtores de conversão — os construtores de um argumento que transformam objetos de outros tipos (incluindo tipos fundamentais) em objetos de uma classe particular.

- Um operador de conversão (também chamado de operador de coerção) pode ser utilizado para converter um objeto de uma classe em um de outra classe ou em um objeto de um tipo fundamental. Esse operador de conversão deve ser uma função-membro não-operadora de coerção sobrecarregadas podem ser definidas para converter objetos de tipos definidos pelo usuário em tipos fundamentais objetos de outros tipos definidos pelo usuário.
- Uma função operadora de coerção sobrecarregada não especifica um tipo de retorno — o tipo de retorno é o tipo no qual o objeto está sendo convertido.
- Uma das características interessantes dos operadores de coerção e construtores de conversão é que, quando necessário, o compilador pode usar essas funções implicitamente para criar objetos temporários.
- Qualquer construtor de um único argumento pode ser considerado um construtor de conversão.
- Sobre carregar o operador de chamada de função poderoso, porque as funções podem aceitar listas de parâmetros arbitrariamente longas e complexas.
- Os operadores de incremento e de decremento prefixado e pós-fixado podem ser inteiramente sobre carregados.
- Para sobre carregar o operador de incremento para permitir tanto o uso de pré-incremento como de pós-incremento, cada função operadora sobre carregada deve ter uma assinatura distinta, de modo que o compilador seja capaz de diferenciar entre as versões de prefixado e pós-fixado.
- A classe `string` padrão é definida no namespace `std`.
- A classe `string` fornece muitos operadores sobre carregados, incluindo operadores de igualdade, relacionais, de atribuição, de atribuição de adição (para concatenação) e de subscripto.
- A classe `string` fornece a função-membro `empty` que retorna `true` se a string estiver vazia; caso contrário, retorna `false`.
- A função-membro `substr` da classe `string` obtém uma substring de um comprimento especificado pelo segundo argumento, iniciando na posição especificada pelo primeiro argumento. Quando o segundo argumento não é especificado, é assumido que ela é chamada.
- O operador sobre carregado `operator<<` da classe `string` realiza nenhuma verificação de limites. Portanto, o programador deve assegurar que as operações que utilizam o operador sobre carregado da classe padrão não manipulam acidentalmente os elementos fora dos limites da string.
- A classe `string` padrão fornece a verificação de limites com a exceção 'argumento inválido' se seu argumento for um subscripto inválido. Por padrão, isso faz com que um programa C++ termine. Se o substituto `nothrow` é usado, a manipulação especificada pelo subscripto é realizada dependendo do contexto em que a chamada aparecer.
- O C++ fornece a palavra-chave `explicit` para suprimir conversões implícitas via construtores de conversão quando essas conversões não puderem ser permitidas. Um construtor `explicit` pode ser utilizado em uma conversão implícita.

Terminologia

| | | |
|---|--|--------------------------------------|
| 'aridade' de um operador | operador sobre carregado | operadores sobre carregáveis |
| Array, classe | operador sobre carregado | operator! |
| associatividade não alterada pela sobre carregada | operador sobre carregado | operator!= |
| auto-atribuição | operador sobre carregado | operator() |
| construtor de conversão | operador sobre carregado | operator, palavra-chave |
| construtor de cópia | operador(int) sobre carregado | operator[] |
| conversão definida pelo usuário | operador sobre carregado | operator+ |
| conversão entre tipos fundamentais e tipos de usuário | operador sobre carregado | operator++ |
| conversões implícitas definidas pelo usuário | operador sobre carregado | operator++(int) |
| empty função-membro | operador sobre carregado | operator< |
| explicit, construtor | operador sobre carregado | operator<< |
| função global para sobre carregar um operador | operador sobre carregado | operator= |
| função operadora | operador sobre carregado | operator== |
| função operadora de coerção | operador sobre carregado | operator>= |
| funções operadoras de atribuição | operador de atribuição sobre carregado | operator>> |
| ignore, função-membro | operador de chamada de função | sobre carga de operadores |
| lvalue& left value | operador de conversão | sobre carregando um operador binário |
| operação de comutatividade | operadores de inserção e extração | sobre carregando um operador unário |
| operador sobre carregado | sobre carregados | string (classe C++ padrão) |

| | | |
|--|---|----------------------------|
| string, concatenação
substr, função-membro de | substring
tipo definido pelo usuário | versão const de operator[] |
|--|---|----------------------------|

Exercícios de revisão

- 11.1 Preencha as lacunas em cada uma das seguintes sentenças:
 a) Suponha quejam variáveis do tipo inteiro e que formularmos a função quejam variáveis de ponto flutuante e formulamos a função dois operadores. Agora suponha quejam variáveis de ponto flutuante e formulamos a função dois operadores. Aqueles estão sendo claramente utilizados para propósitos diferentes. Esse é um exemplo de _____.
 b) A palavra-chave _____ introduz uma definição de função de operador sobrecarregado.
 c) Para utilizar os operadores em objetos de classe, eles devem ser sobrecarregados, com exceção dos operadores de _____.
 d) A _____, a _____ e a _____ de um operador não podem ser alteradas sobrecarregando o operador.
- 11.2 Explique os múltiplos significados dos operadores.
- 11.3 Em que contexto o operador / poderia ser utilizado em C++?
- 11.4 (Verdadeiro/falso) Em C++, apenas os operadores existentes podem ser sobrecarregados.
- 11.5 No que a precedência de um operador sobrecarregado em C++ é comparável com a precedência do operador <?

Respostas dos exercícios de revisão

- 11.1 a) sobrecarga de operador. b) atribuição (vírgula) c) precedência, associatividade, 'aridade'.
 11.2 O operador é tanto o operador de deslocamento para a direita como o operador de extração de fluxo, dependendo de seu contexto. O operador é tanto o operador de deslocamento para a esquerda como o operador de inserção de fluxo, dependendo de seu contexto.
 11.3 Para a sobrecarga de operadores: seria o nome de uma função que forneceria uma versão sobrecarregada do operador específico.
 11.4 Verdadeiro.
 11.5 A precedência é idêntica.

Exercícios

- 11.6 Forneça quantos exemplos puder de sobrecarga de operadores implícita em C++. Dê um exemplo razoável de uma situação em que poderia querer sobrecarregar explicitamente um operador em C++.
- 11.7 Os operadores que não podem ser sobrecarregados são _____, _____, _____ e _____.
- 11.8 A concatenação de string requer dois operandos — as duas strings que devem ser concatenadas. No texto, mostramos como implementar um operador de concatenação sobrecarregado que conseguia concatenar todo primeiro objeto modificando, assim, o próprio tipo. Em alguns aplicativos, é desejável permitir concatenar sem modificar os argumentos. Implemente o operador + para permitir operações como
`string1 = string2 + string3;`
- 11.9 (Exercício final de sobrecarga de operadores) Faça o cuidado que deve haver na seleção de operadores para sobrecarga, liste cada um dos operadores sobrecarregáveis do C++ e, para cada um deles, liste um possível significado (ou vários, se apropriado) para cada de várias classes que você estudou neste texto. Sugerimos que você tente:
 a) Array
 b) Stack
 c) String
- Depois de fazer isso, comente quais operadores parecem ter significado para uma ampla variedade de classes. Que operadores parecem valor para a sobrecarga? Que operadores parecem ambíguos? Liste cada um dos operadores sobrecarregáveis do C++. Para cada um, liste o que você acredita que poderia ser a 'melhor operação' que o operador deve ser utilizado para representar. Se houver operações excelentes, liste todas elas.
- 11.10 Agora que já sabe o processo de sobrecarga, liste cada um dos operadores sobrecarregáveis do C++. Para cada um, liste o que você acredita que poderia ser a 'melhor operação' que o operador deve ser utilizado para representar. Se houver operações excelentes, liste todas elas.
- 11.11 Um exemplo interessante de como sobrecarregar o operador é a chamada de função popular de subscrito de array duplo em algumas linguagens de programação. Em vez de dizer
`chessBoard[row][column]`
- para um array de objetos, sobrecarregue o operador de chamada de função para permitir a forma alternativa

```
chessBoard( row, column )
```

Crie uma classe `DoubleSubscriptedArray` que tem recursos semelhantes às classes 11.6–11.7. Em tempo de construção, a classe deve ser capaz de criar um array com qualquer número de linhas e qualquer número de colunas. A classe deve fornecer o operador() para realizar as operações de subscrito duplo. ~~Por exemplo, para array por 5 changeado usuário poderia escrever~~ para acessar o elemento `pena[0][3]`. Lembre-se de que o operador() pode receber qualquer número de argumentos. ~~Sua implementação deve ser similar à das classes 11.9–11.10 para obter um exemplo de representação subjacente do array de subscrito duplo deve ser um array de um único subscrito *de volta* com o número de elementos. A função operator() deve realizar a aritmética de ponteiro adequada para acessar cada elemento do array.~~ Há duas versões de operator() — uma que retorna `int` (de modo que um elemento de `DoubleSubscriptedArray` possa ser utilizado como um valor) e uma que retorna `int &` (de modo que um elemento de `DoubleSubscriptedArray` possa ser utilizado somente como uma saída). A classe também deve fornecer os seguintes operadores: `<`, `>`, `<=`, `>=`, `==` e `!=`. O operador `<<` deve gerar saída do array no formato tabular (para inserir o conteúdo inteiro do array).

11.12 Sobrecarregue o operador de subscrito para retornar o maior elemento de uma coleção, o segundo maior, o terceiro maior e assim por diante.

11.13 Considere a classe `Complex` mostrada nas figuras 11.19–11.21. A classe permite operações com números complexos. Considere a classe `Complex` mostrada nas figuras 11.19–11.21. A classe permite operações com números complexos.

 $\sqrt{-1}$

- Modifique a classe para permitir a entrada e saída de números complexos pelo operador de inserção e extração (`<<` e `>>`). (você deve remover `print` da classe).
- Sobrecarregue o operador de multiplicação para permitir multiplicação de dois números complexos como em álgebra.
- Sobrecarregue os operadores para permitir comparações de números complexos.

11.14 Uma máquina com inteiros de 32 bits pode representar inteiros no intervalo de aproximadamente -2 bilhões a +2 bilhões. Essa restrição de tamanho fixo raramente causa problemas, mas há aplicativos em que gostaríamos de ser capazes de utilizar um espectro mais amplo de inteiros. É para isso que o C++ foi construído, a saber, para criar novos tipos de dados. Considere a classe `Complex` mostrada nas figuras 11.19–11.21. Estude a classe cuidadosamente, então resolva as seguintes questões:

- Descreva precisamente como ela opera.
- Que restrições a classe tem?
- Sobrecarregue o operador de multiplicação
- Sobrecarregue o operador de divisão
- Sobrecarregue todos os operadores de igualdade e relacionais.

[Nota: Não mostramos um operador de atribuição nem um construtor de cópia para a classe `Complex`. O construtor de cópia fornecidos pelo compilador são capazes de copiar todo o membro de dados do array adequadamente.]

```

1 // Figura 11.19: Complex.h
2 // Definição da classe Complex.
3 #ifndef COMPLEX_H
4 #define COMPLEX_H
5
6 class Complex
7 {
8 public :
9 Complex( double = 0.0, double = 0.0 ); // construtor
10 Complex operator +( const Complex & ) const; // adição
11 Complex operator -( const Complex & ) const; // subtração
12
13 private : print() const; // saída
14     double real; // parte real
15     double imaginary; // parte imaginária
16 }; // fim da classe Complex
17
18 #endif

```

Figura 11.19 Definição da classe `Complex`

```

1 // Figura 11.20: Complex.cpp
2 // Definições de função-membro da classe Complex.
3 #include <iostream>
4 using std::cout;
5
6 #include "Complex.h" // definição da classe Complex
7
8 // Construtor
9 Complex::Complex(double realPart, double imaginaryPart )
10    : real(realPart),
11      imaginary(imaginaryPart)
12 {
13 } // corpo vazio
14 // fim do construtor Complex
15
16 // operador de adição
17 Complex Complex::operator +( const Complex &operand2 )const
18 {
19     return Complex( real + operand2.real,
20                     imaginary + operand2.imaginary );
21 } // fim da função operator+
22
23 // operador de subtração
24 Complex Complex::operator -( const Complex &operand2 )const
25 {
26     return Complex( real - operand2.real,
27                     imaginary - operand2.imaginary );
28 } // fim da função operator-
29
30 // exibe um objeto Complex na fórmula: (a, b)
31
32 void Complex::print() const
33 {
34     cout << "(" << real << ", " << imaginary << ")";
35 } // fim da função print

```

Figura 11.20 Definições de função-membro da classe Complex

```

1 // Figura 11.21: fig11_21.cpp
2 // Programa de teste da classe Complex.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Complex.h"
8
9 int main()
10 {
11     Complex x;
12     Complex y( 4.3, 8.2 );
13     Complex z( 3.3, 1.1 );
14
15     cout << "x: ";
16     x.print();

```

Números complexos \$1.21

(continua)

```

17     cout << "\ny: " ;
18     y.print();
19     cout << "\nz: " ;
20     z.print();
21
22     x = y + z;
23     cout << "\n\nx = y + z:" << endl;
24     x.print();
25     cout << " = " ;
26     y.print();
27     cout << " + " ;
28     z.print();
29
30     x = y - z;
31     cout << "\n\nx = y - z:" << endl;
32     x.print();
33     cout << " = " ;
34     y.print();
35     cout << " - " ;
36     z.print();
37     cout << endl;
38     return 0;
39 } // fim de main

```

x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

$x = y + z:$
 $(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)$

$x = y - z:$
 $(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)$

Números complexos.21

(continuação)

```

1 // Figura 11.22: HugeInt.h
2 // Definição da classe HugeInt.
3 #ifndef HUGEINT_H
4 #define HUGEINT_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class HugeInt
10 {
11 public: ostream &operator <<( ostream &, const HugeInt & );
12 HugeInt( long = 0 ); // construtor de conversão/padrão
13 HugeInt( const char * ); // construtor de conversão
14
15 // operador de adição; HugeInt + HugeInt
16 HugeInt operator +( const HugeInt & ) const;

```

Figura 11.22 Definição da classe HugeInt

(continua)

```

18     // operador de adição; HugeInt + int
19     HugeInt operator +( int ) const;
20
21     // operador de adição;
22     // HugeInt + string que representa o valor de tipo inteiro grande
23     HugeInt operator +( const char * ) const;
24     private :
25         short integer[ 30];
26     }; // fim da classe HugeInt
27
28 #endif

```

Figura 11.22 Definição da classe HugeInt

(continuação)

```

1 // Figura 11.23: Hugeint.cpp
2 // Definições de função-membro e função friend HugeInt.
3 #include <cctype> // protótipo de função isdigit
4 #include <cstring> // protótipo de função strlen
5 #include "Hugeint.h" // definição da classe HugeInt
6
7 // construtor-padrão; construtor de conversão que
8 // converte um inteiro long em um objeto HugeInt
9 HugeInt::HugeInt( long value )
10 {
11     // inicializa o array como zero
12     for ( int i = 0; i <= 29; i++ )
13         integer[ i ] = 0;
14
15     // coloca dígitos de argumento no array
16     for ( int j = 29; value != 0 && j >= 0; j-- )
17     {
18         integer[ j ] = value % 10;
19         value /= 10;
20     } // fim do for
21 } // fim do construtor-padrão/de conversão HugeInt
22
23 // construtor de conversão que converte uma string de caractere
24 // representando um inteiro grande em um objeto HugeInt
25 HugeInt::HugeInt( const char *string )
26 {
27     // inicializa o array como zero
28     for ( int i = 0; i <= 29; i++ )
29         integer[ i ] = 0;
30
31     // coloca os dígitos de argumento no array
32     int length = strlen( string );
33     for ( int j = 30 - length, k = 0; j <= 29; j++, k++ )
34     {
35         if ( isdigit( string[ k ] ) )
36             integer[ j ] = string[ k ] - '0';
37     } // fim do construtor de conversão HugeInt
38

```

Figura 11.23 Definições de função-membro e função friend da classe HugeInt

(continua)

```

39 // operador de adição; HugeInt + HugeInt
40 HugeInt HugeInt:: operator +( const HugeInt &op2 ) const
41 {
42     HugeInt temp; // resultado temporário
43     int carry = 0;
44
45     for ( int i = 29; i >= 0; i-- )
46     {
47         temp.integer[i] =
48             integer[i] + op2.integer[i] + carry;
49
50         if ( temp.integer[i] > 9 )
51         {
52             temp.integer[i] %= 10; // reduz a 0-9
53             carry = 1;
54         }
55         else // não transporta
56             carry = 0;
57     } // fim do for
58
59     return temp; // retorno da cópia do objeto temporário
60 } // fim da função operator+
61
62 // operador de adição; HugeInt + int
63 HugeInt HugeInt:: operator +( int op2 ) const
64 {
65     // converte op2 em um HugeInt, então invoca
66     // operator+ para dois objetos HugeInt
67
68     // retorna *this + HugeInt( op2 );
69 } // fim da função operator+
70
71 // operador de adição;
72 // HugeInt + string que representa o valor de tipo inteiro grande
73 HugeInt HugeInt:: operator +( const char *op2 ) const
74 {
75     // converte op2 em um HugeInt, então invoca
76     // operator+ para dois objetos HugeInt
77     return * this + HugeInt( op2 );
78 } // fim de operator+
79
80 // operador de saída sobrecarregado
81 ostream& operator<<( ostream &output, const HugeInt &num )
82 {
83     int i;
84
85     for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
86     ; // pula zeros à esquerda
87
88     if ( i == 30 )
89         output << 0;
90     else
91
92         for ( ; i <= 29; i++ )
93             output << num.integer[i];
94

```

Figura 11.23 Definições de função-membro e função-função da classe HugeInt

(continua)

```
95  
96     return output;  
97 } // fim da função operator<<
```

Figura 11.23 Definições de função-membro e função end da classe Hugelnt

(continuação)

$$7654321 + 7891234 = 15545555$$

$$7654321 + 9 = 7654330$$

$$7891234 + 10000 = 7901234$$

Figura 11.24 Inteiros muito grandes.

- 11.15 Crie uma classe `Fraction` com as seguintes capacidades:
- Crie um construtor que impede um denominador 0 em uma fração, reduz ou simplifica as frações que não estão na forma reduzida que evita denominadores negativos.
 - Sobrecregue os operadores de adição, subtração, multiplicação e divisão dessa classe.
 - Sobrecregue os operadores relacionais e de igualdade dessa classe.
- 11.16 Estude as funções da biblioteca de tratamento de strings do C e implemente cada uma das funções 11.9–11.10) como parte da classe `String` (Figura 11.10). Então, utilize essas funções para realizar manipulações de texto.
- 11.17 Desenvolva a classe `Polynomial`. A representação interna de `Polynomial` é um array de termos. Cada termo contém um coeficiente e um expoente. O termo $2x^4$ tem o coeficiente 2 e o expoente 4. Desenvolva uma classe completa contendo funções construtoras e destrutoras adequadas, bem como as sobrecargas de operadores de matemática para polinômios. Implemente as seguintes capacidades de operador sobreporregado:
- Sobrecregar o operador de subtração para polinômios.
 - Sobrecregar o operador de atribuição para polinômios.
 - Sobrecregar o operador de multiplicação para polinômios.
 - Sobrecregar o operador de multiplicação de atribuição de subtração.
 - Sobrecregar o operador de multiplicação de atribuição de subtração de multiplicação.
- 11.18 No programa das Figuras 11.3–11.5, a Figura 11.4 contém o comentário `// insertion operator; cannot be a member function if we would like to invoke it with cout << somePhoneNumber;` [operador de inserção de fluxo sobreporregado; não pode ser uma função-membro se quisermos invocá-lo com `cout << somePhoneNumber`]. De fato, o operador de inserção de fluxo poderia ser uma função-membro nestes mesmos dispositivos a invocá-lo `somePhoneNumber.operator<<(cout)` ou `somePhoneNumber << cout`. Reescreva o programa da Figura 11.5 com a inserção de fluxo sobreporregado como uma função-membro e experimente as duas instruções precedentes no programa para demonstrar que elas funcionam.

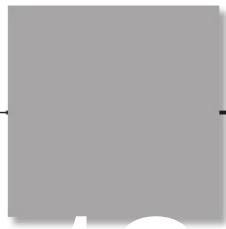


Nunca diga que você conhece inteiramente uma pessoa, até dividir uma herança com ela.
Johann Kaspar Lavater

Esse método é para definir como o número de uma classe, a classe de todas as classes similares à classe fornecida.
Bertrand Russell

Melhor que herdar uma biblioteca, é colecionar uma.
Augustine Birrell

Sempre foi desprezível o lucro que nos vem de pergaminho.
William Shakespeare



Programação orientada a objetos: herança

OBJETIVOS

Neste capítulo, você aprenderá:

- A criar classes herdando de classes existentes.
- Como a herança promove a reutilização de software.
- As noções de classes básicas e classes derivadas e os relacionamentos entre elas.
- O especificador de acesso de `private`.
- O uso de construtores e destrutores em hierarquias de herança.
- As diferenças entre heranças `public`, `protected` e `private`.
- O uso de herança para personalizar software existente.

| | |
|--------|--|
| 12.1 | Introdução |
| 12.2 | Classes básicas e derivadas |
| 12.3 | Membros protected |
| 12.4 | Relacionamento entre classes básicas e derivadas |
| 12.4.1 | Criando e utilizando uma classe CommissionEmployee |
| 12.4.2 | Criando uma classe BasePlusCommissionEmployee para utilizar herança |
| 12.4.3 | Criando uma hierarquia de herança CommissionEmployee–BasePlusCommissionEmployee |
| 12.4.4 | Hierarquia de herança CommissionEmployee–BasePlusCommissionEmployee utilizando dados protected |
| 12.4.5 | Hierarquia de herança CommissionEmployee–BasePlusCommissionEmployee utilizando dados private |
| 12.5 | Construtores e destrutores em classes derivadas |
| 12.6 | Herança public, protected e private |
| 12.7 | Engenharia de software com herança |
| 12.8 | Síntese |

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

12.1 Introdução

Este capítulo continua nossa discussão de programação orientada a objetos (POO) introduzindo outro de seus recursos-chave. A herança é uma forma de reutilização de software em que o programador cria uma classe que absorve dados e comportamentos de uma classe existente e os aprimora com novas capacidades. A reusabilidade de software economiza tempo e recursos durante o desenvolvimento de programa. Ela também encoraja a reutilização de softwares de alta qualidade já testados e deixa aumentar a probabilidade de um sistema ser eficientemente implementado.

Ao criar uma classe, em vez de escrever membros de dados e funções-membro completamente novos, o programador pode garantir que a nova classe deve herdar membros de uma classe existente. Essa classe existente é chamada de classe base ou classe derivada. (Outras linguagens de programação, como Java, referem-se à classe base como superclasse.)

Herança é uma forma de herdar comportamentos de uma classe base de forma direta. Em geral, uma classe derivada pode personalizar comportamentos herdados da classe básica. Ela herda explicitamente uma classe base de dois ou mais níveis na classe. No caso de herança simples, uma classe é derivada de uma classe básica. O C++ também suporta herança múltipla (possivelmente não relacionadas) classes básicas. A herança simples é simples e direta — mostramos vários que devem permitir ao leitor tornar-se proficiente rapidamente. A herança múltipla pode ser complexa e propensa a erros. Discutimos herança múltipla no Capítulo 24, “Outros tópicos”.

O C++ oferece três tipos de herança: public, protected e private. Neste capítulo, concentraremos-nos na herança pública. Explicamos brevemente as outras duas. No Capítulo 21, “Estruturas de dados”, podemos discutir a herança privada como uma alternativa à composição. A terceira forma é herança protegida, utilizada. Compartilhamento de objeto de uma classe derivada também é um objeto de uma classe básica dessa classe derivada. Mas os objetos de classe básica não são suas classes derivadas. Por exemplo, se tivermos veículo como uma classe básica e um carro como uma classe derivada, então carros são veículos, mas nem todos os veículos são carros. À medida que continuamos nosso estudo de programação orientada a objetos nos capítulos 12 e 13, tiramos proveito desse relacionamento para realizar algumas manipulações interessantes.

A experiência na criação de sistemas de software indica que quantidades significativas de código lidam com casos especialmente relacionados. Quando os programadores estão preocupados com casos especiais, os detalhes podem obscurecer a visão geral. Com a programação orientada a objetos, os programadores se concentram nos aspectos comuns entre objetos no sistema em vez de se preocuparem com os casos especiais.

As classes derivadas podem herdar tanto membros quanto comportamentos de suas classes bases. Um membro de uma classe derivada pode ser tratado como um membro de sua classe base — por exemplo, se tivermos carro como uma classe derivada e veículo como uma classe base, então carro é um tipo de veículo. Um comportamento de uma classe derivada pode ser tratado como um comportamento de sua classe base — por exemplo, se tivermos carro como uma classe derivada e veículo como uma classe base, então carro tem comportamentos de veículo.

As funções-membro da classe derivada talvez requerem acesso a membros de dados das classes básicas e funções-membro da classe derivada pode acessar os membros de sua classe básica. Os membros da classe básica que não devem ser acessíveis às funções-membro das classes derivadas devem ser declarados private. Uma classe derivada pode produzir alterações

de estado em membros da classe básica, somente por funções-membro fornecidas na classe básica e herdadas na classe derivada.



Observação de engenharia de software 12.1

As funções-membro de uma classe derivada não podem acessar membros da classe básica.



Observação de engenharia de software 12.2

Se uma classe derivada pudesse acessar membros da classe básica, as classes que herdam dessa classe derivada também poderiam acessar esses dados. Isso propagaria acesso ao que devem ser mantidos em ocultamento de informações seriam perdidos.

Um problema com a herança é que uma classe derivada pode herdar membros de dados e funções-membro de que ela não deve ter. É responsabilidade do programista da classe assinar quais capacidades fornecidas por uma classe sejam aplicáveis às classes derivadas. Nesse caso, uma função-membro da classe básica apropriada para uma classe derivada, aí frequentemente requer que a função-membro se comporte de maneira específica à classe derivada. Nesses casos, a função-membro da classe básica pode ser redefinida na classe derivada com uma implementação apropriada.

12.2 Classes básicas e derivadas

Freqüentemente, um objeto de uma classe pertence a outra classe. Por exemplo, em geometria, um retângulo é um tipo de quadrilátero (assim como o são os quadrados, paralelogramos e trapezóides). Portanto, é correto dizer que a classe da classe quadrilátero. Nesse contexto, o quadrilátero é uma classe básica, e o retângulo é uma classe derivada. Um retângulo é um tipo específico de quadrilátero, mas é incorreto afirmar que um quadrilátero é um retângulo — o quadrilátero poderia ser um paralelogramo ou alguma outra forma. A Figura 12.1 lista vários exemplos simples de classes básicas e classes derivadas.

Como cada objeto de classe pertence sua classe básica e uma classe básica pode ter muitas classes derivadas, o conjunto de objetos representado por uma classe básica é, em geral, maior que o conjunto de objetos representados por qualquer de suas classes derivadas. Por exemplo, a classe Veículo representa todos os veículos, incluindo carros, caminhões, barcos, aviões, bicicletas e assim por diante. Em comparação, a classe carro é um subconjunto menor mais específico de todos os veículos.

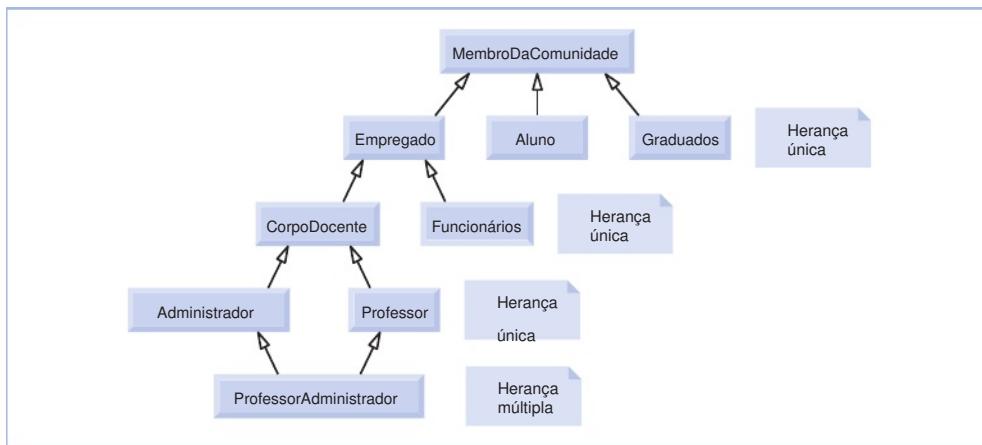
Os relacionamentos de herança formam estruturas hierárquicas do tipo árvore. Uma classe básica existe em um relacionamento hierárquico com suas classes derivadas. Embora as classes possam existir independentemente, depois que são empregadas namentos de herança, elas se tornam afiliadas de outras classes. Uma classe torna-se uma classe básica — fornecendo meio a outras classes, ou uma classe derivada — herdando seus membros de outras classes, ou ambas.

Vamos desenvolver uma hierarquia de herança simples com cinco níveis (representados pelo diagrama de classe UML na 12.2). A comunidade de uma universidade tem milhares de membros. Esses membros consistem em empregados, alunos e professores. Os empregados incluem os membros do corpo docente e os funcionários operacionais. Os membros do corpo docente são administradores (como diretores e chefes de departamento) ou professores. Alguns administradores, porém, também são professores. Utilizamos a herança múltipla para fornecer a classe MembroDaComunidade. Observe também que essa hierarquia de herança poderia conter muitas outras classes. Por exemplo, alunos podem ser graduados ou graduandos. Os alunos graduandos podem ser primários, secundaristas, terceiristas e quartanistas.

Cada seta na hierarquia (Figura 12.2) representa um relacionamento de herança. Quando seguimos as setas nessa hierarquia de classes, podemos declarar que um MembroDaComunidade é um professor é um membro do corpo docente. MembroDaComunidade é uma classe básica direta de Empregado, Aluno e Graduado. Além disso, MembroDaComunidade é uma classe básica indireta de todas as outras classes no diagrama. Iniciando da parte inferior do diagrama, o leitor pode seguir as setas de relacionamento para a classe básica mais alta. Por exemplo, um administrador é um professor, é um membro do corpo docente e é um membro da comunidade.

| Classe básica | Classes derivadas |
|---------------|--|
| Aluno | AlunoDeGraduação, AlunoDePósGraduação |
| Forma | Círculo, Triângulo, Retângulo, Esfera, Cubo |
| Financiamento | FinanciamentoDeCarro, FinanciamentoDeReformaDeCasa, FinanciamentoDeCasaPrópria |
| Empregado | CorpoDocente, Funcionários |
| Conta | ContaCorrente, ContaPoupança |

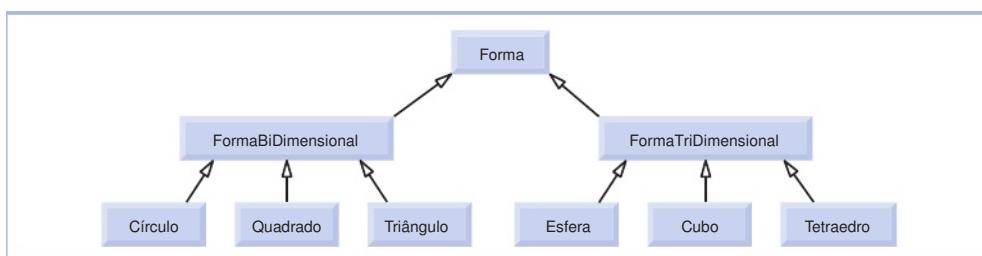
Figura 12.1 Exemplos de herança.

Figura 12.2 Hierarquia de herança de `MembroDaComunidade` da universidade.

Agora considere a hierarquia de herança na Figura 12.3. Essa hierarquia começa com a classe básica `Forma`, que deriva de `FormaBiDimensional` e `FormaTriDimensional`. Ambas as classes derivadas herdam membros da classe básica. A classe `FormaBiDimensional` herda membros da classe básica e adiciona membros próprios. A classe `FormaTriDimensional` herda membros da classe básica e adiciona membros próprios. Como na Figura 12.2, você pode seguir as setas da parte inferior do diagrama para a classe básica superior nessa hierarquia para identificar vários relacionamentos. Por exemplo, um `Triângulo` é uma `FormaBiDimensional` e uma `Forma`, enquanto uma `Esfera` é uma `FormaTriDimensional` e uma `Forma`. Observe que essa hierarquia poderia conter muitas outras classes, como `Elipse`, `Reta`, `Trapezóide`, os quais são `FormaBiDimensional`. Especificar que a classe `Forma` (Figura 12.3) é derivada da (ou herda de) `FormaBiDimensional` poderia ser definida em C++ como segue:

Esse é um exemplo de `public`, a forma mais comumente utilizada. Também discutiremos `protected` (Seção 12.6). Com todas as formas de herança, os membros da classe básica não são acessíveis diretamente de classes derivadas dessa classe, mas esses membros são considerados herdados (isto é, ainda são considerados partes das classes derivadas). Comparemos os outros membros de classe básica retêm seu acesso `private` de membro quando se tornam membros da classe derivada (por exemplo, os membros da classe básica tornam-se membros da classe derivada, e, como veremos em breve, os membros da classe básica tornam-se membros da classe derivada). Por meio desses membros herdados da classe básica, a classe derivada pode adicionar novos membros. Esses membros herdados fornecem tal funcionalidade na classe básica.

A herança não é apropriada a todos os relacionamentos de classe. No Capítulo 10, discutimos o relacionamento classes têm membros que são objetos de outras classes. Esses relacionamentos criam classes compõendo classes existentes. Pode ser incorreto dizer que `Employee` tem `BirthDate` ou que `Employee`

Figura 12.3 Hierarquia de herança para `Forma`.

é um `telephoneNumber`. Entretanto, é apropriado dizer que `Employee` tem um `telephoneNumber`.

E possível tratar os objetos da classe básica e os objetos da classe derivada de modo semelhante; seus aspectos comuns são nos membros da classe básica. Os objetos de todas as classes derivadas de uma classe básica comum podem ser tratados como dessa classe básica (isto é, esses objetos têm `telephoneNumber`). No Capítulo 13, “Programação orientada a objetos: polimorfismo”, consideraremos muitos exemplos que tiram proveito desse relacionamento.

12.3 Membros protected

O Capítulo 3 introduziu os especificadores de acesso. Os membros `protected` de uma classe básica são acessíveis a partir de dentro do corpo dessa classe básica e de qualquer lugar (`protected` é programmaticamente equivalente a um ponteiro) para um objeto dessa classe básica ou uma de suas classes derivadas. Se `protected` são acessíveis somente dentro do corpo dessa classe básica.

`protected` oferece um nível intermediário de proteção entre os membros `private` de uma classe básica e os membros `public` de uma classe básica. Os membros `protected` de uma classe básica podem ser acessados a partir de dentro do corpo dessa classe básica, por exemplo, por membros `protected` de qualquer classe derivada dessa classe básica.

As funções-membro de classe derivada podem referenciar membros de classe básica simplesmente utilizando os nomes de membro. Quando uma função-membro de classe derivada redefine uma função-membro de uma classe básica, o nome da classe básica pode ser acessado a partir da classe derivada precedendo o nome do membro da classe básica com o nome da classe básica e o operador de resolução de escopo `base::` (acesso a membros redefinidos da classe básica na Seção 12.4 e o uso de `dynamic_cast` na Seção 12.4.4).

12.4 Relacionamento entre classes básicas e derivadas

Nesta seção, utilizamos uma hierarquia de herança contendo tipos de empregados no aplicativo de folha de pagamento de um para discutir o relacionamento entre uma classe básica e uma derivada. Os empregados comissionados (que serão representados por objetos de uma classe básica) são pagos com uma porcentagem de suas vendas, enquanto os empregados comissionados com base (que serão representados como objetos de uma classe derivada) recebem um salário-base mais uma porcentagem de suas vendas. Dividimos nossa discussão do relacionamento entre empregados comissionados e empregados comissionados com salário-base em uma série de cinco exemplos cuidadosamente elaborados passo a passo:

1. No primeiro exemplo, criamos a classe `Employee`, que contém como membros `protected` dados de sobrenome, número de seguro social, taxa de comissão (porcentagem) e quantidade de vendas brutas (isto é, total).
2. O segundo exemplo define a classe `CommissionEmployee`, que contém como membros `protected`, sobrenome, número de seguro social, taxa de comissão, quantidade de vendas brutas e salário-base. Criamos a última linha de código que a classe exige — logo veremos que é muito mais eficiente criar essa classe simplesmente herdando da classe `CommissionEmployee`.
3. O terceiro exemplo define uma nova versão da classe `CommissionEmployee`, que herda diretamente da classe `CommissionEmployee` (isto é, `BasePlusCommissionEmployee` herda da classe `CommissionEmployee` que também tem um salário-base) e tenta acessar membros da classe `CommissionEmployee`; isso resulta em erros de compilação, porque a classe derivada não tem acesso a membros `protected` da classe básica.
4. O quarto exemplo mostra que se os dados de dados forem declarados `protected`, uma nova versão da classe `BasePlusCommissionEmployee` herda da classe `CommissionEmployee` e pode acessar esses dados diretamente. Para esse propósito, definimos uma nova versão da classe `protected`. Ambas as classes `BasePlusCommissionEmployee` e `CommissionEmployee` herdam e a não herdada, contêm funcionalidades idênticas, mas mostramos como a versão de `BasePlusCommissionEmployee` é mais fácil de criar e gerenciar.
5. Depois de discutirmos a conveniência de utilizar os membros `protected`, o quinto exemplo, que configura os membros `protected` da classe `CommissionEmployee` de volta para impor boa engenharia de software. Esse exemplo demonstra que a classe `BasePlusCommissionEmployee` pode ser substituída por `CommissionEmployee` e vice-versa.

12.4.1 Criando e utilizando uma classe `CommissionEmployee`

Vamos primeiro examinar a definição da classe `CommissionEmployee` (Figuras 12.4–12.5). O arquivo de cabeçalho (`Figure 12.4`) especifica os serviços da classe `CommissionEmployee`, que incluem um construtor (linhas 12–13) e as funções-membro `earnings` (linha 30) e `setEarnings` (linha 31). As linhas 15–28 declaram funções para manipular os membros de dados da classe (declarados nas linhas 22–27). O arquivo de cabeçalho `CommissionEmployee.h` especifica cada um desses membros de dados como objetos de outras classes

```

1 // Figura 12.4: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public :
12 CommissionEmployee( const string &, const string &, const string &,
13 double = 0.0, double = 0.0 );
14
15 void setFirstName( const string & ); // configura o nome
16 string getFirstName() const; // retorna o nome
17
18 void setLastName(const string &); // configura o sobrenome
19 string getLastName() const; // retorna o sobrenome
20
21 void setSocialSecurityNumber( const string & ); // configura o SSN
22 string getSocialSecurityNumber() const; // retorna o SSN
23
24 void setGrossSales( double ); // configura a quantidade de vendas brutas
25 double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27 void setCommissionRate(double ); // configura a taxa de comissão (porcentagem)
28 double getCommissionRate()const; // retorna a taxa de comissão
29
30 double earnings() const; // calcula os rendimentos
31 void print() const; // imprime o objeto CommissionEmployee
32 private :
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 12.4 Arquivo de cabeçalho da classe CommissionEmployee

não podem acessar diretamente esses dados. Declarar membros de classe como `private` para manipular e validar os membros de dados ajuda a impor boa engenharia de software (definições de membro). As funções `setCommissionRate` definida nas linhas 69–72 da Figura 12.4 e `getCommissionRate` definida nas linhas 69–72 da Figura 12.5, por exemplo, validam seus argumentos antes de atribuir os valores aos membros de dados `commissionRate` e `grossSales`, respectivamente.

A definição do construtor `CommissionEmployee` não utiliza proposidadamente a sintaxe de inicializador de membro nos primeiros vários exemplos desta seção, de modo que podemos demonstrar como proteger os membros de dados de acesso de membro em classe derivada. Como mostrado na Figura 12.5, linhas 13–15, atribuindo valores `private` aos membros de dados `lastName` e `socialSecurityNumber` no corpo do construtor. Mais tarde nesta seção, retornaremos ao uso de listas de inicializadores de membro nos construtores.

Observe que não validamos os valores dos argumentos de constante de atribuí-los aos membros de dados correspondentes. Certamente poderíamos validar o nome e o sobrenome — talvez assegurando que eles tenham um comprimento razoável. De maneira semelhante, um SSN poderia ser validado para assegurar que ele contém nove dígitos, com ou sem traços, exemplo: 123-45-6789 ou 123456789.

A função-membro `getGrossSales` (linhas 81–84) calcula os rendimentos de `Employee`. A linha 83 multiplica o `commissionRate` por `grossSales` e retorna o resultado. A função-membro (linhas 87–93) exibe os valores dos membros de dados de um objeto `CommissionEmployee`.

A Figura 12.6 testa a classe `CommissionEmployee`. As linhas 16–17 instanciam o objeto classe `CommissionEmployee` e invocam o construtor para inicializar o objeto com o nome "Sobrenome", 22-2222 como o número de seguro social, a quantidade de vendas brutas e a taxa de comissão. As linhas 23–29 utilizam as funções-membro para exibir os valores de seus membros de dados. As linhas 31–32 invocam as funções-membro `setGrossSales` e `setCommissionRate` para alterar os valores dos membros de dados respectivamente. A linha 36 então chama a função-membro para gerar saída das informações sobre `Employee` atualizadas. Por fim, a linha 39 exibe os rendimentos calculados pela função-membro `getGrossSales` utilizando os valores atualizados dos membros de dados.

```

1 // Figura 12.5: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // Definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first,    const string &last,    const string &ssn,
11     double sales,    double rate )
12 {
13     firstName = first;    // deve validar
14     lastName = last;    // deve validar
15     socialSecurityNumber = ssn; // deve validar
16     setGrossSales( sales ); // valida e armazena as vendas brutas
17     setCommissionRate( rate ); // valida e armazena a taxa de comissão
18 } // fim do construtor CommissionEmployee
19 // configura o nome
20 void CommissionEmployee::setFirstName(const string &first )
21 {
22     firstName = first;    // deve validar
23 } // fim da função setFirstName
24
25 // retorna o nome
26 string CommissionEmployee::getFirstName() const
27 {
28     return firstName;
29 } // fim da função getFirstName
30
31 // configura o sobrenome
32 void CommissionEmployee::setLastName(const string &last )
33 {
34     lastName = last;    // deve validar
35 } // fim da função setLastName
36
37 // retorna o sobrenome
38 string CommissionEmployee::getLastName()const
39 {
40     return lastName;
41 } // fim da função getLastname
42 }
```

Figura 12.5 O arquivo de implementação para a classe `CommissionEmployee` representa um empregado que ganha uma porcentagem das vendas brutas. (continua)

```

43 // configura o SSN
44 void CommissionEmployee::setSocialSecurityNumber(inst string &ssn )
45 {
46     socialSecurityNumber = ssn; // deve validar
47 } // fim da função setSocialSecurityNumber
48
49 // retorna o SSN
50 string CommissionEmployee::getSocialSecurityNumber()const
51 {
52     return socialSecurityNumber;
53 } // fim da função getSocialSecurityNumber
54
55 // configura a quantidade de vendas brutas
56 void CommissionEmployee::setGrossSales(double sales )
57 {
58     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
59 } // fim da função setGrossSales
60
61 // retorna a quantidade de vendas brutas
62 double CommissionEmployee::getGrossSales(inst)
63 {
64     return grossSales;
65 } // fim da função getGrossSales
66
67 // configura a taxa de comissão
68 void CommissionEmployee::setCommissionRate(double rate )
69 {
70     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
71 } // fim da função setCommissionRate
72
73 // retorna a taxa de comissão
74 double CommissionEmployee::getCommissionRate(inst)
75 {
76     return commissionRate;
77 } // fim da função getCommissionRate
78
79 // calcula os rendimentos
80 double CommissionEmployee::earnings()const
81 {
82     return commissionRate * grossSales;
83 } // fim da função earnings
84
85
86 // imprime o objeto CommissionEmployee
87 void CommissionEmployee::print()const
88 {
89     cout << "commission employee: " << firstName << ' ' << lastName
90     << "\nsocial security number: " << socialSecurityNumber
91     << "\ngross sales: " << grossSales
92 } // fim da função print
93

```

Figura 12.5 O arquivo de implementação para a classe `CommissionEmployee` representa um empregado que ganha uma porcentagem das vendas brutas.

(continuação)

```

1 // Figura 12.6: fig12_06.cpp
2 // Testando a classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "CommissionEmployee.h" Definição da classe CommissionEmployee
12
13 int main()
14 {
15     // instancia um objeto CommissionEmployee
16     CommissionEmployee employee(
17         "Sue", "Jones", "222-22-2222", 10000.06 );
18
19     // configura a formatação de saída de ponto flutuante
20     cout << fixed << setprecision( 2 );
21
22     // obtém os dados do empregado comissionado
23     cout << "Employee information obtained by get functions: \n"
24     << "First name is " << employee.getFirstName()
25     << "Last name is " << employee.getLastName()
26     << "Social security number is "
27     << employee.getSocialSecurityNumber()
28     << "Gross sales is " << employee.getGrossSales()
29     << "Commission rate is " << employee.getCommissionRate() << endl;
30
31     employee.setGrossSales(8000); // configura as vendas brutas
32     employee.setCommissionRate( ); // configura a taxa de comissão
33
34     cout << "\nUpdated employee information output by print function: \n"
35     << endl;
36     employee.print(); // exibe as novas informações do empregado
37
38     // exibe os rendimentos do empregado
39     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
40
41     return 0;
42 } // fim de main

```

Employee information obtained by get functions:

First name is Sue
 Last name is Jones
 Social security number is 222-22-2222
 Gross sales is 10000.00
 Commission rate is 0.06

Updated employee information output by print function:

commission employee: Sue Jones
 social security number: 222-22-2222
 gross sales: 8000.00

Figura 12.6 Programa de teste da classe CommissionEmployee

(continua)

```
commission rate: 0.10
Employee's earnings: $800.00
```

Figura 12.6 Programa de teste da classe CommissionEmployee

(continuação)

12.4.2 Criando uma classe BasePlusCommissionEmployee sem utilizar herança

Agora discutimos a segunda parte da nossa introdução à herança criando e testando uma classe (completamente nova e independente) da classe CommissionEmployee (Figuras 12.7–12.8), que contém nome, sobrenome, número de seguro social, quantidade de vendas brutas, taxa de comissão e salário-base.

Definindo a classe BasePlusCommissionEmployee

O arquivo de cabeçalho BasePlusCommissionEmployee.h (Figura 12.7) especifica os membros da classe BasePlusCommissionEmployee que incluem o construtor BasePlusCommissionEmployee (linhas 13–14) e as funções-membro (linha 34) para setar e retornar os valores.

As linhas 16–32 declaram as funções-membro declarados nas linhas 37–42.

NamastNamesocialSecurityNumber grossSales commissionRate baseSalary da classe. Essas variáveis e funções-membro encapsulam todos os recursos necessários de um empregado comissionado com salário-base. Observe a semelhança entre essa classe e a classe CommissionEmployee (Figuras 12.4–12.5) — neste exemplo, ainda não exploraremos essa semelhança.

A função-membro earnings (definida nas linhas 96–99 da Figura 12.8) da classe CommissionEmployee computa os rendimentos de um empregado comissionado com salário-base. A linha 98 retorna o resultado da adição do salário-base do empregado ao produto da taxa de comissão e das vendas brutas do empregado.

```
1 // Figura 12.7: BasePlusCommissionEmployee.h
2 // Definição da classe BasePlusCommissionEmployee representa um empregado
3 // que recebe um salário-base além da comissão.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 class BasePlusCommissionEmployee
11 {
12 public :
13     BasePlusCommissionEmployee(const string &, const string &,
14         const string &, double = 0.0, double = 0.0, double = 0.0 );
15
16     void setFirstName(const string &); // configura o nome
17     string getFirstName() const; // retorna o nome
18
19     void setLastName(const string &); // configura o sobrenome
20     string getLastName() const; // retorna o sobrenome
21
22     void setSocialSecurityNumber(const string &); // configura o SSN
23     string getSocialSecurityNumber() const; // retorna o SSN
24
25     void setGrossSales( double ); // configura a quantidade de vendas brutas
26     double getGrossSales() const; // retorna a quantidade de vendas brutas
27
28     void setCommissionRate(double ); // configura a taxa de comissão
29     double getCommissionRate() const; // retorna a taxa de comissão
30 }
```

Figura 12.7 Arquivo de cabeçalho da classe BasePlusCommissionEmployee

(continua)

```

31 void setBaseSalary( double ); // configura o salário-base
32 double getBaseSalary() const; // retorna o salário-base
33
34 double earnings() const; // calcula os rendimentos
35 void print() const; // imprime o objeto BasePlusCommissionEmployee
36 private :
37 string firstName;
38 string lastName;
39 string socialSecurityNumber;
40     double grossSales; // vendas brutas semanais
41     double commissionRate; // porcentagem da comissão
42     double baseSalary; // salário-base
43 }; // fim da classe BasePlusCommissionEmployee
44
45 #endif

```

Figura 12.7 Arquivo de cabeçalho da classe BasePlusCommissionEmployee

(continuação)

```

1 // Figura 12.8: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13 {
14     firstName = first; // deve validar
15     lastName = last; // deve validar
16     socialSecurityNumber = ssn; // deve validar
17     setGrossSales( sales ); // valida e armazena as vendas brutas
18     setCommissionRate( rate ); // valida e armazena a taxa de comissão
19     setBaseSalary( salary ); // valida e armazena salário-base
20 } // fim do construtor BasePlusCommissionEmployee
21
22 // configura o nome
23 void BasePlusCommissionEmployee::setFirstName( string &first )
24 {
25     firstName = first; // deve validar
26 } // fim da função setFirstName
27
28 // retorna o nome
29 string BasePlusCommissionEmployee::getFirstName() const
30 {
31     return firstName;
32 } // fim da função getFirstName
33
34 // configura o sobrenome
35 void BasePlusCommissionEmployee::setLastName( string &last )
36 {

```

Figura 12.8 A classe BasePlusCommissionEmployee

(continua)

```

37     lastName = last; // deve validar
38 } // fim da função setLastName
39
40 // retorna o sobrenome
41 string BasePlusCommissionEmployee::getLastName() const
42 {
43     return lastName;
44 } // fim da função getLastname
45
46 // configura o SSN
47 void BasePlusCommissionEmployee::setSocialSecurityNumber(
48     const string &ssn )
49 {
50     socialSecurityNumber = ssn; // deve validar
51 } // fim da função setSocialSecurityNumber
52
53 // retorna o SSN
54 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
55 {
56     return socialSecurityNumber;
57 } // fim da função getSocialSecurityNumber
58
59 // configura a quantidade de vendas brutas
60 void BasePlusCommissionEmployee::setGrossSales(double sales )
61 {
62     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
63 } // fim da função setGrossSales
64
65 // retorna a quantidade de vendas brutas
66 double BasePlusCommissionEmployee::getGrossSales() const
67 {
68     return grossSales;
69 } // fim da função getGrossSales
70
71 // configura a taxa de comissão
72 void BasePlusCommissionEmployee::setCommissionRate(double rate )
73 {
74     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
75 } // fim da função setCommissionRate
76
77 // retorna a taxa de comissão
78 double BasePlusCommissionEmployee::getCommissionRate()
79 {
80     return commissionRate;
81 } // fim da função getCommissionRate
82
83 // configura o salário-base
84 void BasePlusCommissionEmployee::setBaseSalary(double salary )
85 {
86     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
87 }
88
89 // retorna o salário-base
90 double BasePlusCommissionEmployee::getBaseSalary() const
91 {
92     return baseSalary;
93 } // fim da função getBaseSalary

```

Figura 12.8 A classe `BasePlusCommissionEmployee` apresenta um empregado que recebe um salário-base além de uma comissão. (continua)

```

94 // calcula os rendimentos
95 double BasePlusCommissionEmployee::earnings() const
96 {
97     return baseSalary + ( commissionRate * grossSales );
98 } // fim da função earnings
99
100
101 // imprime o objeto BasePlusCommissionEmployee
102 void BasePlusCommissionEmployee::print() const
103 {
104     cout << "base-salaried commission employee: " << firstName << " "
105         << lastName << "\nsocial security number: " << socialSecurityNumber
106         << "\ngross sales: " << grossSales
107         << "\ncommission rate: " << commissionRate
108         << "\nbase salary: " << baseSalary;
109 } // fim da função print

```

Figura 12.8 A classe `BasePlusCommissionEmployee` apresenta um empregado que recebe um salário-base além de uma comissão. (continuação)Testando a classe `BasePlusCommissionEmployee`

A Figura 12.9 testa a classe `BasePlusCommissionEmployee`. As linhas 17–18 instanciam o objeto `BasePlusCommissionEmployee` passando “Bob”, “Lewis”, “333-33-3333”, 5000.04 e 300 para o construtor como nome, sobrenome, número de seguro social, vendas brutas, taxa de comissão e salário-base, respectivamente. As linhas 24–31 utilizam as funções `setBaseSalary` para recuperar os valores dos membros de dados do objeto para a saída. A linha 33 invoca função-membro `setBaseSalary` do objeto para alterar o salário-base. A função `main` (Figura 12.8, linhas 84–87) assegura que o membro de dados `baseSalary` não receba um valor negativo, porque o salário-base de um empregado não pode ser negativo. A linha 37 da Figura 12.9 invoca a função `print` do objeto para gerar saída de informações atualizadas, e a linha 40 chama a função `getEarnings` para exibir os rendimentos do `BasePlusCommissionEmployee`.

```

1 // Figura 12.9: fig12_09.cpp
2 // Testando a classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definição da classe BasePlusCommissionEmployee
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instancia o objeto BasePlusCommissionEmployee
17     BasePlusCommissionEmployee employee("Bob", "Lewis", "333-33-3333", 5000 .04, 300);
18
19     // configura a formatação de saída de ponto flutuante
20     cout << fixed << setprecision( 2 );
21
22     // obtém os dados do empregado comissionado
23

```

Figura 12.9 Programa de teste da classe `BasePlusCommissionEmployee`

(continua)

```

24 cout << "Employee information obtained by get functions: \n"
25 << "\nFirst name is " << employee.getFirstName()
26 << "\nLast name is " << employee.getLastName()
27 << "\nSocial security number is "
28 << employee.getSocialSecurityNumber()
29 << "\nGross sales is " << employee.getGrossSales()
30 << "\nCommission rate is " << employee.getCommissionRate()
31 << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33 employee.setBaseSalary(1000); // configura o salário-base
34
35 cout << "\nUpdated employee information output by print function: \n"
36 << endl;
37 employee.print(); // exibe as novas informações do empregado
38
39 // exibe os rendimentos do empregado
40 cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42 return 0;
43 } // fim de main

```

Employee information obtained by get functions:

First name is Bob
 Last name is Lewis
 Social security number is 333-33-3333
 Gross sales is 5000.00
 Commission rate is 0.04
 Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
 social security number: 333-33-3333
 gross sales: 5000.00
 commission rate: 0.04
 base salary: 1000.00

Employee's earnings: \$1200.00

Figura 12.9 Programa de teste da classe `BasePlusCommissionEmployee`

(continuação)

Explorando as semelhanças entre a classe `BasePlusCommissionEmployee` e a classe `CommissionEmployee`. Observe que grande parte do código para a classe `BasePlusCommissionEmployee` (figuras 12.7–12.8) é semelhante, se não idêntica, ao do código para a classe `CommissionEmployee` (figuras 12.4–12.5). Por exemplo, na classe `CommissionEmployee`, os membros de dados `private firstName, lastName, socialSecurityNumber` e as funções-membro `getFirstName, setLastName, getLastName` são idênticos aos da classe `BasePlusCommissionEmployee`. As classes `CommissionEmployee` e `BasePlusCommissionEmployee` também contêm, ambas, os membros de privado `socialSecurityNumber, commissionRate, grossSales`, bem como as funções para manipular esses membros. Além disso, a classe `BasePlusCommissionEmployee` é quase idêntica à classe `CommissionEmployee`, exceto pelo fato de que o construtor de `CommissionEmployee` também configura `baseSalary`. As outras adições na classe `PlusCommissionEmployee` são os membros `private baseSalary` e as funções-membro `getBaseSalary, setBaseSalary`. A função-membro `print` da classe `BasePlusCommissionEmployee` é quase idêntica à da classe `CommissionEmployee`, exceto pelo fato de que, devido ao uso da classe `BasePlusCommissionEmployee`, também gera saída do valor do membro de dados `baseSalary`.

Cópamos literalmente o código da classe `CommissionEmployee` e colamos na classe `BasePlusCommissionEmployee`. Então modificamos a classe `BasePlusCommissionEmployee` para incluir um salário-base e as funções-membro que manipulam o salário-base. Essa abordagem ‘copiar e colar’ é freqüentemente propensa a erro e demorada. Pior ainda, ela pode espalhar muitas cópias físicas d

código por todo um sistema, criando um pesadelo para a manutenção de código. Existe um modo de ‘absorver’ os membros de funções-membro de uma classe de maneira que façam parte de outras classes sem duplicar o código? Nos próximos vários e fazemos exatamente isso, utilizando a herança.



Observação de engenharia de software 12.3

Copiar e colar código de uma classe para a outra pode espalhar erros por múltiplos arquivos de código-fonte. Para evitar a duplicação de código (e possivelmente erros), utilize a herança, em vez da abordagem ‘copiar e colar’, em situações em que você quer que uma classe ‘absorva’ os membros de dados e as funções-membro de outra classe.



Observação de engenharia de software 12.4

Com a herança, os membros de dados e funções-membro comuns a todas as classes na hierarquia são declarados em uma classe

básica. Quando essas classes derivadas se utilizarem desses membros, não haverá necessidade de redeclará-los em todos os arquivos de código-fonte que contêm uma cópia do código em questão.

12.4.3 Criando uma hierarquia de herança CommissionEmployee–BasePlusCommissionEmployee

Agora criamos e testamos uma nova versão da classe CommissionEmployee (figuras 12.10–12.11) que deriva da classe CommissionEmployee (figuras 12.4–12.5). Nesse exemplo, o objeto CommissionEmployee é herançado por CommissionEmployee. Porque a herança transfere as capacidades da classe base para a classe derivada, a classe BasePlusCommissionEmployee também tem o membro de dados baseSalary (Figura 12.10, linha 24). O sinal de herança public da definição de classe indica herança. A palavra-chave public indica o tipo de herança. Como uma classe derivada (formada com a herança) herda todos os membros da classe base, exceto pelo construtor — cada classe fornece seus próprios construtores que são específicos à classe. [Observe que os destrutores também não são herdados.] Rendimentos, os serviços incluem seu construtor (linhas 15–16) e as funções-membro da classe CommissionEmployee embora

```

1 // Figura 12.10: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee public CommissionEmployee
13 {
14 public :
15     BasePlusCommissionEmployee(const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private :
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 12.10 A definição da classe BasePlusCommissionEmployee, usando o relacionamento de herança com a classe CommissionEmployee

não possamos ver essas funções-membro herdadas no código-fonte de elas são, contudo, uma parte da classe derivada. Os serviços da classe derivada também incluem as funções-membro `Salary, getBaseSalary, earnings e print` (linhas 18–22).

A Figura 12.11 mostra as implementações das funções-membro da classe `BasePlusCommissionEmployee`. O construtor (linhas 10–17) introduz a sintaxe inicializadora da classe básica (linha 14), que utiliza um inicializador de membro para argumentos passados ao construtor da classe básica `CommissionEmployee`. O C++ requer que o construtor da classe derivada chame seu construtor de classe básica para inicializar os membros de dados da classe básica que são herdados na classe derivada. A linha 14 realiza essa tarefa invocando o construtor `CommissionEmployee` pelo nome, passando os parâmetros do construtor `first, last, ssn, sales, rate` como argumentos para inicializar os membros de dados da classe básica `socialSecurityNumber, grossSales e commissionRate`. Se o construtor `BasePlusCommissionEmployee` não invocasse o construtor da classe `CommissionEmployee` explicitamente, o C++ tentaria invocar o construtor-padrão da classe `CommissionEmployee`, mas a classe não tem esse construtor, portanto o compilador emitiria um erro. Considerando a discussão do Capítulo 3, lembre-se de que o compilador fornece um construtor-padrão sem parâmetro para qualquer classe que não inclui explicitamente um construtor. Entretanto, se a classe incluir um construtor explicitamente, estaria construindo o compilador fornecido e qualquer tentativa de chamar implicitamente o construtor-padrão do

```

1 // Figura 12.11: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica
14 { : CommissionEmployee( first, last, ssn, sales, rate )
15     setBaseSalary( salary ); // valida e armazena salário-base
16 } // fim do construtor BasePlusCommissionEmployee
17
18 // configura o salário-base
19 void BasePlusCommissionEmployee::setBaseSalary(double salary )
20 {
21     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22 } // fim da função setBaseSalary
23
24 // retorna o salário-base
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27     return baseSalary;
28 } // fim da função getBaseSalary
29
30 // calcula os rendimentos
31 const
32 {     BasePlusCommissionEmployee::earnings()
33     // a classe derivada não pode acessar dados private da classe básica
34     return baseSalary + ( commissionRate * grossSales );
35 }
36 // fim da função earnings
37
38 // imprime o objeto BasePlusCommissionEmployee

```

Figura 12.11 Arquivo de implementação da classe `BasePlusCommissionEmployee`. Os membros `private` da classe básica não podem ser acessados a partir da classe derivada.

(continua)

```

39 void BasePlusCommissionEmployee::print()const
40 {
41     // a classe derivada não pode acessar dados private da classe básica
42     cout << "base-salaried commission employee: " << firstName << " "
43         << lastName << "\nsocial security number: " << socialSecurityNumber
44         << "\ngross sales: " << grossSales
45         << "\ncommission rate: " << commissionRate
46         << "\nbase salary: " << baseSalary;
47 } // fim da função print

```

```

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(25)
error C2248: 'CommissionEmployee::commissionRate':
cannot access private member declared in class 'CommissionEmployee'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee::commissionRate'
C:\cpphtp5e_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
    see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(25)
error C2248: 'CommissionEmployee::grossSales':
cannot access private member declared in class 'CommissionEmployee'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee::grossSales'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
    see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(25)
error C2248: 'CommissionEmployee::firstName':
cannot access private member declared in class 'CommissionEmployee'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(25)
error C2248: 'CommissionEmployee::lastName':
cannot access private member declared in class 'CommissionEmployee'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee::lastName'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
    see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(25)
error C2248: 'CommissionEmployee::socialSecurityNumber':
cannot access private member declared in class 'CommissionEmployee'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee::socialSecurityNumber'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
    see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(25)
error C2248: 'CommissionEmployee::grossSales':
cannot access private member declared in class 'CommissionEmployee'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee::grossSales'

```

Figura 12.11 Arquivo de implementação da classe derivada `BasePlusCommissionEmployee`. Os membros `private` da classe básica não podem ser acessados a partir da classe derivada.

(continua)

```
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp
error C2248: 'CommissionEmployee::commissionRate':
cannot access private member declared in class 'CommissionEmployee'
    C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
    see declaration of 'CommissionEmployee::commissionRate'
C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'
```

Figura 12.11 Arquivo de implementação `BasePlusCommissionEmployee.cpp`: os membros `private` da classe básica não podem ser acessados a partir da classe derivada.

(continuação)



Erro comum de programação 12.1

Ocorrerá um erro de compilação se um construtor de classe derivada chamar um de seus construtores de classe básica com argumentos que são inconsistentes com o número e os tipos de parâmetros especificados em uma das definições de construtor de classe básica.



Dica de desempenho 12.1

Em um construtor de classe derivada, inicializar os objetos-membro e invocar construtores de classe básica explicitamente na lista de inicializadores de membro impede a inicialização duplicada na qual um construtor-padrão é chamado, então os membros de dados são modificados novamente no corpo do construtor da classe derivada.

O compilador gera erros para a linha 35 da Figura 12.11 porque os membros de dados da classe básica `CommissionEmployee::private` — funções-membro da classe derivada `BasePlusCommissionEmployee` — têm permissão de acessar os dados da classe básica `CommissionEmployee`. Observe que utilizamos texto em itálico na Figura 12.11 para indicar o código incorreto. O compilador emite erros adicionais nas linhas 42–45 da mesma razão.

Erros produzidos por essa razão: `ComissionEmployee::getGrossSales()` não pode ser chamado devido à restrição de visibilidade imposto pelo escopo da classe derivada. [Nota: Para poupar espaço, mostramos somente as mensagens de erro do Visual C++ .NET neste exemplo. As mensagens de erro produzidas por seu compilador podem diferir das mostradas aqui. Note também que destacamos em negrito as partes-chave longas mensagens de erro.]

Incluímos propositalmente o código errado na Figura 12.11 para demonstrar que funções-membro de uma classe derivada podem acessar os dados da sua classe básica. Os membros da classe derivada poderiam ser evitados utilizando as funções-membro dadas da classe básica. Por exemplo, a linha 35 poderia ter sido escrita da seguinte maneira: `getGrossSales()` para acessar, respectivamente, os membros de dados `rate` e `grossSales` da classe `CommissionEmployee`. De maneira semelhante, as linhas 42–45 poderiam utilizar `getBonus()` para recuperar os valores dos membros de dados da classe básica. No próximo exemplo, mostraremos como corrigir os erros encontrados nesse exemplo.

Incluindo o arquivo de cabeçalho da classe básica no arquivo de cabeçalho da classe derivada.

Note que incluirmos (ou) o arquivo de cabeçalho da classe básica no arquivo de cabeçalho da classe derivada (linha 10 da Figura 12.10). Essa é necessária por três razões. Primeiro, para a classe derivada utilizar o nome da classe básica na linha 12, devemos garantir que a classe básica existe — a definição da classe deve exatamente isso.

A segunda razão é que o compilador utiliza uma definição de classe para determinar o tamanho de um objeto dessa classe (discutimos na Seção 3.8). Um programa-cliente que cria um objeto de uma classe derivada para permitir ao compilador reservar a quantidade adequada de memória para o objeto. Ao utilizar herança, o tamanho de um objeto da classe derivada depende dos membros de dados declarados explicitamente na classe derivada e dos membros de suas classes básicas diretas e indiretas. Incluir a definição da classe básica na linha 10 permite ao compilador determinar os requisitos de memória para os membros de dados da classe básica que se tornam parte de um objeto de classe derivada e, assim, controlar o tamanho total do objeto de classe derivada.

A última razão da linha 10 é permitir ao compilador determinar se a classe derivada utiliza ou não os membros herdados da classe básica adequadamente. Por exemplo, no programa das figuras 12.10–12.11, o compilador utiliza o arquivo de cabeçalho da classe básica para determinar que os membros de dados sendo acessados pela classe derivada. Vê-se que esses são inacessíveis à classe derivada, o compilador gera erros. O compilador também utiliza os protótipos de função da classe básica para

as chamadas de função feitas pela classe derivada para as funções de classe básica herdadas — você verá um exemplo des de função na Figura 12.16.

Processo de linkagem em uma hierarquia de herança

Na Seção 3.9, discutimos o processo de linkagem para aplicativos. Nesse exemplo, você viu que o código-objeto do cliente foi linkado com o código-objeto da classe C++ Standard Library utilizada no código-cliente. Vamos ver como o código-objeto para a classe

O processo de linkagem é semelhante a um programa que utiliza classes em uma hierarquia de herança. O processo requer o código-objeto para todas as classes utilizadas no programa e o código-objeto para as classes básicas diretas e indiretas de qual derivada utilizada pelo programa. Suponha que um cliente queira criar uma aplicação que utilize a classe que é uma classe derivada. Vamos vermos um exemplo disso na Seção 12.4.4). Ao compilar o aplicativo-cliente, o código-objeto do cliente deve ser linkado com o código-objeto das classes-pai CommissionEmployee porque BasePlusCommissionEmployee herda funções-membro de sua classe básica. O código também é linkado

com o código-objeto da classe-superior StandardCommissionEmployee, que implementa todas as funcionalidades que o programa pode utilizar.

12.4.4 Hierarquia de herança CommissionEmployee–BasePlusCommissionEmployee utilizando dados protected

Para permitir que a classe CommissionEmployee pusesse diretamente os membros de dados firstName, lastName, socialSecurityNumber, grossSales e commissionRate. Você pode declarar esses membros como membros da classe básica. Como discutimos na Seção 12.3, os membros da classe básica podem ser acessados tanto por membros e membros de qualquer classe derivada dessa classe básica.



Boa prática de programação 12.1

Em primeiro lugar, declare os membros; segundo, os membros e, por último, os membros

Definindo a classe básica CommissionEmployee com dados protected

A classe CommissionEmployee (Figuras 12.12–12.13) agora declara os membros de dados firstName, lastName, socialSecurityNumber, grossSales e commissionRate protected (Figura 12.12, linhas 33–37) para as implementações de função-membro da Figura 12.13 são idênticas àquelas da Figura 12.5.

```

1 // Figura 12.12: CommissionEmployee.h
2 // Definição da classe CommissionEmployee com dados protected.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public :
12     CommissionEmployee( const string &, const string &, const string &,
13                         double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName(const string &); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber(const string &); // configura SSN

```

Figura 12.12 Definição da classe CommissionEmployee que declara dados protected para permitir acesso por classes derivadas. (continua)

```

22     string getSocialSecurityNumber() const; // retorna SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate(double ); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 protected :
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 12.12 Definição da classe `CommissionEmployee`. Ela declara dados `protected` para permitir acesso por classes derivadas. (continuação)

```

1 // Figura 12.13: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee

7 // construtor
8 CommissionEmployee::CommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate )
11 {
12     firstName = first; // deve validar
13     lastName = last; // deve validar
14     socialSecurityNumber = ssn; // deve validar
15     setGrossSales( sales ); // valida e armazena as vendas brutas
16     setCommissionRate( rate ); // valida e armazena a taxa de comissão
17 }
18 // fim do construtor CommissionEmployee
19
20 // configura o nome
21 void CommissionEmployee::setFirstName(const string &first )
22 {
23     firstName = first; // deve validar
24 } // fim da função setFirstName
25
26 // retorna o nome
27 string CommissionEmployee::getFirstName() const
28 {
29     return firstName;
30 } // fim da função getFirstName
31
32 // configura o sobrenome

```

Figura 12.13 A classe `CommissionEmployee` tem dados `protected`.

(continua)

```

33 void CommissionEmployee::setLastName(const string &last)
34 {
35     lastName = last; // deve validar
36 } // fim da função setLastName
37
38 // retorna o sobrenome
39 string CommissionEmployee::getLastName()const
40 {
41     return lastName;
42 } // fim da função getLastname
43
44 // configura o SSN
45 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
46 {
47     socialSecurityNumber = ssn; // deve validar
48 } // fim da função setSocialSecurityNumber
49
50 // retorna o SSN
51 string CommissionEmployee::getSocialSecurityNumber()const
52 {
53     return socialSecurityNumber;
54 } // fim da função getSocialSecurityNumber
55
56 // configura a quantidade de vendas brutas
57 void CommissionEmployee::setGrossSales(double sales)
58 {
59     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60 } // fim da função setGrossSales
61
62 // retorna a quantidade de vendas brutas
63 double CommissionEmployee::getGrossSales()const
64 {
65     return grossSales;
66 } // fim da função getGrossSales
67
68 // configura a taxa de comissão
69 void CommissionEmployee::setCommissionRate(double rate)
70 {
71     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
72 } // fim da função setCommissionRate
73
74 // retorna a taxa de comissão
75 double CommissionEmployee::getCommissionRate()const
76 {
77     return commissionRate;
78 } // fim da função getCommissionRate
79
80 // calcula os rendimentos
81 double CommissionEmployee::earnings()const
82 {
83     return commissionRate * grossSales;
84 } // fim da função earnings
85
86 // imprime o objeto CommissionEmployee
87 void CommissionEmployee::print()const

```

Figura 12.13 A classe CommissionEmployee tem dados protected.

(continua)

```

88 {
89     cout << "commission employee: "<< firstName << ' ' << lastName
90     << "\nsocial security number: " << socialSecurityNumber
91     << "\ngross sales: " << grossSales
92     << "\ncommission rate: " << commissionRate;
93 } // fim da função print

```

Figura 12.13 A classe CommissionEmployee em dados protected.

(continuação)

Modificando a classe derivada BasePlusCommissionEmployee

Após as modificações feitas na Figura 12.13, o código da classe CommissionEmployee é o seguinte. Note que os membros de dados lastName, socialSecurityNumber, grossSales e commissionRate permanecem visíveis para os objetos da classe BasePlusCommissionEmployee, porque podem acessar membros de dados herdados que são membros protected da classe base CommissionEmployee (isto é, membros de dados protected da classe base CommissionEmployee). Como resultado, o compilador não gera erros ao compilar as definições das funções earnings e print na Figura 12.15 (linhas 32–36 e 39–47, respectivamente). Isso mostra os privilégios especiais que são concedidos a uma classe derivada para acessar membros de dados protected da classe básica. Os objetos de uma classe derivada também podem acessar membros protected em qualquer uma das classes básicas indiretas da classe derivada.

A classe BasePlusCommissionEmployee não herda o construtor da classe CommissionEmployee. Entretanto, o construtor da classe BasePlusCommissionEmployee (Figura 12.15, linhas 10–17) chama o construtor da classe CommissionEmployee explicitamente (linha 14). Lembre-se de que o construtor de BasePlusCommissionEmployee deve chamar explicitamente o construtor da classe CommissionEmployee, porque a classe CommissionEmployee não contém um construtor-padrão que poderia ser invocado implicitamente.

```

1 // Figura 12.14: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public :
15     BasePlusCommissionEmployee(const string &, const string &,
16                             const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private :
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 12.14 Arquivo de cabeçalho da classe BasePlusCommissionEmployee

```

1 // Figura 12.15: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica
14     : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary ); // valida e armazena o salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary(double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     // pode acessar dados protected da classe básica
35     return baseSalary + ( commissionRate * grossSales );
36 } // fim da função earnings
37
38 // imprime o objeto BasePlusCommissionEmployee
39 void BasePlusCommissionEmployee::print() const
40 {
41     // pode acessar dados protected da classe básica
42     cout << "base-salaried commission employee: " << firstName << " "
43         << lastName << "\nsocial security number: " << socialSecurityNumber
44         << "\ngross sales: " << grossSales
45         << "\ncommission rate: " << commissionRate
46         << "\nbase salary: " << baseSalary;
47 } // fim da função print

```

Figura 12.15 O arquivo de implementação `BasePlusCommissionEmployee.cpp` a classe `BasePlusCommissionEmployee` herda dados `protected` de `CommissionEmployee`

Testando a classe `BasePlusCommissionEmployee` modificada

A Figura 12.16 utiliza um `objeto` `BasePlusCommissionEmployee` para realizar as mesmas tarefas que a Figura 12.9 realizou em um objeto da primeira versão da classe `BasePlusCommissionEmployee` (figuras 12.7–12.8). Observe que as saídas dos dois programas são idênticas. Criamos a primeira classe `BasePlusCommissionEmployee` sem utilizar a herança e criamos essa versão de `CommissionEmployee` utilizando a herança; entretanto, ambas as classes fornecem as mesmas funcionalidades. Observe que o código para a classe `BasePlusCommissionEmployee` (isto é, o cabeçalho e os arquivos de implementação), que tem 74 linhas, é consideravel-

mente mais curto que o código para a versão da classe não herdada, que tem 154 linhas, porque a versão herdada absorve pa
funcionalidades a partir de `CommissionEmployee`, enquanto a versão não herdada não absorve nenhuma funcionalidade. Além disso, há
agora apenas uma cópia da funcionalidade de `print` declarada e definida na classe `BasePlusCommissionEmployee`. Isso torna o
código-fonte mais fácil de manter, modificar e depurar, porque o código é fonte relacional e pode ser lido diretamente
nos arquivos das figuras 12.12–12.13.

Notas sobre a utilização dos dados `protected`

Neste exemplo, declaramos membros de dados `protected` na classe base, porque as classes derivadas pudessem modificar os dados diretamente. Herdar membros de dados `protected` aumenta ligeiramente o desempenho, porque podemos acessar os membros diretamente sem incorrer no overhead de chamar `setBaseSalary` em todos os casos, porém, é melhor utilizar os membros de `dados` para incentivar a engenharia de software adequada e deixar as questões de otimização de código para o compilador. Seu código será mais fácil de manter, modificar e depurar.

```

1 // Figura 12.16: fig12_16.cpp
2 // Testando a classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definição da classe BasePlusCommissionEmployee
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instancia o objeto BasePlusCommissionEmployee
17     BasePlusCommissionEmployee
18         employee(    "Bob", "Lewis", "333-33-3333", 5000 .04, 300);
19
20     // configura a formatação de saída de ponto flutuante
21     cout << fixed << setprecision( 2 );
22
23     // obtém os dados de empregado comissionado
24     cout << "Employee information obtained by get functions: \n"
25     << "\nFirst name is " << employee.getFirstName()
26     << "\nLast name is " << employee.getLastName()
27     << "\nSocial security number is "
28     << employee.getSocialSecurityNumber()
29     << "\nGross sales is " << employee.getGrossSales()
30     << "\nCommission rate is " << employee.getCommissionRate()
31     << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary(1000); // configura o salário-base
34
35     cout << "\nUpdated employee information output by print function: \n"
36
37     employee.print(); // exibe as novas informações do empregado
38
39     // exibe os rendimentos do empregado
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43 } // fim de main

```

Figura 12.16 Os dados de classe básica `protected` podem ser acessados a partir da classe derivada.

(continua)

```

Employee information obtained by get functions:
First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00

```

Figura 12.16 Os dados de classe básica `protected` podem ser acessados a partir da classe derivada.

(continuação)

Utilizar membros de `protected` cria dois problemas importantes. Primeiro, o objeto de classe derivada não tem de utilizar uma função-membro para configurar o valor do membro de classe `básica`. Portanto, um objeto de classe derivada pode atribuir facilmente um valor inválido ao membro de classe `básica`, assim, o objeto em um estado inconsistente. Por exemplo, com o membro de classe `dados` declarado como `protected`, um objeto de classe derivada (por exemplo, `BasePlusCommissionEmployee`) pode atribuir um valor `negativo`. O segundo problema com o uso de membros de `protected` é que é mais provável que as funções-membro de classe derivada sejam escritas de modo a depender da implementação da classe básica. Na prática, as classes derivadas só devem depender dos serviços da classe básica (isto é, funções-`public`), não da implementação da classe básica. Com membros de classe `protected`, se a implementação de classe básica mudar, podemos precisar mudar todas as classes derivadas dessa classe básica. Podemos por alguma razão querer que uma classe derivada referencia diretamente esses membros de classe básica. Em um caso como esse, é porque uma pequena alteração na classe básica pode ‘quebrar’ a implementação da classe derivada. O programador deve ser capaz de alterar a implementação da classe básica ao mesmo tempo em que ainda fornece os mesmos serviços para as classes derivadas. (Naturalmente, se os serviços de classe básica mudarem, devemos reimplementar nossas classes derivadas — um bom projeto orientado a objetos tenta evitar isso.)

**Observação de engenharia de software 12.5**

É apropriado utilizar o especificador `protected` quando uma classe básica tiver de fornecer um serviço (isto é, uma função-membro) apenas a suas classes derivadas e outros clientes.

**Observação de engenharia de software 12.6**

Declarar membros de classe `protected` (`composição a declarar`) permite aos programadores alterar a implementação da classe básica sem alterar as implementações de classe derivada.

**Dica de prevenção de erro 12.1**

Quando possível, evite incluir membros de classe `protected` em uma classe básica. Em vez disso, inclua funções-membro não-`private` que aceitem membros de classe `protected`, garantindo que o objeto mantenha um estado consistente.

12.4.5 Hierarquia de herança `CommissionEmployee`–`BasePlusCommissionEmployee` utilizando dados `private`

Agora reexamine nossa hierarquia mais uma vez, dessa vez utilizando práticas de engenharia de software melhores. A classe `CommissionEmployee` (figuras 12.17–12.18) agora declara membros de classe `private` `socialSecurityNumber`, `grossSales`

ecommissionRate (Figura 12.17, linhas 33–37) e fornece as funções-membro `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` e `print` para manipular esses valores. Se decidirmos alterar o nome dos membros de dados, as definições `print` não exigirão modificação — somente as definições `get` e `set` precisarão mudar diretamente os membros de dados precisarão mudar. Observe que essas alterações ocorrem unicamente dentro da classe básica. Localizar os efeitos de alterações como esta é uma boa prática de engenharia de software. A classe derivada `CommissionEmployee` (Figuras 12.19–12.20) herda as funções-membro e não precisa alterar suas definições. A classe derivada `CommissionEmployee` (Figuras 12.19–12.20) herda as funções-membro e não precisa alterar suas definições.

Na implementação do construtor `CommissionEmployee` (Figura 12.18, linhas 9–16), observe que utilizamos inicializadores de membro (linha 12) para configurar os valores dos membros `firstName`, `lastName`, `socialSecurityNumber`. Mostramos como a classe derivada `BasePlusCommissionEmployee` (Figuras 12.19–12.20) pode invocar funções-membro da classe base (`getFirstName`, `setLastName`, `getSocialSecurityNumber`, `getSocialSecurityNumber`) para manipular esses membros de dados.

```

1 // Figura 12.17: CommissionEmployee.h
2 // Definição da classe CommissionEmployee com boa engenharia de software.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public :
12     CommissionEmployee( const string &, const string &, const string &,
13                         double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16
17     string getFirstName() const; // retorna o nome
18     void setLastName(const string &); // configura o sobrenome
19     string getLastname() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate(double); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 private :
33     string firstName;
34
35     string lastName;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 12.17 Classe `CommissionEmployee` definida utilizando práticas de boa engenharia de software.

```

1 // Figura 12.18: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )
12 : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14     setGrossSales( sales ); // valida e armazena as vendas brutas
15     setCommissionRate( rate ); // valida e armazena a taxa de comissão
16 } // fim do construtor CommissionEmployee
17
18 // configura o nome
19 void CommissionEmployee::setFirstName(const string &first )
20 {
21     firstName = first; // deve validar
22 } // fim da função setFirstName
23
24 // retorna o nome
25 string CommissionEmployee::getFirstName() const
26 {
27     return firstName;
28 } // fim da função getFirstName
29
30 // configura o sobrenome
31 void CommissionEmployee::setLastName(const string &last )
32 {
33     lastName = last; // deve validar
34 } // fim da função setLastName
35
36 // retorna o sobrenome
37 string CommissionEmployee::getLastName() const
38 {
39     return lastName;
40 } // fim da função getLastname
41
42 // configura o SSN
43 void CommissionEmployee::setSocialSecurityNumber(const string &ssn )
44 {
45     socialSecurityNumber = ssn; // deve validar
46 } // fim da função setSocialSecurityNumber
47
48 // retorna o SSN
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51     return socialSecurityNumber;
52 } // fim da função getSocialSecurityNumber
53
54 // configura a quantidade de vendas brutas
55 void CommissionEmployee::setGrossSales(double sales )
56 {

```

Figura 12.18 Arquivo de implementação da classe `CommissionEmployee`. A classe `CommissionEmployee` utiliza funções-membro para manipular seus dados `private`.

(continua)

```

57     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // fim da função setGrossSales
59
60 // retorna a quantidade de vendas brutas
61 double CommissionEmployee::getGrossSales() const
62 {
63     return grossSales;
64 } // fim da função getGrossSales
65
66 // configura a taxa de comissão
67 void CommissionEmployee::setCommissionRate(double rate)
68 {
69     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // fim da função setCommissionRate
71
72 // retorna a taxa de comissão
73 double CommissionEmployee::getCommissionRate() const
74 {
75     return commissionRate;
76 } // fim da função getCommissionRate
77
78 // calcula os rendimentos
79 double CommissionEmployee::earnings() const
80 {
81     return getCommissionRate() * getGrossSales();
82 } // fim da função earnings
83
84 // imprime o objeto CommissionEmployee
85 void CommissionEmployee::print() const
86 {
87     cout << "commission employee: "
88     << getFirstName() << " " << getLastName()
89     << "\nsocial security number: " << getSocialSecurityNumber()
90     << "\ngross sales: " << getGrossSales()
91     << "\ncommission rate: " << getCommissionRate();
92 } // fim da função print

```

Figura 12.18 Arquivo de implementação da classe `CommissionEmployee` (continuação)



Dica de desempenho 12.2

Utilizar uma função-membro para acessar o valor de um membro de dados é ligeiramente mais lento que acessar os dados diretamente. Entretanto, os compiladores de otimização atuais são projetados com cuidado para realizar muitas otimizações implicitamente (como colocar inline chamadas de função). Como resultado, os programadores devem escrever código que obedeça aos princípios apropriados da engenharia de software e deixar questões de otimização para o compilador. Uma boa regra é 'Não se antecipe ao compilador'.

A classe `BasePlusCommissionEmployee` (Figuras 12.19–12.20) tem várias alterações em suas implementações de função-membro (Figura 12.20) que a distinguem da versão anterior da classe (Figuras 12.18–15). As linhas 30–35 (linhas 38–48) invocam a função `getBaseSalary()` para obter o valor do salário-base, em vez de acessar `baseSalary` diretamente. Isso pode levar a potenciais alterações na implementação da função `getEarnings()`. Por exemplo, se decidirmos renomear o membro de dados `baseSalary`, precisaremos alterar seu tipo, somente as funções `getBaseSalary()` precisarão mudar.

A função `getEarnings()` da classe `BasePlusCommissionEmployee` (Figura 12.20, linhas 32–35) redefine a função do membro classe `CommissionEmployee` (Figura 12.18, linhas 79–82) para calcular os rendimentos de um empregado comissionado com salário-base. A versão de `getEarnings()` da classe `BasePlusCommissionEmployee` calcula a parte dos rendimentos do empregado baseada exclusivamente na comissão, chamando a função da classe básica `CommissionEmployee::getEarnings()`.

(Figura 12.20, linha 34). A função `getEarnings` adiciona o salário-base a esse valor para calcular os rendimentos totais do empregado. Observe a sintaxe utilizada para invocar uma função-membro da classe básica recorrendo ao operador de resolução de escopo binário (`base::getEarnings()`). Essa invocação de função-membro é uma boa prática de engenharia de software: considerando “Observação de engenharia de software 9.9”, lembre-se de que, se a função-membro de um objeto realiza as ações necessárias a outro objeto, devemos chamar essa função-membro em vez de duplicar o código. Fazendo a função `getEarnings` invocar a função `getEarnings` da classe `CommissionEmployee`, evitamos duplicar o código e reduzimos problemas de manutenção de código.

```

1 // Figura 12.19: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee public CommissionEmployee
13 {
14 public :
15     BasePlusCommissionEmployee(const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0);
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private :
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 12.19 Arquivo de cabeçalho da classe `BasePlusCommissionEmployee`

```

1 // Figura 12.20: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica

```

Figura 12.20 A classe `BasePlusCommissionEmployee` herda da classe `CommissionEmployee` e não pode acessar diretamente os dados da classe.
(continua)

```

14     : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary ); // valida e armazena salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // fim da função earnings
36
37 // imprime o objeto BasePlusCommissionEmployee
38 void BasePlusCommissionEmployee::print() const
39 {
40     cout << "base-salaried ";
41
42     // invoca a função print de CommissionEmployee
43     CommissionEmployee::print();
44
45     cout << "\nbase salary: " << getBaseSalary();
46 } // fim da função print

```

Figura 12.20 A classe `BasePlusCommissionEmployee` herda da classe `CommissionEmployee`, mas não pode acessar diretamente os dados da classe.

(continuação)



Erro comum de programação 12.2

Quando uma função-membro de classe básica é redefinida para uma classe derivada, a versão da classe derivada freqüentemente chama a versão da classe básica para fazer o trabalho adicional. A falta de ~~prefixo da classe~~ da classe básica ao referenciar a função-membro da classe básica causa a recursão infinita, porque a função-membro da classe derivada chamaria a si própria.



Erro comum de programação 12.3

Incluir uma função-membro de classe básica com uma assinatura diferente na classe derivada oculta a versão de classe básica da função. Tentativas de chamar a versão de classe ~~básica~~ de classe derivada resultam em erros de compilação.

De maneira semelhante, a função `print` da classe `BasePlusCommissionEmployee` (Figura 12.20, linhas 38–46) redefine a função-membro `print` da classe `CommissionEmployee` (Figura 12.18, linhas 85–92) para gerar a saída de informações que sejam apropriadas para o empregado comissionado com salário-base. ~~BasePlusCommissionEmployee~~ deve parte das informações de um objeto `BasePlusCommissionEmployee` (isto é, a string `first` e `last`, e os valores dos membros de ~~dados da classe~~ `baseSalary` e `sales`). Chamando a função `print` da classe `CommissionEmployee` com o nome qualificado `CommissionEmployee::print()` (Figura 12.20, linha 43), ~~BasePlusCommissionEmployee~~ gera saída das informações restantes do objeto `BasePlusCommissionEmployee` (isto é, o valor do salário-base).

A Figura 12.21 realiza as mesmas manipulações para um objeto de classe derivada que as figuras 12.9 e 12.16 sobre os objetos das classes `CommissionEmployee` e `BasePlusCommissionEmployee`, respectivamente. Embora toda a classe 'empregado comissionado com salário-base' comporte-se de maneira idêntica a classe `employee`, o melhor projeto. Utilizando a herança e chamando as funções-membro que ocultam os dados e asseguram a consistência, criamos eficiente e eficaz uma classe bem-projetada.

Nesta seção, você viu um conjunto evolutivo de exemplos que foi cuidadosamente projetado para ensinar as capacidades-boa engenharia de software com herança. Você aprendeu a criar uma classe derivada usando herança, a utilizar membros de classe `protected` para permitir que uma classe derivada acesse membros de dados da classe básica herdados e a redefinir as funções da classe básica para fornecer versões mais apropriadas aos objetos de classe derivada. Além disso, você aprendeu a aplicar as técnicas de software apresentadas nos capítulos 9–10 e neste capítulo para criar classes que são fáceis de manter, modificar e extender.

```

1 // Figura 12.21: fig12_21.cpp
2 // Testando a classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definição da classe BasePlusCommissionEmployee
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instancia o objeto BasePlusCommissionEmployee
17     BasePlusCommissionEmployee
18         employee(    "Bob", "Lewis", "333-33-3333", 5000 .04, 300);
19
20     // configura a formatação de saída de ponto flutuante
21     cout << fixed << setprecision( 2 );
22
23     // obtém os dados do empregado comissionado
24     cout << "Employee information obtained by get functions: \n"
25     << "\nFirst name is " << employee.getFirstName()
26     << "\nLast name is " << employee.getLastName()
27     << "\nSocial security number is "
28     << employee.getSocialSecurityNumber()
29     << "\nGross sales is " << employee.getGrossSales()
30     << "\nCommission rate is " << employee.getCommissionRate()
31     << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary(1000); // configura o salário-base
34
35     cout << "\nUpdated employee information output by print function: \n"
36     << endl;
37     employee.print(); // exibe as novas informações do empregado
38
39     // exibe os rendimentos do empregado
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43 } // fim de main

```

Figura 12.21 Dados `private` de classe básica são acessíveis a uma classe derivada via função `public` ou `protected` herdada pela classe derivada.

(continua)

```
Employee information obtained by get functions:
```

```
First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00
```

```
Updated employee information output by print function:
```

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

```
Employee's earnings: $1200.00
```

Figura 12.21 Dados `private` de classe básica são acessíveis a uma classe derivada via função `public` ou `protected` herdada pela classe derivada. (continuação)

12.5 Construtores e destrutores em classes derivadas

Como explicamos na seção anterior, instanciar um objeto de classe derivada inicia uma cadeia de chamadas de construtor (o construtor da classe derivada, antes de realizar suas próprias tarefas, invoca o construtor da sua classe básica direta explicitamente (usando o operador `super`) ou implicitamente (chamando o construtor-padrão da classe básica). De maneira lógica, se a classe básica é derivada de outra classe, o construtor da classe básica é solicitado a invocar o construtor da próxima na hierarquia e assim por diante. O último construtor chamado nessa cadeia é o construtor da classe na base da hierarquia, cuja realidade termina de executar primeiro. O corpo do construtor da classe derivada só é executado depois que o construtor da classe básica inicializa os membros de dados da classe básica que o objeto da classe derivada herda. Por exemplo, a hierarquia `CommissionEmployee` → `BasePlusCommissionEmployee` → `Employee` é mostrada na Figura 12.20. Quando um programa chama um objeto da classe `BasePlusCommissionEmployee`, o construtor da classe `CommissionEmployee` é chamado. Visto que a classe `Employee` está na base da hierarquia, seu construtor executa, inicializando os membros de dados que fazem parte do objeto `Employee`. Quando o construtor da classe `CommissionEmployee` termina de executar, ele retorna o controle para o construtor da classe `BasePlusCommissionEmployee`, que inicializa o campo `salary` do objeto `BasePlusCommissionEmployee`.



Observação de engenharia de software 12.7

Quando um programa cria um objeto de classe derivada, o construtor da classe derivada chama imediatamente o construtor da classe básica, o corpo do construtor da classe básica executa, em seguida, os inicializadores de membro da classe derivada executam, e, por fim, o corpo do construtor da classe derivada executa. Esse processo coloca a hierarquia em cascata se ela contiver mais de dois níveis.

Quando um objeto de classe derivada é destruído, o programa chama o destrutor desse objeto. Isso inicia uma cadeia (ou cadeias) de chamadas de destrutor em que o destrutor da classe derivada, os destrutores das classes básicas diretas e indiretas, e os destrutores das classes executam na ordem inversa em que os construtores executaram. Quando o destrutor de um objeto de classe derivada é chamado, o destrutor realiza sua tarefa, então invoca o destrutor da próxima classe básica na hierarquia. Esse processo se repete até que o destrutor da classe básica final na parte superior da hierarquia é chamado. Então o objeto é removido da memória.



Observação de engenharia de software 12.8

Suponha que tivéssemos criado um objeto de uma classe derivada em que tanto a classe básica como a classe derivada contivessem objetos de outras classes. Quando um objeto dessa classe derivada é criado, os construtores para os objetos-membro da classe básica executam primeiro, em seguida, o construtor da classe básica, os construtores para os objetos de membro da classe derivada e o construtor da classe derivada executam, nessa ordem. Os destrutores para objetos de classe derivada são chamados na ordem inversa de seus construtores correspondentes.

Os construtores, destrutores e operadores de atribuição sobrecarregados de uma classe básica (ver Capítulo 11, “Sobre operadores; objetos string e array”) não são herdados por classes derivadas. Entretanto, os construtores, destrutores e operadores de atribuição sobrecarregados de uma classe derivada podem chamar construtores, destrutores e operadores de atribuição sobre da classe básica.

Nosso próximo exemplo revisita a hierarquia de empregados [comissionados definida nas figuras 12.22–12.23](#) e [baseada na figura 12.24–12.25](#) que contêm construtores e destrutores, cada um dos quais imprime uma mensagem quando é invocado. Como você verá na saída da Figura 12.26, essas mensagens demonstram a ordem dos construtores e destrutores são chamados por objetos em uma hierarquia de herança.

```

1 // Figura 12.22: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.

3 #ifndef COMMISSION_H
4
5 #include <string> // classe string padrão C++
6 using std::string;
7
8
9 class CommissionEmployee
10 {
11 public :
12     CommissionEmployee( const string &, const string &, const string &,
13                         double = 0.0, double = 0.0 );
14     ~CommissionEmployee(); // destrutor
15
16     void setFirstName( const string & ); // configura o nome
17     string getFirstName() const; // retorna o nome
18
19     void setLastName(const string &); // configura o sobrenome
20     string getLastname() const; // retorna o sobrenome
21
22     void setSocialSecurityNumber( const string & ); // configura o SSN
23     string getSocialSecurityNumber() const; // retorna o SSN
24
25     void setGrossSales( double ); // configura a quantidade de vendas brutas
26     double getGrossSales() const; // retorna a quantidade de vendas brutas
27
28     void setCommissionRate(double); // configura a taxa de comissão
29     double getCommissionRate() const; // retorna a taxa de comissão
30
31     double earnings() const; // calcula os rendimentos
32     void print() const; // imprime o objeto CommissionEmployee
33 private :
34     string firstName;
35     string lastName;
36     string socialSecurityNumber;
37     double grossSales; // vendas brutas semanais
38     double commissionRate; // porcentagem da comissão
39 }; // fim da classe CommissionEmployee
40
41 #endif

```

Figura 12.22 Arquivo de cabeçalho da classe CommissionEmployee

Neste exemplo, modificamos o construtor `CommissionEmployee` (linhas 10–21 da Figura 12.23) e incluímos um destrutor `missionEmployee` (linhas 24–29), cada um dos quais gera saída de uma linha de texto sobre sua invocação. Também modificamos o construtor `BasePlusCommissionEmployee` (linhas 11–22 da Figura 12.25) e incluímos um destrutor `missionEmployee` (linhas 25–30), cada um dos quais gera saída de uma linha de texto na sua invocação.

```

1 // Figura 12.23: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "CommissionEmployee.h" definição da classe CommissionEmployee
8
9 // construtor
10 CommissionEmployee::CommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate )
13     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
14 {
15     setGrossSales( sales ); // valida e armazena as vendas brutas
16     setCommissionRate( rate ); // valida e armazena a taxa de comissão
17
18     cout << "CommissionEmployee construtor: " << endl;
19     print();
20     cout << "\n\n" ;
21 } // fim do construtor CommissionEmployee
22
23 // destrutor
24 CommissionEmployee::~CommissionEmployee()
25 {
26     cout << "CommissionEmployee destrutor: " << endl;
27     print();
28     cout << "\n\n" ;
29 } // fim do destrutor CommissionEmployee
30
31 // configura o nome
32 void CommissionEmployee::setFirstName(const string &first )
33 {
34     firstName = first; // deve validar
35 } // fim da função setFirstName
36
37 // retorna o nome
38 string CommissionEmployee::getFirstName() const
39 {
40     return firstName;
41 } // fim da função getFirstName
42
43 //configura o sobrenome
44 void CommissionEmployee::setLastName(const string &last )
45 {
46     lastName = last; // deve validar
47 } // fim da função setLastName
48
49 // retorna o sobrenome
50 string CommissionEmployee::getLastName() const

```

Figura 12.23 O construtor de `CommissionEmployee` imprime texto.

(continua)

```

51  {
52      return lastName;
53  } // fim da função getLastName
54
55 // configura o SSN
56 void CommissionEmployee::setSocialSecurityNumber(inst string &ssn )
57 {
58     socialSecurityNumber = ssn; // deve validar
59 } // fim da função setSocialSecurityNumber
60
61 // retorna o SSN
62 string CommissionEmployee::getSocialSecurityNumber()const
63 {
64     return socialSecurityNumber;
65 } // fim da função getSocialSecurityNumber
66
67 // configura a quantidade de vendas brutas
68 void CommissionEmployee::setGrossSales(double sales )
69 {
70     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
71 } // fim da função setGrossSales
72
73 // retorna a quantidade de vendas brutas
74 double CommissionEmployee::getGrossSales()const
75 {
76     return grossSales;
77 } // fim da função getGrossSales
78
79 // configura a taxa de comissão
80 void CommissionEmployee::setCommissionRate(double rate )
81 {
82     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
83 } // fim da função setCommissionRate
84
85 // retorna a taxa de comissão
86 double CommissionEmployee::getCommissionRate()const
87 {
88     return commissionRate;
89 } // fim da função getCommissionRate
90
91 // calcula os rendimentos
92 double CommissionEmployee::earnings()const
93 {
94     return getCommissionRate() * getGrossSales();
95 } // fim da função earnings
96
97 // imprime o objeto CommissionEmployee
98 void CommissionEmployee::print()const
99 {
100    cout << getFirstName() << getLastname()
101    << "\nsocial security number: " << getSocialSecurityNumber()
102    << "\ngross sales: " << getGrossSales()
103    << "\ncommission rate: " << getCommissionRate();
104 } // fim da função print

```

Figura 12.23 O construtor de `CommissionEmployee` em texto.

(continuação)

```

1 // Figura 12.24: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee public CommissionEmployee
13 {
14 public :
15     BasePlusCommissionEmployee(const string &, const string &,
16         const string &, double = 0.0, double = 0.0, double = 0.0 );
17     ~BasePlusCommissionEmployee(); // destrutor
18
19     void setBaseSalary( double ); // configura o salário-base
20     double getBaseSalary() const; // retorna o salário-base
21
22     double earnings() const; // calcula os rendimentos
23     void print() const; // imprime o objeto BasePlusCommissionEmployee
24 private :
25     double baseSalary; // salário-base
26 }; // fim da classe BasePlusCommissionEmployee
27
28 #endif

```

Figura 12.24 Arquivo de cabeçalho da classe `BasePlusCommissionEmployee`

```

1 // Figura 12.25: BasePlusCommissionEmployee.cpp
2 // Definições de funções-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe BasePlusCommissionEmployee
8 #include "BasePlusCommissionEmployee.h"
9
10 // construtor
11 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
12     const string &first, const string &last, const string &ssn,
13     double sales, double rate, double salary )
14     // chama explicitamente o construtor da classe básica
15     : CommissionEmployee( first, last, ssn, sales, rate )
16
17 { setBaseSalary( salary ); // valida e armazena salário-base
18
19     cout << "BasePlusCommissionEmployee construtor:<< endl;
20
21     print();
22     cout << "\n\n" ;
23 } // fim do construtor BasePlusCommissionEmployee
24

```

Figura 12.25 O construtor de `BasePlusCommissionEmployee` exibe texto.

(continua)

```

24 // destrutor
25 BasePlusCommissionEmployee::~BasePlusCommissionEmployee()
26 {
27     cout << "BasePlusCommissionEmployee destrutor:<< endl;
28     print();
29     cout << "\n\n" ;
30 } // fim do destrutor BasePlusCommissionEmployee
31
32 // configura o salário-base
33 void BasePlusCommissionEmployee::setBaseSalary(double salary )
34 {
35     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
36 } // fim da função setBaseSalary
37
38 // retorna o salário-base
39 double BasePlusCommissionEmployee::getBaseSalary() const
40 {
41     return baseSalary;
42 } // fim da função getBaseSalary
43
44 // calcula os rendimentos
45 double BasePlusCommissionEmployee::earnings() const
46 {
47     return getBaseSalary() + CommissionEmployee::earnings();
48 } // fim da função earnings
49
50 // imprime o objeto BasePlusCommissionEmployee
51 void BasePlusCommissionEmployee::print() const
52 {
53     cout << "base-salaried " ;
54
55     // invoca a função print de CommissionEmployee
56     CommissionEmployee::print();
57
58     cout << "\nbbase salary: " << getBaseSalary();
59 } // fim da função print

```

Figura 12.25 O construtor de BasePlusCommissionEmployee em texto.

(continuação)

A Figura 12.26 demonstra a ordem em que construtores e destrutores são chamados para objetos de classes que fazem parte de uma hierarquia de herança. As linhas 15–34) começam instanciando o objeto `BasePlusCommissionEmployee employee` (linhas 21–22) em um bloco separado de escopo (linhas 20–23). O objeto entra e sai imediatamente do escopo (o fim do bloco é alcançado logo que o objeto é criado), então o construtor e o destrutor não são chamados. Em seguida, as linhas 26–27 instanciam o objeto `BasePlusCommissionEmployee employee`, que invoca o construtor `CommissionEmployee` para exibir as saídas com os valores passados a partir do construtor `BasePlusCommissionEmployee`. Então a saída especificada `BasePlusCommissionEmployee` é realizada. As linhas 30–31 instanciam o objeto `BasePlusCommissionEmployee employee`. Novamente, os construtores `BasePlusCommissionEmployee` e `CommissionEmployee` são ambos chamados. Observe que, em cada caso, o corpo do construtor

Este exemplo é só para demonstrar que os destrutores são chamados na ordem inversa de seus construtores correspondentes. A classe `CommissionEmployee` destrói seu objeto `CommissionEmployee` (nessa ordem) para o objeto `employee`, em seguida, os destrutores `BasePlusCommissionEmployee` e `CommissionEmployee` são chamados (nessa ordem) para o objeto `employee`.

```

1 // Figura 12.26: fig12_26.cpp
2 // Ordem de exibição em que a classe básica e construtores e destrutores
3 // da classe derivada são chamados.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // definição da classe BasePlusCommissionEmployee
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17     // configura a formatação de saída de ponto flutuante
18     cout << fixed << setprecision( 2 );
19
20     { // inicia novo escopo
21         CommissionEmployee employee1(
22             "Bob", "Lewis", "333-33-3333", 5000 .04 );
23     } // termina o escopo
24
25     cout << endl;
26     BasePlusCommissionEmployee
27         employee2("Lisa", "Jones", "555-55-5555", 2000 .06, 800);
28
29     cout << endl;
30     BasePlusCommissionEmployee
31         employee3("Mark", "Sands", "888-88-8888", 8000 .15, 2000);
32
33     cout << endl;
34 } // fim de main

```

CommissionEmployee constructor:

commission employee: Bob Lewis

social security number: 333-33-3333

gross sales: 5000.00

commission rate: 0.04

CommissionEmployee destructor:

commission employee: Bob Lewis

social security number: 333-33-3333

gross sales: 5000.00

commission rate: 0.04

CommissionEmployee constructor:

base-salaried commission employee: Lisa Jones

social security number: 555-55-5555

gross sales: 2000.00

commission rate: 0.06

BasePlusCommissionEmployee constructor:

base-salaried commission employee: Lisa Jones

social security number: 555-55-5555

Figura 12. Ordem de chamada do construtor e do destrutor.

(continua)

gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

```
CommissionEmployee constructor:  
commission employee: Mark Sands  
social security number: 888-88-8888  
gross sales: 8000.00  
commission rate: 0.15
```

```
BasePlusCommissionEmployee constructor:  
base-salaried commission employee: Mark Sands  
social security number: 888-88-8888  
gross sales: 8000.00  
commission rate: 0.15  
base salary: 2000.00
```

```
BasePlusCommissionEmployee destructor:  
base-salaried commission employee: Mark Sands  
social security number: 888-88-8888  
gross sales: 8000.00  
commission rate: 0.15  
base salary: 2000.00
```

```
CommissionEmployee destructor:  
commission employee: Mark Sands  
social security number: 888-88-8888  
gross sales: 8000.00  
commission rate: 0.15
```

```
BasePlusCommissionEmployee destructor:  
base-salaried commission employee: Lisa Jones  
social security number: 555-55-5555  
gross sales: 2000.00  
commission rate: 0.06  
base salary: 800.00
```

```
CommissionEmployee destructor:  
commission employee: Lisa Jones  
social security number: 555-55-5555  
gross sales: 2000.00  
commission rate: 0.06
```

Figura 12.20 Ordem de chamada do construtor e do destrutor.

(continuação)

| Especificador de acesso de membro de classe básica | Tipo de herança | | |
|--|---|---|---|
| | Herança public | Herança protected | Herança private |
| public | public na classe derivada.
Pode ser acessada diretamente por funções-membro, funções friend e funções não-membro. | protected na classe derivada.
Pode ser acessada diretamente por funções-membro e função friend . | private na classe derivada.
Pode ser acessada diretamente por funções-membro e função friend . |
| protected | protected na classe derivada.
Pode ser acessada diretamente por funções-membro e função friend . | protected na classe derivada.
Pode ser acessada diretamente por funções-membro e função friend . | private na classe derivada.
Pode ser acessada diretamente por funções-membro e função friend . |
| private | Oculta na classe derivada.
Pode ser acessada por funções-membro e funções friend por meio das funções-membro public ou protected da classe básica. | Oculta na classe derivada.
Pode ser acessada por funções-membro e funções friend por meio das funções-membro public ou protected da classe básica. | Oculta na classe derivada.
Pode ser acessada por funções-membro e funções friend por meio das funções-membro public ou protected da classe básica. |

Figura 12.27 Resumo da acessibilidade do membro de classe básica em uma classe derivada.

Ao derivar de uma classe básica membros protected da classe básica tornam-se os membros da classe derivada. Ao derivar de uma classe básica membros protected da classe básica tornam-se os membros private (por exemplo, as funções tornam-se funções utilitárias) da classe derivada. Assim, a classe derivada

12.7 Engenharia de software com herança

Nesta seção, discutimos o uso de herança para personalizar o software existente. Quando utilizamos herança para criar uma nova classe derivada, a nova classe herda os membros de dados e funções-membro da classe existente, como descrito na Figura 12.27. Podemos personalizar a nova classe para atender às nossas necessidades incluindo membros adicionais e redefinindo membros existentes. O programador de classe derivada faz isso em C++ sem acessar o código-fonte da classe básica. A classe derivada é capaz de se linkar ao código-objeto da classe básica. Essa poderosa capacidade é atraente para fornecedores de softwares independentes (ISVs). Os ISVs podem desenvolver classes 'proprietárias' (patenteadas) para venda ou licenciamento e disponibilizá-las para usuários no formato de código-objeto. Os usuários podem então derivar novas classes dessas classes de forma rápida e sem acessar o código-fonte proprietário (patenteado) dos ISV. Tudo o que os ISV precisam fornecer ao código-fonte é o arquivo de cabeçalho.

As vezes é difícil para os alunos avaliarem o escopo de problemas enfrentados por projetistas que trabalham em projetos de software de larga escala na indústria. Pessoas experientes nesses projetos dizem que a reutilização efetiva de software melhora o desenvolvimento de software. A programação orientada a objetos facilita a reutilização de software, diminuindo, assim, o tempo de desenvolvimento e aprimorando a qualidade do software.

A disponibilidade de bibliotecas de classe substanciais e úteis fornece os benefícios máximos de reutilização de software por meio da herança. Assim, como o software comercial produzido por fornecedores de software independentes tornou-se uma indústria de software explosivo com a chegada do computador pessoal, o interesse pela criação e venda de bibliotecas de classes também está exponencialmente. Os projetistas de aplicativo constroem seus aplicativos com essas bibliotecas e projetistas de biblioteca estão recompensados por ter suas bibliotecas incluídas nos aplicativos. As bibliotecas C++ padrão que vêm com compiladores C++ têm sido de uso mais geral e limitado em escopo. Entretanto, há um compromisso mundial maciço para o desenvolvimento de bibliotecas para uma variedade enorme de áreas de aplicação.



Observação de engenharia de software 12.9

Na etapa de projeto em um sistema orientado a objetos, o projetista freqüentemente determina que certas classes estão intimamente relacionadas. O projetista deve ‘fatorar’ atributos e comportamentos comuns e colocá-los em uma classe básica, depois utilizar a herança para formar classes derivadas, provendo-as de outras capacidades além das herdadas da classe básica.



Observação de engenharia de software 12.10

A criação de uma classe derivada não afeta o código-fonte da sua classe básica. A herança preserva a integridade de uma classe básica.



Observação de engenharia de software 12.11

Assim como os projetistas de sistemas não orientados a objetos devem evitar a proliferação de funções, os projetistas de sistemas orientados a objetos devem evitar proliferação de classes. A proliferação de classes cria problemas de gerenciamento e pode impedir a reusabilidade de software, porque se torna difícil para um cliente localizar a classe mais apropriada em uma enorme biblioteca de classes. A alternativa é criar menos classes que fornecem funcionalidades mais substanciais, mas essas classes talvez fornecem funcionalidades demais.



Dica de desempenho 12.3

Se as classes produzidas por herança forem maiores do que o necessário (isto é, contiverem funcionalidade demais), os recursos de memória e processamento podem ser desperdiçados. Herde da classe cujas funcionalidades estão ‘mais próximas’ daquilo que é necessário.

A leitura de definições de classes derivadas pode ser confusa, porque os membros herdados não são mostrados fisicamente na classe derivada, muito embora estejam presentes. Existe um problema semelhante ao documentar membros de classe derivada.

12.8 Síntese

Este capítulo introduziu a herança — a capacidade de criar uma classe absorvendo membros de dados e funções-membro de uma classe existente e aprimorando-os com novas capacidades. Por meio de uma série de exemplos que utilizam uma hierarquia de herança, você aprendeu a entender a classe básica, a classe derivada e a herança direta, indireta e múltipla. Você também aprendeu a utilizar a herança para proteger membros de classe.

Na herança, a classe existente é chamada de classe básica e a nova classe é referida como classe derivada. A classe derivada pode acessar os membros da classe básica. Você aprendeu a acessar membros de classe básica redefinidos qualificando seus nomes com o nome da classe básica e o operador de resolução de escopo `super` (em que construtores e destrutores são chamados por objetos de classes que fazem parte de uma hierarquia de herança). Por fim, existem três tipos de herança — `protected`, `private` — e a acessibilidade de membros de classe básica em uma classe derivada ao utilizar cada tipo.

No Capítulo 13, “Programação orientada a objetos: polimorfismo”, avançamos nossa discussão de herança introduzindo o polimorfismo — um conceito orientado a objetos que permite escrever programas que tratam, de uma maneira mais geral, objetos de ampla variedade de classes relacionadas por herança. Depois de estudar o Capítulo 13, você estará familiarizado com classes, encapsulamento, herança e polimorfismo — os aspectos essenciais da programação orientada a objetos.

Resumo

- A reutilização de software reduz o tempo e o custo de desenvolvimento de programas.
- A herança é uma forma de reutilização de software em que o programador cria uma classe que absorve dados e comportamentos de uma classe existente e os aprimora com novas capacidades. A classe existente é chamada de classe básica e a nova classe é referida como classe derivada.
- Uma classe básica direta é aquela a partir da qual uma classe derivada explicitamente herda (especificado pelo nome de classe à direita da primeira linha de uma definição de classe). Uma classe básica indireta é herdada de dois ou mais níveis acima na hierarquia de classes.
- Com a herança simples, uma classe é derivada de uma classe básica. Com a herança múltipla, uma classe herda de múltiplas classes (possivelmente não relacionadas).
- Uma classe derivada representa um grupo mais especializado de objetos. Em geral, uma classe derivada contém comportamentos herdados da classe básica mais comportamentos adicionais. Uma classe derivada também pode personalizar comportamentos herdados da classe básica.
- Cada objeto de uma classe derivada é também um objeto da classe básica dessa classe. Entretanto, um objeto de classe básica não é um objeto da classe derivada.

- O relacionamento **herança** representa a herança. Em um **relacionamento** de uma classe derivada também pode ser tratado como um objeto de sua classe básica.
- O relacionamento **composição** representa a composição. Em um **relacionamento** contém um ou mais objetos de outras classes como membros, mas não expõe seu comportamento diretamente em sua interface.
- Uma classe derivada não pode diretamente **acessar os membros** da classe básica; permitir isso violaria o encapsulamento da classe básica. Mas uma classe derivada pode **acessar os membros** de sua classe básica diretamente.
- Uma classe derivada pode produzir alterações de estado **em membros da classe básica**, que não são fornecidas na classe básica e herdadas na classe derivada.
- Quando uma função-membro de classe básica é inadequada a uma classe derivada, ela pode ser redefinida na classe derivada com uma implementação apropriada.
- Os relacionamentos de herança simples formam estruturas hierárquicas do tipo árvore — existe uma classe básica em um relacionamento ráclico com suas classes derivadas.
- É possível tratar objetos de classe básica e de classe derivada de modo semelhante; os aspectos comuns compartilhados entre os tipos são expressos nos membros de dados e funções-membro da classe básica.
- Os membros **privé** de uma classe básica são acessíveis em qualquer lugar que o programa tiver um handle para um objeto dessa classe básica ou para um objeto de uma das classes derivadas dessa classe básica — ou, ao utilizar o operador de resolução de escopo binário, sempre nome da classe estiver no escopo.
- Os membros **protegido** de uma classe básica só são acessíveis dentro da definição dessa classe básica ou de amigas dessa classe.
- Os membros **public** de uma classe básica têm um nível intermediário de proteção entre o escopo local e o escopo global. Os membros **protected** de uma classe básica podem ser acessados por membros e amigos dessa classe básica e por membros e amigos de qualquer classe que herde essa classe básica.
- Infelizmente, os membros **de dados** apresentam com freqüência dois problemas importantes. Primeiro, o objeto de classe derivada não tem de utilizar uma maneira de alterar o valor **dos dados** da classe básica. Segundo, é muito provável que as funções-membro da classe derivada dependam de detalhes de implementação da classe básica.
- Quando uma função-membro da classe derivada redefine uma função-membro da classe básica, a função-membro da classe básica pode ser acessada a partir da classe derivada qualificando o nome da função-membro da classe básica com o nome da classe básica e o operador de resolução de escopo binário (
- Quando um objeto de uma classe derivada é instanciado, o construtor da classe básica é chamado imediatamente (explícita ou implicitamente) para inicializar os membros de dados da classe básica no objeto da classe derivada (antes de os membros de dados da classe derivada serem inicializados).
- Declarar membros **de dados** ao mesmo tempo fornecer funções para manipular e realizar a verificação de validação nesses dados, impõe boa engenharia de software.
- Quando um objeto de classe derivada é destruído, os destrutores são chamados na ordem inversa dos construtores — primeiro o destrutor da classe derivada e, depois, o destrutor da classe básica.
- Ao derivar uma classe de uma classe básica, a classe básica pode ser declarada como **parente qualificado**.
- Ao derivar uma classe de **uma classe**, os membros da classe básica tornam-se **subordinados** à classe derivada, e os membros **protected** da classe básica tornam-se **protegidos** na classe derivada.
- Ao derivar uma classe de **uma classe**, os membros **protected** da classe básica tornam-se **protegidos** membros da classe derivada.
- Ao derivar uma classe de **uma classe**, os membros **protected** da classe básica tornam-se **subordinados** à classe derivada.

Terminologia

| | | |
|------------------------------------|---|----------------------------------|
| classe básica | destrutor de classe derivada | inicializador de classe básica |
| classe básica direta | é um relacionamento | parente qualificado |
| classe derivada | friend de uma classe derivada | private , classe básica |
| composição | herança | private , herança |
| construtor de classe básica | herança múltipla | protected, classe básica |
| construtor de classe derivada | herança única | protected, herança |
| construtor-padrão de classe básica | herdar os membros de uma classe existente | protected, membro, de uma classe |
| destrutor de classe básica | hierarquia de classes | protected, palavra-chave |

| | | |
|--|----------------------------|--------------------|
| public , classe básica | relacionamento hierárquico | subclasse |
| public , herança | software frágil | superclasse |
| redefinir uma função-membro de classe básica quebradiço | | tem relacionamento |

Exercícios de revisão

12.1 Preencha as lacunas em cada uma das seguintes sentenças:

- a) _____ é uma forma de reutilização de software em que novas classes absorvem os dados e comportamentos de classes existentes e aprimoram essas classes com novas capacidades.
 - b) Os membros _____ de uma classe básica podem ser acessados somente na definição de classe básica ou nas definições de classe derivada.
 - c) Em um relacionamento _____, um objeto de uma classe derivada também pode ser tratado como um objeto de sua classe básica.
 - d) Em um relacionamento _____, um objeto de classe tem um ou mais objetos de outras classes como membros.
 - e) Na herança simples, uma classe existe em um relacionamento _____ com suas classes derivadas.
 - f) Os membros _____ de uma classe básica são acessíveis dentro dessa classe básica e em qualquer lugar que o programa tenha handle para um objeto dessa classe básica ou para um objeto de uma de suas classes derivadas.
 - g) Os membros _____ de uma classe básica têm um nível de proteção _____ queões de acesso
 - h) O C++ oferece _____, que permite a uma classe derivada herdar de muitas classes básicas, mesmo se essas classes não estiverem relacionadas.
 - i) Quando um objeto de uma classe derivada é instanciado, o _____ da classe básica é chamado implícita ou explicitamente para fazer qualquer inicialização necessária dos membros de dados de classe básica no objeto de classe derivada.
 - j) Ao derivar uma classe de uma classe básica ~~correspondente~~, os membros _____ da classe básica tornam-se os membros _____ da classe derivada, e os membros _____ da classe básica tornam-se os membros _____ da classe derivada.
 - k) Ao derivar uma classe de uma classe básica ~~correspondente~~, os membros _____ da classe básica tornam-se os membros _____ da classe derivada, e os membros _____ da classe básica tornam-se os membros _____ da classe derivada.
- 12.2 Determine se cada uma das seguintes afirmações é verdadeira ou falsa e explique por quê.
- a) Os construtores de classe básica não são herdados por classes derivadas.
 - b) Um relacionamento _____ é implementado via herança.
 - c) Uma classe _____ tem um relacionamento _____ com a classe Volante.
 - d) A herança estimula a reutilização de software de alta qualidade comprovada.
 - e) Quando um objeto de classe derivada é destruído, os destrutores são chamados na ordem inversa dos construtores.

Respostas dos exercícios de revisão

12.1 a) Herança.protected . c)é ou herança.todou ou composição ou agregação. e) hierárquica f) private . h) herança múltipla. i) constructor.protected. k)protected,protected.

12.2 a) Verdadeira. b) Falsa. Um relacionamento _____ é implementado via composição. Um relacionamento _____ é implementado via herança. c) Falsa. Esse é um exemplo de _____ que tem um relacionamento _____ com a classe Veículo. d) Verdadeira. e) Verdadeira.

Exercícios

12.3 Muitos programas escritos com herança podem ser escritos com composição. Responda: Por que? Responda: Porque a classe ployee da hierarquia _____ pode utilizar composição em vez de herança. Depois de fazer isso, avalie os méritos relativos das duas abordagens para programar a classe CommissionEmployee. Eles são diferentes?

12.4 Discuta de que maneira a herança promove a reutilização de software, economiza tempo durante o desenvolvimento de programa e a evita erros.

12.5 Alguns programadores preferem não utilizar herança porque acreditam que ele quebra o encapsulamento da classe básica. Discuta os méritos relativos a esse aspecto.

12.6 Desenhe uma hierarquia de herança para alunos universitários semelhante à hierarquia mostrada na Figura 12.2. Utilize a classe básica da hierarquia, então _____ que derivam de _____ Continue a estender a hierarquia o mais profundamente (isto é, com muitos níveis). _____ que _____ poderiam derivar de _____ e _____ poderiam derivar de _____ Depois de desenhar a hierarquia, discuta os relacionamentos entre as classes [não escrever nenhum código para este exercício.]

- 12.7** O mundo das formas é muito mais rico que as formas incluídas na hierarquia de herança da Figura 12.3. Anote todas as formas que você imagina — bidimensionais e tridimensionais — e as formas que possam ser criadas com o maior número de níveis possíveis. Sua hierarquia deve ter ~~uma classe base~~ qual arquitetura? [Nota: Você não precisa escrever nenhum código para este exercício.] Utilizaremos essa hierarquia nos exercícios do Capítulo 13 para processar um conjunto de formas distintas como ~~classe base~~ polimorfismo, é o assunto do Capítulo 13.)
- 12.8** Desenhe uma hierarquia de herança ~~com classes~~ que possa representar um quadrado, um retângulo e um paralelogramo. Utilize Quadrilátero como a classe básica da hierarquia. Tome a hierarquia o mais profunda possível.
- 12.9** (Hierarquia de herança) Os serviços de correio expresso ~~oferecem~~ oferecem várias opções de entrega, cada qual com custos específicos. Crie uma hierarquia de herança para representar ~~varias~~ tipos de pacotes. Utilize classe básica da hierarquia, então ~~incluir~~ ~~criar~~ classes para derivar. A classe básica Package deve incluir membros de dados que representam nome, endereço, cidade, estado e CEP tanto do remetente quanto do destinatário. Po pacote, além dos membros de dados que armazem o peso (em quilos), o custo por quilograma para a entrega do pacote. O construtor deve inicializar esse membro de dados. Assure que esse o peso (em quilos) e o custo por quilograma para a entrega do pacote. O construtor deve fornecer uma função ~~membro~~ calculateCost que retorna o custo associado com a entrega do pacote. A função calculateCost deve determinar o custo multiplicando o peso pelo custo (em quilos). A classe derivada TwoDayPackage deve herdar a funcionalidade da classe básica e também incluir um membro de dados que representa uma taxa fixa que a empresa de entrega cobra pelo serviço de entrega de dois dias. O construtor deve receber um valor para inicializar esse membro. A classe TwoDayPackage deve redefinir a função calculateCost para que ela calcule o custo de entrega adicionando a taxa fixa ao custo baseado em peso ~~calculado~~ do pacote. A classe OvernightPackage deve herdar diretamente de TwoDayPackage e conter um membro de dados adicional para representar uma taxa adicional por quilo cobrada pelo serviço de entrega noturna. A classe OvernightPackage deve redefinir a função calculateCost para que ela acrescente a taxa adicional por quilo ao custo-padrão por quilo antes de calcular o custo da entrega. Escreva um programa de teste que cria objetos de todas as classes e a função-membro calculateCost.
- 12.10** (Hierarquia de herança) Crie uma hierarquia de herança que um banco possa utilizar para representar as contas bancárias dos clientes. Todos os clientes nesse banco podem depositar (isto é, creditar) dinheiro em suas contas e retirar (isto é, debitar) o dinheiro delas. Há também tipos mais específicos de contas. As contas de poupança, por exemplo, recebem juros pelo dinheiro depositado nelas. As contas bancárias, por outro lado, cobram uma taxa por transação (isto é, crédito ou débito).
- Crie uma hierarquia de herança contendo classes ~~de dados~~ de contas. A classe Account deve herdar da classe ~~básica~~ BankAccount que herda da classe ~~base~~ Account. A classe BankAccount deve incluir um membro de dados ~~double~~ para representar o saldo da conta. A classe deve fornecer um construtor que recebe um saldo inicial e o utiliza para inicializar o membro de dados. O construtor deve validar o saldo inicial para assegurar que ele é maior que zero. O saldo deve ser configurado no construtor. A classe deve exibir uma mensagem de erro, indicando que o saldo inicial era inválido. A classe deve fornecer três funções-membro. A função-membro ~~deposit~~ deve adicionar uma quantia ao saldo atual. A função-membro ~~withdraw~~ deve garantir que o valor do débito não exceda o saldo. Se o débito exceder, o saldo deve permanecer inalterado e a função deve imprimir a mensagem "Debit amount exceeded account balance". A função-membro ~~balance~~ deve retornar o saldo atual.
- A classe derivada SavingsAccount deve herdar a funcionalidade de Account e também incluir um membro de dados do tipo double para indicar a taxa de juros (porcentagem) anual. A classe SavingsAccount deve receber o saldo inicial, bem como um valor inicial para a taxa de juros. A classe SavingsAccount deve fornecer uma função-membro calculateInterest que retorna double para indicar os juros auferidos por uma conta. A função-membro deve determinar esse valor multiplicando a taxa de juros pela taxa de juros da conta. [herdar as funções-membro credit e debit exatamente como são sem redefini-las.]
- A classe derivada CheckingAccount deve herdar da classe ~~básica~~ BankAccount e incluir um membro adicional de ~~dados~~ do tipo double que representa a taxa cobrada por transações. A classe CheckingAccount deve receber o saldo inicial, bem como um parâmetro que indica o valor de uma taxa de juros. A classe CheckingAccount deve redefinir as funções-membro ~~deposit~~ e ~~withdraw~~ para que subtraiam a taxa do saldo da conta sempre que qualquer uma das transações for realizada. As funções devem invocar a função-membro ~~withdraw~~ da classe ~~básica~~ BankAccount para realizar a subtração do saldo da conta, o valor da taxa de juros exceder ao do saldo da conta). [Defina a função ~~deposit~~ para que ela retorne o valor indicando se houve retirada de dinheiro. Em seguida, utilize o valor de retorno para determinar se uma taxa deve ser cobrada.]
- Depois de definir as classes nessa hierarquia, escreva um programa que cria objetos de cada classe e testa suas funções-membro. Adicione os juros ao objeto SavingsAccount invocando primeiro sua função ~~interest~~ e, então, passando o valor retornado dos juros para a função ~~deposit~~ do objeto.



Um Anel para todos governar,
Um Anel para encontrá-los,
Um Anel para todos reunir e na
escuridão aprisioná-los.

John Ronald Reuel Tolkien
O silêncio muitas vezes de pura
inocência
Convence quando a fala falha.
William Shakespeare

Propostas genéricas não
decidem casos concretos.
Oliver Wendell Holmes

Um filósofo de estatura
imponente não pensa em um
vazio. Mesmo suas idéias mais
abstratas são, em alguma
medida, condicionadas pelo que
é ou não conhecido na época
em que ele vive.

Alfred North Whitehead

Programação orientada a objetos: polimorfismo

OBJETIVOS

Neste capítulo, você aprenderá:

O que é polimorfismo, como ele torna a programação mais conveniente e os sistemas mais extensíveis e sustentáveis.

A declarar e utilizar funções virtuais para produzir polimorfismo.

A distinguir entre classes abstratas e classes concretas.

A declarar funções virtuais puras para criar classes abstratas.

A utilizar informações de tipo em tempo de execução (Time Type Information RTTI) com downcasting, dynamic_cast, typeid e type_info.

Gramática C++ implementa as funções virtuais e a

A utilizar os destrutores virtuais para assegurar que todos os destrutores apropriados executem em um objeto.

| | | |
|---|--------|--|
| P | 13.1 | Introdução |
| á | 13.2 | Exemplos de polimorfismo |
| m | 13.3 | Relacionamentos entre objetos em uma hierarquia de herança |
| | 13.3.1 | Invocando funções de classe básica a partir de objetos de classe derivada |
| | 13.3.2 | Apontando ponteiros de classe derivada para objetos da classe básica |
| | 13.3.3 | Chamadas de função-membro de classe derivada via ponteiros de classe básica |
| | 13.3.4 | Funções virtuais |
| | 13.3.5 | Resumo das atribuições permitidas entre objetos de classe básica e de classe derivada e ponteiros |
| S | 13.4 | Campos de tipo e instruções switch |
| | 13.5 | Classes abstratas e funções virtual puras |
| | 13.6 | Estudo de caso: sistema de folha de pagamento utilizando polimorfismo |
| | 13.6.1 | Criando a classe básica abstrata Employee |
| | 13.6.2 | Criando a classe derivada concreta SalariedEmployee |
| | 13.6.3 | Criando a classe derivada concreta HourlyEmployee |
| | 13.6.4 | Criando a classe derivada concreta CommissionEmployee |
| | 13.6.5 | Criando a classe derivada concreta indireta BasePlusCommissionEmployee |
| | 13.6.6 | Demonstrando o processamento polimórfico |
| | 13.7 | Polimorfismo, funções virtual e vinculação dinâmica ‘sob o capô’ (opcional) |
| | 13.8 | Estudo de caso: sistema de folha de pagamento utilizando polimorfismo e informações de tipo em tempo de execução com downcasting, dynamic_cast<typeid> e type_info |
| | 13.9 | Destrutores virtuais |
| | 13.10 | Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional) |
| | 13.11 | Síntese |

[Resumo](#) | [Terminologia](#) | [Exercícios de revisão](#) | [Respostas dos exercícios de revisão](#) | [Exercícios](#)

13.1 Introdução

Nos capítulos 9–12, discutimos as tecnologias-chave de programação orientada a objetos incluindo classes, objetos, encapsular sobrecarga de operadores e herança. Agora continuamos nosso estudo de [Polimorfismo](#) e demonstrando o uso de heranças. O polimorfismo permite ‘programar no geral’ em vez de ‘programar no específico’. Em particular, o polimorfismo permite escrever programas que processam objetos de classes que fazem parte da mesma hierarquia de classes como se todos os objetos da classe básica da hierarquia. Como veremos em breve, o polimorfismo se livra dos handles de ponteiro de classe básica handles de referência de classe básica, mas não dos handles de nome.

Considere o exemplo de polimorfismo a seguir. Suponha que criamos um programa que simula o movimento de vários tipos de animais para um estudo biológico. As classes `Animal`, `Passaro` e `Reptil` representam os três tipos de animais sob investigação. Imagine que cada uma dessas classes herda da classe básica `Animal` uma função que mantém a localização atual de um animal. Toda classe derivada implementa a função `move`. Nossa programa manteve os ponteiros para objetos das várias classes derivadas de `Animal`. Para simular os movimentos dos animais, o programa envia a mesma mensagem a cada objeto uma vez por segundo — a saber, `move`. Entretanto, cada tipo específico responde a uma mensagem de maneira própria e única — um passaro poderia pular três metros, enquanto um réptil — um tartaruga — poderia nadar dez metros. O programa emite a mesma mensagem (`move`) para cada objeto animal genericamente, mas cada objeto sabe como modificar sua posição apropriadamente de acordo com seu tipo de movimento específico. Contar com o fato de que cada objeto sabe ‘fazer a coisa certa’ (isto é, faz o que é apropriado para esse tipo de objeto) em resposta à mesma chamada de método é o conceito-chave do polimorfismo. A mesma mensagem (nessa caso, `move`) enviada a uma variedade de objetos tem muitas formas de resultados — daí o termo polimorfismo.

Com o polimorfismo podemos projetar e implementar sistemas que são facilmente extensíveis — novas classes podem ser adicionadas com pouca ou nenhuma modificação a partes gerais do programa, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que o programador adiciona à hierarquia. Por exemplo, se criarmos a classe `Tartaruga` que herda da classe `Animal` (que poderia responder a uma mensagem `move` andando cinco centímetros), precisaremos escrever somente a parte da simulação que instanciaria a classe `Tartaruga`. As partes da simulação que processam cada tipo de animal genericamente podem permanecer idênticas.

Começamos com uma seqüência de pequenos exemplos focalizados que levam a um entendimento das funções vinculação dinâmica — duas tecnologias subjacentes do polimorfismo. Então apresentamos um estudo de caso que revê a hierarquia Employee do Capítulo 12. No estudo de caso, definimos uma ‘interface’ comum (isto é, um conjunto de funcionalidades) para todas classes na hierarquia. Essas funcionalidades comuns entre empregados são definidas na classe básica abstrata, qual as classes `BaseEmployee`, `HourlyEmployee` e `CommissionEmployee` herdam diretamente, `BaseEmployee` é a classe base.

Nessa hierarquia, cada empregado tem uma função particular o salário semanal do empregado. Essas funções variam por tipo de empregado — por exemplo, se recebe um salário semanal fixo independente do número de horas trabalhadas, enquanto os são pagos por hora e recebem pagamento de horas extras. Mostramos como processar cada empregado ‘de maneira mais geral’ — isto é, utilizando ponteiros de classe básica para obter essa função de classe derivada. Desse modo, o programador precisa se preocupar apenas com um tipo de chamada de função, que pode ser executado para executar várias funções diferentes com base nos objetos referenciados pelos ponteiros de classe básica.

Um recurso-chave deste capítulo é sua discussão detalhada (opcional) de polimorfismo dinâmico.

‘soO caso principal é utilizar realizações de trabalho para empregar o polimorfismo para operações específicas que precisam ser realizadas operações em um tipo de objeto específico em uma hierarquia — a operação não pode ser aplicada de maneira genérica a vários tipos de objetos. Reutilizamos nessa seção para demonstrar as poderosas capacidades de tempo de execução (run-time type information RTTI) e a execução dinâmica que permite a um programa determinar o tipo de objeto em tempo de execução e agir sobre esse objeto de maneira correspondente. Utilizamos essas capacidades para determinar se o empregado particular é um `CommissionEmployee`. Se é, então damos para esse empregado um bônus de 10% em seu salário-base.

13.2 Exemplos de polimorfismo

Nesta seção, discutimos vários exemplos de polimorfismo. Com polimorfismo, uma função pode produzir a ocorrência de ações diferentes, dependendo do tipo do objeto em que a função é invocada. Isso fornece tremenda capacidade expressiva para o programador. Se a classe é derivada da classe `Quadrilatero`, então um objeto `Quadrilatero` é uma versão mais específica de um objeto `Quadrilatero`. Portanto, qualquer operação (como calcular o perímetro ou a área) que pode ser realizada em um objeto da classe `Quadrilatero` também pode ser realizada em um objeto da classe `Triangulo`. As operações também podem ser realizadas em outros tipos de quadriláteros, como `Quadrado`, `Paralelogramo` e `Trapezoides`. O polimorfismo ocorre quando um programa invoca uma função atual por meio de um ponteiro de classe básica (isto é, referência — o C++ escolhe dinamicamente (isto é, em tempo de execução) a função correta para a classe a partir da qual o objeto foi instanciado. Veremos um exemplo de código que ilustra esse processo na Seção 13.3.

Como outro exemplo, suponha que vamos projetar um videogame que manipula objetos de vários tipos diferentes, incluindo os das classes `Mercuriano`, `Venustiano`, `NaveEspacial`, `CañaoDeLaser`. Imagine que cada uma dessas classes herda da classe básica `ObjetoEspacial` que contém a função `move`. Toda classe derivada implementa essa função de maneira apropriada a essa classe. Um programa gerenciador de tela mantém um `contêiner` (ou seja, um `ponteiro`) de `ObjetoEspacial` para objetos das várias classes. Para atualizar a tela, o gerenciador de tela envia periodicamente a mesma mensagem a cada objeto — `move`. Cada tipo de objeto responde de maneira única. Por exemplo, um `Mercuriano` se move no mesmo em vermelho com o número apropriado de velocidade. Ele se move para si mesmo como um disco voador brilhante. Um `ObjetoDelas` poderia se desenhar como um feixe vermelho brilhante através da tela. Mais uma vez, a mesma mensagem (neste caso, enviada a uma variedade de objetos tem ‘muitas formas’ de resultados).

Um gerenciador de tela polimórfico facilita adicionar novas classes a um sistema com modificações mínimas no seu código. Basta que quiséssemos adicionar objetos da classe `Mercuriano` ao nosso videogame. Para fazer isso, devemos construir uma classe `Mercuriano` que herda de `ObjetoEspacial` mas fornece sua própria definição de função `move`. Entretanto, quando ponteiros para objetos da classe `Mercuriano` aparecem no contêiner, o programador não precisa modificar o código para o gerenciador de tela. O gerenciador de tela invoca a função `move` de cada objeto no contêiner, independentemente do tipo de objeto, então os novos objetos `Mercuriano` simplesmente se ‘conectam’. Portanto, sem modificar o sistema (além de construir e incluir as próprias classes), os programadores podem utilizar o polimorfismo para acomodar classes adicionais, inclusive aquelas que nem mesmo foram consideradas quando o sistema foi criado.



Observação de engenharia de software 13.1

Com as funções virtuais e o polimorfismo, você pode tratar generalidades e deixar a questão do ambiente de tempo de execução em si para as especificidades. Você pode instruir uma variedade de objetos a se comportar de maneiras apropriadas a esses objetos sem mesmo conhecer seus tipos (contanto que esses objetos pertençam à mesma hierarquia de herança e estejam sendo acessados a partir de um ponteiro de classe básica comum).



Observação de engenharia de software 13.2

O polimorfismo promove extensibilidade: o software escrito para invocar comportamento polimórfico é escrito independentemente dos tipos dos objetos para os quais as mensagens são enviadas. Portanto, novos tipos de objetos que podem responder a mensagens existentes podem ser incorporados nesse sistema sem modificar o sistema de base. Somente o código de cliente que instancia os novos objetos deve ser modificado para acomodar os novos tipos.

13.3 Relacionamentos entre objetos em uma hierarquia de herança

A Seção 12.4 criou uma hierarquia de classes de empregados com que a classe-padrão da classe `Employee`. Os exemplos do Capítulo 12 manipulam os objetos `Employee`, `BaseCommissionEmployee` e `PlusCommissionEmployee`, utilizando os nomes dos objetos para invocar suas funções-membro. Agora examinamos os relacionamentos entre classes em uma hierarquia rigorosamente. As próximas várias seções apresentam uma série de exemplos que demonstram como ponteiros das classes básicas e derivadas podem ser apontados para objetos das classes básicas e derivadas e como esses ponteiros podem ser utilizados para funções-membro que manipulam esses objetos. Perto do final desta seção, demonstramos como obter o comportamento polimórfico de ponteiros das classes básicas apontados para objetos de classes derivadas.

Na Seção 13.3.1 atribuímos o endereço de um objeto de classe derivada a um ponteiro de classe básica, então mostramos invocar uma função via ponteiro de classe básica invoca as funcionalidades da classe básica — isto é, o tipo do handle determina que função é chamada. Na Seção 13.3.2, atribuímos o endereço de um objeto de classe básica a um ponteiro de classe derivada; resulta em um erro de compilação. Discutimos a mensagem de erro e investigamos por que o compilador não permite essa atribuição. Na Seção 13.3.3, atribuímos o endereço de um objeto de classe derivada a um ponteiro de classe básica, depois examinamos como o ponteiro de classe derivada pelo ponteiro de classe básica, ocorrem erros de compilação. Por fim, na Seção 13.3.4, introduzimos as funções-membro `getBaseSalary()` e `getTotalSalary()` declarando uma função de classe `base` e apontando um objeto de classe derivada ao ponteiro de classe básica e utilizamos esse ponteiro para invocar as funcionalidades da classe derivada — exatamente a capacidade que precisamos para alcançar o comportamento polimórfico.

Um conceito-chave nesses exemplos é demonstrar que um objeto de uma classe derivada pode ser tratado como um objeto de classe básica. Isso permite várias manipulações interessantes. Por exemplo, um programa pode criar um array de ponteiros de classe básica que aponta para objetos de muitos tipos de classes derivadas. Apesar de os objetos de classes derivadas serem de tipos diferentes, o compilador permite isso porque cada objeto é tratado desse jeito. Entretanto, não podemos tratar um objeto de classe básica como um objeto de qualquer uma de suas classes derivadas. Por exemplo, um `BaseCommissionEmployee` não tem um membro de dados `baseSalary`, mas tem um membro de dados `commissionRate`. O relacionamento de herança não aplica somente de uma classe derivada para suas classes-sóciais diretas e indiretas.

13.3.1 Invocando funções de classe básica a partir de objetos de classe derivada

O exemplo nas figuras 13.1–13.5 demonstra três maneiras de apontar ponteiros de classe básica e ponteiros de classe derivada para objetos de classe básica e de classe derivada. As duas primeiras são simples e diretas — apontamos um ponteiro de classe básica para um objeto de classe básica (e invocamos as funcionalidades da classe básica) e apontamos um ponteiro de classe derivada para um objeto de classe derivada (e invocamos as funcionalidades da classe derivada). Em seguida, demonstramos o relacionamento entre as classes derivadas e básicas (isto é, o relacionamento de herança) — um ponteiro de classe básica para um objeto de classe derivada (e mostrando que as funcionalidades de classe básica estão, de fato, disponíveis no objeto de classe derivada).

A classe `CommissionEmployee` (Figuras 13.1–13.2), que discutimos no Capítulo 12, é utilizada para representar os empregados que recebem uma porcentagem de suas vendas. A classe `BaseCommissionEmployee` (Figuras 13.3–13.4), que também discutimos no Capítulo 12, é utilizada para representar os empregados que recebem um salário-base mais uma porcentagem de suas vendas. A classe `PlusCommissionEmployee` (Figura 13.5) é um `CommissionEmployee` que também tem um salário-base. A função-membro `earnings` da classe `BasePlusCommissionEmployee` (linhas 32–35 da Figura 13.4) redefine a função-membro `earnings` da classe `CommissionEmployee` (linhas 79–82 da Figura 13.2) para incluir o salário-base do objeto da classe derivada. A função-membro `getTotalSalary` (linhas 38–46 da Figura 13.4) redefine a função-membro `getTotalSalary` da classe `CommissionEmployee` (linhas 85–92 da Figura 13.2) para exibir as mesmas informações exibidas na classe `CommissionEmployee`.

No Figura 13.5, as linhas 19–20 criam um objeto `Employee`; a linha 23 cria um ponteiro para esse objeto. As linhas 26–27 criam um objeto `BaseCommissionEmployee`; a linha 30 cria um ponteiro para esse objeto. As linhas 37 e 39 utilizam o nome de cada objeto (`baseEmployee` e `plusEmployee`, respectivamente) para invocar a função-membro `getTotalSalary` do objeto. A linha 42 atribui o endereço do objeto à classe básica `CommissionEmployee`; a linha 45 utiliza para invocar a função-membro `getTotalSalary` do objeto. A linha 48 atribui o endereço do objeto da classe derivada `PlusCommissionEmployee` ao ponteiro de classe derivada `commissionEmployee`. Porque a linha 52 utiliza para invocar a função-membro `getTotalSalary` do objeto `PlusCommissionEmployee`, o programa exibe a mesma informação que a linha 45. A linha 55 então atribui o endereço de objeto

```

1 // Figura 13.1: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                         double= 0.0, double= 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName(const string &); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber(const string &); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate(double); // configura a taxa de comissão
28     double getCommissionRate()const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate;// porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 13.1 Arquivo de cabeçalho da classe CommissionEmployee

```

1 // Figura 13.2: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )

```

Figura 13.2 Arquivo de implementação da classe CommissionEmployee

(continua)

```

12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14     setGrossSales( sales ); // valida e armazena as vendas brutas
15     setCommissionRate( rate ); // valida e armazena a taxa de comissão
16 } // fim do construtor CommissionEmployee
17
18 // configura o nome
19 void CommissionEmployee::setFirstName(const string &first )
20 {
21     firstName = first; // deve validar
22 } // fim da função setFirstName
23
24 // retorna o nome
25 string CommissionEmployee::getFirstName() const
26 {
27     return firstName;
28 } // fim da função getFirstName
29
30 // configura o sobrenome
31 void CommissionEmployee::setLastName(const string &last )
32 {
33     lastName = last; // deve validar
34 } // fim da função setLastName
35
36 // retorna o sobrenome
37 string CommissionEmployee::getLastName() const
38 {
39     return lastName;
40 } // fim da função getLastname
41
42 // configura o SSN
43 void CommissionEmployee::setSocialSecurityNumber(const string &ssn )
44 {
45     socialSecurityNumber = ssn; // deve validar
46 } // fim da função setSocialSecurityNumber
47
48 // retorna o SSN
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51     return socialSecurityNumber;
52 } // fim da função getSocialSecurityNumber
53
54 // configura a quantidade de vendas brutas
55 void CommissionEmployee::setGrossSales(double sales )
56 {
57     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // fim da função setGrossSales
59
60 // retorna a quantidade de vendas brutas
61 double CommissionEmployee::getGrossSales(const
62     return grossSales;
63 } // fim da função getGrossSales
64
65 // configura a taxa de comissão
66 void CommissionEmployee::setCommissionRate(double rate )
67 {
68 }
```

Figura 13.2 Arquivo de implementação da classe CommissionEmployee

(continua)

```

69     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // fim da função setCommissionRate
71
72 // retorna a taxa de comissão
73 double CommissionEmployee::getCommissionRate() const
74 {
75     return commissionRate;
76 } // fim da função getCommissionRate
77
78 // calcula os rendimentos
79 double CommissionEmployee::earnings() const
80 {
81     return getCommissionRate() * getGrossSales();
82 } // fim da função earnings
83
84 // imprime o objeto CommissionEmployee
85 void CommissionEmployee::print() const
86 {
87     cout << "commission employee: "
88         << getFirstName() << ' ' << getLastName()
89         << "\nsocial security number: " << getSocialSecurityNumber()
90         << "\ngross sales: " << getGrossSales()
91         << "\ncommission rate: " << getCommissionRate();
92 } // fim da função print

```

Figura 13.2 Arquivo de implementação da classe CommissionEmployee

(continuação)

```

1 // Figura 13.3: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee(const string &, const string &,
16         const string &, double= 0.0, double= 0.0, double= 0.0);
17
18     void setBaseSalary( double); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private:
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 13.3 Arquivo de cabeçalho da classe BasePlusCommissionEmployee

```

1 // Figura 13.4: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // Definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica
14     : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary );    // valida e armazena o salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary(double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // fim da função earnings
36
37 // imprime o objeto BasePlusCommissionEmployee
38 void BasePlusCommissionEmployee::print() const
39 {
40     cout << "base-salaried ";
41
42     // invoca a função print de CommissionEmployee
43     CommissionEmployee::print();
44
45     cout << "\nbase salary: " << getBaseSalary();
46 } // fim da função print

```

Figura 13.4 Arquivo de implementação da classe `BasePlusCommissionEmployee`

de classe derivada `BasePlusCommissionEmployee`.²⁰ O ponteiro de classe básica `CommissionEmployee* handle` a linha 59 utiliza para invocar a função-membro `print`. O compilador C++ permite esse ‘*crossing pointer*’ que um objeto de uma classe derivada é um objeto de sua classe básica. Observe que, apesar de o `handle` ser apontado para um objeto `BasePlusCommissionEmployee`, de classe derivada, a função-membro `print` da classe básica `CommissionEmployee` é invocada (em vez da função-membro `print` da classe derivada). A saída de cada invocação de função `print` no programa revela que as funcionalidades invocadas dependem do tipo do handle (isto é, o tipo de ponteiro ou referência) utilizado para invocar a função, não do tipo do objeto para o qual o handle aponta.²¹ Na Seção 13.3.4, quando introduzimos as *demóstrasmos* que é possível invocar as funcionalidades do tipo de objeto, em vez de invocar as funcionalidades do tipo de handle. Veremos que isso é crucial implementar o comportamento polimórfico — o tópico-chave deste capítulo.

```

1 // Figura 13.5: fig13_05.cpp
2 // Apontando ponteiros de classe básica e classe derivada para objetos de classe
3 // básica e classe derivada, respectivamente.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // inclui definições de classe
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18     // cria objeto de classe básica
19     CommissionEmployee commissionEmployee(
20         "Sue", "Jones", "222-22-2222", 10000 .06 );
21
22     // cria ponteiro de classe básica
23     CommissionEmployee *commissionEmployeePtr = 
24
25     // cria objeto de classe derivada
26     BasePlusCommissionEmployee basePlusCommissionEmployee(
27         "Bob", "Lewis", "333-33-3333", 5000 .04, 300);
28
29     // cria ponteiro de classe derivada
30     BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 
31
32     // configura a formatação de saída de ponto flutuante
33     cout << fixed << setprecision( 2 );
34
35     // gera saída dos objetos commissionEmployee e basePlusCommissionEmployee
36     cout << "Print base-class and derived-class objects:\n\n" ;
37     commissionEmployee.print(); // invoca print da classe básica
38     cout << "\n\n" ;
39     basePlusCommissionEmployee.print(); // invoca print da classe derivada
40
41     // aponta o ponteiro de classe básica para o objeto de classe básica e imprime
42     // perfeitamente natural
43     cout << "\n\n\nCalling print with base-class pointer to "
44     << "\nbase-class object invokes base-class print function:\n\n" ;
45     commissionEmployeePtr->print(); // invoca print da classe básica
46
47     // aponta o ponteiro de classe derivada para o objeto de classe derivada e imprime
48     // perfeitamente natural
49     cout << "\n\n\nCalling print with derived-class pointer to "
50     << "\nderived-class object invokes derived-class "
51     << "print function.\n\n" ;
52     basePlusCommissionEmployeePtr->print(); // invoca print da classe derivada
53
54     // aponta ponteiro de classe básica para o objeto de classe derivada e imprime
55     commissionEmployeePtr = &basePlusCommissionEmployee;
56     cout << "\n\n\nCalling print with base-class pointer to "
57     << "derived-class object\ninvokes base-class print "

```

Figura 13.5 Atribuindo endereços de objetos das classes básica e derivada aos ponteiros de classe básica e derivada.

(continua)

```

58     <<     "function on that derived-class object:\n\n"    ;
59     commissionEmployeePtr->print(); // invoca print da classe básica
60     cout << endl;
61     return 0;
62 } // fim de main

```

Print base-class and derived-class objects:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling print with base-class pointer to
base-class object invokes base-class print function:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

```

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

```

Figura 13.5 Atribuindo endereços de objetos das classes básica e derivada aos ponteiros de classe básica e derivada.

(continuação)

13.3.2 Apontando ponteiros de classe derivada para objetos da classe básica

Na Seção 13.3.1, atribuímos o endereço de um objeto de classe derivada a um ponteiro de classe básica e explicamos que o compilador C++ permite essa atribuição, porque um objeto de classe derivada é um objeto de classe básica. Adotamos a abordagem da Figura 13.6, já que apontamos um ponteiro de classe derivada para uma ~~uma classe derivada~~
classe CommissionEmployeeBasePlusCommissionEmployee [das figuras 13.1–13.4.] As linhas 8–9 da Figura 13.6 criam um objeto CommissionEmployee a linha 10 cria um ~~ponteiro~~
ponteiro CommissionEmployee. A linha 14 tenta atribuir o endereço de objeto de classe básica CommissionEmployee ao ponteiro de classe derivada CommissionEmployee, mas o compilador C++ gera um erro. O compilador impede essa atribuição, porque o tipo é ~~uma classe derivada~~
BasePlusCommissionEmployee. Considere as

```

1 // Figura 13.6: fig13_06.cpp
2 // Apontando um ponteiro de classe derivada para um objeto de classe básica.
3 #include "CommissionEmployee.h"
4 #include "BasePlusCommissionEmployee.h"
5
6 int main()
7 {
8     CommissionEmployee commissionEmployee(
9         "Sue", "Jones", "222-22-2222", 10000.06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr =
11        // aponta o ponteiro de classe derivada para objeto de classe básica
12        // Erro: um CommissionEmployee não é um BasePlusCommissionEmployee
13        basePlusCommissionEmployeePtr = &commissionEmployee;
14
15    return 0;
16 } // fim de main

```

Mensagens de erro do compilador de linha de comando Borland C++:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
to 'BasePlusCommissionEmployee *' in function main()
```

Mensagens de erro do compilador GNU C++:

```
fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
`BasePlusCommissionEmployee*'
```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```
C:\cpphtp5\examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
'=' : cannot convert from 'CommissionEmployee * __w64' to
'BasePlusCommissionEmployee *'
Cast from base to derived requires dynamic_cast or static_cast
```

Figura 13.6 Apontando um ponteiro de classe derivada para um objeto de classe básica.

consequências se o compilador precisasse permitir essa atribuição. Por exemplo, se pudéssemos invocar cada função-membro de classe derivada diretamente através de um ponteiro de classe derivada, teríamos que o compilador fornecesse uma função-membro para cada membro da classe derivada. Entretanto, o compilador forneceria uma função-membro para o objeto de classe básica, e não para a classe derivada. Isso poderia resultar em problemas, porque a função-membro fornecida para a classe derivada poderia sobrescrever a função-membro fornecida para a classe básica. Essa memória não pertence ao objeto de classe derivada, então a função-membro Salary poderia sobrescrever outros dados importantes na memória, possivelmente dados que pertencem a um objeto diferente.

13.3.3 Chamadas de função-membro de classe derivada via ponteiros de classe básica

A partir de um ponteiro de classe básica, o compilador permite invocar somente funções-membro de classes básicas. Portanto,

parte desse código não é permitida pelo compilador. A Figura 13.7 mostra as consequências de se tentar invocar uma função-membro de classe derivada a partir de um ponteiro de classe básica. [Nota: Estamos utilizando novamente as classes CommissionEmployee e BasePlusCommissionEmployee das figuras 13.1–13.4.] A linha 9 cria um ponteiro CommissionEmployeePtr para um objeto CommissionEmployee, e as linhas 10–11 criam um objeto BasePlusCommissionEmployee. A linha 14 aponta basePlusCommissionEmployeePtr para o objeto de classe derivada CommissionEmployee. Considerando a Seção 13.3.1, lembre-se de que o compilador C++ permite invocar uma função-membro de classe derivada a partir de um ponteiro para um objeto de classe básica. As linhas 18–22 invocam as funções-membro de classe básica getSocialSecurityNumber, getGrossSales e getCommissionRate a partir do ponteiro de classe básica. Todas essas chamadas são legítimas, porque

```

1 // Figura 13.7: fig13_07.cpp
2 // Tentando invocar as funções-membro exclusivas da classe derivada
3 // por um ponteiro de classe básica.
4 #include "CommissionEmployee.h"
5 #include "BasePlusCommissionEmployee.h"
6
7 int main()
8 {
9     CommissionEmployee *commissionEmployeePtr; // classe básica
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11        "Bob", "Lewis", "333-33-3333", 5000.04, 300); // classe derivada
12
13    // aponta o ponteiro de classe básica para o objeto de classe derivada
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoca as funções-membro de classe básica no objeto de classe derivada
17    // por ponteiro de classe básica
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // tentativa de invocar funções exclusivas de classe derivada
25    // em objeto de classe derivada por meio de um ponteiro de classe básica
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28
29 } // fim de main

```

Mensagens de erro do compilador de linha de comando Borland C++:

```

Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
'CommissionEmployee' in function main()

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
    'getBaseSalary' : is not a member of 'CommissionEmployee'
    C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
        see declaration of 'CommissionEmployee'
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
    'setBaseSalary' : is not a member of 'CommissionEmployee'
    C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
        see declaration of 'CommissionEmployee'

```

Mensagens de erro do compilador GNU C++:

```

fig13_07.cpp:26: error: 'getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
each function it appears in.)
fig13_07.cpp:27: error: 'setBaseSalary' undeclared (first use this function)

```

Figura 13.7 Tentando invocar funções-membro exclusivas da classe derivada via um ponteiro da classe básica.

herda essas funções-membro da classe base. Sabemos que o tipo do handle determina as funções-membro que estão nas linhas 26–27 tentamos invocar as funções-membro da classe base. O compilador C++ gera erros nessas duas linhas, porque elas não são funções-membro da classe base. O handle pode invocar somente aquelas funções que são membros do tipo de classe associada desse handle. (Nesse caso, o tipo de handle é `BaseEmployee`, então podemos invocar somente as funções-membro da classe base.)

Conclui-se que o compilador C++ realmente permite acesso a membros exclusivos de classe derivada a partir de um ponteiro de classe básica que aponta para um objeto de classe derivada se explicitamente fizermos a coerção do ponteiro de classe básica para o tipo de classe derivada — uma técnica conhecida como *downcasting*. Com isso, você aprendeu na Seção 13.3.1, é possível apontar um ponteiro de classe básica para um objeto de classe derivada. Entretanto, como demonstramos na Figura 13.7, um ponteiro de classe básica pode ser utilizado para invocar apenas as funções declaradas na classe básica. O *downcasting* permite a um programa uma operação específica de classe derivada em um objeto de classe derivada apontado por um ponteiro de classe básica. De

downcasting na Seção 13.8 pode invocar funções de classe derivada que não estão na classe básica. Mostraremos um exemplo co



Observação de engenharia de software 13.3

Se o endereço de um objeto de classe derivada foi atribuído a um ponteiro de uma de suas classes básicas diretas ou indiretas, é aceitável fazer coerção desse ponteiro de classe básica de volta para um ponteiro do tipo da classe derivada. De fato, isso deve ser feito para enviar a esse objeto de classe derivada mensagens que não aparecem na classe básica.

13.3.4 Funções virtuais

Na Seção 13.3.1, apontamos um ponteiro de classe básica para um objeto de classe derivada e, então, invocamos a função-membro desse ponteiro. Lembre-se de que o tipo do handle determina que funções da classe invocar. Nesse caso, o ponteiro invoca a função-membro `getBaseSalary()` sobre o objeto `BaseEmployee`, embora o ponteiro estivesse apontando para um objeto que tem sua própria função personalizada, `getGrossSales()`. O tipo do objeto apontado, não o tipo do handle, determina qual versão de uma função invocar.

Primeiro, consideramos as razões pelas quais *são úteis*. Suponha que um conjunto de classes de formas como Círculo, Triângulo, Retângulo e Quadrado são todas derivadas da classe `Básica`. A capacidade de desenhar a si própria por meio de uma função `desenhar()` poderia ser fornecida a cada uma dessas classes. Embora cada classe tenha sua própria função `desenhar()` para cada forma e de forma diferente. Em um programa que desenha um conjunto de formas, seria útil poder tratar as formas genericamente como objetos da classe `Básica`. Para desenhar qualquer forma, poderíamos simplesmente utilizar um ponteiro da classe `Básica` e invocar a função `desenhar()`. No entanto, deixar o programa determinar (isto é, em tempo de execução) qual tipo de classe derivada utilizar, com base no tipo do objeto para o qual o ponteiro da classe básica aponta a qualquer dado momento.

Para permitir esse tipo de comportamento de herança, chamado de *polimorfismo*, devemos sobrescrevermos a função `desenhar()` em cada uma das classes derivadas para desenhar a forma apropriada. Da perspectiva da implementação, sobrescreve função não é diferente de redefinir uma função (que é a abordagem que utilizamos até agora). Uma função sobrescrita em uma classe derivada tem a mesma assinatura e tipo de retorno (isto é, protótipo) que a função que ela sobrescreve em sua classe básica. Declaramos a função de classe `básica` e podemos redefinir essa função. Por contraste, se declararmos a função de classe `básica`, podemos sobrescrever essa função para ativar comportamento polimórfico. Declaramos uma função precedendo o protótipo da função com a palavra-chave `virtual`.

```
virtual void desenhar() const;
```

apareceria na classe `básica`. O protótipo precedente declara uma função `virtual` que não aceita argumentos e não retorna nada. A função é declarada em geral, uma função não faria alterações no objeto em que ela é invocada. As funções virtuais não têm necessariamente de ser funções



Observação de engenharia de software 13.4

Uma vez que uma função é declarada `virtual`, ela permanece `virtual` por toda a hierarquia de herança a partir desse ponto, mesmo que essa função não seja declarada explicitamente quando uma classe a sobrescreve.



Boa prática de programação 13.1

Mesmo que uma função seja implicitamente `virtual` por causa de uma declaração feita em um ponto mais alto da hierarquia de classes, declare explicitamente essa função em cada nível da hierarquia para promover a clareza do programa.



Dica de prevenção de erro 13.1

Quando um programador navega por uma hierarquia de classes para localizar uma classe para reutilização, é possível que uma função nessa classe exiba um comportamento **desito do** mesmo que ele esteja declarado explicitamente. Isso acontece quando a classe herda uma **função** de sua classe básica e pode produzir erros de lógica sutis. Erros como esses podem ser evitados declarando explicitamente todas as funções **virtual** por toda a hierarquia de herança.



Observação de engenharia de software 13.5

Quando uma classe derivada escolhe não sobreescrver uma **função** de sua classe básica, a classe derivada simplesmente herda a implementação de **função** de sua classe básica.

Se o programa invocar uma **função** meio de um ponteiro de classe básica para um objeto de classe derivada (por exemplo `BasePtr->desenhar()`), o programa escolherá a **função** da classe derivada corretamente (isto é, em tempo de execução) com base no tipo do ponteiro — **coerção dinâmica de ponteiro**. Escolher a função apropriada para a chamada em tempo de execução.

Quando uma **função** é chamada referenciando um objeto específico por nome e utilizando o operador de seleção de membro ponto (por exemplo `Quadrado.desenhar()`), a invocação de função é convertida em tempo de compilação (isso é denominado **vinculação estática**). A função que é chamada é aquela definida para (ou herdada pela) a classe desse objeto particular — esse não é um comportamento polimórfico. Portanto, a vinculação dinâmica cessa a partir de handles de ponteiro (e, como veremos, de referência).

Agora vejamos como as **figuras 13.8–13.9** são os arquivos de cabeçalho da classe `CommissionEmployee` e `BasePlusCommissionEmployee`, respectivamente. Observe que a única diferença entre esses arquivos e os das figuras 13.1 e 13.3 é que especificamos as funções-membro `print` de cada classe `como` (linhas 30–31 da Figura 13.8 e linhas 21–22 da Figura 13.9). Como as funções `print` são **virtual** na classe `CommissionEmployee`, elas **funcionam** sobrescrevem as da classe `BasePlusCommissionEmployee`. Agora, se apontarmos um ponteiro da classe básica para um objeto da classe derivada `BasePlusCommissionEmployee`, o programa usar esse ponteiro para chamar a função correspondente do objeto `BasePlusCommissionEmployee`, que será invocada. Não há nenhuma alteração nas implementações de função-membro das classes `CommissionEmployee` e `BasePlusCommissionEmployee`, então reutilizamos as versões das figuras 13.2 e 13.4.

```

1 // Figura 13.8: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                         double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName( const string & ); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas

```

Figura 13.8 O arquivo de cabeçalho da classe `CommissionEmployee` declara as funções `earnings` e `print` como **virtual**.

(continua)

```

26
27     void setCommissionRate(double); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     virtual double earnings() const; // calcula os rendimentos
31     virtual void print() const; // imprime o objeto CommissionEmployee
32 private :
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate;// porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 13.8 O arquivo de cabeçalho da classe `CommissionEmployee.h` declara as funções `earnings` e `print` como virtual . (continuação)

```

1 // Figura 13.9: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee public CommissionEmployee
13 {
14 public :
15     BasePlusCommissionEmployee(const string &, const string &,
16         const string &, double= 0.0, double= 0.0, double= 0.0);
17
18     void setBaseSalary( double); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     virtual double earnings() const; // calcula os rendimentos
22     virtual void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private :
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 13.9 O arquivo de cabeçalho da classe `BasePlusCommissionEmployee.h` declara as funções `earnings` e `print` como virtual .

Modificamos a Figura 13.5 para criar o programa da Figura 13.10. As linhas 46–57 demonstram novamente que um ponteiro de classe básica pode ser utilizado para invocar a função `print` de uma classe derivada. A linha 46 aponta para um objeto `CommissionEmployee`, que é derivado de `Employee`. A linha 47 aponta para um objeto `BasePlusCommissionEmployee`, que é derivado de `CommissionEmployee`. A linha 50 aponta para um objeto de classe `BasePlusCommissionEmployee`. Observe que, quando a linha 67 invoca a função `print` do ponteiro de classe básica, a função da classe derivada `BasePlusCommissionEmployee` é invocada, então a linha 67 gera saída de um texto diferente do gerado pela linha 59 na Figura 13.5 (quando o identificador `base`

Vemos que declarar uma função `print` é suficiente para que o programa determine dinamicamente qual função invocar com base no tipo de objeto para o qual o handle aponta, em vez de no tipo do handle. A decisão sobre qual função chamar é um exemplo de polimorfismo. Observe novamente que quando `commissionEmployee` aponta para um objeto `CommissionEmployee` (linha 46), a função `print` da classe `CommissionEmployee` é invocada; e, quando `commissionEmployee` aponta para um objeto `BasePlusCommissionEmployee`, a função `print` da classe `BasePlusCommissionEmployee` é invocada. Portanto, a mesma mensagem — enviada (a partir de um ponteiro de classe básica) para uma variedade de objetos relacionados por herança com essa classe — assume muitas formas — esse é o comportamento polimórfico.

```

1 // Figura 13.10: fig13_10.cpp
2 // Introduzindo polimorfismo, funções virtuais e vinculação dinâmica.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // inclui definições de classe
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17     // cria objeto de classe básica
18     CommissionEmployee commissionEmployee(
19         "Sue", "Jones", "222-22-2222", 10000, .06);
20
21     // cria ponteiro de classe básica
22     CommissionEmployee *commissionEmployeePtr = NULL;
23
24     // cria objeto de classe derivada
25     BasePlusCommissionEmployee basePlusCommissionEmployee(
26         "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
27
28     // cria ponteiro de classe derivada
29     BasePlusCommissionEmployee *basePlusCommissionEmployeePtr =
30
31     // configura a formatação de saída de ponto flutuante
32     cout << fixed << setprecision(2);
33
34     // gera saída de objetos utilizando vinculação estática
35     cout << "\nInvoking print function on base-class and derived-class \""
36     << "\nobjects with static binding\n\"";
37     commissionEmployee.print(); // vinculação estática
38     cout << "\n\"";
39     basePlusCommissionEmployee.print(); // vinculação estática
40
41     // gera saída de objetos utilizando vinculação dinâmica
42     cout << "\n\n\nInvoking print function on base-class and \""
43     << "derived-class \nobjects with dynamic binding\"";
44
45     // aponta o ponteiro de classe básica para o objeto de classe básica e imprime
46     commissionEmployeePtr = &commissionEmployee;

```

Figura 13.10 Demonstrando polimorfismo invocando uma função `virtual` de classe derivada via um ponteiro de classe básica para um objeto de classe derivada. (continua)

```

47 cout << "\n\nCalling virtual function print with base-class pointer"
48 << "\nto base-class object invokes base-class "
49 << "print function:\n\n" ;
50 commissionEmployeePtr->print(); // invoca print da classe básica
51
52 // aponta o ponteiro de classe derivada para o objeto de classe derivada e imprime
53 basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54 cout << "\n\nCalling virtual function print with derived-class "
55 << "pointer\nto derived-class object invokes derived-class "
56 << "print function:\n\n" ;
57 basePlusCommissionEmployeePtr->print(); // invoca print da classe derivada
58
59 // aponta o ponteiro de classe básica para o objeto de classe derivada e imprime
60 commissionEmployeePtr = &basePlusCommissionEmployee;
61 cout << "\n\nCalling virtual function print with base-class pointer"
62 << "\nto derived-class object invokes derived-class "
63 << "print function:\n\n" ;
64
65 // polimorfismo; invoca print de BasePlusCommissionEmployee;
66 // ponteiro de classe básica para objeto de classe derivada
67 commissionEmployeePtr->print();
68 cout << endl;
69 return 0;
70 } // fim de main

```

Invoking print function on base-class and derived-class objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222

gross sales: 10000.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Invoking print function on base-class and derived-class objects with dynamic binding

Calling virtual function print with base-class pointer to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis

Figura 13.10 Demonstrando polimorfismo invocando uma função virtual de classe derivada via um ponteiro de classe básica para um objeto de classe derivada.

(continua)

```

social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Figura 13.10 Demonstrando polimorfismo invocando uma função virtual de classe derivada via um ponteiro de classe básica para um objeto de classe derivada.

(continuação)

13.3.5 Resumo das atribuições permitidas entre objetos de classe básica e de classe derivada e ponteiros

Agora que você viu um aplicativo completo que processa diversos objetos polimorficamente, resumimos o que você pode e o que não pode fazer com objetos e ponteiros de classe básica e derivada. Embora um objeto da classe derivada também seja um objeto de classe básica, os dois objetos são, apesar disso, diferentes. Como discutido anteriormente, os objetos de classe derivada podem ser tratados como se fossem objetos de classe básica. Esse é um relacionamento lógico, porque a classe derivada contém todos os membros da classe básica. Entretanto, os objetos de classe básica não podem ser tratados como se fossem de classe derivada — a classe básica pode ter membros exclusivos de classes derivadas adicionais. Por essa razão, não é permitido apontar um ponteiro de classe derivada para um objeto de classe básica sem uma coerção explícita — essa atribuição deixaria os membros exclusivos da classe derivada indisponíveis para o objeto de classe básica. A coerção alivia o compilador da responsabilidade de emitir uma mensagem de erro. De certo modo, utilizando a coerção você está dizendo: ‘sei que o que estou fazendo é perigoso e assumo toda a responsabilidade por minhas ações’.

Na seção atual e no Capítulo 12, discutimos quatro maneiras de apontar ponteiros de classe básica e ponteiros de classe derivada para objetos de classe básica e objetos de classe derivada:

1. Apontar um ponteiro de classe básica para um objeto de classe básica é simples e direto — as chamadas feitas a partir de um ponteiro de classe básica simplesmente invocam as funcionalidades da classe básica.
2. Apontar um ponteiro de classe derivada para um objeto de classe derivada é simples e direto — as chamadas feitas a partir de um ponteiro de classe derivada simplesmente invocam as funcionalidades da classe derivada.
3. É seguro apontar um ponteiro de classe básica para um objeto de classe derivada, mas não para um objeto de classe derivada de sua classe básica. Entretanto, esse ponteiro pode ser utilizado para invocar apenas as funções-membro da classe básica. Se o programador tentar referenciar um membro exclusivo da classe derivada por meio do ponteiro de classe básica, o compilador informa um erro. Para evitar esse erro, o programador deve fazer coerção do ponteiro de classe básica para um ponteiro de classe derivada. O ponteiro de classe derivada então pode ser utilizado para invocar as funcionalidades completas do objeto de classe derivada. Entretanto, essa técnica — chamada de *downcasting* — é uma operação potencialmente perigosa. A Seção 13.8 demonstra como utilizar downcasting com segurança.
4. Apontar um ponteiro de classe derivada para um objeto de classe básica gera um erro de compilação. O relacionamento é que se aplica apenas de uma classe derivada para suas classes básicas direta e indireta, e não vice-versa. Um objeto de classe básica não contém membros exclusivos de classe derivada que podem ser invocados a partir de um ponteiro de classe derivada.



Erro comum de programação 13.1

Depois de apontar um ponteiro de classe básica para um objeto de classe derivada, tentar referenciar membros exclusivos de classe derivada com o ponteiro de classe básica é um erro de compilação.



Erro comum de programação 13.2

Tratar um objeto de classe básica como um objeto de classe derivada pode causar erros.

13.4 Campos de tipo e instruções switch

Uma maneira de determinar o tipo de um objeto que é incorporado em um programa é utilizar uma instrução `switch` para realizar uma instrução que distingue entre os tipos de objeto e, então, invocar uma ação apropriada para um objeto particular. Por exemplo, em uma hierarquia de classes, cada classe pode ter uma implementação de uma função que realiza uma ação específica para aquele tipo de classe.

Entretanto, utilizar a lógica `switch` expõe os programas a uma variedade de problemas potenciais. Por exemplo, o programador poderia se esquecer de incluir um teste de tipo quando um é garantido ou poderia se esquecer de testar todos os casos possíveis de uma instrução. Ao modificar um sistema baseado em `switch`, adicionando novos tipos, o programador poderia se esquecer de inserir os novos casos em todas as instruções. Cada adição ou exclusão de uma classe requer a modificação de todas as instruções `switch` no sistema; a monitoração dessas instruções pode ser demorada e propensa a erro.



Observação de engenharia de software 13.6

A programação polimórfica pode eliminar a necessidade de lógica desse tipo. Usando o mecanismo de polimorfismo do C++, para realizar uma lógica equivalente, os programadores podem evitar os tipos de enfoque normalmente associados com a lógica `switch`.



Observação de engenharia de software 13.7

Uma consequência interessante de utilizar polimorfismo é que os programas assumem uma aparência simplificada. Eles contêm menos lógica de desvio e código mais simples, seqüencial. Essa simplificação facilita o teste, depuração e manutenção do programa.

13.5 Classes abstratas e funções virtual puras

Quando pensamos em classe como um tipo, supomos que os programas criam objetos desse tipo. Entretanto, há casos em que as classes a partir das quais o programador nunca pretenderá instanciar qualquer objeto. Essas classes são chamadas de **abstratas**. Como essas classes são normalmente utilizadas como classes básicas em hierarquias de herança, elas são referidas como **classes abstratas**. Essas classes não podem ser utilizadas para instanciar objetos, porque, como veremos em breve, as classes abstratas incompletas — as classes derivadas devem definir as ‘partes ausentes’. Construímos programas com classes abstratas na Seção 12.3.

O propósito de uma classe abstrata é fornecer uma classe básica apropriada a partir da qual outras classes podem herdar. As classes abstratas fornecem implementações de cada função membro que elas definem. Podemos também ter uma classe abstrata que só define suas classes concretas, como

quadradocirculo triangulo rombo cilindro . As classes básicas abstratas são muito genéricas para definir objetos reais; precisamos ser mais específicos antes de podermos pensar em instanciar objetos. Por exemplo, se alguém lhe disser para ‘desenhar a forma bidimensional que forma você desenharia? As classes concretas fornecem os aspectos específicos que tornam razoável instanciar objetos.

Uma hierarquia de herança não precisa conter nenhuma classe abstrata, mas, como veremos, muitos bons sistemas orientados a objetos têm hierarquias de classe encabeçadas por classes básicas abstratas. Em alguns casos, as classes abstratas constituem níveis superiores da hierarquia. Um bom exemplo disso é a hierarquia de formas na Figura 12.3, que começa com a classe básica `Forma`. No próximo nível da hierarquia temos mais duas classes básicas abstratas: `bidimensional` e `tridimensional`. O próximo nível da hierarquia define classes concretas para formas bidimensionais (`circulo` e `Tetraedro`), para formas tridimensionais (`esfera` e `prisma`).

Tornamos uma classe abstrata declarando uma ou mais funções virtuais puras. Uma função virtual pura é especificada colocando `pure` em sua declaração, como em

```
virtual void draw() const = 0; // função virtual pura
```

O ‘`= 0`’ é conhecido como **implementador puro**. As funções virtuais puras não fornecem implementações. Toda classe derivada concreta deve sobreescriver todas as funções da classe básica com implementações concretas dessas funções. A diferença entre uma função e uma função pura é que uma função tem uma implementação e dá à classe derivada a opção de sobreescrivê-la; em contraste, uma função pura fornece uma implementação que a classe derivada

sobrecreva a função (para que a classe derivada seja concreta; caso contrário, a classe derivada permanece abstrata). As funções virtuais puras são utilizadas quando não faz sentido a classe básica ter uma implementação de uma função, mas o programador quer que todas as classes derivadas concretas implementem a função. Retornando ao nosso exemplo anterior de espaciais, não faz sentido a classe `Espaciotecnico` ter uma implementação para a função `draw()` que não há nenhuma maneira de desenhar um objeto espacial genérico sem ter as informações adicionais sobre que tipo de objeto espacial está sendo desenhado. Um exemplo de uma função que seria definida como pura (por exemplo, `getNome()`) seria uma que retornasse um nome para o objeto. Podemos nomear esse espacial genérico (por exemplo, ‘`objeto espacial`’), assim, uma implementação-padrão para essa função pode ser fornecida e a função não precisa ser sobreescrita. Entretanto, a função ainda é declarada se espera que as classes derivadas sobrecrevam essa função para fornecer nomes mais específicos para os objetos de classe de



Observação de engenharia de software 13.8

Uma classe abstrata define uma interface pública comum para as várias classes em uma hierarquia de classes. Uma classe abstrata contém uma ou mais funções puras que as classes derivadas concretas devem sobrepor.



Erro comum de programação 13.3

Tentar instanciar um objeto de uma classe abstrata causa um erro de compilação.



Erro comum de programação 13.4

A falha em sobrepor uma função pura em uma classe derivada, e então tentar instanciar objetos dessa classe, é um erro de compilação.



Observação de engenharia de software 13.9

Uma classe abstrata tem pelo menos uma função pura. Uma classe abstrata também pode ter membros de dados e funções concretas (incluindo construtores e destrutores), que estão sujeitos às regras normais de herança por classes derivadas.

Embora não possamos instanciar objetos de uma classe básica abstrata, podemos utilizar a classe básica abstrata para definir ponteiros e referências que podem referenciar objetos de qualquer classe concreta derivada da classe abstrata. Em geral, os programas utilizam esses ponteiros e referências para manipular objetos de classe derivada polimorficamente.

Consideremos outra aplicação de polimorfismo. Um gerenciador de tela precisa exibir uma variedade de objetos, inclusive os vários tipos de objetos que o programador adicionará ao sistema depois de escrever o gerenciador de tela. O sistema pode precisar exibir formas, como círculos, triângulos ou retângulos, que são derivadas da classe `básica`. O gerenciador de tela utiliza ponteiros para gerenciar os objetos que são exibidos. Para desenhar qualquer objeto (independentemente do nível em que a classe desse objeto aparece na hierarquia de herança), o gerenciador de tela usa um ponteiro de classe básica para o objeto invocar a função `desenhar`. Cada objeto na hierarquia de herança tem a capacidade de desenhar a si mesmo. O gerenciador de tela não precisa se preocupar com o tipo de cada objeto, nem se o gerenciador de tela encontrou ou não alguma vez objetos desse tipo.

O polimorfismo é particularmente eficaz para implementar sistemas em camadas de software. Em sistemas operacionais, por exemplo, cada tipo de dispositivo físico poderia operar diferentemente dos outros. Mesmo assim, os drivers para

dispositivos diferentes podem operar no contexto de um driver de dispositivo diverso. Um driver de dispositivo manipula dispositivos de um tipo específico. Entretanto, a própria classe geralmente difere de qualquer outro dispositivo no sistema — simplesmente transfira um número de bytes da memória para esse dispositivo. Um sistema operacional orientado a objetos talvez utilize uma classe básica abstrata para fornecer uma interface apropriada a todos os drivers de dispositivo. Então, por meio da herança dessa classe básica abstrata, todas as classes derivadas que são formadas operam de maneira semelhante. As capacidades, as funções e as implementações dessas funções são fornecidas pelas classes derivadas que correspondem aos tipos específicos de drivers de dispositivo. Essa arquitetura também permite que os novos dispositivos sejam facilmente adicionados a um sistema, mesmo depois que o sistema operacional tenha sido definido. O usuário pode simplesmente conectar o dispositivo e instalar seu novo driver de dispositivo no sistema operacional ‘conversa’ com esse novo dispositivo por meio de seu driver de dispositivo, que tem as mesmas funções-membro públicas que todos os outros drivers de dispositivo — aqueles definidos na classe básica abstrata de driver de dispositivo.

E comum em programação orientada a objetos definir que pode percorrer todos os objetos em um contêiner (como um array). Por exemplo, um programa pode imprimir uma lista de objetos usando um iterador e, então, utilizar o iterador para obter o próximo elemento da lista toda vez que o iterador for chamado. Os iteradores são freqüentemente usados na programação polimórfica para percorrer um array ou uma lista vinculada de ponteiros para objetos de vários níveis de uma hierarquia. Os ponteiros em uma lista dessas são todos ponteiros de classe básica. (O Capítulo 23, “Standard Template Library (STL)”, apresenta um tratamento completo de iteradores.) Uma lista de ponteiros para objetos da classe básica

permite que o programador trate todos os tipos de objetos da mesma maneira, independentemente de qual classe estiverem.

13.6 Estudo de caso: sistema de folha de pagamento utilizando polimorfismo

Esta seção reexamina a hierarquia `Employee-BasePlusCommissionEmployee`, que exploramos integralmente na Seção 12.4. Neste exemplo, utilizamos uma classe abstrata e polimorfismo para realizar cálculos de folha de pagamento baseados no tipo de empregado. Criamos uma hierarquia de funcionários aprimorada para resolver o problema a seguir:

Uma empresa paga seus empregados semanalmente. Os funcionários são de quatro tipos: funcionários assalariados que recebem salários fixos semanalmente do número de horas trabalhadas, funcionários que trabalham por hora e são pagos da mesma forma e recebem horas extras por todas as horas trabalhadas além das 40 horas normais, funcionários comissionados que recebem uma porcentagem sobre suas vendas e funcionários assalariados-comissionados que recebem um salário-base mais uma porcentagem sobre suas vendas. Para o período de pagamento atual, a empresa decidiu recompensar os empregados comissionados assalariados adicionando 10 por cento ao salário-base. A empresa quer implementar um programa C++ que realiza o pagamento polimorficamente na folha de pagamento.

Utilizamos a classe `Employee` para representar o conceito geral de um funcionário. As classes que derivam diretamente de `Employee` são `SalariedEmployee`, `CommissionEmployee` e `HourlyEmployee`. A classe `BasePlusCommissionEmployee`, derivada de `CommissionEmployee`, representa o último tipo de empregado. O diagrama de classes UML na Figura 13.11 mostra a hierarquia de herança do nosso aplicativo polimórfico de folha de pagamento de funcionários. Observe que o nome da classe abstrata `Employee` é escrito em itálico, de acordo com a convenção da UML.

A classe básica `Employee` declara a ‘interface’ para a hierarquia — isto é, o conjunto de funções-membro que um programa pode invocar em todos `Employee`s. Cada funcionário, independentemente de como seus vencimentos são calculados, tem um `HomePhone`, um `SocialSecurityNumber` e um `private ostream& print(ostream& os)`. Essas informações aparecem na classe de base abstrata.



Observação de engenharia de software 13.10

Uma classe derivada pode herdar a interface ou implementação de uma classe básica. As hierarquias projetadas para a implementação tendem a ter suas funcionalidades na parte superior da hierarquia — cada nova classe derivada herda uma ou mais funções-membro que foram definidas em uma classe básica e a classe derivada utiliza as definições de classe básica. As hierarquias projetadas para herança de interface tendem a ter suas funcionalidades na parte inferior da hierarquia — uma classe básica especifica uma ou mais funções que devem ser definidas para cada classe na hierarquia (isto é, elas têm o mesmo protótipo), mas as classes derivadas individuais fornecem suas próprias implementações da(s) função(ões).

As seções a seguir implementam a hierarquia de classes das cinco primeiras implementam uma das classes abstratas ou concretas. A última seção implementa um programa de teste que constrói objetos de todas essas classes e processa c polimorficamente.

13.6.1 Criando a classe básica abstrata Employee

Antes de vermos o código da Figura 13.11, discutimos mais detalhadamente o que acontece quando o projeto direciona a todos os empregados, mas todos os cálculos de rendimentos dependem da classe do empregado. Portanto, declaramos `virtual` pura na classe `Employee` porque uma implementação-padrão não faz sentido para essa função — não há informações suficientes para determinar que função deve retornar. Toda classe derivada sobrescreverá a implementação apropriada. Para calcular os rendimentos de um empregado, o programa atribui o endereço do objeto de um empregado a um `Employee` da classe básica e, então, invoca a função `print` desse objeto. Mantemos os ponteiros `Employee` para cada um dos quais aponta para um objeto. Naturalmente, não pode haver objetos que `Employee` é uma classe abstrata — por causa da herança, porém, todos os objetos de todas as classes derivadas desse disso, ser considerados objetos `Employee`. O programa iterará pelo loop de chamadas para cada objeto `Employee`. O C++ processa essas chamadas de função polimorficamente, talvez como uma função pura em `Employee`. Força cada classe derivada a direcionar que deseja ser uma classe concreta, e isso permite ao projetista da hierarquia de classes exigir que cada classe derivada forneça um cálculo de salário apropriado, se de fato essa classe derivada precisar ser concreta.

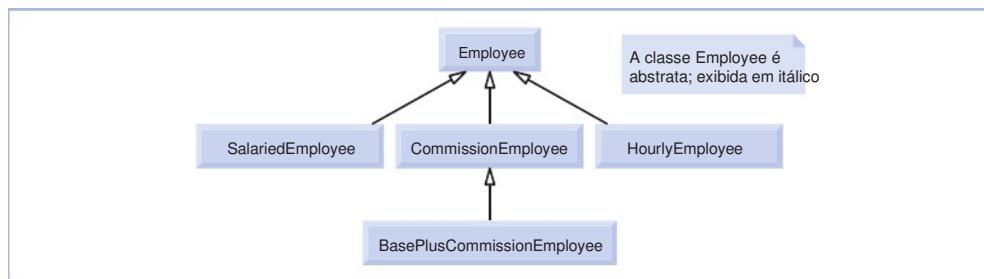


Figura 13.11 Diagrama de classes UML da hierarquia de classes.

A função `print` na classe `Employee` exibe o nome, o sobrenome e o número de seguro social do empregado. Como veremos, cada classe derivada de `Employee` sobrescreve a função para gerar saída do tipo do empregado (por exemplo, "hourly employee") seguido pelas demais informações do empregado.

O diagrama na Figura 13.12 mostra cada uma das cinco classes da hierarquia no lado esquerdo e as funções `earnings` e `print` no lado esquerdo e as funções `print` no lado direito. Observe que a classe `Employee` para a função `earnings` indicar que essa é uma função pura. Cada classe derivada sobrescreve essa função para fornecer uma implementação apropriada. Não listamos as funções básicas porque elas não são sobrescritas em qualquer classe derivada — cada uma dessas funções é herdada e utilizada "como é" por cada uma das classes derivadas.

Consideremos o arquivo de cabeçalho da classe 13.13). As funções membros incluem um construtor que aceita o nome, o sobrenome e o número de seguro social como argumentos (linhas 12) e `print` que sobrenome e o número de seguro social (linhas 14, 17 e 20, respectivamente). A função `earnings` calcula o rendimento do empregado (linhas 15, 18 e 21, respectivamente). A função `print` (linha 24) e a função `print` (linha 25).

Lembre-se de que declararmos como uma função pura porque primeiro precisamos conhecer o tipo de

~~capacidade para determinar se é ou não uma função pura. Declarar essa função pura pode violar os protocolos da classe básica Employee para invocar a função polimorficamente para qualquer tipo de~~

A Figura 13.14 contém as implementações de função `Employee` para a classe `Employee`. A implementação é oferecida para a função `earnings` virtual. Observe que o construtor (linhas 10–15) não valida o número do seguro social. Normalmente, essa validação deve ser fornecida. Um exercício do Capítulo 12 pede para você validar um número de seguro social a fim de ass que ele esteja na forma ###### onde cada # representa um dígito.

Observe que a função `print` (Figura 13.14, linhas 54–58) fornece uma implementação que será sobrescrita em cada uma das classes derivadas. Mas cada uma dessas funções ~~utiliza essa estrutura~~ para imprimir informações comuns a todas as classes na hierarquia.

13.6.2 Criando a classe derivada concreta `SalariedEmployee`

A classe `SalariedEmployee` (Figuras 13.15–13.16) herda da classe 8 da Figura 13.15). As funções membros incluem um construtor que aceita um nome, um sobrenome, um número de seguro social e um salário semanal como argumento (linhas 11–12); uma função para atribuir um novo valor não negativo ao membro de dados (linhas 4); uma função `getSalary` para retornar o valor de `salary` (linha 15); uma função `earnings` que calcula os rendimentos da empregada (linha 18) e uma função `print` que gera saída do tipo do empregado, `salaried employee`: " seguido por informações específicas do empregado produzidas pela função `WeeklySalary` de

`SalariedEmployee` (linha 19).

| | earnings | print |
|------------------------------|--|---|
| Employee | = 0 | firstName lastName
social security number: SSN |
| Salaried-Employee | weeklySalary | salaried employee: firstName lastName
social security number: SSN
weekly salary: weeklySalary |
| Hourly-Employee | If hours <= 40
wage * hours
If hours > 40
(40 * wage) + (hours - 40) * wage * 1.5) | hourly employee: firstName lastName
social security number: SSN
hourly wage: wage hours worked: hours |
| Commission-Employee | commissionRate * grossSales | commission employee: firstName lastName
social security number: SSN
gross sales: grossSales
commission rate: commissionRate |
| BasePlus-Commission-Employee | baseSalary + (commissionRate * grossSales) | base salaried commission employee: firstName lastName
social security number: SSN
gross sales: grossSales
commission rate: commissionRate
base salary: baseSalary |

Figura 13.12 Interface polimórfica para as classes na hierarquia `Employee`

```

1 // Figura 13.13: Employee.h
2 // Classe básica abstrata Employee.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class Employee
10 {
11 public:
12     Employee( const string &, const string &, const string & );
13
14     void setFirstName( const string & ); // configura o nome
15     string getFirstName() const; // retorna o nome
16
17     void setLastName( const string & ); // configura o sobrenome
18     string getLastName() const; // retorna o sobrenome
19
20     void setSocialSecurityNumber( const string & ); // configura o SSN
21     string getSocialSecurityNumber() const; // retorna o SSN
22
23     // a função virtual pura cria a classe básica abstrata Employee
24     virtual double earnings() const = 0; // virtual pura
25     virtual void print() const; // virtual
26 private:
27     string firstName;
28     string lastName;
29     string socialSecurityNumber;
30 }; // fim da classe Employee
31
32 #endif // EMPLOYEE_H

```

Figura 13.13 Arquivo de cabeçalho da classe Employee

```

1 // Figura 13.14: Employee.cpp
2 // Definições de função-membro da classe básica abstrata Employee.
3 // Nota: Nenhuma definição recebe funções virtuais puras.
4 #include <iostream>
5 using std::cout;
6
7 #include "Employee.h" // Definição da classe Employee
8
9 // construtor
10 Employee::Employee( const string &first, const string &last,
11                     const string &ssn )
12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13     // corpo vazio
14
15 } // fim do construtor Employee
16
17 // configura o nome
18 void Employee::setFirstName( const string &first )
19 {

```

Figura 13.14 Arquivo de implementação da classe Employee

(continua)

```

20     firstName = first;
21 } // fim da função setFirstName
22
23 // retorna o nome
24 string Employee::getFirstName() const
25 {
26     return firstName;
27 } // fim da função getFirstName
28
29 // configura o sobrenome
30 void Employee::setLastName(const string &last)
31 {
32     lastName = last;
33 } // fim da função setLastName
34
35 // retorna o sobrenome
36 string Employee::getLastName() const
37 {
38     return lastName;
39 } // fim da função getLastname
40
41 // configura o SSN
42 void Employee::setSocialSecurityNumber(const string &ssn)
43 {
44     socialSecurityNumber = ssn; // deve validar
45 } // fim da função setSocialSecurityNumber
46
47 // retorna o SSN
48 string Employee::getSocialSecurityNumber() const
49 {
50     return socialSecurityNumber;
51 } // fim da função getSocialSecurityNumber
52
53 // imprime informações de Employee (virtual, mas não virtual pura)
54 void Employee::print() const
55 {
56     cout << getFirstName() << " " << getLastName()
57         << "\nsocial security number: " << getSocialSecurityNumber();
58 } // fim da função print

```

Figura 13.14 Arquivo de implementação da classe Employee

(continuação)

```

1 // Figura 13.15: SalariedEmployee.h
2 // Classe SalariedEmployee derivada de Employee.
3 #ifndef SALARIED_H
4 #define SALARIED_H
5
6 #include "Employee.h" // definição da classe Employee
7
8 class SalariedEmployee : public Employee
9 {
10 public:
11     SalariedEmployee( const string &, const string &,
12                       const string &, double= 0.0 );

```

Figura 13.15 Arquivo de cabeçalho da classe SalariedEmployee

(continua)

```
13     void setWeeklySalary( double ); // configura o salário semanal
14     double getWeeklySalary() const; // retorna o salário semanal
15
16
17     // palavra-chave virtual assinala intenção de sobreescriver
18     virtual double earnings() const; // calcula os rendimentos
19     virtual void print() const; // imprime objeto SalariedEmployee
20 private :
21     double weeklySalary; // salário por semana
22 }; // fim da classe SalariedEmployee
23
24 #endif // SALARIED_H
```

Figura 13.15 Arquivo de cabeçalho da classe SalariedEmployee

(continuação)

A Figura 13.16 contém as implementações de função-membro. O construtor da classe passa o nome, o sobrenome e o número de seguro social para o construtor a fim de inicializar os membros de dados. São herdados da classe básica, mas não acessíveis na classe derivada (linhas 8–10). Ela escreve a função earnings para empregar para fornecer uma implementação concreta que retorna o salário de comissão de implementações sem usar a classe base. Será uma classe abstrata e qualquer tentativa de instanciar um objeto da classe resultaria em um erro de compilação (e, naturalmente, em um erro de runtime se essa classe fosse usada como uma classe concreta). Observe que no arquivo de cabeçalho da classe Employee, declaramos as funções earnings e print como virtual (linhas 18–19 da Figura 13.15) — na realidade, é redundante colocá-las na classe base, já que elas permanecem funções-membro. Definimos essas funções como virtual na classe básica, e então elas permanecem funções-membro toda a hierarquia de classes. Considerando a “Boa prática de programação 13.1”, lembre-se de que declarar explicitamente cada nível da hierarquia pode promover clareza de programa.

A função `print` da classe `SalariedEmployee` (linhas 36–41 da Figura 13.16) sobrecreve a função `print` da classe `SalariedEmployee`, não sobrecrevendo a função `print` herdada a versão de `Employee.print`. Nesse caso, a função `print` da classe `SalariedEmployee` simplesmente retornaria o nome completo e o número de seguro social do funcionário, o que não representa adequadamente a classe `SalariedEmployee`. Para imprimir informações completas de um `Employee`, da classe derivada gera a saída de `Employee.print`: “ Seguida pelas informações específicas de `SalariedEmployee`, o sobrenome e o número de seguro social) impressas juntas, utilizando o operador de resolução de escopo (linha 39) — esse é um exemplo elegante de reutilização de código. A saída produzida pelo comando `print` o salário semanal do empregado obtido invocando a função `getTotalPay` da classe.

```
1 // Figura 13.16: SalariedEmployee.cpp
2 // Definições de função-membro da classe SalariedEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "SalariedEmployee.h" // definição da classe SalariedEmployee
7
8 // construtor
9 SalariedEmployee::SalariedEmployee(const string &first,
10 : Employee(first, last, ssn) string &ssn, double salary )
11 {
12     setWeeklySalary( salary );
13 } // fim do construtor SalariedEmployee
14
15
16 // configura o salário
17 void SalariedEmployee::setWeeklySalary(double salary )
```

Figura 13.16 Arquivo de implementação da classe SalariedEmployee

(continua)

```

18 {
19     weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // fim da função setWeeklySalary
21
22 // retorna o salário
23 double SalariedEmployee::getWeeklySalary() const
24 {
25     return weeklySalary;
26 } // fim da função getWeeklySalary
27
28 // calcula os rendimentos;
29 // sobrescreve a função virtual pura earnings em Employee
30 double SalariedEmployee::earnings() const
31 {
32     return getWeeklySalary();
33 } // fim da função earnings
34
35 // imprime informações de SalariedEmployee
36 void SalariedEmployee::print() const
37 {
38     cout << "salaried employee: ";
39     Employee::print(); // reutiliza função print da classe básica abstrata
40     cout << "\nweekly salary: " << getWeeklySalary();
41 } // fim da função print

```

Figura 13.16 Arquivo de implementação da classe SalariedEmployee

(continuação)

13.6.3 Criando a classe derivada concreta HourlyEmployee

A classe HourlyEmployee (figuras 13.17–13.18) também herda de Employee (classe 8 da Figura 13.17). As funções-membro

public incluem um construtor (linhas 11–12) que aceita como argumentos um nome, um sobrenome, um número de seguro social e uma hora e o número de horas trabalhadas; atribui novos valores aos membros de dados respectivamente (linhas 14 e 17); uma função que retorna os valores de horas, respectivamente (linhas 15 e 18); uma função virtual earnings que calcula os rendimentos de um empregado (linha 21) e uma função print que gera saída do tipo do empregado, a saber: "e informações específicas do empregado (linha 22).

A Figura 13.18 contém as implementações de função-membro para as classes 18–21 e 30–34 definem as funções que atribuem novos valores aos membros de dados respectivamente. As funções linhas 18–21 asseguram que não negativo, e a função (linhas 30–34) assegura que o membro é maior ou igual a zero e menor ou igual a 168 (o número total de horas em uma semana). As funções 24–27 e 37–40. Não declaramos essas funções, portanto as classes derivadas da classe Employee podem sobrepor elas (embora as classes derivadas certamente possam redefiní-las). Observe que o construtor constroi a classe SalariedEmployee passa o nome, o sobrenome e o número de seguro social da classe básica (linha 11) para inicializar os membros de dados private herdados declarados na classe básica. Além disso, a função print chama a função da classe básica (linha 56) para gerar saída de informações específicas, nome, o sobrenome e o número de seguro social — esse é outro exemplo elegante de reutilização de código.

13.6.4 Criando a classe derivada concreta CommissionEmployee

A classe CommissionEmployee (figuras 13.19–13.20) deriva da classe Employee (classe 8 da Figura 13.19). As implementações de

funcão-membro (Figura 13.20) incluem um construtor (linhas 9–15) que aceita um nome, um sobrenome, um número de seguro social, a quantidade de vendas e uma taxa de comissão (linhas 30–33), para atribuir novos valores aos membros de dados CommissionRategrossSales respectivamente; as funções (linhas 24–27 e 36–39), que recuperam os valores desses membros de dados; a função (linhas 43–46) para calcular os rendimentos de um empregado; e a função (linhas 49–55), que gera saída do tipo do empregado, a saber: "e informações específicas do empregado. O construtor CommissionEmployee também passa o nome, o sobrenome e o número de seguro social (linha 11) a fim de inicializar os membros de dados Employee. A função print chama a função da classe básica (linha 52) para exibir informações específicas (isto é, o nome, o sobrenome e o número de seguro social).

```

1 // Figura 13.17: HourlyEmployee.h
2 // Definição da classe HourlyEmployee.
3 #ifndef HOURLY_H
4 #define HOURLY_H
5
6 #include "Employee.h" // definição da classe Employee
7
8 class HourlyEmployee : public Employee
9 {
10 public :
11 HourlyEmployee( const string &, const string &,
12 const string &, double= 0.0, double= 0.0 );
13
14 void setWage(double); // configura o salário por hora
15 double getWage() const; // retorna o salário por hora
16
17 void setHours( double); // configura as horas trabalhadas
18 double getHours() const; // retorna as horas trabalhadas
19
20 // a palavra-chave virtual assinala intenção de sobrescrever
21 virtual double earnings() const; // calcula os rendimentos
22 virtual void print() const; // imprime o objeto HourlyEmployee
23 private :
24 double wage; // salário por hora
25 double hours; // horas trabalhadas durante a semana
26 }; // fim da classe HourlyEmployee
27
28 #endif // HOURLY_H

```

Figura 13.17 Arquivo de cabeçalho da classe HourlyEmployee

```

1 // Figura 13.18: HourlyEmployee.cpp
2 // Definições de função-membro da classe HourlyEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "HourlyEmployee.h"// definição da classe HourlyEmployee
7
8 // construtor
9 HourlyEmployee::HourlyEmployee(const string &first, const string &last,
10 const string &ssn, double hourlyWage, double hoursWorked )
11 : Employee( first, last, ssn )
12 {
13     setWage( hourlyWage ); // valida a remuneração por hora
14     setHours( hoursWorked ); // valida as horas trabalhadas
15 } // fim do construtor HourlyEmployee
16
17 // configura a remuneração
18 void HourlyEmployee::setWage(double hourlyWage )
19 {
20     wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // fim da função setWage
22
23 // retorna a remuneração

```

Figura 13.18 Arquivo de implementação da classe HourlyEmployee

(continua)