```
   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, disciplinada. Para alguns problemas, é útil ter as funções chamando umas às outras. Uma **função recursiva** é uma função que chama a si mesma, direta ou indiretamente (por outra função).[2] A recursão é um tópico importante, discutido detalhadamente em cursos de nível superior de ciência da computação. Esta seção e a próxima apresentam exemplos simples de recursão. Este livro contém um extenso 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) **caso(s) básico(s)**. Se a função é chamada com um caso básico, ela simplesmente retorna um resultado. Se a função é chamada com um problema mais complexo, em geral, ela divide o problema em duas partes conceituais — uma parte que a função sabe fazer e outra que não sabe. Para tornar a recursão realizável, a última parte deve parecer-se com o problema original, mas ser uma versão ligeiramente mais simples ou ligeiramente menor. Esse novo problema é parecido com o problema original; portanto, a função carrega (chama) uma cópia nova dela própria para trabalhar no problema menor — isso é referido como **chamada recursiva** e também é chamado de **passo de recursão**. O passo de recursão freqüentemente inclui a palavra-chave `return`, porque 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 `main`.

O passo de recursão executa enquanto a chamada original à função ainda está aberta, isto é, não terminou de executar. O passo de recursão pode resultar em muito mais dessas chamadas recursivas, uma vez que a função continua dividindo cada novo subproblema com o qual a função é chamada em duas partes conceituais. Para que a recursão, por fim, termine, toda vez que a função chamar a si mesma 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, e uma seqüência de retornos se segue até que a chamada de função original por fim retorne o resultado final para `main`. Tudo isso soa bem estranho comparado ao tipo de resolução de problemas 'convencional' que utilizamos até agora. Como um exemplo desses conceitos em operação, vamos escrever um programa recursivo para realizar um cálculo matemático popular.

O fatorial de um inteiro n não negativo, escrito $n!$ (e pronunciado como 'n fatorial'), é o produto

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

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, `number`, maior que ou igual a 0, pode ser calculado **iterativamente** (não recursivamente) utilizando uma instrução `for` como mostrado a seguir:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

Chega-se a uma definição recursiva da função fatorial observando o seguinte relacionamento:

$$n! = n \cdot (n-1)!$$

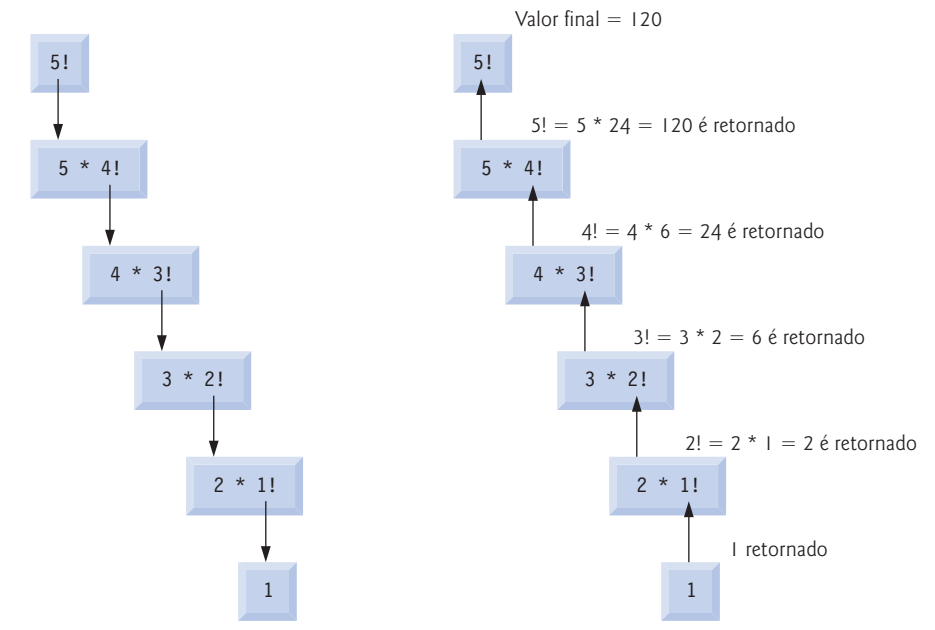
Por exemplo, 5! é claramente igual a 5 * 4! Como mostrado a seguir:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

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 0–10. (A escolha do tipo de dados `unsigned long` é explicada daqui a pouco.) A função recursiva `factorial` (linhas 23–29) primeiro determina se a condição de terminação

[2] Embora muitos compiladores permitam que a função `main` chame a si própria, a Seção 3.6.1, parágrafo 3, da documentação do padrão C++ indica que `main` não deve ser chamado a partir de dentro de um programa. Seu único propósito é ser o ponto inicial para a execução de programa.



(a) Seqüência de chamadas recursivas. (b) Valores retornados a partir de cada chamada recursiva.

Figura 6.28 Avaliação recursiva de 5!.

`number <= 1` (linha 25) é verdadeira. Se `number` for de fato menor que ou igual a 1, a função `factorial` retornará 1 (linha 26), nenhuma recursão adicional será necessária e a função terminará. Se `number` for maior que 1, a linha 28 expressa o problema como o produto de `number` e uma chamada recursiva para `factorial` avaliar o fatorial de `number - 1`. Observe que `factorial( number - 1 )` é um problema ligeiramente mais simples que o cálculo original `factorial( number )`.

A função `factorial` foi declarada para receber um parâmetro do tipo `unsigned long` e retornar um resultado de tipo `unsigned long`. Essa é a notação abreviada para `unsigned long int`. A documentação do padrão C++ requer que uma variável de tipo `unsigned 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 de dados `long int` 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 grandes rapidamente. Escolhemos o tipo de dados `unsigned long` para que o programa possa calcular fatoriais maiores que 7! em computadores com inteiros pequenos (como dois bytes). Infelizmente, a função `factorial` produz valores grandes com tanta rapidez que até `unsigned long` não nos ajuda a calcular muitos valores fatoriais antes mesmo de o tamanho de uma variável `unsigned long` ser excedido.

Os exercícios exploram o uso de variáveis do tipo de dados `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 requisitos 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 já estão disponíveis em bibliotecas de classes populares,[3] e trabalhamos 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 convirja 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 subseqüente é a soma dos dois números de Fibonacci anteriores.

[3] Essas classes podem ser encontradas em `shoup.net/ntl`, `cliodhna.cop.uop.edu/~hetrick/c-sources.html` e `www.trumphurst.com/cpplibs/datapage.phtml?category='intro'`.

```
1  // Figura 6.29: fig06_29.cpp
2  // Testando a função fatorial 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 fatorial 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 fatorial
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 fatorial
```

```
 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 converge para um valor constante de 1,618.... Esse número, também, ocorre freqüentemente na natureza e é chamado de **relação áurea** ou **média áurea**. Humanos tendem a achar a média áurea esteticamente agradável. Os arquitetos freqüentemente projetam janelas, salas e edifícios cujo comprimento e largura estão na relação da média áurea. Os cartões-postais freqüentemente são projetados com uma relação de comprimento/largura da média áurea.

A série de Fibonacci pode ser definida recursivamente como segue:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci (n) = fibonacci (n – 1) + fibonacci (n – 2)

O programa da Figura 6.30 calcula o n-ésimo número de Fibonacci recursivamente utilizando a função `fibonacci`. Note que os números de Fibonacci também tendem a se tornar rapidamente grandes, embora mais lentamente do que os fatoriais. Portanto, escolhemos o tipo de dados `unsigned long` para o tipo de parâmetro e o tipo de retorno na função `fibonacci`. 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 `for` 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 `fibonacci` (linhas 15, 18, 19 e 20) provenientes de `main` não são recursivas, mas as chamadas de `fibonacci` da linha 30 são recursivas. Toda vez que o programa invoca `fibonacci` (linhas 25–31), a função testa imediatamente o caso básico para determinar se `number` é igual a `0` ou `1` (linha 27). Se isso for verdadeiro, a linha 28 retorna `number`. Curiosamente, se `number` for maior que 1, o passo de recursão (linha 30) gera duas chamadas recursivas, cada uma para um problema ligeiramente menor do que a chamada original a `fibonacci`. A Figura 6.31 mostra como a função `fibonacci` avaliaria `fibonacci( 3 )`.

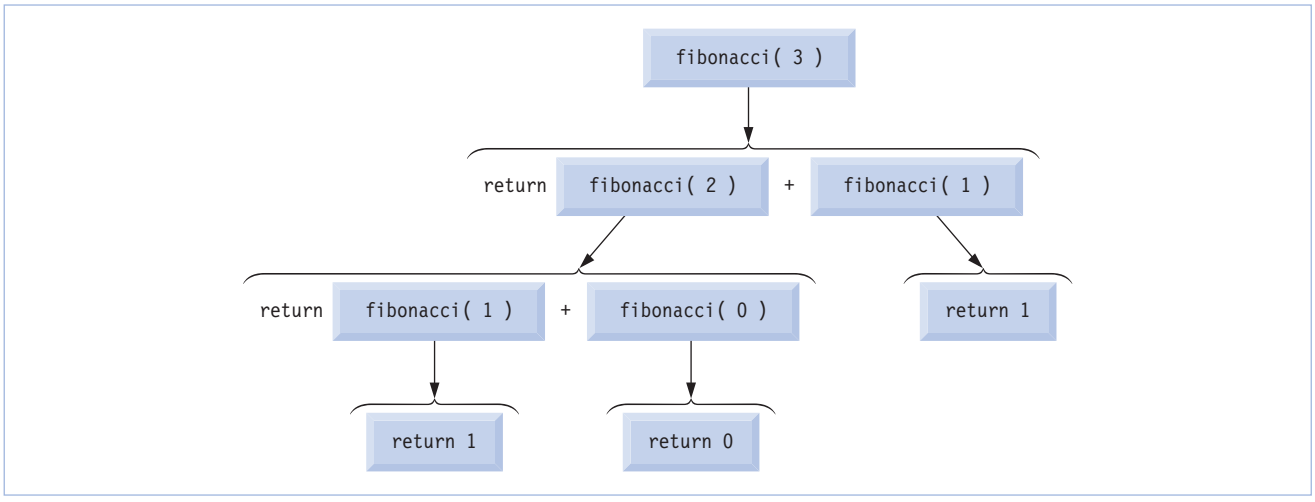


Figura 6.31 Conjunto de chamadas recursivas à função `fibonacci`.

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 pelas regras de precedência e associatividade de operadores. A Figura 6.31 mostra que avaliar `fibonacci( 3 )` produz duas 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 da maioria dos operadores (inclusive +) devem ser avaliados. Portanto, o programador não deve fazer nenhuma suposição sobre a ordem em que essas chamadas executam. As chamadas poderiam de fato executar `fibonacci( 2 )` primeiro e, então, `fibonacci( 1 )`, ou poderiam executar na ordem inversa: `fibonacci( 1 )`, 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 um operando pode ter **efeitos colaterais** (alterações 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, `&&`, `||`, vírgula (`,`) e `?:`. Os três primeiros são operadores binários cujos dois operandos são garantidamente avaliados da esquerda para a direita. O último operador é o único operador ternário do C++. Seu operando mais à esquerda sempre é avaliado primeiro; se for avaliado como não-zero (verdadeiro), o operando do meio é avaliado em seguida e o último operando é ignorado; se o operando mais à esquerda for avaliado como zero (falso), o terceiro operando é avaliado em seguida 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 `&&`, `||`, `?:` e vírgula (`,`) pode levar a erros de lógica.

Dica de portabilidade 6.3

Os programas que dependem da ordem de avaliação dos operandos de operadores diferentes dos operadores `&&`, `||`, `?:` e vírgula (`,`) 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 na função `fibonacci` tem o efeito de duplicar o número de chamadas de função; isto é, o número de chamadas recursivas que é requerido para calcular o $n^{\text{ésimo}}$ número de Fibonacci está na ordem de 2^n. Isso rapidamente foge do controle. Calcular somente o vigésimo número de Fibonacci exigiria um número de chamadas na ordem de 2^{20} ou cerca de um milhão de chamadas, calcular o trigésimo número de Fibonacci exigiria um número de chamadas na ordem de 2^{30} ou cerca de um bilhão de chamadas e assim por diante. Os cientistas da computação se referem a isso como **complexidade exponencial**. Os problemas dessa natureza humilham 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. Esta seção 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 alcança 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 controlada por contador e a recursão gradualmente se aproximam do término: a iteração modifica um contador até que o contador assume um valor que faz a condição de continuação do loop falhar; a recursão produz versões mais simples do problema original até que o caso básico 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 recursiva de uma maneira que convirja para o caso básico.

Para ilustrar as diferenças entre iteração e recursão, examinemos uma solução iterativa do problema fatorial (Figura 6.32). Observe 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 quanto ao caso básico. Na solução iterativa, a linha 28 testa a condição de continuação do loop — se o teste falhar, o loop termina. Por fim, observe que em vez de produzir a versão mais simples do problema original, a solução iterativa utiliza um contador que é modificado até a condição de continuação do loop tornar-se falsa.

A recursão tem muitos pontos negativos. Ela invoca repetidamente o mecanismo, e conseqüentemente 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 com que outra cópia da função (na realidade, somente as variáveis da função) seja criada; isso pode consumir memória considerável. A iteração 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 acidentalmente 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 fatorial iterativa.
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
```

Figura 6.32 Solução fatorial iterativa. (continua)

```
12  int main()
13  {
14     // calcula o fatorial 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 fatorial iterativa
23  unsigned long factorial( unsigned long number )
24  {
25     unsigned long result = 1;
26
27     // declaração iterativa da função fatorial
28     for ( unsigned long i = number; i >= 1; i-- )
29        result *= i;
30
31     return result;
32  } // fim da função fatorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figura 6.32 Solução fatorial iterativa. (continuação)

Localização no texto		Exemplos e exercícios de recursão
Capítulo 6	Seção 6.19, Figura 6.29	Função fatorial
	Seção 6.19, Figura 6.30	Função de Fibonacci
	Exercício 6.7	Soma de dois inteiros
	Exercício 6.40	Elevando um inteiro a uma potência inteira
	Exercício 6.42	Torres de Hanói
	Exercício 6.44	Visualizando a recursão
	Exercício 6.45	Máximo divisor comum recursivo
	Exercícios 6.50 e 6.51	Exercício Mistério "O que esse programa faz?"
Capítulo 7	Exercício 7.18	Exercício Mistério "O que esse programa faz?"
	Exercício 7.21	Exercício Mistério "O que esse programa faz?"
	Exercício 7.31	Classificação por seleção

Figura 6.33 Resumo de exemplos e exercícios de recursão no texto. (continua)

Localização no texto		Exemplos e exercícios de recursão
	Exercício 7.32	Determine se uma string é um palíndromo
	Exercício 7.33	Pesquisa linear
	Exercício 7.34	Pesquisa binária
	Exercício 7.35	Oito Rainhas
	Exercício 7.36	Imprima um array
	Exercício 7.37	Imprima uma string de trás para a frente
	Exercício 7.38	Valor mínimo em um array
Capítulo 8	Exercício 8.24	Quicksort
	Exercício 8.25	Percorrendo um labirinto
	Exercício 8.26	Gerador de labirintos aleatório
	Exercício 8.27	Labirintos de qualquer tamanho
Capítulo 20	Seção 20.3.3, figuras 20.5–20.7	Classificação por intercalação
	Exercício 20.8	Pesquisa linear
	Exercício 20.9	Pesquisa binária
	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	Imprima uma lista vinculada de trás para a frente
	Exercício 21.21	Pesquise uma lista vinculada
	Exercício 21.22	Exclusão de árvore binária
	Exercício 21.25	Impressão de árvore

Figura 6.33 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 orientado a objetos do nosso sistema ATM. No Capítulo 3, identificamos as classes que precisaremos implementar e criamos nosso primeiro diagrama de classes. No Capítulo 4, descrevemos alguns atributos das nossas classes. No Capítulo 5, examinamos estados dos objetos e 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 mundo real. As operações de um rádio incluem configurar sua estação e volume (em geral invocadas por uma pessoa que ajusta os controles do rádio). As operações de um carro incluem acelerar (invocada pelo motorista ao pressionar pedal do acelerador), desacelerar (invocada 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.