Técnicas de Programação

TP0401

Prof. Giovane Barcelos
giovane_barcelos@uniritter.edu.br

Plano de Ensino
Conteúdo programático

1. Introdução à programação em C 2. Desenvolvimento estruturado de programas em C 3. Controle de programa 4. Funções 5. Arrays 6. Ponteiros	N1
7. Caracteres e strings 8. Entrada/Saída formatada 9. Estruturas, uniões, manipulações de bits e enumerações 10. Processamento de arquivos 11. Estruturas de dados 12. O pré-processador 13. Outros tópicos sobre C	N2

Objetivos

- A construir programas de forma modular a partir de pequenas partes chamadas funções.
- As funções matemáticas comuns na biblioteca-padrão em C.
- A criar novas funções.
- Os mecanismos usados para passar informações entre funções.
- Como o mecanismo de chamada/retorno de função é aceito pela pilha de chamada de função e pelos registros de ativação.
- Técnicas de simulação a partir da geração de números aleatórios.
- Como escrever e usar funções que chamam a si mesmas.

Introdução

- Quase todos os programas de computador que resolvem problemas do mundo real são muito maiores que os programas que foram apresentados nos primeiros capítulos.
- A experiência vem mostrando que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de partes menores, ou de módulos, cada um mais facilmente administrável que o programa original.
- Essa técnica é chamada de dividir e conquistar.
- Este capítulo descreve os recursos da linguagem em C que facilitam o projeto, a implementação, a operação e a manutenção de programas de grande porte.

Módulos de programa em C

- Os módulos em C são chamados de **funções**.
- Os programas em C normalmente são escritos combinando-se novas funções com funções 'pré-definidas', disponíveis na **biblioteca-padrão de C**.
- A biblioteca-padrão de C oferece uma rica coleção de funções para a realização de cálculos matemáticos comuns, manipulação de strings, manipulação de caracteres, entrada/saída e muitas outras operações úteis. Isso torna seu trabalho mais fácil, pois essas funções oferecem muitas das capacidades de que você precisa.

Módulos de programa em C

Boa prática de programação 5.1
Familiarize-se com a rica coleção de funções da biblioteca-padrão de C.

Observação sobre engenharia de software 5.1
Evite reinventar a roda. Quando possível, use as funções da biblioteca-padrão de C em vez de escrever novas funções. Isso pode reduzir o tempo de desenvolvimento do programa.

Dica de portabilidade 5.1
O uso das funções da biblioteca-padrão de C ajuda a tornar os programas mais portáveis.

Módulos de programa em C

- As funções printf, scanf e pow que discutimos nos capítulos anteriores, são funções da biblioteca-padrão.
- Você pode escrever funções que definam tarefas específicas que poderão ser usadas em muitos pontos de um programa.
- Elas também são chamadas de funções definidas pelo programador.
- As instruções reais que definem a função são escritas apenas uma vez e ficam escondidas de outras funções.
- As funções são chamadas (ou invocadas) por uma chamada de função, que especifica o nome da função e oferece informações (como argumentos) de que a função chamada precisa para realizar sua tarefa designada.

Módulos de programa em C

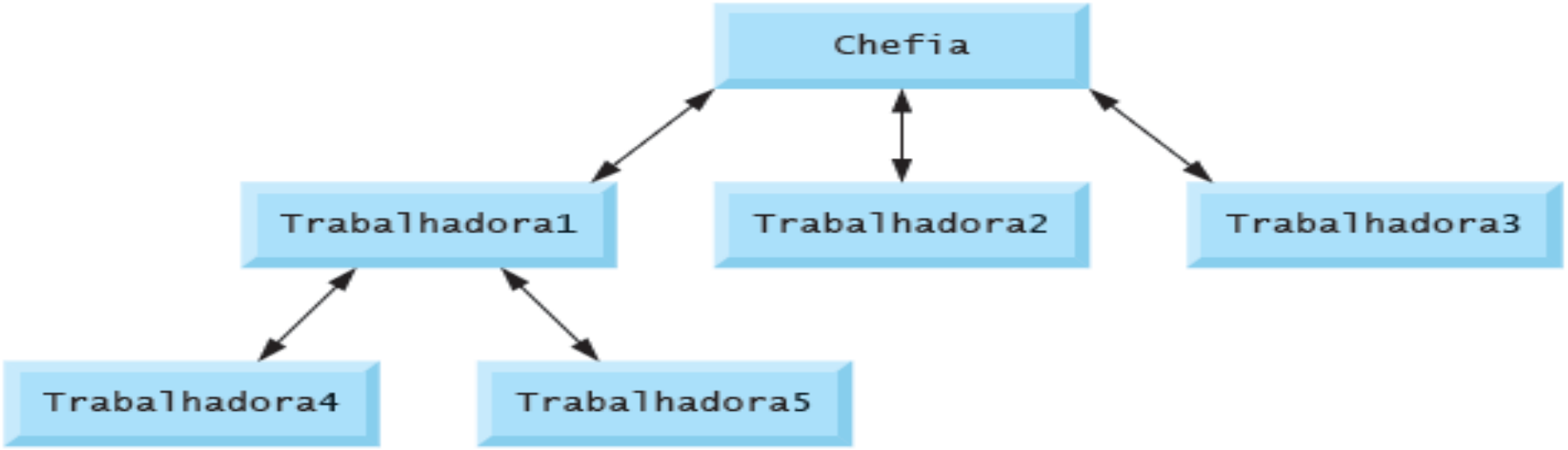


Figura 5.1 ■ Relacionamento hierárquico entre função chefia e função trabalhadora.

Módulos de programa em C

- Uma analogia comum para isso é a forma hierárquica de gerência.
- Uma chefia (a função chamadora) pede a uma trabalhadora (a função chamada) que realize uma tarefa e informe quando ela tiver sido concluída (Figura 5.1).
- Por exemplo, uma função que precise exibir informações na tela chama a função trabalhadora printf para realizar essa tarefa, depois printf exibe a informação e informa de volta — ou retorna — à função chamadora quando sua tarefa é concluída.
- A função chefia não sabe como a função trabalhadora realiza suas tarefas designadas.

Módulos de programa em C

- A trabalhadora pode chamar outras funções trabalhadoras, e a chefia não saberá disso.
- Logo veremos como essa 'ocultação' de detalhes da implementação promove a boa engenharia de software.
- A Figura 5.1 mostra a função chefia comunicando-se com várias funções trabalhadoras de maneira hierárquica.
- Observe que Trabalhadora1 atua como uma função chefia para Trabalhadora4 e Trabalhadora5.
- Os relacionamentos entre as funções podem diferir da estrutura hierárquica mostrada na figura 5.1.

Funções da biblioteca matemática

- Muitas das funções da biblioteca matemática permitem realizar vários cálculos matemáticos comuns.
- Normalmente, as funções são usadas em um programa ao escrevermos o nome da função seguido de um parêntese à esquerda, seguido do argumento (ou uma lista de argumentos separados por vírgula) da função, seguido do parêntese à direita.
- Por exemplo, um programador que queira calcular e exibir a raiz quadrada de 900.0 poderia escrever

```
printf( "%.2f", sqrt( 900.0 ) );
```

- Quando esse comando é executado, a função sqrt da biblioteca matemática é chamada para calcular a raiz quadrada do número contido nos parênteses (900.0).

Funções da biblioteca matemática

- O número 900.0 é o argumento da função sqrt.
- Esse comando mostraria 30.00.
- A função sqrt pede um argumento do tipo double e retorna um resultado do tipo double.
- Todas as funções na biblioteca matemática que retornam valores de ponto flutuante retornam o tipo de dados double.
- Observe que valores double, assim como valores float, podem ser exibidos ao usarmos a especificação de conversão %f.

Funções da biblioteca matemática

Dica de prevenção de erro 5.1

Inclua o cabeçalho math usando a diretiva do pré-processador #include <math.h> ao usar funções na biblioteca matemática.

Funções da biblioteca matemática

Função	Descrição	Exemplo
`sqrt( x )`	raiz quadrada de *x*	`sqrt( 900,0 )` é `30,0` `sqrt( 9,0 )` é `3,0`
`exp( x )`	função exponencial e^x	`exp( 1,0 )` é `2,718282` `exp( 2,0 )` é `7,389056`
`log( x )`	logaritmo natural de *x* (base *e*)	`log( 2,718282 )` é `1,0` `log( 7,389056 )` é `2,0`

`log10( x )`	logaritmo de *x* (base 10)	`log10( 1,0 )` é `0,0`
		`log10( 10,0 )` é `1,0`
		`log10( 100,0 )` é `2,0`

`fabs( x )`	valor absoluto de *x*	`fabs( 13,5 )` é `13,5`
		`fabs( 0,0 )` é `0,0`
		`fabs( -13,5 )` é `13,5`

`ceil( x )`	arredonda *x* ao menor inteiro não menor que *x*	`ceil( 9,2 )` é `10,0` `ceil( -9,8 )` é `-9,0`
`floor( x )`	arredonda *x* ao maior inteiro não maior que *x*	`floor( 9,2 )` é `9,0` `floor( -9,8 )` é `-10,0`
`pow( x, y )`	*x* elevado à potência *y* (x^y)	`pow( 2, 7 )` é `128,0` `pow( 9, 0,5 )` é `3,0`
`fmod( x, y )`	módulo (resto) de *x*/*y* como um número em ponto flutuante	`fmod( 13,657, 2,333 )` é `1,992`
`sin( x )`	seno trigonométrico de *x* (*x* em radianos)	`sin( 0,0 )` é `0,0`
`cos( x )`	cosseno trigonométrico de *x* (*x* em radianos)	`cos( 0,0 )` é `1,0`
`tan( x )`	tangente trigonométrica de *x* (*x* em radianos)	`tan( 0,0 )` é `0,0`

Figura 5.2 ■ Funções da biblioteca matemática comumente usadas.

Funções da biblioteca matemática

- Os argumentos de função podem ser constantes, variáveis ou expressões.
- Se c1 = 13.0, d = 3.0 and f = 4.0, então a instrução

  printf("%.2f", sqrt(c1 + d * f));
- calcula e exibe a raiz quadrada de 13.0 + 3.0 * 4.0 = 25.0, a saber 5.00.
- Algumas funções da biblioteca matemática de C estão resumidas na Figura 5.2.
- Na figura, as variáveis x e y são do tipo double.

Funções

- As funções permitem a criação de um programa em módulos.
- Todas as variáveis descritas nas definições de função são variáveis locais — elas são conhecidas apenas na função em que são definidas.
- A maioria das funções possui uma lista de parâmetros que oferecem meios de transmissão de informações entre as funções.
- Os parâmetros de uma função também são variáveis locais dessa função.

Funções

Observação sobre engenharia de software 5.2

Nos programas que contêm muitas funções, main normalmente é implementada como um grupo de chamadas para funções que realizam a maior parte do trabalho no programa.

Funções

- Existem várias motivações para se ‘funcionalizar’ um programa.
- A técnica de dividir e conquistar torna o desenvolvimento do programa mais administrável.
- Outra motivação é a reutilização do software — o uso de funções existentes como blocos de montagem para criar novos programas.
- A reutilização do software é um fator importante no movimento da programação orientada a objeto, a respeito da qual você aprenderá mais quando estudarmos as linguagens derivadas da C, como C++, Java e C# (diz-se ‘C sharp’).
- Utilizamos a abstração toda vez que usamos funções da biblioteca-padrão, como printf, scanf e pow.
- A terceira motivação é evitar a repetição do código em um programa.
- Empacotar o código como uma função permite que ele seja executado a partir de diversos locais em um programa, bastando para isso que a função seja chamada.

Funções

Observação sobre engenharia de software 5.3

Cada função deve ser limitada a realizar uma única tarefa bem-definida, e o nome dela deve expressar essa tarefa. Isso facilita a abstração e promove a reutilização do software.

Observação sobre engenharia de software 5.4

Se você não puder escolher um nome curto que expresse o que a função faz, é possível que sua função esteja tentando realizar muitas tarefas diversas. Normalmente, é melhor quebrar essa função em várias funções menores — às vezes chamamos isso de decomposição.

Funções

- Com uma boa nomeação e definição de função, os programas podem ser criados a partir de funções padronizadas que realizam tarefas específicas, em vez de serem criados a partir do uso de um código personalizado.
- Isso é chamado de abstração.
- Utilizamos a abstração toda vez que usamos funções da biblioteca-padrão, como printf, scanf e pow.
- A terceira motivação é evitar a repetição do código em um programa.
- Empacotar o código como uma função permite que ele seja executado a partir de diversos locais em um programa, bastando para isso que a função seja chamada.

Observação sobre engenharia de software 5.5

Geralmente, uma função não deve ocupar mais que uma página. Melhor ainda, as funções não devem ocupar mais que meia página. Funções pequenas promovem a reutilização do software.

Observação sobre engenharia de software 5.6

Os programas devem ser escritos como coleções de pequenas funções. Isso os torna mais fáceis de serem escritos, depurados, mantidos e modificados.

Definições de funções

```c
/* Fig. 5.3: fig05_03.c
   Criando e usando uma função definida pelo programador */
#include <stdio.h>

int square( int y ); /* protótipo da função */

/* função main inicia execução do programa */
int main( void )
{
   int x; /* contador */

   /* loop 10 vezes e calcula e exibe quadrado de x a cada vez */
   for ( x = 1; x <= 10; x++ ) {
      printf( "%d  ", square( x ) ); /* chamada da função */
   } /* fim do for */

   printf( "\n" );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */

/* definição de função square retorna quadrado do parâmetro */
int square( int y ) /* y é uma cópia do argumento à função */
{
   return y * y; /* retorna o quadrado de y como um int */
} /* fim da função square */
```

```
1   4   9   16   25   36   49   64   81   100
```

Figura 5.3 ■ Usando uma função definida pelo programador.

Definições de funções

- Os programas que apresentamos até agora consistem em uma função chamada main que chama as funções da biblioteca-padrão para realizar suas tarefas.
- Agora, refletiremos sobre como escrever funções personalizadas.
- Considere um programa que use uma função square para calcular e exibir os quadrados dos inteiros de 1 a 10 (Figura 5.3).

Definições de funções

Boa prática sobre programação 5.2

Insira uma linha em branco entre as definições de função para separar as funções e melhorar a legibilidade do programa.

Definições de funções

- A função square é invocada ou chamada em main dentro da instrução printf (linha 14)

```
printf( "%d  ", square( x ) ); /* chamada da função */
```

- A função square recebe uma cópia do valor de x no parâmetro y (line 22).
- Depois, square calcula y * y (linha 24).
- O resultado é passado de volta à função printf em main, onde square foi chamada (linha 14), e printf exibe o resultado.
- Esse processo é repetido 10 vezes, usando a estrutura de repetição for.

Definições de funções

- A definição da função square mostra que square espera por um parâmetro inteiro y.
- A palavra-chave int antes do nome da função (linha 22) indica que square retorna um resultado inteiro.
- A instrução return dentro de square passa o resultado do cálculo de volta à função chamadora.
- A linha 5

  ```
  int square( int y ); /* protótipo da função */
  ```

  é um protótipo da função.
- O int nos parênteses informa ao compilador que square espera receber um valor inteiro da chamadora.

Definições de funções

- O int à esquerda do nome da função square informa ao compilador que square retorna um resultado inteiro à chamadora.
- O compilador se refere ao protótipo da função para verificar se as chamadas a square (line 14) (linha 14) contêm o tipo de retorno correto, o número correto de argumentos, os tipos corretos de argumentos e se eles estão na ordem correta.
- Os protótipos de função serão discutidos com detalhes na Seção 5.6.
- O formato de uma definição de função é

tipo-valor-retorno nome-função (*lista de parâmetros*)
{
  definições
  instruções
}

Definições de funções

- O *nome-função* é qualquer identificador válido.
- O tipo-valor-retorno é o tipo de dado do resultado retornado à chamadora.
- O *tipo-valor-retorno* void indica que uma função não retorna um valor.
- Juntos, tipo-valor-retorno, *nome-função* e *lista de parâmetros* às vezes são chamados de cabeçalho da função.

Definições de funções

Erro comum de programação 5.1
Esquecer de retornar um valor de uma função que deveria retornar um valor pode gerar erros inesperados. O padrão
em C indica que o resultado dessa omissão é indefinido.

Erro comum de programação 5.2
Retornar um valor de uma função com um tipo de retorno void é um erro de compilação.

Definições de funções

- A **lista de parâmetros** é uma lista separada por vírgula que especifica os parâmetros recebidos pela função quando ela é chamada.
- Se uma função não recebe nenhum valor, a *lista de parâmetros* é void.
- Um tipo precisa ser listado explicitamente para cada parâmetro.

Definições de funções

Erro comum de programação 5.3
Especificar parâmetros de função do mesmo tipo como double x, y em vez de double x, double y, resulta em um erro de compilação.

Erro comum de programação 5.4
Colocar um ponto e vírgula após o parêntese à direita que delimita a lista de parâmetros de uma definição de
função é um erro de sintaxe.

Definições de funções

Erro comum de programação 5.5

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

Boa prática de programação 5.3

Embora não seja errado, não use os mesmos nomes para os argumentos de uma função e para os parâmetros correspondentes na definição da função. Isso ajuda a evitar ambiguidades.

Definições de funções

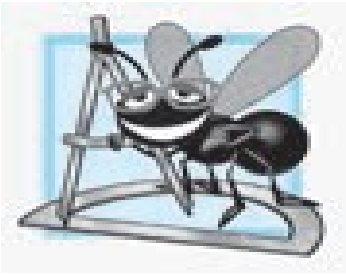
- As *definições* e *instruções* dentro das chaves formam o corpo da função.
- O corpo da função também é chamado de bloco.
- As variáveis podem ser declaradas em qualquer bloco, e os blocos podem ser aninhados.
- *Uma função não pode ser definida dentro de outra função.*

Definições de funções

Erro comum de programação 5.6
Definir uma função dentro de outra função é um erro de sintaxe.

Boa prática de programação 5.4
Escolher nomes de função e de parâmetro significativos torna os programas mais legíveis e evita o uso excessivo
de comentários.

Definições de funções



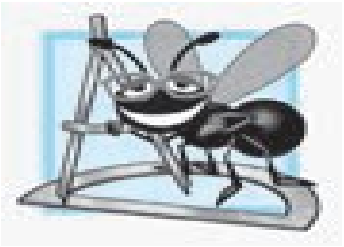
Observação sobre engenharia de software 5.7

Uma função que exige um grande número de parâmetros pode estar realizando tarefas demais. Considere dividi-la em funções menores, que realizem as tarefas separadamente. O cabeçalho da função deverá caber em uma linha, se possível.

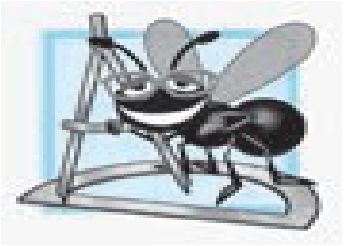
Observação sobre engenharia de software 5.8

O protótipo da função, o cabeçalho da função e as chamadas de função devem combinar em número, tipo e ordem de argumentos e de parâmetros, e também no tipo do valor de retorno.

Definições de funções



Observação sobre engenharia de software 5.9
Um protótipo de função colocado fora de qualquer definição de função se aplica a todas as chamadas para a função que aparecem após o protótipo de função no arquivo. Um protótipo de função colocado dentro de uma função se aplica apenas às chamadas feitas nessa função.



Observação sobre engenharia de software 5.10
O armazenamento automático é um exemplo do **princípio do menor privilégio** *— ele permite o acesso aos dados somente quando eles são realmente necessários. Por que ter variáveis acessíveis e armazenadas na memória quando, na verdade, elas não são necessárias?*

Definições de funções

- **Existem três maneiras de retornar o controle de uma função para o ponto em que uma função foi invocada.**
- **Se a função não retornar um resultado, simplesmente, quando a chave direita de término da função for alcançada, ou ao se executar o comando**

  return;
- **Se a função retornar um resultado, o comando**

  return *expressão;*
- **retorna o valor da *expressão* à chamadora.**

Definições de funções

```
1  /* Fig. 5.4: fig05_04.c
2     Achando o máximo de três inteiros */
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z ); /* protótipo de função */
6
7  /* função main inicia a execução do programa */
8  int main( void )
9  {
10    int number1; /* primeiro inteiro */
11    int number2; /* segundo inteiro */
12    int number3; /* terceiro inteiro */
13
14    printf( "Digite três inteiros: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    /* number1, number2 e number3 são argumentos
18       da chamada da função maximum */
19    printf( "Máximo é: %d\n", maximum( number1, number2, number3 ) );
20    return 0; /* indica conclusão bem-sucedida */
21 } /* fim do main */
22
23 /* Definição da função maximum */
24 /* x, y e z são parâmetros */
25 int maximum( int x, int y, int z )
26 {
```

Figura 5.4 ■ Encontrando o máximo de três inteiros. (Parte 1 de 2.)

Definições de funções

```
27    int max = x; /* considera que x é o maior */
28
29    if ( y > max ) { /* se y é maior que max, atribui y a max */
30       max = y;
31    } /* fim do if */
32
33    if ( z > max ) { /* se z é maior que max, atribui z a max */
34       max = z;
35    } /* fim do if */
36
37    return max; /* max é o maior valor */
38 } /* fim da função maximum */
```

```
Digite três inteiros: 22 85 17
Máximo é: 85
```

```
Digite três inteiros: 85 22 17
Máximo é: 85
```

```
Digite três inteiros: 22 17 85
Máximo é: 85
```

Figura 5.4 ■ Encontrando o máximo de três inteiros. (Parte 2 de 2.)

Definições de funções

- **Nosso segundo exemplo utiliza uma função definida pelo programador, maximum, para determinar e retornar o maior de três inteiros (Figura 5.4).**
- **Em seguida, os inteiros são passados a maximum (linha 19), que determina o maior inteiro.**
- **Esse valor é retornado a main pelo comando return em maximum (linha 37).**

Protótipos de funções

- Um dos recursos mais importantes de C é o protótipo de função.
- Esse recurso foi emprestado pelo comitê do padrão em C dos desenvolvedores em C++.
- O protótipo de função diz ao compilador o tipo de dado retornado pela função, o número de parâmetros que a função espera receber, os tipos dos parâmetros e a ordem em que esses parâmetros são esperados.
- O compilador utiliza protótipos de função para validar as chamadas de função.

Protótipos de funções

- As versões anteriores de C não realizavam esse tipo de verificação, de modo que era possível chamar funções incorretamente sem que o compilador detectasse os erros.
- Essas chamadas poderiam resultar em erros fatais no tempo de execução, ou em erros não fatais que causavam erros lógicos sutis e difíceis de detectar.

Protótipos de funções

Boa prática de programação 5.5

Inclua protótipos de função em todas as funções para tirar proveito das capacidades de verificação de tipo da C.

Utilize diretivas do pré-processador #include para obter protótipos de função para as funções da biblioteca-padrão

a partir dos cabeçalhos para as bibliotecas apropriadas, ou para obter cabeçalhos que contenham protótipos de função para funções desenvolvidas por você e/ou pelos membros do seu grupo.

Protótipos de funções

- O protótipo de função para maximum na Figura 5.4 (linha 5)

```
/* protótipo de função */
int maximum( int x, int y, int z );
```

- Esse protótipo de função indica que maximum utiliza três argumentos do tipo int e retorna um resultado do tipo int.
- Observe que o protótipo de função é igual à primeira linha da definição da função maximum.

Protótipos de funções

Boa prática de programação 5.6

Às vezes, os nomes de parâmetros são incluídos nos protótipos de função (nossa preferência) para fins de documentação.
O compilador ignora esses nomes.

Erro comum de programação 5.7

Esquecer de colocar o ponto e vírgula ao final de um protótipo de função é um erro de sintaxe.

Protótipos de funções

- Uma chamada de função que não corresponde ao protótipo de função consiste em um erro de compilação.
- Outro erro também é gerado se o protótipo de função e a definição da função divergirem.
- Por exemplo, na Figura 5.4, se o protótipo de função tivesse sido escrito como

```
void maximum( int x, int y, int z );
```

- o compilador geraria um erro, pois o tipo de retorno void no protótipo de função seria diferente do tipo de retorno int no cabeçalho da função.

Protótipos de funções

- Outro recurso importante dos protótipos de função é a coerção de argumentos, ou seja, forçar os argumentos para o tipo apropriado.
- Por exemplo, a função sqrt da biblioteca matemática pode ser chamada com um argumento inteiro, embora o protótipo de função em <math.h> especifique um argumento double e, ainda assim, ela desempenhará seu papel corretamente.
- A instrução

```
printf( "%.3f\n", sqrt( 4 ) );
```

avalia corretamente sqrt(4), e imprime o valor 2.000.

Protótipos de funções

- O protótipo de função faz com que o compilador converta o valor inteiro 4 para o valor double 4.0 antes que o valor seja passado para sqrt.
- Em geral, os valores de argumento que não correspondem exatamente aos tipos de parâmetro no protótipo de função são transformados no tipo apropriado antes que a função seja chamada.
- Essas conversões podem gerar resultados incorretos se as regras de promoção da C não forem seguidas.
- As regras de promoção especificam como os tipos podem ser convertidos para outros tipos sem que haja perda de dados.

Protótipos de funções

- Em nosso exemplo de sqrt, um int é convertido automaticamente para um double sem mudar seu valor.
- Porém, um double convertido para um int trunca a parte fracionária do valor double.
- Converter tipos inteiros grandes para tipos inteiros pequenos (por exemplo, long para short) pode resultar em valores alterados.
- As regras de promoção se aplicam automaticamente a expressões que contenham valores de dois ou mais tipos de dados (também chamados de **expressões de tipo misto**).

Protótipos de funções

Tipo de dados	Especificação de conversão de printf	Especificação de conversão de scanf
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%c	%c

Figura 5.5 ■ Hierarquia de promoção para tipos de dados.

Protótipos de funções

- O tipo de cada valor em uma expressão de tipo misto é automaticamente promovido para o tipo ‘mais alto’ na expressão (na realidade, uma versão temporária de cada valor é criada e usada na expressão; os valores originais permanecem inalterados).
- A Figura 5.5 lista os tipos de dados na ordem do mais alto para o mais baixo, com as especificações de conversão printf e scanf de cada tipo.

Protótipos de funções

- A conversão de valores em tipos inferiores normalmente resulta em um valor incorreto.
- Portanto, um valor pode ser convertido em um tipo inferior somente pela atribuição explícita do valor a uma variável do tipo inferior, ou usando-se um operador de coerção.
- Os valores de argumento de função são convertidos para tipos de parâmetro de um protótipo de função como se estivessem sendo atribuídos diretamente às variáveis desses tipos.
- Se nossa função square que usa um parâmetro inteiro (Figura 5.3), for chamada com um argumento de ponto flutuante, o argumento será convertido em int (um tipo inferior), e square normalmente retornará um valor
- incorreto.
- Por exemplo, square(4.5) retorna 16, e não 20.25.

Protótipos de funções

Erro comum de programação 5.8

Converter um tipo de dado mais alto na hierarquia de promoção em um tipo inferior pode alterar o valor do dado. Muitos compiladores emitem advertências nesses casos.

Protótipos de funções

- **Se não há protótipo de função para uma função, o compilador forma seu próprio protótipo, usando a primeira ocorrência da função — ou a definição de função, ou uma chamada para a função.**
- **Normalmente, isso causa advertências ou erros, a depender do compilador.**

Protótipos de funções

Dica de prevenção de erro 5.2
Sempre inclua protótipos de função nas funções que você define ou usa em seu programa; isso ajuda a evitar erros
e advertências na compilação.

Observação sobre engenharia de software 5.9
Um protótipo de função colocado fora de qualquer definição de função se aplica a todas as chamadas para a função que aparecem após o protótipo de função no arquivo. Um protótipo de função colocado dentro de uma função se aplica apenas às chamadas feitas nessa função.

Pilha de chamada de funções e registros de ativação

- Para entender como C realiza chamadas de função, precisamos, em primeiro lugar, considerar uma estrutura de dados (ou seja, a coleção de itens de dados relacionados) conhecida como **pilha**.
- Os estudantes podem pensar na pilha como algo semelhante a uma pilha de pratos.
- Quando um prato é colocado na pilha, ele normalmente é colocado no topo (chamamos isso de **empilhar**).
- De modo semelhante, quando um prato é removido da pilha, ele sempre é removido do topo (chamamos isso de **desempilhar**).
- As pilhas são conhecidas como estruturas de dados **last-in, first-out (LIFO)** — o último item empilhado (inserido) na pilha é o primeiro item a ser desempilhado (retirado) da pilha.

Pilha de chamada de funções e registros de ativação

- Quando um programa chama uma função, a função chamada precisa saber como retornar ao chamador, de modo que o endereço de retorno da função chamadora é colocado na pilha de execução do programa (às vezes chamada de pilha de chamada de função).
- Se houver uma série de chamadas de função, os endereços de retorno sucessivos são empilhados na ordem 'último a entrar, primeiro a sair', de modo que cada função possa retornar à sua chamadora.
- A pilha de execução do programa também contém a memória para as variáveis locais usadas em cada chamada de função durante a execução do programa.

Pilha de chamada de funções e registros de ativação

- **Esses dados, armazenados como uma parte da pilha de execução do programa, são conhecidos como registros de ativação ou quadros de pilha da chamada de função.**
- **Quando uma chamada de função é feita, o registro de ativação para essa chamada de função é colocado na pilha de execução do programa.**
- **Quando a função retorna ao seu chamador, o registro de ativação para essa chamada de função é retirado da pilha, e essas variáveis locais não são mais conhecidas do programa.**

Pilha de chamada de funções e registros de ativação

- **Naturalmente, a quantidade de memória em um computador é finita, de modo que apenas certa quantidade de memória pode ser usada para armazenar registros de ativação na pilha de execução do programa.**
- **Se houver mais chamadas de função do que é possível armazenar nos registros de ativação da pilha de execução do programa, um erro conhecido como estouro de pilha (stack overflow) ocorrerá.**

Cabeçalhos

Cabeçalho	Explicação
<assert.h>	Contém macros e informações que acrescentam diagnósticos que auxiliam a depuração do programa.
<ctype.h>	Contém protótipos de função para funções que testam certas propriedades dos caracteres, e protótipos de função para funções que podem ser usadas para converter letras minúsculas em maiúsculas, e vice-versa.
<errno.h>	Define macros que são úteis na comunicação de condições de erro.
<float.h>	Contém os limites de tamanho de ponto flutuante do sistema.
<limits.h>	Contém os limites de tamanho de inteiros do sistema.
<locale.h>	Contém protótipos de função e outras informações que permitem que um programa seja modificado para o local em que estiver sendo executado. A noção de local permite que o sistema de computação trate de diferentes convenções que expressam dados como datas, horas, valores monetários e números grandes em qualquer lugar do mundo.
<math.h>	Contém protótipos de função para funções da biblioteca matemática.
<setjmp.h>	Contém protótipos de função para funções que permitem evitar a sequência normal de chamada e de retorno de função.
<signal.h>	Contém protótipos de função e macros que lidam com diversas condições que podem surgir durante a execução do programa.
<stdarg.h>	Define macros que lidam com uma lista de argumentos para uma função cujo número e cujo tipo são desconhecidos.
<stddef.h>	Contém as definições comuns de tipo usadas pela C para realizar cálculos.
<stdio.h>	Contém protótipos de função para as funções da biblioteca-padrão de entrada/saída, e informações usadas por eles.
<stdlib.h>	Contém protótipos de função para conversões de números em texto e de texto em números, alocação de memória, números aleatórios e outras funções utilitárias.
<string.h>	Contém protótipos de função para funções de processamento de strings.
<time.h>	Contém protótipos de função e tipos para manipulação de hora e data.

Figura 5.6 ■ Alguns dos cabeçalhos da biblioteca-padrão.

Cabeçalhos

- Cada biblioteca-padrão tem um cabeçalho correspondente que contém os protótipos de função para todas as funções nessa biblioteca, e definições de vários tipos de dados e constantes necessárias a essas funções.
- A Figura 5.6 lista alfabeticamente alguns dos cabeçalhos da biblioteca-padrão que podem ser incluídos nos programas.
- O termo ‘macros’, que é usado várias vezes na Figura 5.6, será discutido com detalhes adiante em Pré-processador em C.
- Você pode criar cabeçalhos personalizados. Os cabeçalhos definidos pelo programador também devem usar a extensão de nome de arquivo .h.

Cabeçalhos

- Um cabeçalho definido pelo programador pode ser incluído utilizando-se a diretiva do pré-processador #include.
- Por exemplo, se o protótipo de nossa função square.h estivesse localizado no cabeçalho square.h, incluiríamos esse cabeçalho em nosso
- programa usando a diretiva a seguir do código do programa:

```
#include "square.h"
```

Chamando funções por valor e por referência

- Em muitas linguagens de programação, existem duas maneiras de se chamar funções — a chamada por valor e a chamada por referência.
- Quando os argumentos são passados por valor, uma *cópia* do valor do argumento é feita e passada para a função chamada.
- As mudanças na cópia não afetam o valor original da variável na chamadora.
- Quando um argumento é passado por referência, o chamador permite que a função chamada modifique o valor da variável original.
- A chamada por valor deverá ser usada sempre que a função chamada não precisar modificar o valor da variável original da chamadora.

Chamando funções por valor e por referência

- Isso evita efeitos colaterais (modificações de variável) acidentais que tanto atrapalham o desenvolvimento de sistemas de software corretos e confiáveis.
- A chamada por referência deve ser usada apenas nos casos de funções chamadas confiáveis, que precisam modificar a variável original.
- Em C, todas as chamadas são feitas por valor.
- Adiante veremos que os arrays são automaticamente passados por referência.

Geração de números aleatórios

- Agora, faremos um rápido e divertido (ao menos, é o que esperamos) desvio em direção a uma aplicação de programação popular: a simulação e os jogos..
- O elemento da sorte pode ser introduzido nas aplicações de computador com o uso da função rand da biblioteca-padrão de C a partir do cabeçalho <stdlib.h>.
- Considere a seguinte instrução:

```
i = rand();
```

- A função rand gera um inteiro entre 0 e RAND_MAX (uma constante simbólica definida no cabeçalho <stdlib.h>).

Geração de números aleatórios

- A C padrão indica que o valor de RAND_MAX deve ser pelo menos 32767, que é o valor máximo para um inteiro de dois bytes (ou seja, 16 bits).
- Se rand realmente produz inteiros aleatórios, cada número entre 0 e RAND_MAX tem a mesma chance (ou probabilidade) de ser escolhido toda vez que rand chamada.
- O intervalo de valores produzidos diretamente por rand, normalmente, é diferente daquele que é necessário em uma aplicação específica..

Geração de números aleatórios

- Por exemplo, um programa que simula o lançamento de uma moeda poderia exigir apenas 0 para ‘cara’ e 1 para ‘coroa’.
- Um programa de jogo de dados, que simula o rolar de um dado de seis lados, exigiria inteiros aleatórios de 1 a 6.
- Para demonstrar rand, desenvolveremos um programa que simulará 20 lançamentos de um dado de seis lados e exibirá o valor de cada lançamento.
- O protótipo de função para a funçãorand está em <stdlib.h>.
- Usaremos o operador de módulo (%) em conjunto com rand da seguinte forma:

  rand() % 6

- para produzir inteiros no intervalo de 0 a 5.

Geração de números aleatórios

```
1  /* Fig. 5.7: fig05_07.c
2     Inteiros escalados e deslocados, produzidos por 1 + rand() % 6 */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* função main inicia a execução do programa */
7  int main( void )
8  {
9     int i; /* contador */
10
11    /* loop 20 vezes */
12    for ( i = 1; i <= 20; i++ ) {
13
14       /* escolhe número aleatório de 1 a 6 e imprime na tela */
15       printf( "%10d", 1 + ( rand() % 6 ) );
16
17       /* se contador é divisível por 5, inicia nova linha de impressão */
18       if ( i % 5 == 0 ) {
```

Figura 5.7 ■ Números inteiros escalados e deslocados produzidos por 1 + rand( ) % 6. (Parte 1 de 2.)

Geração de números aleatórios

```
19            printf( "\n" );
20         } /* fim do if */
21      } /* fim do for */
22
23      return 0; /* indica conclusão bem-sucedida */
24   } /* fim do main */
```

```
         6        6        5        5        6
         5        1        1        5        3
         6        6        2        4        2
         6        2        3        4        1
```

Figura 5.7 ■ Números inteiros escalados e deslocados produzidos por 1 + rand( ) % 6. (Parte 2 de 2.)

Geração de números aleatórios

- Isso é chamado de escala.
- O número 6 é chamado de fator de escala.
- Depois, deslocamos o intervalo de números produzidos somando 1 ao nosso resultado anterior.
- A saída da Figura 5.7 confirma que os resultados estão no intervalo de 1 a 6 — a saída poderia variar conforme o compilador.

Geração de números aleatórios

```c
/* Fig. 5.8: fig05_08.c
   Lançando um dado de seis lados 6000 vezes */
#include <stdio.h>
#include <stdlib.h>

/* função main inicia a execução do programa */
int main( void )
{
   int frequency1 = 0; /* contador de lançamento 1 */
   int frequency2 = 0; /* contador de lançamento 2 */
   int frequency3 = 0; /* contador de lançamento 3 */
   int frequency4 = 0; /* contador de lançamento 4 */
   int frequency5 = 0; /* contador de lançamento 5 */
   int frequency6 = 0; /* contador de lançamento 6 */

   int roll; /* contador de lançamento, valor de 1 a 6000 */
   int face; /* representa o valor de um dado lançado, de 1 a 6 */

   /* loop 6000 vezes e resume resultados */
   for ( roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6; /* número aleatório de 1 a 6 */

      /* determina valor da face e incrementa contador apropriado */
      switch ( face ) {

         case 1: /* valor foi 1 */
            ++frequency1;
            break;

```

Figura 5.8 ■ Lançando um dado de seis lados 6000 vezes. (Parte 1 de 2.)

Geração de números aleatórios

```
30            case 2: /* valor foi 2 */
31               ++frequency2;
32               break;
33
34            case 3: /* valor foi 3 */
35               ++frequency3;
36               break;
37
38            case 4: /* valor foi 4 */
39               ++frequency4;
40               break;
41
42            case 5: /* valor foi 5 */
43               ++frequency5;
44               break;
45
46            case 6: /* valor foi 6 */
47               ++frequency6;
48               break; /* opcional */
49         } /* fim do switch */
50      } /* fim do for */
51
52      /* exibe resultados em formato tabular */
53      printf( "%s%13s\n", "Face", "Frequência" );
54      printf( "    1%13d\n", frequency1 );
55      printf( "    2%13d\n", frequency2 );
56      printf( "    3%13d\n", frequency3 );
57      printf( "    4%13d\n", frequency4 );
58      printf( "    5%13d\n", frequency5 );
59      printf( "    6%13d\n", frequency6 );
60      return 0; /* indica conclusão bem-sucedida */
61   } /* fim do main */
```

```
Face         Frequência
   1               1003
   2               1017
   3                983
   4                994
   5               1004
   6                999
```

Figura 5.8 ■ Lançando um dado de seis lados 6000 vezes. (Parte 2 de 2.)

Geração de números aleatórios

- **Para mostrar que esses números ocorrem com, aproximadamente, a mesma probabilidade, simularemos 6000 lançamentos de um dado usando o programa da Figura 5.8.**
- **Cada inteiro de 1 a 6 deverá aparecer, aproximadamente, 1000 vezes.**
- **Como vemos na saída do programa, ao escalar e deslocar, usamos a função rand para simular de modo realista o lançamento de um dado de seis lados.**
- ***Nenhum* caso default case é fornecido na estrutura switch.**

Geração de números aleatórios

- Observe também o uso do especificador de conversão %s para imprimir as strings de caracteres "Face" e "Frequency" como cabeçalhos de coluna (linha 53).
- Depois de estudarmos os arrays no Capítulo 6, mostraremos como substituir a estrutura switch inteira por uma instrução de linha única de modo elegante.

Geração de números aleatórios

- **A execução do programa da Figura 5.7 novamente produz exatamente a mesma sequência de valores..**
- **Como eles podem ser números aleatórios? Ironicamente, esse recurso de repetição é uma característica importante da função rand.**

Geração de números aleatórios

- Ao depurar um programa, isso é essencial para provar que as correções em um programa funcionam de modo apropriado.
- A função rand, na verdade, gera números pseudoaleatórios.
- Chamar rand repetidamente produz uma sequência de números que parece aleatória.
- Porém, a sequência se repete toda vez que o programa é executado.
- Quando um programa tiver sido totalmente depurado, ele poderá ser condicionado a produzir uma sequência diferente de números aleatórios a cada execução.

Geração de números aleatórios

```c
/* Fig. 5.9: fig05_09.c
   Randomizando o programa de lançamento de dado */
#include <stdlib.h>
#include <stdio.h>

/* função main inicia a execução do programa */
int main( void )
{
   int i; /* contador */
   unsigned seed; /* número usado para criar semente do gerador de número aleatório */

   printf( "Digite a semente: " );
   scanf( "%u", &seed ); /* observe o %u de unsigned */

   srand( seed ); /* inicia gerador de número aleatório */

   /* loop 10 vezes */
   for ( i = 1; i <= 10; i++ ) {

      /* escolhe número aleatório de 1 a 6 e o imprime */
      printf( "%10d", 1 + ( rand() % 6 ) );

      /* se o contador é divisível por 5, inicia nova linha de impressão */
      if ( i % 5 == 0 ) {
         printf( "\n" );
      } /* fim do if */
   } /* fim do for */

   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */
```

Geração de números aleatórios

```
Digite a semente: 67
        6       1       4       6       2
        1       6       1       6       4

Digite a semente: 867
        2       4       6       1       6
        1       1       3       6       2

Digite a semente: 67
        6       1       4       6       2
        1       6       1       6       4
```

Figura 5.9 ■ Randomizando o programa de lançamento de um dado.

Geração de números aleatórios

- Isso é chamado de randomização, e ocorre na função srand da biblioteca-padrão.
- A função srand usa um argumento inteiro unsigned e semeia a função rand para que ela produza uma sequência diferente de números aleatórios a cada execução do programa.
- Demonstramos srand na Figura 5.9.

Geração de números aleatórios

- Uma variável do tipo unsigned também é armazenada em pelo menos dois bytes da memória.
- Um unsigned int de dois bytes só pode ter valores positivos no intervalo de 0 a 65535.
- Um unsigned int de quatro bytes só pode ter valores positivos no intervalo de 0 a 4294967295.
- A função srand usa um valor unsigned como um argumento.
- O especificador de conversão %u é usado para ler um valor unsigned com scanf.
- O protótipo de função para srand é encontrado em <stdlib.h>.

Geração de números aleatórios

- Execute o programa várias vezes e observe os resultados. Note que uma sequência diferente de números aleatórios é obtida toda vez que o programa é executado, desde que uma semente diferente seja fornecida.
- Para randomizar sem incluir uma semente a cada vez, use um comando como

```
srand( time( NULL ) );
```

- Isso faz com que o computador leia seu clock para obter o valor da semente automaticamente.
- A função time retorna o número de segundos que se passaram desde a meia-noite de 1º de janeiro de 1970.

Geração de números aleatórios

- Esse valor é convertido em um inteiro não sinalizado, e é usado como semente do gerador de números aleatórios.
- A função time recebe NULL como um argumento (time é capaz de lhe oferecer uma string que represente o valor que ela retorna; NULL desativa essa capacidade para uma chamada específica a time).
- O protótipo de função para time está em <time.h>.

Geração de números aleatórios

- Os valores produzidos diretamente por rand estão no intervalo:

  0 £ rand() £ RAND_MAX

- Como você já sabe, o comando a seguir simula o lançamento de um dado de seis lados:

  face = 1 + rand() % 6;

- Essa instrução sempre atribui um valor inteiro (aleatório) à variável face no intervalo1 £ face £ 6.
- A largura desse intervalo (ou seja, o número de inteiros consecutivos no intervalo) é 6, e o número inicial do intervalo é 1.

Geração de números aleatórios

- Com relação ao comando anterior, vemos que a largura do intervalo é determinada pelo número usado para escalar rand o operador de módulo (ou seja, 6), e que o número inicial do intervalo é igual ao número (ou seja, 1) que é somado a rand % 6.
- Podemos generalizar esse resultado da seguinte forma:

```
n = a + rand() % b;
```

- onde a é o valor de deslocamento (que é igual ao primeiro número do intervalo desejado de inteiros consecutivos) e b é o fator de escala (que é igual à largura do intervalo desejado de inteiros consecutivos).

Geração de números aleatórios

Erro comum de programação 5.9

Usar srand *no lugar de* rand *para gerar números aleatórios.*

Exemplo: um jogo de azar

```c
/* Fig. 5.10: fig05_10.c
   Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contém protótipo para função time */

/* constantes de enumeração representam status do jogo */
enum Status { CONTINUE, WON, LOST };

int rollDice( void ); /* protótipo de função */

/* função main inicia a execução do programa */
int main( void )
{
   int sum; /* soma dos dados lançados */
   int myPoint; /* ponto ganho */

   enum Status gameStatus; /* pode conter CONTINUE, WON ou LOST */

   /* randomiza gerador de número aleatório usando hora atual */
   srand( time( NULL ) );

   sum = rollDice(); /* primeiro lançamento dos dados */

   /* determina status do jogo com base na soma dos dados */
   switch( sum ) {

      /* vence na primeira jogada */
      case 7:
      case 11:
         gameStatus = WON;
         break;

      /* perde na primeira jogada */
```

Exemplo: um jogo de azar

```
35          case 2:
36          case 3:
37          case 12:
38             gameStatus = LOST;
39             break;
40
41          /* lembra ponto */
42          default:
43             gameStatus = CONTINUE;
44             myPoint = sum;
45             printf( "Ponto é %d\n", myPoint );
46             break; /* optional */
47       } /* fim do switch */
48
49       /* enquanto jogo não termina */
50       while ( gameStatus == CONTINUE ) {
51          sum = rollDice(); /* joga dados novamente */
52
53          /* determina status do jogo */
54          if ( sum == myPoint ) { /* vence fazendo ponto */
55             gameStatus = WON; /* jogo termina, jogador vence */
56          } /* fim do if */
57        else {
```

Figura 5.10 ■ Programa que simula o jogo de craps. (Parte 1 de 2.)

Exemplo: um jogo de azar

```
58            if ( sum == 7 ) { /* perde por lançar 7 */
59               gameStatus = LOST; /* jogo termina, jogador perde */
60            } /* fim do if */
61         } /* fim do else */
62      } /* fim do while */
63
64      /* mostra mensagem de vitória ou perda */
65      if ( gameStatus == WON ) { /* jogador venceu? */
66         printf( "Jogador vence\n" );
67      } /* fim do if */
68      else { /* jogador perdeu */
69         printf( "Jogador perde\n" );
70      } /* fim do else */
71
72      return 0; /* indica conclusão bem-sucedida */
73   } /* fim do main */
74
75   /* lança dados, calcula soma e exibe resultados */
76   int rollDice( void )
77   {
78      int die1; /* primeiro dado */
79      int die2; /* segundo dado */
80      int workSum; /* soma dos dados */
81
82      die1 = 1 + ( rand() % 6 ); /* escolhe valor aleatório die1 */
83      die2 = 1 + ( rand() % 6 ); /* escolhe valor aleatório die2 */
84      workSum = die1 + die2; /* soma die1 e die2 */
85
86      /* exibe resultados dessa jogada */
87      printf( "Jogador rolou %d + %d = %d\n", die1, die2, workSum );
88      return workSum; /* retorna soma dos dados */
89   } /* fim da função rollRice */
```

Figura 5.10 ■ Programa que simula o jogo de craps. (Parte 2 de 2.)

Exemplo: um jogo de azar

```
    Jogador lançou 5 + 6 = 11
    Jogador vence
```

```
    Jogador lançou 4 + 1 = 5
    Ponto é 5
    Jogador lançou 6 + 2 = 8
    Jogador lançou 2 + 1 = 3
    Jogador lançou 3 + 2 = 5
    Jogador vence
```

```
    Jogador lançou 1 + 1 = 2
    Jogador perde
```

```
    Jogador lançou 6 + 4 = 10
    Ponto é 10
    Jogador lançou 3 + 4 = 7
    Jogador perde
```

Figura 5.11 ■ Exemplos de jogadas de craps.

Exemplo: um jogo de azar

- Um dos jogos de azar mais populares é o jogo de dados conhecido como ‘craps’, que é jogado em cassinos e becos no mundo inteiro. As regras do jogo são simples:
  - Um jogador lança dois dados. Cada dado tem seis faces. Essas faces contêm 1, 2, 3, 4, 5 e 6 pontos. Depois que os dados param, a soma dos pontos nas duas faces voltadas para cima é calculada. Se a soma for 7 ou 11 na primeira jogada, o jogador vence. Se a soma for 2, 3 ou 12 na primeira jogada (chamado ‘craps’), o jogador perde (ou seja, a ‘casa’ vence). Se a soma for 4, 5, 6, 8, 9 ou 10 na primeira jogada, então a soma se torna o ‘ponto’ do jogador. Para vencer, o jogador precisa continuar lançando os dados até que ‘faça seu ponto’. O jogador perde lançando um 7 antes de fazer o ponto.
- A Figura 5.10 simula o jogo de craps, e a Figura 5.11 mostra vários exemplos de execução.

Exemplo: um jogo de azar

- Observe que, de acordo com as regras, o jogador deve começar o jogo lançando dois dados, e deverá fazer o mesmo em todos os lançamentos seguintes.
- Definimos uma função rollDice para lançar os dados e calcular e imprimir sua soma.
- A função rollDice é definida uma vez, mas é chamada em dois lugares no programa (linhas 23 e 51).
- O interessante é que rollDice não usa argumentos,
- de modo que indicamos void na lista de parâmetros (linha 76).
- A função rollDice retorna a soma dos dois dados, de modo que um tipo de retorno int é indicado no cabeçalho da função.

Exemplo: um jogo de azar

- O jogador pode vencer ou perder no primeiro lançamento, ou pode vencer ou perder em qualquer um dos lançamentos seguintes.
- A variável gameStatus, definida para ser de um novo tipo — enum Status —, armazena o status atual.
- A linha 8 cria um tipo definido pelo programador, chamado de enumeração.
- Uma enumeração, introduzida pela palavra-chave enum, um conjunto de constantes inteiras representadas por identificadores.
- Às vezes, constantes de enumeração são chamadas
- de constantes simbólicas.
- Os valores em um enum começam com 0 e são aumentados em 1.

Exemplo: um jogo de azar

- **Na linha 8, a constante CONTINUE tem o valor 0, WON tem o valor 1 e LOST tem o valor 2.**
- **Também é possível atribuir um valor inteiro para cada identificador em um enum (ver Capítulo 10).**
- **Os identificadores em uma enumeração devem ser exclusivos, mas os valores podem ser duplicados.**

Exemplo: um jogo de azar

Erro comum de programação 5.10
Atribuir um valor a uma constante de enumeração depois de ela ter sido definida é um erro de sintaxe.

Erro comum de programação 5.11
Use apenas letras maiúsculas nos nomes de constantes de enumeração para fazer com que elas se destaquem em
um programa e indicar que as constantes de enumeração não são variáveis.

Exemplo: um jogo de azar

- Quando se vence o jogo, seja no primeiro lançamento seja em outro qualquer, o gameStatus é definido como WON.
- Quando se perde o jogo, seja no primeiro lançamento seja em outro qualquer, o gameStatus é definido como LOST.
- Caso contrário, o gameStatus é definido como CONTINUE e o jogo continua.
- Depois do primeiro lançamento, se o jogo terminar, a estrutura while (linha 50) é desprezada, pois o gameStatus não é CONTINUE.
- O programa prossegue para a estrutura if...else na linha 65, que exibe "Jogador vence", se o gameStatus for WON e "Jogador perde" se o gameStatus for LOST.

Exemplo: um jogo de azar

- Depois do primeiro lançamento, se o jogo não terminar, então sum é salva em myPoint.
- A execução prossegue com a estrutura while (linha 50), pois o gameStatus é CONTINUE.
- Toda vez que o while se repete, rollDice é chamada para produzir uma nova sum.
- Se sum combinar com myPoint, o gameStatus é definido como WON para indicar que o jogador venceu, o teste do while falha, a estrutura if...else (linha 65) exibe "Jogador vence" e a execução é concluída.

Exemplo: um jogo de azar

- Se sum é igual a 7 (linha 58), gameStatus é definido como LOST para indicar que o jogador perdeu, o teste de while falha, a estrutura if...else a estrutura 'Jogador perde' e a execução é concluída.
- Observe a arquitetura de controle interessante do programa.
- Usamos duas funções — main e rollDice —, e as estruturas switch, while, if...else aninhada e if aninhada.
- Nos exercícios, investigaremos diversas características interessantes do jogo de craps.

Classes de armazenamento

- Nos capítulos 2 a 4, usamos identificadores para nomes de variáveis.
- Os atributos das variáveis incluem nome, tipo, tamanho e valor.
- Neste capítulo, também usamos identificadores para nomes de funções definidas pelo usuário.
- Em um programa, na realidade, os identificadores têm outros atributos, como classe de armazenamento, duração do armazenamento, escopo e vinculação.
- C oferece quatro classes de armazenamento, indicadas pelos especificadores de classe de armazenamento: auto, register, extern e static.
- A classe de armazenamento de um identificador determina sua duração de armazenamento, escopo e vinculação.
- A duração de armazenamento de um identificador é o período durante o qual o identificador existe na memória.

Classes de armazenamento

- Alguns existem por pouco tempo, alguns são criados e destruídos repetidamente e outros existem por toda a execução de um programa.
- O escopo de um identificador é o *local* em que o identificador pode ser referenciado em um programa.
- Alguns podem ser referenciados por um programa; outros, apenas por partes de um programa.
- A vinculação de um identificador é determinante em um programa de múltiplos arquivos-fonte (assunto que veremos no Capítulo 14), seja o identificador conhecido apenas no arquivo-fonte atual, seja em qualquer arquivo-fonte com declarações apropriadas.
- Esta seção discute as classes de armazenamento e a duração do armazenamento.

Classes de armazenamento

- A Seção 5.13 abordará o escopo.
- Adiante discutiremos sobre a vinculação do identificador e a programação com múltiplos arquivos fonte.
- Os quatro especificadores de classe de armazenamento podem ser divididos em duas durações de armazenamento: duração de armazenamento automático e duração de armazenamento estático.
- As palavras-chave auto e register são usadas para declarar variáveis de duração de armazenamento automático.
- As variáveis com duração de armazenamento automático são criadas quando o bloco em que estão definidas é iniciado; elas existirão enquanto o bloco estiver ativo, e serão destruídas quando o bloco terminar sua execução.

Classes de armazenamento

- Apenas variáveis podem ter duração de armazenamento automático.
- As variáveis locais de uma função (aquelas declaradas na lista de parâmetros ou no corpo da função) normalmente possuem duração de armazenamento automático.
- A palavra-chave auto declara explicitamente variáveis de duração de armazenamento automático..

Classes de armazenamento

- Por exemplo, a declaração a seguir indica que as variáveis double x e y são variáveis locais automáticas, e existem apenas no corpo da função em que a declaração aparece:

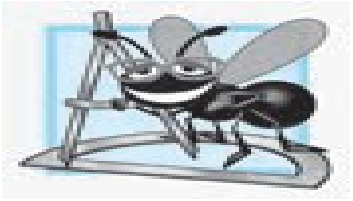
```
auto double x, y;
```

- As variáveis locais possuem duração de armazenamento automático como padrão, de modo que a palavra-chave auto raramente é usada.
- A partir de agora, passaremos a nos referir a variáveis com duração de armazenamento automático simplesmente como variáveis automáticas.

Classes de armazenamento

Dica de desempenho 5.1

O armazenamento automático é um meio de economizar memória, pois as variáveis automáticas existem apenas quando são necessárias. Elas são criadas quando uma função é iniciada, e são destruídas quando a função termina.



Observação sobre engenharia de software 5.10

O armazenamento automático é um exemplo do **princípio do menor privilégio** *— ele permite o acesso aos dados somente quando eles são realmente necessários. Por que ter variáveis acessíveis e armazenadas na memória quando, na verdade, elas não são necessárias?*

Classes de armazenamento

- **Os dados na versão da linguagem de máquina de um programa normalmente são carregados em registradores para cálculos e outros tipos de processamento.**

Classes de armazenamento

Dica de desempenho 5.2

O especificador de classe de armazenamento register pode ser colocado antes de uma declaração de variável automática para sugerir que o compilador mantenha a variável em um dos registradores de hardware de alta velocidade do computador. Se variáveis muito utilizadas, como contadores ou totais, puderem ser mantidas nos registradores do hardware, poderemos eliminar o overhead da carga repetitiva das variáveis da memória para os registradores e o armazenamento dos resultados na memória.

Classes de armazenamento

- É possível que o compilador ignore declarações register.
- Por exemplo, pode não haver um número suficiente de registradores disponíveis para o compilador utilizar.
- A declaração a seguir sugere que a variável inteira contador seja colocada em um dos registradores
- do computador e inicializada em 1:
  - `register int contador = 1;`
- A palavra-chave register só pode ser usada com variáveis de duração de armazenamento automática.

Classes de armazenamento

Dica de desempenho 5.3

Normalmente, declarações register são desnecessárias. Os compiladores otimizados de hoje são capazes de reconhecercas variáveis usadas com frequência, e podem decidir colocá-las nos registradores sem que uma declaração
register seja necessária.

Classes de armazenamento

- As palavras-chave extern e static são usadas nas declarações de identificadores para variáveis e funções de duração de armazenamento estático.
- Os identificadores de duração de armazenamento estático existem a partir do momento em que um programa inicia sua execução.
- Para variáveis estáticas, o armazenamento é alocado e inicializado uma vez, quando o programa é iniciado.
- Para funções, o nome da função existe quando o programa inicia sua execução.

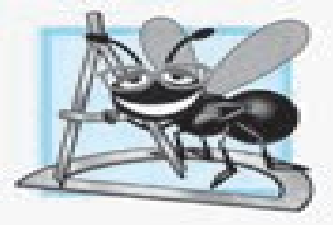
Classes de armazenamento

- Embora as variáveis e os nomes de funções existam desde o início da execução do programa, isso não significa que esses identificadores possam ser acessados do começo ao fim do programa.
- A duração do armazenamento e o escopo (onde um nome pode ser usado) são duas questões diferentes, como veremos na Seção 5.13.
- Existem dois tipos de identificadores com duração de armazenamento estático: identificadores externos (como variáveis globais e nomes de funções) e variáveis locais declaradas com o especificador de classe de armazenamento static.
- Variáveis globais e nomes de funções são da classe de armazenamento extern , como padrão.

Classes de armazenamento

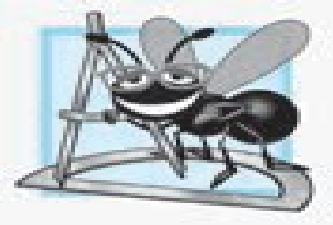
- As variáveis globais são criadas a partir da colocação das declarações de variável fora de qualquer definição de função, e elas reterão os seus valores durante toda a execução do programa.
- As variáveis globais e funções podem ser referenciadas por qualquer função que siga suas declarações ou definições no arquivo.
- Este é um motivo para que se use protótipos de função — ao incluirmos stdio.h em um programa que chama printf, o protótipo de função é colocado no início de nosso arquivo para tornar o nome printf conhecido do restante do arquivo.

Classes de armazenamento



Observação sobre engenharia de software 5.11

Descrever uma variável como global em vez de local permite que efeitos colaterais involuntários ocorram quando uma função que não precisa de acesso à variável acidentalmente, ou intencionalmente, a modifica. Em geral, o uso de variáveis globais deve ser evitado, exceto nas situações em que um desempenho exclusivo é requerido (conforme discutiremos no Capítulo 14).



Observação sobre engenharia de software 5.12

As variáveis usadas em apenas uma função em particular devem ser definidas como variáveis locais, e não variáveis externas, nessa mesma função.

Classes de armazenamento

- **Variáveis locais declaradas que contenham a palavra-chave static também são conhecidas apenas na função em que são definidas, porém, diferentemente das variáveis automáticas, variáveis locais static retêm seu valor quando a função termina.**
- **Da próxima vez em que a função for chamada, a variável local static conterá o valor que ela tinha quando a função foi executada pela última vez.**
- **O comando a seguir declara a variável local count com uma static, e ela será inicializada em 1.**
  - `static int count = 1;`

Classes de armazenamento

- **Todas as variáveis numéricas de duração de armazenamento estático são inicializadas em zero se não forem inicializadas explicitamente.**
- **As palavras-chave extern e static possuem um significado especial quando aplicadas explicitamente a identificadores externos.**
- **No Capítulo 14, discutiremos o uso explícito de extern e static com identificadores externos e programas com múltiplos arquivos-fonte.**

Regras de escopo

- O escopo de um identificador é a parte do programa em que o identificador pode ser referenciado.
- Por exemplo, ao definirmos uma variável local em um bloco, ela só poderá ser referenciada após sua definição nesse bloco, ou em blocos aninhados dentro desse bloco.
- Os quatro escopos de identificador são escopo de função, escopo de arquivo, escopo de bloco e escopo de protótipo de função.
- Os labels (identificadores seguidos por um sinal de dois pontos, como start:) são os únicos identificadores com escopo de função.
- Eles podem ser usados em qualquer lugar na função em que aparecerem, mas não podem ser referenciados fora do corpo da função.

Regras de escopo

- Os labels são usados em estruturas switch (como nos labels de case) e em comandos goto
- Eles são detalhes de implementação que as funções ocultam uma da outra.
- Essa ocultação — mais formalmente chamada ocultação de informações — é um meio de implementar o princípio do menor privilégio, um dos princípios mais fundamentais da boa engenharia de software.
- Um identificador declarado fora de qualquer função tem escopo de arquivo.
- Esse identificador é 'conhecido' (ou seja, acessível) em todas as funções, a partir do ponto em que é declarado até o final do arquivo.

Regras de escopo

- Todas as variáveis globais, definições de função e protótipos de função colocados fora de uma função possuem escopo de arquivo.
- Os identificadores definidos dentro de um bloco têm escopo de bloco.
- O escopo de bloco termina na chave direita (}) do bloco.
- As variáveis locais definidas no início de uma função possuem escopo de bloco, assim como os parâmetros de função, que são considerados variáveis locais pela função.
- Qualquer bloco pode conter definições de variável.

Regras de escopo

- **Quando os blocos são aninhados e um identificador em um bloco mais externo tem o mesmo nome de um identificador em um bloco interno, o identificador no bloco externo é 'ocultado' até que o bloco interno termine.**
- **Isso significa que, ao executar no bloco interno, este vê o valor de seu próprio identificador local, e não o valor do identificador com o mesmo nome no bloco que o delimita.**
- **As variáveis locais declaradasd static ainda possuem escopo de bloco, embora existam desde o momento em que o programa iniciou sua execução.**

Regras de escopo

- Assim, a duração do armazenamento não afeta o escopo de um identificador.
- Os únicos identificadores com escopo de protótipo de função são aqueles usados na lista de parâmetros de um protótipo de função.
- Como já dissemos, os protótipos de função não exigem nomes na lista de parâmetros — apenas os tipos são obrigatórios.
- Se um nome for usado na lista de parâmetros de um protótipo de função, o compilador desprezará esse nome.
- Os identificadores usados em um protótipo de função podem ser reutilizados em qualquer lugar no programa sem causar ambiguidades.

Regras de escopo

Erro comum de programação 5.12

Usar acidentalmente o mesmo nome para um identificador em um bloco interno e em um bloco externo, quando na verdade você deseja que o identificador no bloco externo esteja ativo durante a execução do bloco interno.

Dica de prevenção de erro 5.3

Evite nomes de variáveis que ocultem nomes em escopos externos. Isso pode ser feito simplesmente ao evitar o uso
de identificadores duplicados em um programa.

Regras de escopo

```
1  /* Fig. 5.12: fig05_12.cC
2     Um exemplo de escopo */
3  #include <stdio.h>
4
5  void useLocal( void ); /* protótipo de função */
6  void useStaticLocal( void ); /* protótipo de função */
7  void useGlobal( void ); /* protótipo de função */
8
9  int x = 1; /* variável global */
10
11 /* função main inicia a execução do programa */
12 int main( void )
13 {
14    int x = 5; /* variável local para main */
15
16    printf("x local no escopo externo de main é %d\n", x );
17
18    { /* inicia novo escopo */
19       int x = 7; /* variavel local para novo escopo */
20
21       printf( "x local no escopo interno de main é %d\n", x );
22    } /* fim do novo escopo */
23
24    printf( "x local no escopo externo de main é %d\n", x );
25
26    useLocal(); /* useLocal tem x local automática */
27    useStaticLocal(); /* useStaticLocal tem x local estática */
28    useGlobal(); /* useGlobal usa x global */
29    useLocal(); /* useLocal reinicializa x local automática */
30    useStaticLocal(); /* x local estática retém seu valor anterior */
31    useGlobal(); /* x global também retém seu valor */
32
33    printf( "\nx local em main é %d\n", x );
34    return 0; /* indica conclusão bem-sucedida */
35 } /* fim do main */
36
```

Regras de escopo

```
37  /* useLocal reinicializa variável local x durante cada chamada */
38  void useLocal( void )
39  {
40     int x = 25; /* inicializada toda vez que useLocal é chamada */
41
42     printf( "\nx local em useLocal é %d após entrar em useLocal\n", x );
43     x++;
44     printf( "x local em useLocal é %d antes de sair de useLocal\n", x );
45  } /* fim da função useLocal */
46
47  /* useStaticLocal inicializa variável local estática x somente na
48     primeira vez em que essa função é chamada; o valor de x é
49     salvo entre as chamadas a essa função */
50  void useStaticLocal( void )
```

Figura 5.12 ■ Exemplo de escopo. (Parte 1 de 2.)

Regras de escopo

```
51  {
52     /* inicializada apenas na primeira vez que useStaticLocal é chamada */
53     static int x = 50;
54
55     printf( "\nx estática local é %d na entrada de useStaticLocal\n", x );
56     x++;
57     printf( "x estática local é %d na saída de useStaticLocal\n", x );
58  } /* fim da função useStaticLocal */
59
60  /* função useGlobal modifica variável global x durante cada chamada */
61  void useGlobal( void )
62  {
63     printf( "\nx global é %d na entrada de useGlobal\n", x );
64     x *= 10;
65     printf( "x global é %d na saída de useGlobal\n", x );
66  } /* fim da função useGlobal */
```

Regras de escopo

```
x local no escopo externo de main é 5
x local no escopo interno de main é 7
x local no escopo externo de main é 5

x local em useLocal é 25 após entrar em useLocal
x local em useLocal é 26 antes de sair de useLocal

x local estática é 50 na entrada de useStaticLocal
x local estática é 51 na saída de useStaticLocal

x global é 1 na entrada de useGlobal
x global é 10 na saída de useGlobal

x local em useLocal é 25 após entrar em useLocal
x local em useLocal é 26 antes de sair de useLocal

x local estática é 51 na entrada de useStaticLocal
x local estática é 52 na saída de useStaticLocal

x global é 10 na entrada de useGlobal
x global é 100 na saída de useGlobal

x local em main é 5
```

Figura 5.12 ■ Exemplo de escopo. (Parte 2 de 2.)

Regras de escopo

- A Figura 5.12 demonstra os problemas de escopo com as variáveis globais, variáveis locais automáticas e variáveis locais static.
- Uma variável global x é definida e inicializada em 1 (linha 9).
- Essa variável global é ocultada em qualquer bloco (ou função) em que um nome de variável x é definido.
- Em main, uma variável local x definida e inicializada em 5 (linha 14).
- Essa variável é então exibida para mostrar que o x global está ocultado em main.
- Em seguida, um novo bloco é definido em main com outra variável local x inicializada em 7 (linha 19).

Regras de escopo

- Essa variável é exibida para mostrar que ela oculta x no bloco externo de main.
- A variável x com valor 7 é automaticamente destruída quando o bloco termina, e a variável local x no bloco externo de main é exibida novamente para mostrar que ela não está mais oculta.
- O programa define três funções que não utilizam nenhum argumento, nem retornam nada.
- A função useLocal define uma variável automática x e a inicializa em 25 (linha 40).
- Quando useLocal é chamada, a variável é exibida, incrementada e exibida novamente antes da saída da função.

Regras de escopo

- Toda vez que essa função é chamada, a variável automática x é reinicializada em 25.
- A função useStaticLocal define uma variável static x e a inicializa em 50 (linha 53).
- As variáveis locais declaradas como static retêm seus valores mesmo quando estão fora do escopo.
- Quando useStaticLocal é chamada, x é exibida, incrementada e exibida novamente antes da saída da função.
- Da próxima vez em que essa função for chamada, a variável local static x terá o valor 51.
- A função useGlobal não define variável alguma.

Regras de escopo

- Portanto, quando ela se refere à variável x, o x global (linha 9) é usado.
- Quando useGlobal é chamada, a variável global é exibida, multiplicada por10 e exibida novamente antes da saída da função.
- Da próxima vez em que a função useGlobal for chamada, a variável global ainda terá seu
- valor modificado, 10.
- Por fim, o programa exibe a variável local x em main novamente (linha 33) para mostrar que nenhuma das chamadas de função modificou o valor de x, pois todas as funções se referiam a variáveis em outros escopos.

Recursão

- Os programas que discutimos geralmente são estruturados como funções que chamam umas às outras de maneira disciplinada, hierárquica.
- Para alguns tipos de problemas, é útil ter funções que chamam a si mesmas.
- Uma **função recursiva** é uma função que chama a si mesma direta ou indiretamente, por meio de outra função.
- A recursão é um assunto complexo, discutido em detalhes em cursos de ciência da computação de nível mais alto.
- Nesta e na próxima seção, apresentaremos exemplos simples de recursão.

Recursão

- A Figura 5.17, na Seção 5.16, resume os 31 exemplos e exercícios de recursão encontrados neste livro.
- Em primeiro lugar, consideramos a recursão conceitualmente, e depois examinamos vários programas que contêm funções recursivas.
- As técnicas recursivas para solução de problemas possuem diversos elementos em comum.
- Uma função recursiva é chamada para resolver um problema. Na verdade, a função sabe somente como resolver os casos mais simples, ou os chamados **casos básicos**.

Recursão

- **Se a função é chamada com um caso básico, ela simplesmente retorna um resultado.**
- **Se uma função é chamada com um problema mais complexo, ela divide o problema em duas partes conceituais: uma parte que ela sabe como fazer e uma parte que ela não sabe como fazer.**
- **Para tornar a recursão viável, a segunda parte precisa ser semelhante ao problema original, mas uma versão ligeiramente mais simples ou ligeiramente menor.**

Recursão

- Como esse novo problema se parece com o problema original, a função inicia (chama) uma nova cópia de si mesma para atuar sobre o problema menor — esse processo é conhecido como chamada recursiva, e também como etapa de recursão.
- A etapa de recursão também inclui a palavra-chave return, pois seu resultado será combinado com a parte do problema que a função sabia como resolver para formar um resultado que será passado de volta a quem a chamou originalmente, possivelmente, main.
- A etapa de recursão é executada enquanto a chamada original à função está aberta, ou seja, enquanto sua execução ainda não foi concluída..

Recursão

- A etapa de recursão pode resultar em muito mais dessas chamadas recursivas, pois a função continua dividindo cada problema com que é chamada em duas partes conceituais.
- Para que a recursão termine, toda vez que a função chama a si mesma dentro de uma versão ligeiramente mais simples do problema original, essa sequência de problemas menores, eventualmente, deverá convergir no caso básico.
- Nesse ponto, a função reconhece o caso básico, retorna um resultado para a cópia anterior da função e a sequência de retornos segue na fila até que a chamada original da função finalmente retorne o resultado final a main.

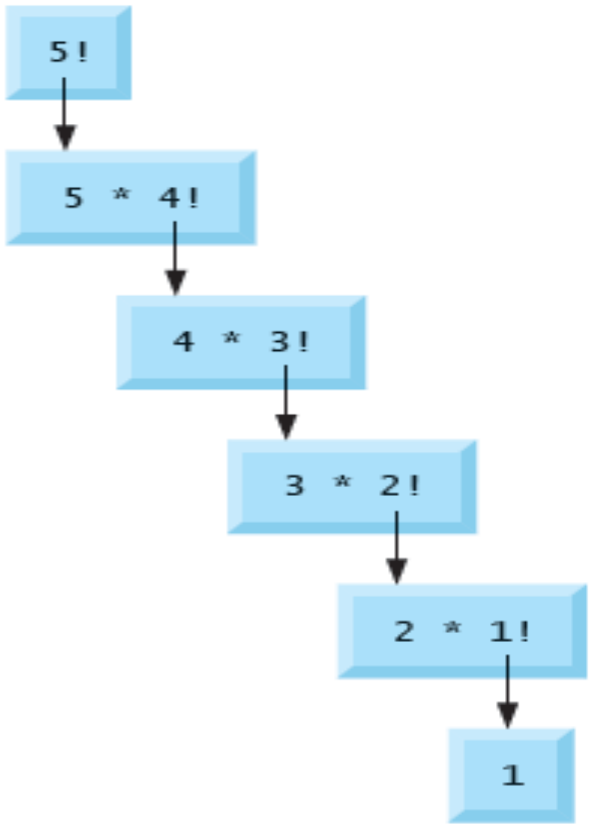
Recursão

- O fatorial de um inteiro não negativo *n*, escrito como *n*! (diz-se '*n* fatorial'), é o produto
  - $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$

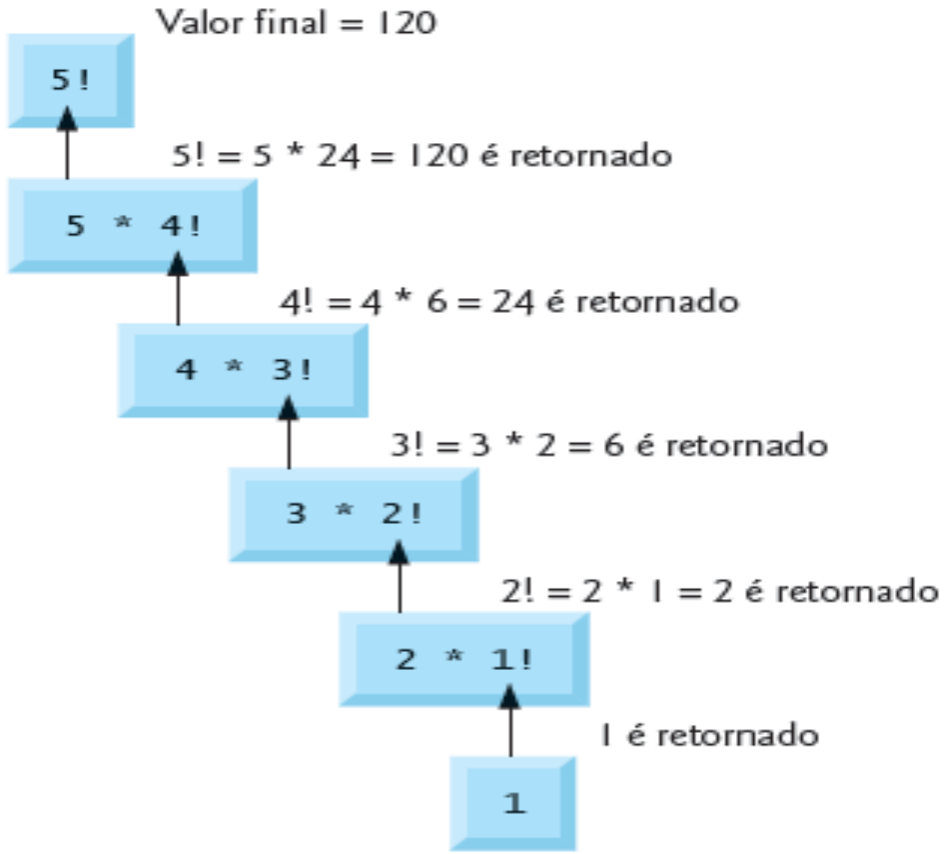
  com 1! igual a 1, e 0! definido como 1.
- Por exemplo, 5! é o produto 5 * 4 * 3 * 2 * 1, que é igual a 120.
- O fatorial de um inteiro, número, maior ou igual a 0, pode ser calculado iterativamente (não recursivamente) usando uma estrutura for da seguinte forma:

```
fatorial = 1;
for ( contador = número ; contador >= 1; contador-- )
   fatorial *= contador;
```

Recursão



(a) Sequência de chamadas recursivas.



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

Figura 5.13 ■ Avaliação recursiva de 5!.

Recursão

```
1  /* Fig. 5.14: fig05_14.c
2     Função recursiva fatorial */
3  #include <stdio.h>
4
5  long factorial( long number ); /* protótipo de função */
6
7  /* função main inicia a execução do programa */
8  int main( void )
9  {
10    int i; /* contador */
11
12    /* loop 11 vezes; durante cada iteração, calcula
13       fatorial( i ) e mostra o resultado */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, fatorial( i ) );
16    } /* fim do for */
17
18    return 0; /* indica conclusão bem-sucedida */
19 } /* fim do main */
20
21 /* definição recursiva da função fatorial */
22 long factorial( long number )
23 {
24    /* caso básico */
25    if ( number <= 1 ) {
26       return 1;
27    } /* fim do if */
28    else { /* etapa recursiva */
29       return ( number * factorial( number - 1 ) );
30    } /* fim do else */
31 } /* fim da função fatorial */
```

Figura 5.14 ■ Calculando fatoriais com uma função recursiva. (Parte 1 de 2.)

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

Figura 5.14 ■ Calculando fatoriais com uma função recursiva. (Parte 2 de 2.)

Recursão

- **Uma definição recursiva da função de fatorial é obtida observando-se o seguinte relacionamento:**

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

- **Por exemplo, 5! é nitidamente igual a 5 * 4!, como mostramos em:**

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

- **A avaliação de 5! prosseguiria como se pode ver na Figura 5.13.**

Recursão

- A Figura 5.13(a) mostra como a sucessão de chamadas recursivas prossegue até 1! ser avaliado como 1, o que encerra a recursão.
- A Figura 5.13(b) apresenta os valores retornados de cada chamada recursiva para a função que a chamou até que o valor final seja calculado e retornado.
- A Figura 5.14 usa a recursão para calcular e exibir os fatoriais dos inteiros de 0 a 10 (a escolha do tipo long será explicada em breve).
- A função recursiva fatorial primeiro testa se uma condição de término é verdadeira, ou seja, se number é menor ou igual a 1.

Recursão

- Se number é realmente menor ou igual a 1, fatorial retorna 1, nenhuma outra recursão é necessária e o programa é encerrado.
- Se number é maior que 1, o comando

  ```
  return number * fatorial( number - 1 );
  ```

- expressa o problema como o produto de number e de uma chamada recursiva a fatorial avaliando o fatorial de number - 1.
- A chamada fatorial(number - 1) é um problema ligeiramente mais simples do que o cálculo original fatorial(number).

Recursão

- A função fatorial (line 22) foi declarada para receber um parâmetro do tipo long e retornar um resultado do tipo long.
- Essa é uma notação abreviada para long int.
- O padrão em C especifica que uma variável do tipo long int armazenada em pelo menos 4 bytes, e que, portanto, pode manter um valor tão grande quanto +2147483647.
- Como podemos ver na Figura 5.14, os valores fatoriais crescem rapidamente. Escolhemos o tipo de dado long para que o programa possa calcular fatoriais maiores que 7! em computadores com inteiros pequenos (como 2 bytes).

Recursão

- O especificador de conversão %ld é usado para exibir valoreslong.
- Infelizmente, a função fatorial produz valores grandes tão rapidamente que até mesmo long int não nos ajuda a exibir muitos valores fatoriais antes que o tamanho de uma variável long int seja excedido.
- Ao explorar os exercícios, é possível que o usuário que queira calcular fatoriais de números maiores precise de double.

Recursão

- **Isso mostra uma fraqueza na C (e na maioria das outras linguagens de programação procedurais), já que a linguagem não é facilmente estendida a ponto de lidar com os requisitos exclusivos de diversas aplicações.**
- **Como veremos adiante, C++ é uma linguagem extensível que, por meio de 'classes', permite que criemos inteiros arbitrariamente grandes, se quisermos.**

Erro comum de programação 5.13

Esquecer de retornar um valor de uma função recursiva quando isso é necessário.

Erro comum de programação 5.14

Omitir o caso básico, ou escrever a etapa de recursão incorretamente, de modo que ela não convirja no caso básico, causará recursão infinita, o que, eventualmente, esgotará toda a memória. Esse problema é semelhante ao de um loop infinito em uma solução iterativa (não recursiva). A recursão infinita também pode ser causada por uma entrada não esperada.

Exemplo de uso da recursão: a série de Fibonacci

- A série de Fibonacci
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- começa com 0 e 1 e tem a propriedade de estabelecer que cada número de Fibonacci subsequente é a soma dos dois números de Fibonacci anteriores.
- A série ocorre na natureza e, em particular, descreve uma forma de espiral.
- A razão de números de Fibonacci sucessivos converge para um valor constante 1,618...

Exemplo de uso da recursão: a série de Fibonacci

- Esse número, também, ocorre repetidamente na natureza, e tem sido chamado de razão áurea, ou proporção áurea.
- As pessoas tendem a considerar a proporção áurea esteticamente atraente.
- Os arquitetos normalmente projetam janelas, salas e prédios cujo comprimento e largura estão na razão da proporção áurea.
- Cartões-postais normalmente são desenhados com uma razão comprimento/largura da proporção áurea.

Exemplo de uso da recursão: a série de Fibonacci

```
1  /* Fig. 5.15: fig05_15.c
2     Função recursiva fibonacci */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* protótipo de função */
6
7  /* função main inicia a execução do programa */
8  int main( void )
9  {
10    long result; /* valor de fibonacci */
11    long number; /* número fornecido pelo usuário */
12
13    /* obtém inteiro do usuário */
14    printf( "Digite um inteiro: " );
15    scanf( "%ld", &number );
16
17    /* calcula valor de fibonacci para número informado pelo usuário */
18    result = fibonacci( number );
19
20    /* mostra resultado */
21    printf( "Fibonacci( %ld ) = %ld\n", number, result );
22    return 0; /* indica conclusão bem-sucedida */
23 } /* fim do main */
24
```

Exemplo de uso da recursão: a série de Fibonacci

```
25  /* Definição recursiva da função fibonacci */
26  long fibonacci( long n )
27  {
28     /* caso básico */
29     if ( n == 0 |n == 1 ) {
30        return n;
31     } /* fim do if */
32     else { /* etapa recursiva */
33        return fibonacci( n - 1 ) + fibonacci( n - 2 );
34     } /* fim do else */
35  } /* fim da função fibonacci */
```

```
Digite um inteiro:   0
Fibonacci( 0 ) = 0
```

```
Digite um inteiro:   1
Fibonacci( 1 ) = 1
```

Figura 5.15 ■ Gerando números de Fibonacci por meio de recursão. (Parte 1 de 2.)

Exemplo de uso da recursão: a série de Fibonacci

```
Digite um inteiro:   2
Fibonacci( 2 ) = 1
```

```
Digite um inteiro:   3
Fibonacci( 3 ) = 2
```

```
Digite um inteiro:   4
Fibonacci( 4 ) = 3
```

```
Digite um inteiro:   5
Fibonacci( 5 ) = 5
```

```
Digite um inteiro:   6
Fibonacci( 6 ) = 8
```

```
Digite um inteiro:   10
Fibonacci( 10 ) =    55
```

```
Digite um inteiro:   20
Fibonacci( 20 ) =    6765
```

```
Digite um inteiro:   30
Fibonacci( 30 ) =    832040
```

```
Digite um inteiro:   35
Fibonacci( 35 ) =    9227465
```

Figura 5.15 ■ Gerando números de Fibonacci por meio de recursão. (Parte 2 de 2.)

Exemplo de uso da recursão: a série de Fibonacci

- A série de Fibonacci pode ser recursivamente definida da seguinte forma::

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

- A Figura 5.15 calcula recursivamente o *n*-ésimo número de Fibonacci, usando a função fibonacci.
- Observe que os números da série de Fibonacci tendem a crescer rapidamente.
- Portanto, escolhemos o tipo de dadolong para os tipos de parâmetro e de retorno na função fibonacci.
- Na Figura 5.15, cada par de linhas de saída mostra uma execução separada do programa.

Exemplo de uso da recursão: a série de Fibonacci

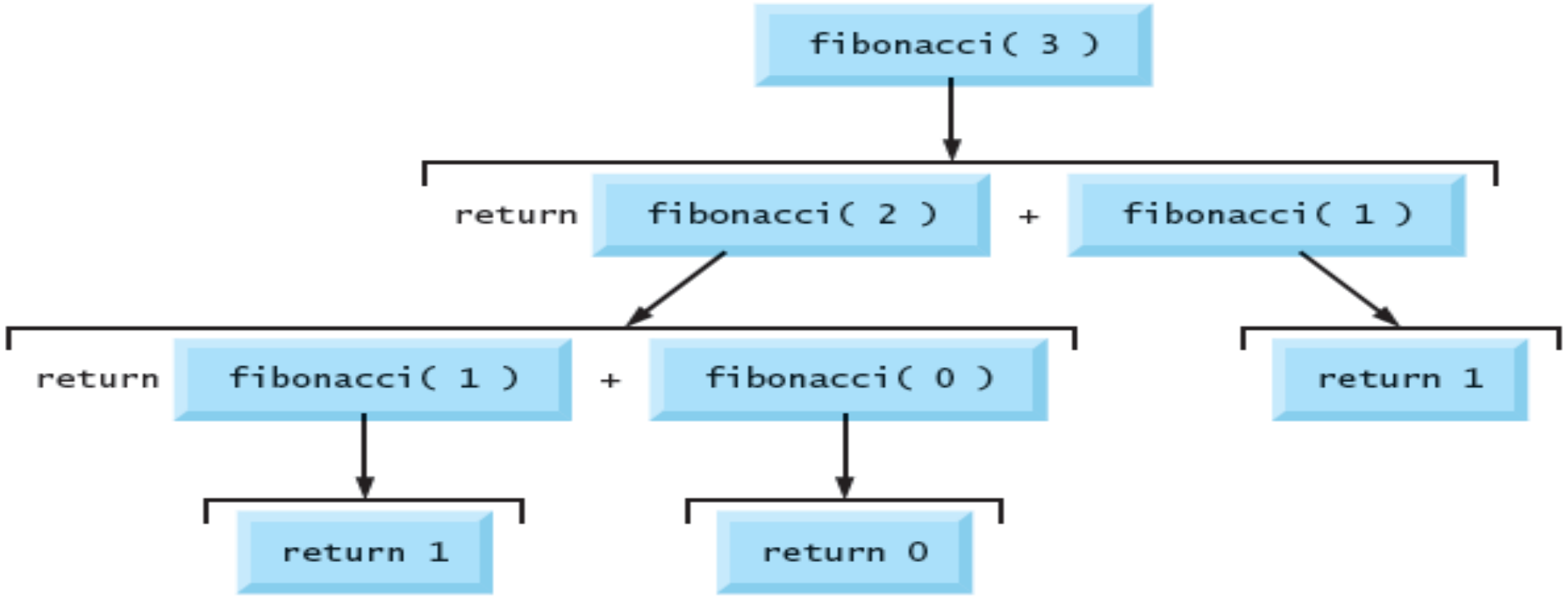


Figura 5.16 ■ Conjunto de chamadas recursivas para fibonacci(3).

Exemplo de uso da recursão: a série de Fibonacci

- A chamada para fibonacci a partir de main não é uma chamada recursiva (linha 18), mas todas as outras chamadas para fibonacci são recursivas (linha 33).
- Toda vez que fibonacci é chamada, ela imediatamente testa o caso básico — n é igual a 0 ou 1.
- Se isso for verdadeiro, n é retornado. É interessante que, se n for maior que 1, a etapa de recursão gerará *duas* chamadas recursivas, cada uma para um problema ligeiramente mais simples do que a chamada original para fibonacci.
- A Figura 5.16 mostra como a função fibonacci avaliaria fibonacci(3).

Exemplo de uso da recursão: a série de Fibonacci

- Essa figura levanta algumas questões interessantes sobre a ordem em que os compiladores em C avaliarão os operandos dos operadores.
- Este é um assunto diferente do da ordem em que os operadores são aplicados aos seus operandos, a saber, a ordem ditada pelas regras de precedência de operadores.
- Pela Figura 5.16, parece que, enquanto fibonacci(3), é avaliada, duas chamadas recursivas serão feitas, a saber, fibonacci(2) e fibonacci(1).
- Mas em que ordem essas chamadas serão feitas? A maioria dos programadores simplesmente supõe que os operandos serão avaliados da esquerda para a direita.

Exemplo de uso da recursão: a série de Fibonacci

- Estranhamente, o padrão em C não especifica a ordem em que os operandos da maioria dos operadores (incluindo +) devem ser avaliados.
- Portanto, você não deve fazer qualquer suposição sobre a ordem em que essas chamadas serão executadas.
- As chamadas poderiam realmente executar fibonacci(2) primeiro e, depois, fibonacci(1), , ou então poderiam executar na ordem contrária, fibonacci(1) e depois fibonacci(2).
- Nesse programa, e na maioria dos outros, o resultado final seria o mesmo..

Exemplo de uso da recursão: a série de Fibonacci

- **Porém, em alguns programas, a avaliação de um operando pode ter efeitos colaterais que podem afetar o resultado final da expressão.**
- **Dos muitos operadores da C, o padrão em 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, garantidamente, serão avaliados da esquerda para a direita.**

Exemplo de uso da recursão: a série de Fibonacci

- [Nota: as vírgulas usadas para separar os argumentos em uma chamada de função não são operadores de vírgula.]
- Seu operando mais à esquerda sempre é avaliado primeiro; se o operando mais à esquerda for avaliado como diferente de zero, o operando do meio será avaliado em seguida, e o último será ignorado; se o operando mais à esquerda for avaliado como zero, o terceiro será avaliado em seguida, e o operando do meio será ignorado.

Exemplo de uso da recursão: a série de Fibonacci

Erro comum de programação 5.15
Escrever programas que dependem da ordem de avaliação dos operandos dos operadores diferentes de &&, ||, ?: e
do operador de vírgula (,) pode ocasionar erros, pois não há garantia de que os compiladores avaliarão os operandos
na ordem em que você espera.

Dica de portabilidade 5.2
Os programas que dependem da ordem de avaliação dos operandos dos operadores diferentes de &&, ||, ?: e do operador de vírgula (,) podem funcionar de modos diferentes em compiladores distintos.

Exemplo de uso da recursão: a série de Fibonacci

- Um aviso deve ser dado sobre programas recursivos como aquele que usamos aqui para gerar os números de Fibonacci.
- Cada nível de recursão na função fibonacci tem um efeito duplo sobre o número de chamadas; ou seja, o número de chamadas recursivas que serão executadas para calcular o *n*-ésimo número de Fibonacci está na ordem de 2^n. Isso fica fora de controle rapidamente.
- Calcular apenas o 20º número de Fibonacci exigiria 2^{20}, ou cerca de um milhão de chamadas, e calcular o 30o número de Fibonacci exigiria 2^{30}, ou cerca de um bilhão de chamadas, e assim por diante.

Exemplo de uso da recursão: a série de Fibonacci

- Os cientistas da computação chamam isso de complexidade exponencial.
- Problemas dessa natureza humilham até mesmo os computadores mais poderosos!
- Questões de complexidade em geral e de complexidade exponencial em particular são discutidas em detalhes em uma cadeira do curso de nível superior de ciências da computação, geralmente chamada de 'Algoritmos'.

Exemplo de uso da recursão: a série de Fibonacci

Dica de desempenho 5.4

Evite programas recursivos no estilo Fibonacci, que resultam em uma 'explosão' exponencial de chamadas.

Exemplo de uso da recursão: a série de Fibonacci

- O exemplo que mostramos nesta seção usou uma solução intuitivamente atraente para calcular os números de Fibonacci, mas existem técnicas melhores.
- O Exercício 5.48 pede que você investigue a recursão com mais profundidade e propõe técnicas alternativas de implementação do algoritmo de Fibonacci recursivo.

Recursão versus interação

- Nas seções anteriores, estudamos duas funções que podem ser facilmente implementadas, recursiva ou iterativamente.
- Nesta seção, compararemos as duas técnicas e discutiremos por que você deveria escolher uma ou outra técnica em uma situação em particular.
- Tanto a iteração quanto a recursão se baseiam em uma estrutura de controle: a iteração usa uma estrutura de repetição; a recursão usa uma estrutura de seleção.
- Ambas envolvem repetição: a iteração usa explicitamente uma estrutura de repetição; a recursão consegue a repetição por meio de chamadas de função repetitivas.

Recursão versus interação

- **Tanto uma quanto a outra envolvem testes de término: a iteração termina quando a condição de continuação do loop falha; a recursão termina quando um caso básico é reconhecido.**
- **A iteração continua a modificar um contador até que ele passe a ter um valor que faça com que a condição de continuação do loop falhe; a recursão continua a produzir versões mais simples do problema original até que o caso básico seja alcançado.**

Recursão versus interação

- **Tanto a iteração quanto a recursão podem ocorrer infinitamente: um loop infinito ocorre com iteração se o teste de continuação do loop nunca se tornar falso; a recursão infinita ocorre se a etapa de recursão não reduzir o problema a cada vez, de maneira que ele convirja para o caso básico.**
- **A recursão tem muitos pontos negativos. Ela chama o mecanismo repetidamente, e, por conseguinte, também gera um overhead (sobrecarga) com as chamadas de função.**
- **Isso pode ser dispendioso em tempo de processador e espaço de memória.**

Recursão versus interação

- Cada chamada recursiva faz com que outra cópia da função (na realidade, apenas as variáveis da função) seja criada; isso pode consumir uma memória considerável.
- A iteração normalmente ocorre dentro de uma função, de modo que o overhead de chamadas de função repetidas e a atribuição extra de memória sejam omitidos.
- Logo, por que escolher a recursão?

Recursão versus interação

Observação sobre engenharia de software 5.13

Qualquer problema que pode ser resolvido recursivamente também pode ser resolvido iterativamente (não recursivamente).

A técnica iterativa normalmente é preterida em favor da técnica recursiva quando esta espelha o problema mais naturalmente e resulta em um programa mais fácil de entender e depurar. Outro motivo para escolher uma solução recursiva é que uma solução iterativa pode não ser aparente.

Recursão versus interação

Dica de desempenho 5.5

Evite usar a recursão em situações de desempenho. As chamadas recursivas gastam tempo e consomem
memória adicional.

Erro comum de programação 5.16

Ter, acidentalmente, uma função não recursiva chamando a si mesma, direta ou indiretamente, por meio
de outra função.

Recursão versus interação

- **A Figura 5.17 resume, por capítulo, os 31 exemplos e exercícios de recursão no texto.**

Recursão versus interação

- Encerraremos este capítulo com algumas observações que fazemos repetidamente ao longo do livro.
- A boa engenharia de software é importante.
- O alto desempenho é importante.
- Infelizmente, esses objetivos normalmente estão em conflito um com o outro.
- A boa engenharia de software é a chave para tornar mais controlável a tarefa de desenvolver os sistemas de software maiores e mais complexos de que precisamos.
- Alto desempenho é a chave para realizar os sistemas do futuro, que aumentarão as demandas de computação do hardware.
- Onde as funções se encaixam aqui?

Recursão versus interação

Dica de desempenho 5.6

Criar programas utilizando funções de uma maneira elegante e hierárquica promove a boa engenharia de software. Mas isso tem um preço. Um programa rigorosamente dividido em funções — em comparação a um programa monolítico (ou seja, em uma parte) sem funções — cria, potencialmente, grandes quantidades de chamadas de função, que consomem tempo de execução no(s) processador(es) de um computador. Assim, embora os programas monolíticos possam funcionar melhor, eles são mais difíceis de programar, testar, depurar, manter e desenvolver.

Lembre-se

“ Primeiro, resolva o problema. Em seguida, escreva o código. ” John Johnson