CAPÍTULO 7

REÚSO DE CÓDIGO E DEPURAÇÃO DE PROGRAMAS

OBJETIVOS

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Reúso de código
  - ❒ Erro de execução
  - ❒ Erro de lógica
  - ❒ Erro de truncamento
  - ❒ IEEE 754
- Descrever vantagens e desvantagens decorrentes do uso de componentes de biblioteca num programa
- Beneficiar-se do uso de abreviações no IDE CodeBlocks
- Classificar erros de programação de acordo com sua complexidade
- Diferenciar teste e depuração de programas
- Empregar o método de busca binária em depuração
- Explicar como o compilador pode ajudar na tarefa de depuração de um programa
- Discorrer sobre a técnica de depuração que utiliza **printf()**
- Minuciar a diferença entre número real em Matemática e número de ponto flutuante em programação
- Descrever os significados dos **%f** e **%.*nf*** na função **printf()**
- Esclarecer por que apenas alguns poucos números reais podem ser representados precisamente num computador

7.1 Introdução

ESTE CAPÍTULO APRESENTA três tópicos de natureza prática que são essenciais para a evolução do aprendiz de programação. O primeiro tópico lida com **reúso de código**, que se refere ao uso de parte de um programa na construção de outro programa. Isto é, reúso é um recurso utilizado com o objetivo de economizar esforços, reduzindo trabalho considerado redundante, visto que já foi realizado anteriormente. Assim reúso de código busca evitar a reinvenção da roda e este capítulo apresenta recomendações práticas para aplicação eficiente desse recurso.

O segundo tópico tratado neste capítulo é depuração de programas. Esse tema é complexo quando os programas a serem depurados são de natureza complexa, mas aqui será tratado num nível compatível com o caráter introdutório deste livro.

O terceiro tema deste capítulo é um pouco indigesto para um principiante e pode ser saltado. Ele refere-se a uma categoria de erros que infeniza a vida do programador inexperiente (e muitos programadores experientes também), que é imprecisão na representação de números reais e, mais especificamente, erros decorrentes de truncamento. O leitor pode deixar para reportar-se à **Seção 7.5** que lida com essa agrura, apenas quando se deparar com ela.

7.2 Reúso de Código

Reúso de código refere-se ao fato de partes de um programa poderem ser usadas, com poucas alterações, na construção de outros programas. Esse é um tópico fundamental em programação que, infelizmente, só tem recebido a devida atenção em disciplinas de engenharia de software nas quais ele é tratado em profundidade. Mas, mesmo programas simples podem beneficiar-se do uso prático desse conceito e, de fato, ele tem sido usado diversas vezes ao longo dos capítulos precedentes. Por exemplo, quando usa uma função da biblioteca padrão de C num programa, o programador está beneficiando-se de reúso de software. Isto é, bibliotecas são criadas levando em consideração componentes que provavelmente serão necessários em programas diferentes. Bibliotecas oferecem ainda o benefício de serem suficientemente testadas, de modo que raramente encontra-se um bug. Entretanto, normalmente, bibliotecas não permitem que seus componentes sejam adaptados para necessidades específicas.

Em programação, muitos problemas são recorrentes, de modo que um programador experiente é capaz de identificar semelhanças entre um novo problema e um problema já resolvido e incorporar partes da solução do problema conhecido na solução do novo problema. Mesmo quando dois programas parecem ser completamente diferentes, sempre há algo de um programa que se pode usar em outro, como você aprenderá neste capítulo.

Costuma-se dizer que funções constituem a forma mais rudimentar de reúso de código, mas reúso pode usar unidades menores, como blocos e outros trechos de código. Ou seja, a prática generalizada de copiar e colar trechos de um programa em outro, apesar de rudimentar, também pode ser considerada reúso de código. Portanto tente encontrar semelhanças entre um problema que você esteja tentando resolver e outros problemas resolvidos antes. Talvez, você possa utilizar algum programa antigo como base e modificá-lo, acrescentar ou remover partes etc.

Programadores com alguma experiência raramente começam um programa a partir do nada. Muitas vezes, um programa é começado a partir de outro já existente, mesmo quando os dois programas resolvem problemas totalmente diferentes. Por exemplo, suponha que você tenha construído um programa que determina se um número é primo ou não, tal qual aquele apresentado na **Seção 5.11.2** (considere-o como sendo o *programa 1*) e que, agora, você precisa construir um programa para calcular o MDC de dois números, como o programa

apresentado na **Seção 5.11.3** (considere-o *programa 2*). Essas duas tarefas parecem ser bem díspares e, de fato, o são. Mas, há mais do primeiro programa que pode ser usado no segundo do que, talvez, você possa imaginar:

- Os dois programas devem iniciar com um comentário de bloco que os apresentam. Logo pode-se copiar o bloco inicial de comentário do programa 1 para o programa 2 e, depois, editar o comentário copiado no programa 2 onde for necessário.
- Os dois programas incluem os mesmos cabeçalhos. Portanto as diretivas de inclusão do programa 1 podem ser copiadas para o programa 2 sem necessidade de edição.
- Os dois programas usam a função `LeNatural()`. Novamente, a definição dessa função pode ser copiada para o programa 2 sem necessidade de edição.
- Qualquer programa hospedado de console requer uma função **main()**. A função **main()** do programa 2 deve ser diferente daquela do programa 1, mas, mesmo assim, é possível aproveitar alguns trechos:
  - As duas funções começam do mesmo modo. Isto é, pode-se copiar o comentário de bloco que inicia a função, seu cabeçalho e até o abre-chaves ({) pode ser aproveitado. Assim você pode copiar essas linhas e precisará editar apenas a parte do comentário que descreve aquilo que a função faz.
  - No corpo da função **main()** do programa 1, pode-se ainda usar: a instrução que apresenta o programa (que precisará ser editada), a instrução de despedida do programa (que, talvez, não precise ser editada), a instrução **return** e o fecha-chaves.
  - Programadores mais experientes são capazes de aproveitar outros trechos da função **main()**, mas, enquanto você não atinge o apogeu da programação, é melhor parar por aqui.

Depois de todo esse banho de reúso de código, que, provavelmente, não lhe tomará mais do que cinco minutos, o que resta a ser feito para obter um programa que calcula o MDC de dois números a partir de um programa que verifica se um número é primo? Bem, falta o principal que é resolver o problema proposto. Mas, pelo menos, você estará mais motivado pelo fato de ter ganho muito tempo.

7.3 Usando Abreviações com CodeBlocks

O editor de programas do IDE CodeBlocks oferece uma característica muito interessante que, quando utilizada, agiliza a produção de código. Trata-se da opção denominada **abreviações** (originalmente, *abbreviations*). Utilizando uma abreviação, o programador precisa apenas pressionar algumas poucas teclas para obter o esboço de uma instrução. Por exemplo, suponha que num dado local de um programa-fonte você deseja digitar uma instrução **if-else**. Então, você precisará digitar apenas *ife* seguido de `[CTRL]+[J]` para obter automaticamente a inclusão das seguintes linhas no seu programa:

```
if () {
   ;
} else {
   ;
}
```

Após incluir, essas linhas, o editor do CodeBlocks posiciona o cursor de edição no espaço entre parênteses para que você possa editar a expressão condicional da instrução **if**.

Além da vantagem de natureza prática mencionada, o uso de abreviações pode prevenir a ocorrência de erros de sintaxe (p. ex., esquecimento de abre ou fecha-chaves).

O editor do IDE CodeBlocks vem pré-configurado com várias abreviações que podem ser editadas ou removidas. O programador pode ainda incluir novas abreviações para instruções que ele usa com frequência. Para acessar as configurações de abreviações do editor do CodeBlocks, siga o seguinte procedimento:

1. Clique na opção *Editor...* do menu *Settings*.
2. Na janela de configuração do editor, clique sobre o ícone intitulado *Abbreviations* no painel da esquerda e você obterá dois painéis à direita denominados *Keywords* e *Code*, como mostra a **Figura 7–1**.

Para examinar o fragmento de programa associado a uma palavra enumerada no painel *Keywords*, clique sobre a palavra desejada e o fragmento aparecerá no painel *Code*. Por exemplo, na **Figura 7–1**, a palavra selecionada é *ife* e o fragmento de programa inserido quando essa palavra é acionada é mostrado no painel *Code*.

Existem três convenções básicas utilizadas na escrita dos fragmentos de programa que serão inseridos:

- Pontos representam espaços em branco (v. **Figura 7–1**).
- Seta apontando para a direita (→) representa tabulação (v. **Figura 7–2**). Para evitar futuras decepções, nunca use tabulações (v. **Seção 3.17.3**).
- Barra vertical (|) representa o local onde o cursor de edição será posicionado após a inserção do fragmento de programa. Portanto deve haver apenas uma barra vertical em qualquer fragmento de programa.

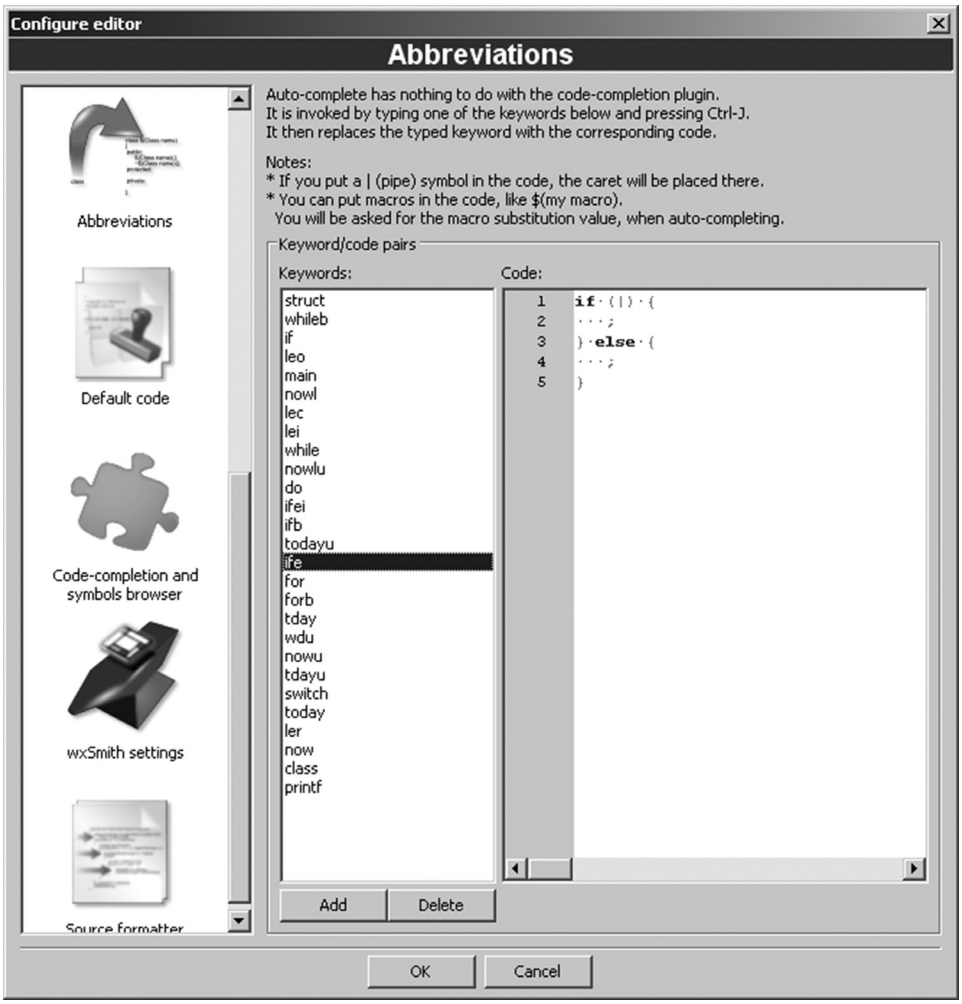

Figura 7–1: Abreviações em CodeBlocks 1

A **Figura 7–2** mostra o fragmento de programa que será inserido quando a palavra *if* é digitada seguida de `[CTRL]+[J]`. De acordo com a configuração deste fragmento, após sua inserção o cursor será posicionado entre os parênteses, conforme indicado pela barra vertical.

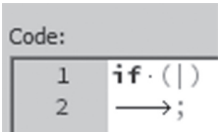

FIGURA 7–2: ABREVIAÇÕES EM CODEBLOCKS 2

Além das recomendações de configuração apresentadas na **Seção 3.17.3**, para usar abreviações sem inclusão de caracteres de tabulação, é preciso substituir manualmente todas as tabulações encontradas nos fragmentos de programas associados às abreviações que você usará por espaços em branco, como foi feito no fragmento de programa associado à palavra *ife* e mostrado na **Figura 7–1**.

7.4 Depuração de Programas

Nem mesmo os programadores mais experientes escrevem programas livres de erros em suas primeiras tentativas. Assim, uma grande parcela do tempo gasto em programação é dedicada à tarefa de encontrar e corrigir erros. **Depurar** um programa significa localizar e consertar trechos do programa que provocam seu mau funcionamento. Apesar de estarem intimamente relacionados, teste e depuração de um programa não significam a mesma coisa. Um bom teste deve ser capaz de apontar um comportamento anormal de um programa, mas não indica com exatidão quais são as causas de tal comportamento. Por outro lado, a depuração deve determinar precisamente as instruções que causam o mau funcionamento do programa e corrigi-las, reescrevendo-as ou substituindo-as.

7.4.1 Classificação de Erros de Programação

Erros de programação são usualmente classificados em três categorias:

- **Erros de compilação** (ou **erros de sintaxe**) ocorrem em virtude de violações das regras de sintaxe da linguagem de programação utilizada e já foram suficientemente discutidas na **Seção 3.18**. Erros de ligação (v. **Seção 3.18.6**), apesar de, rigorosamente, serem distintos de erros apontados pelo compilador também estão incluídos nessa categoria.
- Um **erro de execução** não impede um programa de ser compilado, mas faz com que sua execução seja interrompida de maneira anormal (algumas vezes, causando até mesmo a falha de todo o sistema operacional no qual o programa é executado). Um exemplo comum desse tipo de erro é uma tentativa de divisão de um valor inteiro por zero.
- **Erro de lógica** é um erro que nem impede a compilação nem acarreta interrupção da execução de um programa. Entretanto, um programa contendo um erro desse tipo não funciona conforme o esperado. Por exemplo, o usuário solicita que o programa execute uma determinada tarefa e o programa não a realiza satisfatoriamente. Um erro de lógica ocorre quando o algoritmo utilizado é incorreto, mesmo que ele tenha sido implementado corretamente ou quando ele é correto, mas sua implementação é equivocada.

7.4.2 Poupando Depuração

Pode parecer surpreendente para um iniciante em programação, mas depuração é uma atividade muito mais difícil e desgastante para o programador do que a escrita de programas. Em resumo, depuração requer paciência, criatividade, esperteza e, principalmente, muita experiência por parte do programador. Idealmente, além de possuir profundo conhecimento sobre a linguagem utilizada, o programador que atua na depuração de programas deve possuir outros conhecimentos, tais como sobre compiladores, assembly e arquitetura de computadores, que transcendem a tarefa básica de construção de programas.

Levando em consideração as prováveis dificuldades que tipicamente cercam as atividades de depuração, é mais sensato tentar evitar que essas atividades se façam necessárias. Infelizmente, alguns programadores adotam equivocadamente uma estratégia contrária a esse argumento. O raciocínio utilizado por esses maus programadores é aproximadamente o seguinte:

1. Tão logo o programador adquire uma vaga ideia do problema em questão, ele constrói um programa para resolvê-lo.
2. Então, ele verifica se o programa funciona com alguns poucos casos de teste.
3. Se o programa funcionar, ele considera-se satisfeito.
4. Se o programa não funcionar, ele passa a depurá-lo, muitas vezes, por tentativa e erro, até que ele seja aprovado nos testes de verificação. Aqui, provavelmente, o mau programador despenderá muito mais tempo do que na escrita do programa.

A abordagem apresentada é equivocada em termos de alocação de esforços porque ela transfere para a fase de depuração, que é exatamente a mais árdua, a tarefa de colocar o programa em funcionamento. No restante da corrente seção, serão apresentadas algumas atitudes que um bom programador deve adotar para poupar atividades de depuração.

Para poupar tempo, antes de iniciar uma sessão de depuração de programa, certifique-se que:

- **Você entende realmente o algoritmo seguido pelo programa**. É praticamente impossível depurar um programa cujo funcionamento não seja completamente entendido.
- **O programa foi compilado usando o máximo nível de apresentação de mensagens de advertência** (no compilador GCC, use as opções `-Wall` e `-pedantic`). Além disso, **todas as mensagens de advertência emitidas pelo compilador devem ter sido atendidas**. Quando utilizado com essas opções, o compilador é capaz de apontar muitas causas de possíveis erros (v. **Seção 3.18.4**). Isto é, o uso preventivo do compilador pode ajudá-lo a evitar que muitos erros ocorram antes mesmo de o programa ser executado pela primeira vez. Examine cuidadosamente cada mensagem de advertência emitida pelo compilador e corrija todas as instruções que correspondam a uma dada advertência, mesmo que você tenha certeza que elas não causarão problemas. Agindo de modo contrário, uma mensagem de advertência importante poderá deixar de ser notada.
- **O programa-fonte foi analisado com o auxílio de uma lista de verificação de programas contendo questões relacionadas a erros comuns de programação em C**. Existem muitos erros de programação que são comuns. Assim, verificar se algum desses erros ocorre em seu programa pode fazê-lo economizar muito tempo. No o site dedicado a este livro (***www.ulysseso.com/ip***), existe uma imensa coleção de erros comuns de programação em C que pode ser usada como lista de verificação.
- **Outros programadores examinaram seu programa**. Não se iluda imaginando que o erro não se encontra numa determinada seção de seu programa apenas porque você já a examinou várias vezes. Do mesmo modo que um texto pode conter erros gramaticais evidentes que o autor não percebe, um programa pode conter erros que são óbvios para terceiros, mas que o programador que o escreveu não consegue notá-los.

7.4.3 Técnicas Elementares de Depuração

A etapa mais difícil de depuração de um programa consiste em localizar precisamente a instrução ou o conjunto de instruções que causa o mau funcionamento do programa. Como já foi dito (v. **Seção 3.18**), encontrar erros sintáticos não é difícil, mesmo quando o compilador não é capaz de apontá-los com precisão e, portanto, esse tópico não merece maiores considerações.

A seguir, serão apresentadas algumas técnicas comuns utilizadas para localizar causas de erros lógicos e de execução em programas. Porém, antes de utilizar alguma destas técnicas, é importante que o programador determine

precisamente a natureza do erro e quando ele ocorre (p. ex., sempre que o programa recebe tal entrada, ele apresenta tal e tal comportamento). A situação ideal acontece quando o programador é capaz de reproduzir um determinado erro sempre que são introduzidos dados possuindo as mesmas características (i.e., quando o erro é sistemático, e não quando ele é aparentemente aleatório).

As técnicas descritas abaixo devem servir apenas como um guia introdutório. À medida que você se tornar um programador experiente, será capaz de desenvolver suas próprias técnicas e de utilizar versões mais sofisticadas do que aquelas apresentadas aqui. Além disso, as duas técnicas discutidas não se aplicam a programas muito pequenos, porque elas não serão de grande utilidade.

Uso de printf()

A função **printf()** e algumas outras funções de saída constituem uma ferramenta rudimentar bastante útil em depuração. Existem dois usos principais de **printf()** em depuração:

- Examinar o valor de uma ou mais variáveis em vários pontos de um programa
- Verificar o fluxo de execução do programa (p. ex., para determinar se uma determinada instrução é executada).

A técnica consiste em distribuir chamadas de **printf()** em vários pontos estratégicos do programa. Enquanto realiza essa tarefa, certifique-se de que você será capaz de distinguir cada uma dessas chamadas quando ela for executada. Como exemplos de uso de **printf()** na depuração de um programa têm-se:

```
printf("Valor de x antes de tal instrucao: %f\n", x);
printf("Valor de x apos o segundo while: %f\n", x);
printf("Instrucoes seguindo else do if 3 sendo executadas");
```

No caso de um programa que é abortado em virtude de um erro de execução, as chamadas de **printf()** que foram executadas, evidentemente, estão antes do erro que causou o aborto. Além disso, quando uma chamada de **printf()** não é concluída com êxito, o erro encontra-se exatamente num dos parâmetros que essa função tenta exibir ou num especificador de formato usado inadequadamente. Por exemplo, considere o seguinte programa:

```
#include <stdio.h>   /* printf()    */
#include "leitura.h" /* LeInteiro() */

int x;

int main(void)
{
   printf("\nDigite um numero inteiro: ");
   x = LeInteiro();

   printf("\nValor introduzido: %s\n", x);

   return 0;
}
```

Esse programa pode escrever o seguinte na tela e, em seguida, ser abortado:

```
Digite um numero inteiro: 5
Valor introduzido:
```

Portanto o erro deve estar na segunda chamada de **printf()** que não foi concluída. (Se você ainda não descobriu qual é o erro, trata-se do uso do especificador **%s**, que deveria ser usado na escrita de strings, em vez de **%d**, que é usado na escrita de valores inteiros).

Obviamente, encerrada a fase de depuração, as chamadas de **printf()** usadas com essa finalidade devem ser removidas. Para que essas chamadas de **printf()** não sejam confundidas com chamadas legítimas dessa função, sugere-se que as chamadas usadas em depuração sejam colocadas em destaque que facilite suas identificações e remoções. Por exemplo, acrescente vários espaços verticais antes e depois de chamadas de **printf()** usadas em depuração e não as endente. Essas medidas facilitarão a rápida identificação das chamadas de **printf()** que precisarão ser removidas, uma vez concluída a fase de depuração.

Uso de Comentários

Comentários são utilizados para excluir da compilação um trecho de programa no interior do qual se suspeita que esteja a origem do mau funcionamento do programa. Essa técnica de depuração segue o seguinte procedimento:

1. Um programa apresenta comportamento inesperado e você suspeita que um determinado trecho dele está provocando esse comportamento indesejável.
2. O referido trecho de programa é envolto em comentários de modo a ser excluído do código executável. Então, compila-se o programa e verifica-se como o programa executável resultante se comporta. Talvez, algumas outras adaptações no programa, como, por exemplo, remoção de comentários preexistentes, sejam necessárias antes de compilá-lo novamente. A propósito, o editor do CodeBlocks possui uma opção, denominada *Comment*, no menu *Edit* que comenta trechos de programa automaticamente. Esse mesmo menu possui uma opção, denominada *Uncomment*, que remove comentários de trechos de programa automaticamente. Essas duas opções ajudam bastante o programador a implementar essa abordagem de depuração
3. Se o programa continuar apresentando o mesmo erro, é provável que esse erro não seja provocado pelo trecho de programa comentado. Assim, você deve procurar o erro em outro local do programa. Então, remova os comentários usados com a finalidade descrita nesta seção, eleja um novo trecho como suspeito e recomece a busca pelo erro a partir do passo 2.
4. Se o programa não apresentar o mesmo erro, é provável que sua conjetura sobre a causa do erro tenha sido adequada e que o trecho de programa comentado seja realmente o causador do erro. Se esse trecho for grande ao ponto de não permitir identificar exatamente qual é a instrução causadora do erro, repita o procedimento a partir do passo 2, mas agora comente um trecho de programa menor dentro da porção de programa anteriormente comentada.

O procedimento descrito acima constitui caso especial de uma abordagem mais geral de depuração (ou, mais precisamente, de procura de erros) denominada **busca binária**. Essa abordagem pode ser utilizada associada ao uso de **printf()** prescrito antes.

7.5 Números Reais Não São Realmente Reais

Esta seção lida com um tópico que incomoda muitos iniciantes em programação que não possuem pleno conhecimento sobre implementação de números reais como **números de ponto flutuante**. Esses números recebem essa denominação porque, nessa forma de representação de números reais, o ponto decimal que separa as partes inteira e fracionária do número é movido (para a direita ou para a esquerda, dependendo do valor do número) de tal modo que antes do ponto decimal se tenha sempre `1` (em base binária). Desse modo, a parte inteira do número na base binária não precisa ser armazenada (pois é sempre igual a `1`). Logo, nesse contexto, *flutuar* significa *mover o ponto decimal* conforme foi descrito. Por outro lado, numa representação de números reais de **ponto fixo**, o ponto decimal não *flutua*... Linguagens de programação modernas tipicamente usam representações de ponto flutuante que seguem o padrão **IEEE 754**.

Para começar a entender o drama, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   double x = 9.90,
          y = 12.0,
          diferenca,  /* Armazenará y - x */
          parteFrac;  /* Parte fracionária da diferença */
   int    parteInt,   /* Parte inteira da diferença     */
          centesimos; /* Parte fracionária em centésimos */

   diferenca = y - x; /* Calcula a diferença */

      /* Obtém a parte inteira da diferença */
   parteInt = (int) diferenca;

      /* Obtém a parte fracionária da diferença */
   parteFrac = diferenca - (double) parteInt;

      /* Apresenta os valores obtidos até aqui */
   printf( "\nx = %f\ny = %f\ndiferenca = %f\nparteFrac = %f\n",
           x, y, diferenca, parteFrac );

      /* Calcula os centésimos da parte fracionária da diferença */
   centesimos = (int) (parteFrac*100.0);

      /* Exibe na tela a parte inteira e os centésimos */
   printf( "\nparteInt = %d\ncentesimos = %d\n", parteInt, centesimos );

   return 0;
}
```

Esse programa é simples e suas pretensões são bastante modestas. Isto é, o que ele pretende fazer é apenas calcular a parte inteira e os centésimos da diferença entre os valores das variáveis `y` (que armazena `12.0`) e `x` (que armazena `9.90`). Mesmo que você tenha pouca intimidade com Matemática, não terá dificuldade para concluir que essa parte inteira deverá ser `2` e os aludidos centésimos deverão resultar em `10`. Para tentar obter esses resultados o programa acima segue os seguintes passos:

1. Calcule a diferença `y - x`.
2. Obtenha a parte inteira dessa diferença. Na instrução do programa que efetua essa operação, usa-se o operador de conversão explícita **(int)**, mas, de fato, ele não é necessário, pois, de qualquer modo, ocorreria conversão implícita de atribuição (v. **Seção 3.10.1**). Assim, esse operador foi usado apenas por uma questão de legibilidade (v. **Seção 3.10.2**).
3. Obtenha a parte fracionária da diferença citada subtraindo dessa diferença a parte inteira obtida no passo anterior. Novamente, o operador **(double)** é usado apenas por questão de legibilidade.
4. Obtenha os centésimos desejados multiplicando a parte fracionária obtida no passo anterior por `100`. Mais uma vez, o operador **(int)** não influi no resultado; ele apenas melhora a legibilidade do programa.

Se você seguir o raciocínio empregado pelo programa e descrito acima usando papel e lápis (ou mesmo mentalmente), obterá o resultado esperado. Contudo, para desespero do programador, quando esse programa é executado, ele exibe o seguinte resultado:

```
x = 9.900000
y = 12.000000
diferenca = 2.100000
parteFrac = 0.100000

parteInt = 2
centesimos = 9          [Deveria ser 10]
```

Ora, se o raciocínio usado pelo programa parece ser absolutamente correto, como se pode comprovar com papel e lápis, por que ele apresenta esse decepcionante resultado? O que há de errado com esse programa?

O pecado cometido pelo programa em questão é que ele considera uma representação de números reais em computador como se ela fosse fiel a números reais de Matemática. Mas, na verdade, não existem legítimos números reais em computação e o raciocínio empregado para se chegar a essa conclusão é muito simples. Conforme você deve ter aprendido em Matemática elementar, por menor que seja um intervalo de números reais, ele será sempre infinito. Mas, por outro lado, por maior que seja o número de bytes usados para representar números reais em computador, esse número de bytes será sempre finito. Portanto poucos números reais de um intervalo qualquer podem ser representados precisamente num computador. Conclusão: números reais são representados apenas aproximadamente em qualquer computador.

Se a conclusão acima lhe deixou surpreso, você poderá ficar ainda mais perplexo ao saber que a representação binária frequentemente usada para números reais é incapaz de replicar números reais simples, que requerem poucos dígitos para serem representados com precisão em base decimal[1]. E um desses infames números é **`9.9`**, que o programa acima, aparentemente, representa com exatidão. Além disso, **`0.1`** (em base decimal), que seria o resultado preciso da parte fracionária que o programa calcula, também não possui representação binária exata.

A origem do problema em questão é ocultada pelo uso do especificador **%f**, que é comumente usado com **printf()** para exibir valores do tipo **double**. Porém, usando-se esse especificador, apenas seis casas decimais de um valor desse tipo podem ser expostas e, para piorar o jogo de esconde-esconde, a função **printf()** efetua arredondamento, como você deverá constatar. Enfim, a origem do problema pode ser pressentida usando-se um especificador de formato que possibilite a apresentação de um número maior de casas decimais (p. ex., **%.16f**, em vez de **%f**) na primeira chamada de **printf()** do programa. O uso do especificador **%.16f** faz com que valores do tipo **double** sejam apresentados com 16 casas decimais. Efetuando-se essa alteração no programa, o resultado que ele apresenta é o seguinte:

```
x = 9.9000000000000004              [Deveria ser 9.9]
y = 12.0000000000000000             [OK]
diferenca = 2.0999999999999996      [Deveria ser 2.1]
parteFrac = 0.0999999999999996      [Deveria ser 0.1]

parteInt = 2                        [OK]
centesimos = 9                      [Deveria ser 10]
```

O *zoom* proporcionado pelo uso do especificador **%.16f** permite diagnosticar claramente a causa do erro do programa em discussão. Ou seja, quando a parte fracionária:

`0.0999999999999996`

é multiplicada por **`100.0`**, obtém-se:

`9.99999999999996`

Logo, quando esse valor é convertido em **int** na atribuição:

```
centesimos = (int) (parteFrac*100.0);
```

ocorre truncamento (e não arredondamento) da parte fracionária do último valor acima. Assim, o resultado atribuído à variável **`centesimos`** é **`9`**, e não **`10`**, como se esperava.

O tipo de problema apresentado pelo programa acima é denominado **erro de truncamento** mas, nesse caso específico, ele não ocorre em virtude de truncamento em si: ele é decorrente do modo como números reais são representados em memória. Por exemplo, se o número em questão fosse representado como **`10.0000000`**, ocorreria truncamento, mas não ocorreria erro de truncamento.

[1] A justificativa para essa afirmação requer uma digressão sobre representação de números reais em base binária que está bem além do escopo desse livro. Portanto, aqui, apenas se ilustrará a veracidade dessa assertiva por meio de exemplos.

A solução mais comum para erro de truncamento decorrente da conversão de um número real positivo em número inteiro consiste em adicionar um pequeno valor ao número real antes que ele seja convertido em inteiro. Obviamente, esse valor deve ser suficientemente pequeno para que não acrescente ainda mais imprecisão ao resultado.

Na representação do tipo **double** especificada pelo padrão IEEE 754, o número máximo de casas decimais significativas é `15`, que é o valor da constante **DBL_DIG** definida no cabeçalho `<float.h>`. Portanto uma escolha segura para o valor a ser acrescentado para evitar os erros de truncamento em questão é `1.0E-14`. Mas, se você estiver escrevendo um programa que requer maior precisão, o valor dessa constante pode ser ligeiramente menor. Nesse caso, consulte um texto mais completo sobre o assunto (v. **Bibliografia**).

Levando em consideração as conclusões derivadas da discussão acima, o programa apresentado no início desta seção poderia ser corrigido como mostrado a seguir:

```
#include <stdio.h>

#define DELTA 1.0E-14

int main(void)
{
   double x = 9.90,
          y = 12.0,
          diferenca,  /* Armazenará y - x */
          parteFrac;  /* Parte fracionária da diferença */
   int    parteInt,   /* Parte inteira da diferença     */
          centesimos; /* Parte fracionária em centésimos */

      /* Calcula a diferença */
   diferenca = y - x;

      /* Efetua a correção da diferença */
   diferenca = diferenca + DELTA;

      /* Obtém a parte inteira da diferença */
   parteInt = (int) diferenca;

      /* Obtém a parte fracionária da diferença */
   parteFrac = diferenca - (double) parteInt;

      /* Apresenta os valores obtidos até aqui */
   printf("\nx = %.16f\ny = %.16f\ndiferenca = %.16f"
          "\nparteFrac = %.16f\n", x, y, diferenca, parteFrac);

      /* Calcula os centésimos da parte fracionária da diferença */
   centesimos = (int) (parteFrac*100.0);

      /* Exibe na tela a parte inteira e os centésimos */
   printf( "\nparteInt = %d\ncentesimos = %d\n", parteInt, centesimos );

   return 0;
}
```

Quando executado, esse novo programa apresenta o seguinte resultado:

```
x = 9.9000000000000004
y = 12.0000000000000000
diferenca = 2.1000000000000099
parteFrac = 0.1000000000000099

parteInt = 2
centesimos = 10
```

Para obter o resultado esperado, as únicas alterações introduzidas no último programa com relação ao programa anterior foram a definição da constante `DELTA`:

```
#define DELTA 1.0E-14
```

e a inclusão da instrução:

```
diferenca = diferenca + DELTA;
```

Agora, é importante destacar que o programa acima só funciona porque se sabe de antemão que o valor da variável `diferenca` é positivo. Quer dizer, se seu valor fosse negativo, o valor da constante `DELTA` deveria ser subtraído, em vez de somado. Em qualquer caso, é melhor substituir essa última instrução pela seguinte instrução **if** que lida com os dois casos:

```
if (diferenca >= 0.0) {
   diferenca = diferenca + DELTA;
} else {
   diferenca = diferenca - DELTA;
}
```

Antes de concluir esta seção, é importante salientar que os problemas aqui discutidos não são inerentes à linguagem C. Ou seja, a origem desses problema é de natureza teórica (i.e., o fato de não ser possível representar um intervalo infinito de números reais) e de representação (i.e., a impossibilidade de representar com exatidão alguns números, como `0.9`, em base binária).

7.6 Exemplos de Programação

7.6.1 Leitura de Notas

Problema: (a) Escreva uma função que lê notas de alunos. Uma nota é considerada válida se ela se encontra entre `0.0` e `10.0` (inclusive). (b) Escreva um programa que teste a função descrita em (a).

Solução de (a):

```
/****
 * LeNota(): Lê um valor x do tipo double tal que: 0.0 <= x <= 10.0
 *
 * Parâmetros: Nenhum
 *
 * Retorno: O valor lido
 ****/
double LeNota(void)
{
   double valorLido;

      /* O laço while encerra apenas quando for lido um valor válido */
   while (1) {
      valorLido = LeReal();

         /* Verifica se o valor lido está de acordo com o esperado */
      if (valorLido >= 0.0 && valorLido <= 10.0) {
         break; /* O valor lido está OK */
      } else {
         printf("\n\t>>> Nota incorreta. Tente novamente\n\t> ");
      }
   }

   return valorLido;
}
```

Análise: Na função `LeNota()`, os valores reais lidos são comparados usando operadores relacionais na expressão que acompanha a instrução **if**:

```
valorLido >= 0.0 && valorLido <= 10.0
```

Isto é, a função `ComparaDoubles()` não é utilizada, conforme é preconizado na Seção 5.11.6, para comparação de números reais. Acontece que, nesse caso, essa função não se faz necessária, visto que os valores `0.0` e `10.0` são representados precisamente em todas as implementações de números reais que seguem o padrão IEEE 754, que é aquele recomendado pelos padrões mais recentes de C. Entretanto, em caso de dúvida entre usar ou não a referida função, use-a.

Solução de (b):

```
/****
 * main(): Testa a função LeNota()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   double umaNota;

   printf("\n\t>>> Digite uma nota: ");
   umaNota = LeNota();

   printf( "\n\t>>> A nota introduzida foi %3.1f\n", umaNota );

   return 0;
}
```

Análise: Esse programa é trivial demais para requerer comentários adicionais.

Exemplo de execução do programa:

```
>>> Digite uma nota: -7

>>> Nota incorreta. Tente novamente
> 7

>>> A nota introduzida foi 7.0
```

7.6.2 Números Primos 2

Problema: A definição de número primo foi apresentada na Seção 5.11.2. (a) Levando em consideração que, no máximo, um número é divisível por sua raiz quadrada, escreva uma função, denominada `EhPrimo2()`, que determina se um número natural é primo ou não. (b) Escreva um programa que lê números inteiros não negativos como entrada e determina se cada um deles é primo ou não. O programa deve encerrar quando o usuário digitar zero ou um.

Solução de (a): A função apresentada a seguir é uma versão melhorada daquela apresentada na Seção 5.11.2.

```
/****
 * EhPrimo2(): Verifica se um número inteiro maior do que um é primo ou não
 *
 * Parâmetros: n (entrada): o número que será testado
 *
 * Retorno: 1, se o número for primo
 *          0, se o número não for primo
 *         -1, se for indefinido (i.e., se n <= 1)
 ****/
```

```
int EhPrimo2(int n)
{
   int i, raiz;

      /* O conceito de número primo não é definido */
      /* para números inteiros menores do que dois */
   if (n <= 1) {
      return -1; /* Indefinido */
   }

   /*                                                                    */
   /* No máximo, um número é divisível por sua raiz quadrada. Portanto, se não for */
   /* encontrado um divisor para o parâmetro no máximo igual a sua raiz, a busca   */
   /* por um divisor é encerrada. Para evitar que essa raiz seja recalculada a cada */
   /* passagem no laço abaixo, armazena-se esse valor numa variável local.         */
   /*                                                                    */

   raiz = sqrt(n);  /* Calcula a raiz quadrada do número a ser testado */

      /* Verifica se o número tem algum divisor. No  */
      /* máximo, um número é divisível por sua raiz. */
   for (i = 2; i <= raiz; ++i) {
      if (!(n%i)) {
         return 0; /* Encontrado um divisor */
      }
   }

      /* Não foi encontrado nenhum divisor para o número dado. Logo ele é primo */
   return 1;
}
```

Solução de (b): Para obter a função **main**() solicitada, copie e cole a função **main**() apresentada na Seção 5.11.2. Em seguida, substitua a chamada de `EhPrimo()` por uma chamada de `EhPrimo2()`. Essa é a única alteração necessária. Para completar o programa, você deve copiar e colar, sem fazer nenhuma alteração adicional, as diretivas **#include** e a função `LeNatural()` do programa apresentado na Seção 5.11.2. Em seguida, você deve acrescentar:

```
#include <math.h> /* sqrt() */
```

na seção do programa que contém as demais diretivas **#include**.

7.6.3 Números Primos 3

Problema: (a) Escreva uma função, denominada `ExibePrimos()`, que recebe um número inteiro maior do que `1` como parâmetro e apresenta na tela todos os número primos menores ou iguais a ele. (b) Escreva um programa que lê números inteiros não negativos como entrada e exibe na tela todos os números primos menores do que cada um deles. O programa deve encerrar quando o usuário digitar zero ou um.

Solução de (a): A função `ExibePrimos()` apresentada a seguir usa a função `EhPrimo2()` da Seção 7.6.2.

```
/****
 * ExibePrimos(): Exibe todos os números primos menores ou iguais
 *                a um determinado inteiro maior do que um
 *
 * Parâmetros: n (entrada): o número que servirá de referência
 *
 * Retorno: Nada
 ****/
```

```
void ExibePrimos(int n)
{
   int i;

      /* O valor do parâmetro deve ser maior do que 1 */
   if (n <= 1) {
      printf("\nValor invalido\n");
      return;
   }

      /* O menor número primo é 2 */
   printf("\n>>> Numeros primos entre 2 e %d:\n\t> ", n);

      /* Verifica quais são os primos */
   for (i = 2; i <= n; ++i) {
      if (EhPrimo2(i) > 0) {
         printf("%d\t", i); /* Encontrado um primo */
      }
   }

   printf("\n"); /* Embelezamento apenas */
}
```

Solução de (b): A função **main()** apresentada abaixo usa uma grande porção do código da função **main()** da Seção 7.6.2.

```
/****
 * main(): Exibe todos os números primos menores ou iguais a um determinado
 *         inteiro maior do que um valor introduzido via teclado
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int num;

      /* Apresenta o programa e explica seu funcionamento */
   printf( "\n\t>>> Este programa verifica quais sao os numeros"
           "\n\t>>> primos entre dois e o valor digitado."
           "\n\t>>> Para encerra-lo, digite zero ou um.\n" );

      /* O laço principal do programa encerra */
      /* quando o usuário introduz 0 ou 1     */
   while (1) {
      printf("\n\t>>> Digite um numero inteiro que nao seja negativo:\n\t> ");
      num = LeNatural(); /* Lê o número */

      if (num <= 1) { /* Encerra o laço */
         break;
      }

         /* Encontra os números primos entre 2 e o número introduzido */
      ExibePrimos(num);
   }

      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Para completar o programa, copie e cole as definições das funções `LeNatural()` e `EhPrimo2()` e as diretivas **#include** do programa apresentado na Seção 7.6.2.

Exemplo de execução do programa:

```
        >>> Este programa verifica quais sao os numeros
        >>> primos entre dois e o valor digitado.
        >>> Para encerra-lo, digite zero ou um.

        >>> Digite um numero inteiro que nao seja negativo:
        > 14
>>> Numeros primos entre 2 e 14:
        > 2     3       5       7       11      13

        >>> Digite um numero inteiro que nao seja negativo:
        > 1

        >>> Obrigado por usar este programa.
```

7.6.4 Verificando Ordenação de Inteiros

Problema: Escreva um programa que recebe um número inteiro positivo `n` como entrada. Então, o programa solicita que o usuário introduza `n` números inteiros (sem restrição) e informa se eles estão em ordem crescente ou não.

Solução:

```
#include <stdio.h>   /* printf()    */
#include "leitura.h" /* LeInteiro() */

/****
 * main(): Verifica se uma lista de números inteiros está ordenada em ordem crescente
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int nValores, /* Número de valores introduzidos */
       valorCorrente, /* Armazena o valor corrente */
       valorAnterior, /* Armazena o valor anterior */
       emOrdem = 1, /* Informa se os valores estão em ordem crescente */
       i;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa verifica se uma lista de N"
           "\n\t>>> numeros inteiros esta' em ordem crescente."
           "\n\t>>> O valor de N deve ser maior do que 1.\n" );

      /* Lê o número de valores a serem introduzidos. */
      /* Esse valor deve ser maior do que 1.          */
   while (1) {
         /* Tenta ler um valor correto */
      printf("\n\t>>> Digite o numero de valores: ");
      nValores = LeInteiro();

         /* Se o valor está correto, encerra o laço */
      if (nValores > 1) {
         break;
      }
```

```
      /* O valor não era o que o programa esperava */
      printf( "\nO numero de valores deve ser maior do que 1\n");
   }

      /* Lê o primeiro valor */
   printf("\n\t>>> Valor 1: ");
   valorAnterior = LeInteiro();

      /* Lê os demais valores. Para que eles estejam  */
      /* em ordem crescente, qualquer valor lido deve */
      /* ser maior do que ou igual ao anterior        */
   for (i = 2; i <= nValores; ++i) {
      printf("\n\t>>> Valor %d: ", i);
      valorCorrente = LeInteiro();

         /* Verifica se os valores estão em ordem crescente */
      if (valorCorrente < valorAnterior) {
         emOrdem = 0; /* Valores não estão em ordem */
         break; /* Não adianta prosseguir */
      } else {
            /* Até aqui, os valores estão em ordem. Então o */
            /* valor anterior passa a ser o valor corrente. */
         valorAnterior = valorCorrente;
      }
   }

      /* Apresenta o resultado */
   printf( "\n\t>>> Os valores %sestao em ordem "
           "crescente\n", emOrdem ? "" : "NAO " );

   return 0;
}
```

Análise: Os comentários inseridos no programa devem ser suficientes para seu entendimento.

Exemplo de execução do programa:

```
>>> Este programa verifica se uma lista de N
>>> numeros inteiros esta' em ordem crescente.
>>> O valor de N deve ser maior do que 1.

>>> Digite o numero de valores: 5

>>> Valor 1: -5
>>> Valor 2: 2
>>> Valor 3: -4

>>> Os valores NAO estao em ordem crescente
```

7.6.5 Desenhando Quadrados

Problema: Escreva um programa que lê um valor `L` e desenha um quadrado de lado `L` delimitado por asteriscos conforme ilustrado abaixo:

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

Solução:

```
#include <stdio.h>   /* printf() e putchar() */
#include "leitura.h" /* LeInteiro()          */

#define MAIOR_LADO 20 /* Maior lado permitido para um quadrado */

/****
 * main(): Desenha um quadrado de asteriscos
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int lado, /* Lado do quadrado */
       i, j;

      /* Apresenta o programa e explica seu funcionamento */
   printf("\n\t>>> Este programa desenha um quadrado de lado L."
          "\n\t>>> O valor de L deve ser um inteiro maior do"
          "\n\t>>> que 1 e menor do que %d.\n", MAIOR_LADO + 1);

      /* O laço encerra quando o valor digitado for válido */
   while (1) {
      printf("\n>>> Digite o lado do quadrado:\n\t> ");
      lado = LeInteiro();

         /* Verifica se o valor é válido */
      if (lado > 1 && lado <= MAIOR_LADO) {
         break; /* Valor é válido. Encerra o laço. */
      }

         /* Valor digitado não é válido */
      printf( "\a\n>>> O lado deve ser maior do que 1 e "
              "menor do que %d <<<\n", MAIOR_LADO + 1 );
   }

   printf("\n\t>>> Eis o seu quadrado:\n\n");

      /* Desenha a linha superior do quadrado */
   for (i = 1; i <= lado; ++i) {
      putchar('*');
   }

      /* Desenha as demais linhas */
   for (j = 2; j < lado; ++j) {
      putchar('\n'); /* Pula linha */
      putchar('*'); /* Primeira coluna */

         /* Da segunda coluna até a penúltima */
         /* escreve espaços em branco         */
      for (i = 2; i < lado; ++i) {
         putchar(' ');
      }

      putchar('*'); /* Última coluna */
   }
      /* Desenha a linha inferior do quadrado */
   putchar('\n'); /* Pula linha */
   for (i = 1; i <= lado; ++i) {
      putchar('*');
   }
```

```
   putchar('\n'); /* Apenas embelezamento */

   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa desenha um quadrado de lado L.
        >>> O valor de L deve ser um inteiro maior do
        >>> que 1 e menor do que 21.

>>> Digite o lado do quadrado:
        > 5

        >>> Eis o seu quadrado:

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

7.6.6 Tabuada de Multiplicação

Problema: Escreva um programa que crie uma tabuada de multiplicação com cinco linhas e cinco colunas.

Solução:

```
#include <stdio.h>

#define N_LINHAS  5
#define N_COLUNAS 5

/****
 * main(): Apresenta uma tabuada de multiplicação na tela
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int i, j;

      /* Apresenta o programa ao usuário */
   printf( "\n>>> Este programa exibe uma tabuada de "
           "multiplicacao\n\n" );

      /* Desenha a linha superior da tabuada */
   printf("  X |\t");
   for (i = 1; i <= N_COLUNAS; ++i) {
      printf("%3d\t", i);
   }

      /* Um pouco de embelezamento */
   printf("\n ---|----------------------------------------");

   for (j = 1; j <= N_LINHAS; ++j) {
         /* Escreve primeira coluna da próxima linha */
      printf("\n%3d |\t", j);

         /* Escreve as demais colunas da linha corrente */
      for (i = 1; i <= N_COLUNAS; ++i) {
```

```
         printf("%3d\t", i*j);
      }
   }

   putchar('\n'); /* Embelezamento apenas */

   return 0;
}
```

Resultado de execução do programa:

```
>>> Este programa exibe uma tabuada de multiplicacao
  X |     1       2       3       4       5
 ---|-----------------------------------------
  1 |     1       2       3       4       5
  2 |     2       4       6       8      10
  3 |     3       6       9      12      15
  4 |     4       8      12      16      20
  5 |     5      10      15      20      25
```

7.6.7 Números de Fibonacci 1

Problema: Escreva um programa que verifique se um número inteiro positivo faz parte de uma sequência de Fibonacci (v. Seção 2.10.4) e qual é o menor número de termos que uma sequência contendo esse número possui.

Solução:

```
#include <stdio.h>   /* printf()    */
#include "leitura.h" /* LeInteiro() */

/****
 * main(): Verifica se um número lido via teclado é um número de Fibonacci
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int antecedente1, antecedente2, atual, numeroTestado, i;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa verifica se um numero inteiro"
           "\n\t>>> positivo faz parte de uma sequencia de Fibonacci\n" );

      /* Lê o número que será testado */
   while (1) {
      printf("\n>>> Digite o numero a ser testado:\n\t> ");
      numeroTestado = LeInteiro();

         /* Verifica se o valor é válido */
      if (numeroTestado > 0) {
         break; /* Valor está OK */
      }

         /* Valor não é válido */
      printf("\a\n\t>>> O numero deve ser maior do que 0 <<<\n");
   }
```

```
      /* Inicia os dois primeiros termos e o termo atual da série */
   atual = antecedente1 = antecedente2 = 1;

      /* Gera os termos da sequência a partir do terceiro termo até */
      /* encontrar um número maior ou igual ao número sendo testado */
   for(i = 3; atual < numeroTestado; ++i) {
      atual = antecedente1 + antecedente2;

         /* Atualiza os termos antecedentes */
      antecedente1 = antecedente2;
      antecedente2 = atual;
   }

      /* Apresenta o resultado do teste. Se o último      */
      /* número gerado for igual ao número sendo testado, */
      /* este faz parte de uma sequência de Fibonacci.    */
   printf( "\n\t>>> %d %s e' um numero de Fibonacci\n",
           numeroTestado, numeroTestado == atual ? "" : "NAO" );

   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa verifica se um numero inteiro
        >>> positivo faz parte de uma sequencia de Fibonacci

>>> Digite o numero a ser testado:
        > 5

        >>> 5  e' um numero de Fibonacci
```

7.6.8 Leitura de Datas com Validação 1

Problema: Escreva um programa que lê uma data introduzida via teclado e verifica sua validade.

Solução:

```
/********************** Includes *************************/

#include <stdio.h>   /* printf()    */
#include "leitura.h" /* LeInteiro() */

/***************** Constantes Simbólicas ******************/

#define PRIMEIRO_ANO_BISSEXTO 1752

/************************ Alusões ************************/

extern int LeNatural(void);
extern int EhAnoBissexto(int ano);
extern int EhDataValida(int dia, int mes, int ano);

/***************** Definições de Funções ******************/

/****
 * EhAnoBissexto(): Verifica se um ano é bissexto
 *
 * Parâmetros: ano (entrada): o ano que será testado
 *
 * Retorno: 1, se o ano for bissexto
 *          0, se o ano não for bissexto
 *
 * Observação: Um ano é bissexto quando:
```

```
 *             * Ele é múltiplo de 400 ou
 *             * Ele é múltiplo de 4, mas não é múltiplo de 100
 ****/
int EhAnoBissexto(int ano)
{
      /* Se o ano for anterior ao primeiro ano considerado */
      /* bissexto, ele não pode ser assim considerado      */
   if (ano < PRIMEIRO_ANO_BISSEXTO) {
      return 0;
   }

   return !(ano%400) || (!(ano%4) && ano%100);
}

/****
 * EhDataValida(): verifica se uma data é válida
 *
 * Parâmetros: dia (entrada) - o dia
 *             mes (entrada) - o mês
 *             ano (entrada) - o ano
 *
 * Retorno: 1, se a data for válida; 0, em caso contrário
 ****/
int EhDataValida(int dia, int mes, int ano)
{
      /* Não existe dia menor do que ou igual */
      /* a zero, nem dia maior do que 31      */
   if ( dia <= 0 || dia > 31) {
      return 0; /* Dia inválido */
   }

      /* Verifica se o dia é válido, o que depende do mês */
   switch(mes) {
      case 1: /* Estes meses sempre têm 31 dias */
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
         return 1;
      case 4: /* Estes meses sempre têm 30 dias */
      case 6:
      case 9:
      case 11:
            /* Se o mês tiver mais de 30 dias, ele é inválido */
         if (dia > 30) {
            return 0;
         } else {
            return 1;
         }
      case 2: /* Este mês pode ter 28 ou 29 dias */
            /* Se o mês tiver mais de 29 dias, ele é inválido */
         if (dia > 29) {
            return 0;
         } else if (dia < 29) {
            return 1; /* Mês tem 28 dias */
         } else if (EhAnoBissexto(ano)) {
```

```
            return 1; /* Mês tem 29 dias e ano é bissexto */
         } else {
            return 0;
         }
      default: /* A execução não deve chegar até aqui */
         printf("\nOcorreu um erro sobrenatural");
   }

      /* A execução também não se deve chegar até aqui */
   printf("\nOcorreu outro erro sobrenatural");

   return 0; /* Esta instrução nunca deve ser executada */
}

/****
 * main(): Lê uma data e verifica sua validade
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int d, m, a; /* Dia, mês e ano, respectivamente */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa verifica a validade de"
           "\n\t>>> uma data introduzida pelo usuario.\n" );

   printf("\n\t>>> Digite o dia:\n\t> ");
   d = LeNatural();

   printf("\n\t>>> Digite o mes:\n\t> ");
   m = LeNatural();

   printf("\n\t>>> Digite o ano:\n\t> ");
   a = LeNatural();

      /* Apresenta o resultado da validação de acordo com */
      /* o valor retornado pela função EhDataValida()     */
   printf( "\n\t>>> A data %.2d/%.2d/%.2d %se\' valida\n",
           d, m, a, EhDataValida(d, m, a) ? "" : "NAO " );

   return 0;
}
```

Análise:

- A função **main()** chama a função `LeNatural()` definida no exemplo da Seção 5.11.1. Portanto faz-se desnecessário apresentar sua definição no programa, mas ela realmente faz parte dele (apenas foi omitida).
- A função **main()** poderia ser mais simpática com o usuário e permitir a correção de alguns enganos básicos nas leituras do dia e do ano. Por exemplo, se o usuário digitasse `32` como dia do mês, ela deveria permitir que o usuário corrigisse o erro antes de chamar a função `EhDataValida()`. Reimplementar a função **main()** de modo que ela seja mais amigável ao usuário é deixado como exercício.

Exemplo de execução do programa:

```
>>> Este programa verifica a validade de
>>> uma data introduzida pelo usuario.

>>> Digite o dia:
> 29

>>> Digite o mes:
> 2

>>> Digite o ano:
> 2000

>>> A data 29/02/2000 e' valida
```

7.6.9 Calculando MDC Usando o Algoritmo de Euclides

Preâmbulo: O **algoritmo de Euclides** para cálculo do MDC (máximo divisor comum) de dois números naturais é baseado no fato de o MDC de `x` e `y` ser o mesmo que o MDC de `x%y` e `y`, se `x > y`.

Problema: (a) Utilizando conhecimento apresentado no preâmbulo, escreva uma função que implementa o algoritmo de Euclides. (b) Escreva uma função **main()**, semelhante àquela da **Seção 5.11.3**, que lê repetidamente dois valores inteiros não negativos, calcula o MDC deles usando a função especificada no item (a) e apresenta os resultados.

Solução de (a):

```
/****
 * MDC2(): Calcula o MDC de dois números naturais usando o algoritmo de Euclides
 *
 * Parâmetros: x, y (entrada): números cujo MDC será calculado
 *
 * Retorno: O MDC dos dois números recebidos como parâmetros
 ****/
int MDC2(int x, int y)
{
   int resto; /* Armazena o resto da divisão de x por y */

      /* O laço encerra quando o MDC for encontrado */
   while (1) {
      resto = x % y; /* Calcula o resto da divisão de x por y */

         /* Se a divisão não deixou resto, o MDC é y */
      if (!resto) {
         break;
      }

         /* Atualiza os valores de x e y para uma nova tentativa de encontrar MDC */
      x = y;
      y = resto;
   }
   return y;
}
```

Solução de (b): Para obter a função solicitada, copie e cole a função **main()** apresentada na **Seção 5.11.3** e substitua a chamada de `MDC()` por uma chamada de `MDC2()`. Para completar o programa, copie e cole as definições das funções `LeOpcaoSimNao()` e `LeNatural()` e as diretivas **#include**. Todos esses demais componentes fazem parte do programa apresentado na **Seção 5.11.3**.

7.6.10 Conjectura de Collatz 2

Problema: Escreva um programa que encontra o número entre `1` e `100000` que produz a maior sequência de Collatz (v. Seção 5.11.10).

Solução:

```
#include <stdio.h> /* printf() */

   /* Maior valor que pode ser usado como primeiro */
   /* termo de uma sequência de Collatz            */
#define MAX 100000

/****
 * Collatz2(): Determina o número de termos da sequência de
 *             Collatz que começa com um dado inteiro positivo
 *
 * Parâmetro: n (entrada) - termo inicial da sequência
 *
 * Retorno: O número de termos da sequência
 ****/
int Collatz2(int n)
{
   int cont = 1; /* Conta o número de termos da sequência e é iniciado com 1    */
                 /* pois a sequência tem pelo menos um termo, que é o parâmetro */

      /* O primeiro termo deve ser positivo */
   if (n <= 0) {
      return 0; /* Não existe sequência */
   }

      /* Determina os demais termos. O laço encerra */
      /* quando é encontrado um termo igual a 1.    */
   while (1) {
         /* Gera o próximo termo */
      if (!(n%2)) { /* O termo corrente é par */
         n = n/2; /* Calcula o próximo termo */
      } else { /* O termo corrente é ímpar */
         n = 3*n + 1; /* Calcula o próximo termo */
      }

      ++cont; /* Mais um termo foi gerado */

         /* Se o novo termo for igual a 1,  */
         /* chegou-se ao final da sequência */
      if (n == 1) {
         break; /* Encerra o laço */
      }
   }

   return cont;
}

/****
 * main(): Encontra o número entre 1 e MAX que produz a maior sequência de Collatz
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
```

```
int main(void)
{
   int nMax, /* Número que gera a maior sequência */
       maxTermos, /* Número de termos da maior sequência */
       nTermos, /* Número de termos de cada sequência */
       i;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa encontra o numero entre 1 e %d que"
           "\n\t>>> produz a maior sequencia de Collatz.\n", MAX );
      /* Inicia as variáveis 'nMax' e 'maxTermos' com 1 */
   nMax = 1;
   maxTermos = 1; /* A sequência que começa com 1 tem apenas 1 termo */

      /* Determina os números de termos das sequências  */
      /* começando com 2 até MAX e verifica qual desses */
      /* valores produz a maior sequência               */
   for (i = 2; i <= MAX; ++i) {
         /* Determina o número de termos */
         /* da sequência corrente        */
      nTermos = Collatz2(i);

         /* Verifica se o número de termos da sequência */
         /* corrente é maior do que o maior valor atual */
      if (nTermos > maxTermos) {
            /* A sequência corrente possui o maior número de termos */
         maxTermos = nTermos;

         nMax = i; /* i produz a maior sequência até aqui */
      }
   }

      /* Apresenta o número que produz a maior sequência */
   printf( "\n\t>>> O numero entre 1 e %d que produz a"
           "\n\t>>> maior sequencia de Collatz e' %d.\n", MAX, nMax );

      /* Apresenta o número de termos da maior sequência */
   printf( "\n\t>>> Essa sequencia possui %d termos.\n", maxTermos );

   return 0;
}
```

Resultado de execução do programa:

```
>>> Este programa encontra o numero entre 1 e 100000 que
>>> produz a maior sequencia de Collatz.

>>> O numero entre 1 e 100000 que produz a
>>> maior sequencia de Collatz e' 77031.

>>> Essa sequencia possui 351 termos.
```

7.7 Exercícios de Revisão

Introdução (Seção 7.1)

1. O que é reúso de código?

Reúso de Código (Seção 7.2)

2. Por que programadores experientes beneficiam-se mais de reúso de código do que programadores iniciantes?
3. Cite três formas de reúso de código.

4. (a) Qual é a principal vantagem obtida pelo programador com o uso de bibliotecas? (b) Qual é a maior desvantagem que bibliotecas podem apresentar?
5. Qual é a forma mais rudimentar de reúso de código?

Usando Abreviações com CodeBlocks (Seção 7.3)

6. O que são abreviações no IDE CodeBlocks?
7. Como abreviações facilitam a escrita de código?

Depuração de Programas (Seção 7.4)

8. Como são classificados os erros de programação?
9. Apresente duas causas de erros de execução comuns.
10. Qual é a diferença entre teste e depuração de programas?
11. Como funciona o método de busca binária em depuração?
12. Como um compilador pode ajudar na tarefa de depuração de um programa?
13. Ambos os trechos de programa a seguir contêm erros:

Trecho de programa 1	Trecho de programa 2

`int x = 0;`
`if (x = 10)`
`  y = y + 1;`
	`int x = 0;`
	`if (10 = x)`
	`  y = y + 1;`

Por que o compilador indica uma ocorrência de erro no trecho de programa 2, mas o mesmo não ocorre com o trecho de programa 1?

14. (a) Como funciona a técnica de depuração que utiliza **printf()**? (b) Compare essa técnica de depuração com a técnica de depuração que faz uso de comentários.
15. Por que erros de execução são mais difíceis de corrigir do que erros de sintaxe?
16. Por que erros de lógica são mais difíceis de corrigir do que erros de execução?
17. Por que mensagens de advertência emitidas por um compilador não devem ser negligenciadas?

Números Reais Não São Realmente Reais (Seção 7.5)

18. (a) O que é truncamento? (b) O que é erro de truncamento?
19. (a) O que é representação (número) de ponto flutuante? (b) O que é representação (número) de ponto fixo?
20. Qual é a diferença entre número real e número de ponto flutuante?
21. Quantas casas decimais são exibidas quando se usa o especificador **%f** com **printf()**?
22. (a) Explique o uso do especificador **%.*n*f** com **printf()**, sendo ***n*** um número inteiro positivo. (b) Em que situações esse especificador deve ser usado em detrimento a **%f**?
23. Por que nenhum intervalo de números reais pode ser representado precisamente em computador?
24. A que se refere o padrão IEEE 754?
25. Por que o valor da constante `DELTA` foi escolhido como `1.0E-14`?

7.8 Exercícios de Programação

7.8.1 Fácil

EP7.1 Escreva um programa que lê dois números inteiros positivos como entrada e informa quais são os números primos que estão entre os dois valores introduzidos. [**Sugestão**: Este é um exercício sobre reúso de código. Use o exemplo apresentado na **Seção 7.6.3** como base do seu programa.]

EP7.2 Um **número composto** é um número natural (i.e., inteiro não negativo) maior do que `1` e que não é primo. Escreva um programa que lê um número inteiro como entrada e determina se ele é composto ou não. [**Sugestão**: Este é um exercício sobre reúso de código. Compare a definição de número composto com aquela de número primo apresentado na **Seção 5.11.2** e verifique o que precisa ser alterado no programa sobre números primos apresentado naquela seção para obter a solução para este exercício de programação.]

EP7.3 Escreva um programa que lê um número inteiro como entrada e apresenta na tela quais são os números compostos menores do que ele. [**Sugestão**: Este é um exercício sobre reúso de código. Compare o enunciado deste exercício com aquele do exemplo apresentado na **Seção 7.6.3**, que exibe na tela os números primos menores do que determinado valor e verifique o que precisa ser alterado para obter a solução para este exercício de programação. Note que a função `ExibePrimos()` não poderá ser usada na íntegra, mas a maior parte de seu código poderá ser reutilizada.]

EP7.4 Escreva um programa que recebe um número inteiro positivo `n` como entrada. Então, o programa solicita que o usuário introduza `n` números inteiros (sem restrição) e informa se eles estão em ordem decrescente ou não. [**Sugestão**: Este é um exercício sobre reúso de código. Compare o enunciado deste exercício com aquele do exemplo apresentado na **Seção 7.6.4** que verifica se uma lista de valores está em ordem crescente e reutilize o código daquele programa fazendo as devidas alterações.]

EP7.5 Escreva um programa que lê dois valores `L` e `A` e desenha um retângulo com `L` asteriscos de largura e `A` asteriscos de altura, conforme ilustrado abaixo para `L` igual a `5` e `A` igual a `3`:

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

[**Sugestão**: Reutilize parte do código que desenha quadrados de asteriscos apresentado na **Seção 7.6.5**.]

EP7.6 Modifique o programa apresentado na **Seção 7.6.7** de tal modo que, se for o caso, o programa informe qual é o número de termos da menor sequência de Fibonacci que contém o número introduzido pelo usuário. [**Sugestão**: Você precisará apenas acrescentar algumas linhas ao referido programa.]

EP7.7 Escreva um programa que apresenta na tela uma tabela de multiplicação com `n` linha e `n` colunas, sendo `n` um valor inteiro positivo introduzido pelo usuário. O valor de `n` deve ser limitado entre `2` e `15`. [**Sugestão**: Use o exemplo apresentado na **Seção 7.6.6** como ponto inicial do seu programa.]

EP7.8 Escreva um programa que apresenta uma tabuada de soma na tela, conforme mostrado a seguir:

```
 1   2   3   4   5   6   7   8
 2   4   5   6   7   8   9  10
 3   5   6   7   8   9  10  11
 4   6   7   8   9  10  11  12
 5   7   8   9  10  11  12  13
 6   8   9  10  11  12  13  14
 7   9  10  11  12  13  14  15
 8  10  11  12  13  14  15  16
```

[**Sugestão**: Reutilize parte do código do programa que exibe uma tabela de multiplicação apresentado na **Seção 7.6.6**.]

EP7.9 **Preâmbulo**. Um número **deficiente** é um número natural maior do que `1` cuja soma de seus divisores, incluindo `1` mas excluindo ele próprio, é menor do que ele. Por exemplo, *8* é um número deficiente, já que *1* + *2* + *4* é igual a *7*, que é menor do que *8*. Um número natural maior do que *1* é **abundante** se a soma de seus divisores, incluindo *1* mas excluindo ele próprio, for maior do que menor do que o número. Por exemplo, *12* é um número abundante, pois *1* + *2* + *3* + *4* + *6* = *16*, que é maior do que

12. Um número **perfeito** é um número natural maior do que *1* cuja soma de seus divisores, incluindo *1* mas excluindo ele próprio, é igual ele. Por exemplo, *6* é um número perfeito, já que *1 + 2 + 3* é igual a *6*. **Problema**. Escreva um programa que lê números inteiros naturais via teclado e classifica-os como primo, perfeito, deficiente ou abundante. A entrada de dados deve encerrar quando o usuário digitar zero. [**Sugestões**: (1) Crie uma função que retorna a soma dos divisores de um número natural maior do que *1* recebido como parâmetro. (2) Use a função `LeNatural()` definida na **Seção 5.11.1** para ler o valor introduzido pelo usuário. (3) Chame a função sugerida no item (1) para obter a soma dos divisores desse valor. (4) Use instruções **if** para classificar o número.]

EP7.10 Escreva um programa que exibe na tela uma tabela de soma, subtração, multiplicação ou divisão com `n` linhas e `n` colunas, sendo `n` um valor inteiro positivo introduzido pelo usuário. O programa deve apresentar o seguinte menu de opções:

```
1. Soma
2. Subtracao
3. Divisao
4. Multiplicacao
5. Encerra o programa

Escolha a opcao:
```

Após escolher a opção de operação, o usuário deve introduzir o valor de `n`. [**Sugestão**: Estude a **Seção 5.8** e o exemplo apresentado na **Seção 7.6.6**.]

EP7.11 Escreva um programa que calcula as áreas das seguintes figuras geométricas planas: retângulo, triângulo, círculo, paralelogramo e trapézio. As fórmulas usadas para calcular essas áreas encontram-se na **Tabela 7–1**.

Figura Plana	Área
Retângulo (a e b são os lados)	$a \times b$
Triângulo (b é a base; h é a altura)	$\frac{b \times h}{2}$
Círculo (r é o raio)	$\pi \times r^2$
Paralelogramo (*b* é a base; *h* é a altura)	$b \times h$
Trapézio (*b* é a base menor; *B* é a base maior; *h* é a altura)	$\frac{(b + B) \times h}{2}$

Tabela 7–1: Áreas de Figuras Planas

O programa deve oferecer ao usuário o seguinte menu de opções:

```
1. Area de retangulo
2. Area de triangulo
3. Area de circulo
4. Area de paralelogramo
5. Area de trapezio
6. Encerra o programa
```

EP7.12 Escreva um programa que calcula os volumes dos seguintes sólidos geométricos: paralelepípedo, cone, esfera, pirâmide quadrangular e cilindro. As fórmulas usadas para calcular esses volumes encontram-se na **Tabela 7–2**.

Sólido Geométrico	Volume
Paralelepípedo (a, b e c são os lados)	$a \times b \times c$
Cone (r é o raio da base; h é a altura)	$\frac{\pi \times r^2 \times h}{3}$
Esfera (r é o raio)	$\frac{4 \times \pi \times r^3}{3}$
Pirâmide quadrangular (a é o lado da base; h é a altura)	$\frac{a^2 \times h}{3}$
Cilindro (r é o raio da base; h é a altura)	$\pi \times r2 \times h$

Tabela 7–2: Volumes de Figuras Sólidas

O programa deve oferecer ao usuário o seguinte menu de opções:

```
1. Volume de paralelepipedo
2. Volume de cone
3. Volume de esfera
4. Volume de piramide quadrangular
5. Volume de cilindro
6. Encerra o programa
```

7.8.2 Moderado

EP7.13 **Preâmbulo**: Uma tripla pitagórica (v. exercício **EP4.27**) é **primitiva** quando os três números que a compõem são primos entre si. De acordo com o célebre matemático Euclides, uma tripla pitagórica é primitiva se e somente se os valores de *m* e *n* usados nas fórmulas apresentadas no exercício **EP4.27** forem primos entre si e tiverem paridades diferentes (i.e., se *m* for par, *n* será ímpar e vice-versa). **Problema**: Escreva um programa que apresenta as triplas de Pitágoras primitivas quando os valores de `m` e `n` variam entre *1* e *5*. O resultado do programa deverá ser o seguinte:

```
>>> Triplas Pitagoricas Primitivas <<<

a =  3, b =  4, c =  5  (m = 2, n = 1)
a =  5, b = 12, c = 13  (m = 3, n = 2)
a = 15, b =  8, c = 17  (m = 4, n = 1)
a =  7, b = 24, c = 25  (m = 4, n = 3)
a = 21, b = 20, c = 29  (m = 5, n = 2)
a =  9, b = 40, c = 41  (m = 5, n = 4)
```

[**Sugestão**: Use as sugestões apresentadas para o exercício **EP4.27** e as funções solicitadas nos exercícios **EP5.9** e **EP5.10**.]