Capítulo 3

Introdução à Linguagem C

OBJETIVOS

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia associada à linguagem C:
  - ❒ Padrão ISO
  - ❒ Tipo de dado
  - ❒ Tipo primitivo
  - ❒ Palavra-chave
  - ❒ Palavra reservada
  - ❒ Efeito colateral
  - ❒ Ordem de avaliação
  - ❒ Biblioteca padrão
  - ❒ Cabeçalho
  - ❒ Programa robusto
  - ❒ Programa interativo
  - ❒ Curto-circuito
  - ❒ Definição de variável
  - ❒ Iniciação de variável
  - ❒ Comentário
  - ❒ Constante simbólica
  - ❒ Instrução portável
  - ❒ Programa de console
  - ❒ Hospedeiro
  - ❒ Sistema livre
  - ❒ Conversão explícita
  - ❒ Prompt
  - ❒ Terminal de instrução
  - ❒ Código de caracteres
  - ❒ Especificador de formato
  - ❒ Diretivas #include e #define
  - ❒ Caractere constante
  - ❒ Sequência de escape
  - ❒ String constante
  - ❒ Especificador de formato
  - ❒ String de formatação
  - ❒ Mensagem de erro
  - ❒ Mensagem de advertência
  - ❒ Teste de programa
  - ❒ Depuração de programa
- Detalhar as regras para formação de identificadores da linguagem C
- Identificar formatos de escrita de constantes de C
- Justificar o uso de nomenclaturas diferentes na escrita de identificadores que pertencem a categorias diferentes
- Descrever e usar os seguintes operadores da linguagem C:
  - ❒ Aritméticos ❒ Lógicos ❒ Relacionais ❒ De incremento ❒ De decremento
- Explicar diferenças entre os operadores prefixo e sufixo de incremento e decremento
- Empregar os tipos **int**, **char** e **double** da linguagem C
- Usar **printf()** para escrita na tela
- Apresentar situações nas quais um compilador efetua conversões implícitas e quais são suas consequências
- Explicar todas as propriedades dos operadores da linguagem C
- Classificar erros de programação
- Desenvolver as etapas de construção de um programa de pequeno porte em linguagem algorítmica

3.1 A Linguagem C

LINGUAGEM C foi desenvolvida no início da década de 1970 num laboratório da AT&T por Dennis Ritchie como uma linguagem de alto nível dispondo de facilidades de baixo nível, de modo que fosse possível contemplar satisfatoriamente todas as qualidades desejáveis de um bom programa (v. **Seção 1.3**).

Com o ganho de popularidade a partir dos anos 1980, a linguagem C passou a contar com vários compiladores que apresentavam características diferentes daquelas que faziam parte da especificação original da linguagem. Essas discrepâncias comprometiam uma das principais características desejáveis em programas escritos em linguagens de alto nível: a portabilidade. Para que houvesse compatibilidade entre compiladores, foi necessário criar uma padronização da linguagem que fosse seguida pelos fabricantes de compiladores.

A padronização da linguagem C foi inicialmente realizada por um comitê do *American National Standards Institute* (ANSI) em 1989 e, por causa disso, foi denominada de **ANSI C**. Posteriormente, essa padronização passou a ser conhecida como **C89**. Em 1990, um comitê ISO assumiu a responsabilidade pela padronização de C e aprovou o padrão ANSI. Popularmente, esse padrão é denominado **C90**, que é, essencialmente, o mesmo padrão C89. Em 1999, o comitê ISO responsável pela padronização de C aprovou o padrão de C popularmente conhecido como **C99** e, finalmente, em dezembro de 2011, ratificou o mais recente padrão da linguagem, conhecido como **C11**. Entretanto, até o final da escrita deste livro, este último padrão ainda recebia pouco suporte por parte dos fabricantes de compiladores.

Apesar de muitos fabricantes de compiladores de C desenvolverem suas próprias versões dessa linguagem, eles oferecem a opção ANSI C (ou ISO C) que pode ser escolhida pelo programador, por exemplo, por meio de uma opção de compilação. Em nome da portabilidade, é sempre recomendado utilizar uma opção ANSI/ISO C (v. **Seção 1.7.2**).

Não obstante todos os esforços de padronização, muitas construções da linguagem C ainda têm interpretações que variam de compilador para compilador. Isso ocorre em virtude de ambiguidades e omissões ora na definição original da linguagem ora em antigas versões de padronização. Esses problemas, apesar de reconhecidos, são perpetuados a cada novo padrão para garantir **compatibilidade histórica**; isto é, para garantir que programas antigos continuem podendo ser compilados num padrão mais recente. Ao longo do texto, cada característica de C que pode ter mais de uma interpretação é identificada como **dependente de implementação**. Esse termo é mais preciso do que *dependente de compilador*, pois diferentes versões (i.e., implementações de C) de um mesmo compilador podem ter interpretações diferentes para uma dada construção da linguagem.

3.2 Identificadores

Um **identificador** serve para nomear um componente utilizado num programa escrito numa dada linguagem de programação.

3.2.1 Regras para Criação de Identificadores

Cada linguagem de programação possui regras próprias para formação de seus identificadores. As regras para criação de identificadores em C são as seguintes:

- **Caracteres permitidos**. Um identificador em C deve ser constituído apenas de letras, dígitos e _ (subtraço). Aqui, letra não inclui, por exemplo, cedilha ou caracteres acentuados de português.
- **Restrição**. O primeiro caractere de um identificador não pode ser um dígito.

- **Tamanho**. O número de caracteres permitido num identificador depende da versão do padrão de C utilizada. O padrão ISO C90 requer que compiladores de C aceitem, pelo menos, 31 caracteres; nos padrões mais recentes (C99 e C11), esse número passou a ser 63. Normalmente, compiladores modernos dão liberdade para programadores *sensatos* escreverem identificadores do tamanho desejado, de modo que não é necessário memorizar esses números.

Uma característica importante da linguagem C é que ela, diferentemente de algumas outras linguagens (p. ex., Pascal), faz distinção entre letras maiúsculas e minúsculas. Por exemplo, as variáveis denominadas `umaVar` e `UmaVar` são consideradas diferentes.

3.2.2 Palavras-chaves

Palavras que possuem significado especial numa linguagem de programação (p. ex., **while** e **for** em C) são denominadas **palavras-chaves** da linguagem. Essas palavras *não podem* ser redefinidas pelo programador. Por exemplo, não é permitido ter uma variável denominada *while* em C.

3.2.3 Palavras Reservadas

Identificadores associados a componentes disponibilizados pela biblioteca padrão de C (v. **Seção 3.13**) são conhecidos como **palavras reservadas** (p. ex., *printf*) e o programador deve evitar redefini-los em seus programas. Identificadores que começam por subtraço também são reservados pela linguagem C e, portanto, também devem ser evitados.

3.3 Códigos de Caracteres

Os caracteres que devem fazer parte de uma implementação de C são denominados coletivamente **conjunto básico de caracteres**. Cada conjunto básico de caracteres deve conter um subconjunto mínimo de 96 caracteres especificado pelo padrão de C e o valor inteiro atribuído a cada caractere deve caber em um byte. Devem fazer parte do conjunto básico de caracteres os caracteres que aparecem na **Tabela 3–1**. Nessa tabela, **caractere gráfico** é o termo usado pelo padrão ISO de C para denominar um caractere com representação gráfica que não se enquadra em nenhuma das demais categorias de caracteres.

Categoria	Caracteres
Letras Maiúsculas	`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
Letras Minúsculas	`a b c d e f g h i j k l m n o p q r s t u v w x y z`
Dígitos	`0 1 2 3 4 5 6 7 8 9`
Caracteres Gráficos	`! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~`
Espaços em Branco	❒ espaço (`' '`) ❒ tabulação horizontal (`'\t'`) ❒ tabulação vertical (`'\v'`) ❒ quebra de linha (`'\n'`) ❒ quebra de página (`'\f'`)

Tabela 3–1: Conjunto Básico de Caracteres de C

Um **código de caracteres** consiste num mapeamento entre um conjunto de caracteres e um conjunto de inteiros. Como exemplos de código de caracteres bem conhecidos têm-se: ASCII, EBCDIC e ISO 8859-1.

O código **ASCII** é um dos mais antigos e conhecidos códigos de caracteres. Esse código de caracteres usa apenas 7 bits (e não 8 como muitos pensam), de modo que o número total de caracteres no código ASCII original é *128* (i.e., 2^7) e eles são mapeados em inteiros no intervalo de *0* a *127*.

Outro antigo código de caracteres é **EBCDIC**, que contém todos os caracteres que constam no código ASCII. Porém, os mapeamentos desses dois códigos de caracteres não são equivalentes. Além disso, os valores inteiros que mapeiam as letras do código EBCDIC não são consecutivos. Por exemplo, os valores inteiros associados às letras *i* e *j* no código EBCDIC são, respectivamente, *137* e *145*. No código ASCII, essas mesmas letras são representadas por *105* e *106*, respectivamente.

Apesar de o padrão ISO de C não especificar um código de caracteres que deva ser usado com a linguagem, a maioria dos códigos de caracteres em uso atualmente em implementações de C pode ser considerada extensão do código ASCII. Ou seja, os códigos de caracteres mais usados correntemente preservam os caracteres do código ASCII e seus mapeamentos. Exemplos de tais códigos de caracteres são aqueles especificados pelo padrão ISO 8859.

Em resumo, como nenhum padrão de C requer o uso de qualquer código de caracteres específico, o programador *jamais* deve fazer qualquer suposição sobre o código de caracteres utilizado por uma dada implementação de C.

3.4 Tipos de Dados Primitivos

Um **tipo de dado** (ou, simplesmente, **tipo**) consiste num conjunto de valores munido de uma coleção de operações permitidas sobre eles. Um tipo **primitivo** (ou **embutido**) é um tipo incorporado numa linguagem de programação e representado por uma palavra-chave. A linguagem C oferece ainda a possibilidade de criação de inúmeros outros tipos. Esses tipos, denominados **tipos derivados**, podem ser criados pelo próprio programador (v. **Seção 10.2**) ou providos pela biblioteca padrão de C (v. **Seção 3.13**).

Existem muitos tipos de dados primitivos em C, mas, em virtude da natureza introdutória deste livro, apenas dois tipos inteiros (**int** e **char**) e um real (**double**) serão discutidos aqui.

3.4.1 int

Existem diversos tipos inteiros primitivos em C. Neste livro, o tipo **int** foi escolhido para representar essa categoria porque ele é o tipo inteiro mais largamente usado e, principalmente, porque é fácil de lembrar (i.e., *int* corresponde às três primeiras letras de *int*eiro). O tipo **int**, por outro lado, herda um grave defeito desde os primórdios da criação da linguagem C: sua **largura** (i.e., o número de bytes ocupados por um valor desse tipo) não é precisamente especificada. Portanto esse tipo pode ser usado sem problemas com objetivos didáticos (como é o caso aqui), mas *nunca* deve ser usado na prática.

3.4.2 char

O espaço ocupado em memória por valores do tipo **char** é sempre um byte. Esse tipo é comumente usado para representar caracteres, mas ele pode representar quaisquer inteiros que requerem apenas um byte de memória. Isto é, uma variável do tipo **char** pode representar tanto um caractere quanto um inteiro que não está associado a um caractere, desde que esse inteiro caiba em um byte. Neste livro, esse tipo será usado apenas em processamento de caracteres.

3.4.3 double

Existem três tipos reais em C, mas, neste livro introdutório, apenas o tipo **double** será utilizado.

É importante salientar que, assim como alguns números reais não podem ser exatamente representados em base decimal (p. ex., `1/3`), o mesmo ocorre com representação na base binária, que é usada por computadores para representar números reais. Portanto muitos valores reais que possuem representação exata em base decimal são representados de modo aproximado em base binária. Por exemplo, o valor `0.1` pode ser representado exatamente em base decimal, mas esse não é o caso em base binária.

3.5 Constantes

Existem cinco tipos de constantes em C: **inteiras**, **reais**, **caracteres**, **strings** e de **enumerações**. Constantes desse último tipo serão vistas no **Capítulo 10**, enquanto as demais serão apresentadas a seguir.

3.5.1 Constantes Inteiras

Constantes inteiras em C podem ser classificadas de acordo com a base numérica utilizada em: **decimais**, **octais** e **hexadecimais**. Em programação de alto nível, apenas constantes em base decimal são usadas. Por isso, este livro só usa constantes dessa categoria.

Constantes inteiras em base decimal são escritas utilizando-se os dígitos de `0` a `9`, sendo que o primeiro dígito não pode ser zero, a não ser quando o próprio valor é zero. Exemplos válidos de constantes inteiras decimais são: `0`, `12004`, `-67`. Um exemplo inválido seria `06`, pois o número começa com zero e seria interpretado como uma constante na base octal.

3.5.2 Constantes Reais

Constantes reais podem ser escritas em duas notações:

- **Notação convencional**. Nessa notação, um ponto decimal separa as partes inteira e fracionária do número, como, por exemplo:

  `3.1415`

  `.5` [a parte inteira é zero]

  `7.` [a parte fracionária é zero]

- **Notação científica**. Nessa notação, um número real consiste em duas partes: (1) **mantissa** e (2) **expoente**. Essas partes são separadas por `e` ou `E` e a segunda parte representa uma potência de *10*. Por exemplo, o número `2E4` deve ser interpretado como 2×10^4 e lido como *dois vezes dez elevado à quarta potência*. A mantissa de um número real pode ser inteira ou fracionária, enquanto o expoente pode ser positivo ou negativo, mas deve ser inteiro. Por exemplo, `2.5E-3` representa o número 2.5×10^{-3}; ou seja, *0.0025*.

3.5.3 Caracteres Constantes

Existem duas formas básicas de representação de caracteres constantes em C:

- **Escrevendo-se o caractere entre apóstrofos** (p. ex., `'A'`). Apesar de essa forma de representação ser a mais legível e portável, ela é factível apenas quando o caractere possui representação gráfica e o programador possui meios para introduzi-lo (p. ex., uma configuração adequada de teclado).
- **Por meio de uma sequência de escape**, que consiste do caractere \ (barra inversa) seguido por um caractere com significado especial, sendo ambos envolvidos entre apóstrofos. Sequências de escape devem ser usadas para representar certos caracteres que não possuem representação gráfica (p. ex., quebra de linha) ou que têm significados especiais em C (p. ex., apóstrofo). Essa forma de representação de caracteres também é portável, apesar de ser menos legível do que a forma de representação

anterior. A Tabela 3–2 apresenta as sequências de escape mais comuns que podem ser utilizadas num programa escrito em C.

Sequência de Escape	Descrição	Frequência de Uso
`'\a'`	Campainha (alerta)	Raramente usada
`'\t'`	Tabulação	Frequentemente usada
`'\n'`	Quebra de linha	Muito usada
`'\r'`	Retorno (início da linha)	Raramente usada
`'\b'`	Retrocede um caractere	Raramente usada
`'\0'`	Caractere nulo	Muito usada
`'\\'`	Barra invertida	Eventualmente usada
`'\''`	Apóstrofo	Eventualmente usada
`'\"'`	Aspas	Eventualmente usada

Tabela 3–2: Sequências de Escape e seus Significados

As sequências de escape `'\t'` e `'\n'` são usadas com muita frequência em formatação de dados na tela usando **printf()** (v. Seção 3.14.1). Em instruções de saída na tela, `'\r'` faz o cursor mover-se para o início da linha corrente, enquanto `'\b'` faz o cursor mover-se uma posição para trás. A sequência de escape `'\0'` é usada em processamento de strings (v. Capítulo 9).

É importante salientar que um caractere constante pode ser representado tanto em um dos formatos descritos quanto por um número inteiro capaz de ser contido em um byte. Isto é, uma constante composta de um caractere ou uma sequência de escape entre apóstrofos apenas informa o compilador que ele deve considerar o valor correspondente ao caractere ou à sequência de escape no código de caracteres que ele usa. Por exemplo, se o código de caracteres utilizado for ASCII, quando o compilador encontra a constante `'A'`, ele a interpreta como `65` (esse valor corresponde ao mapeamento do caractere `'A'` no código ASCII).

Em resumo, qualquer que seja o formato de caractere constante utilizado, o compilador interpreta-o como o valor inteiro correspondente ao caractere no código de caracteres utilizado na respectiva implementação de C. Entretanto, não é uma boa ideia representar caracteres num programa por constantes inteiras por duas razões. A primeira razão é legibilidade (por exemplo, será que alguém vai entender o que significa `65` quando ler seu programa?). Requerer que alguém use uma tabela de caracteres para entender seu programa não é sensato. A segunda razão é que o programa pode ter sua portabilidade comprometida, pois o padrão de C não especifica que o código de caracteres utilizado por um compilador deva ser ASCII ou qualquer outro (v. Seção 3.3).

3.5.4 Strings Constantes

Um **string constante** consiste numa sequência de caracteres constantes. Em C, um string constante pode conter caracteres constantes em qualquer dos formatos apresentados e deve ser envolvido entre aspas. Exemplos de strings constantes são:

```
"bola"
"Dia\tMes\tAno\n"
```

Strings constantes separados por um ou mais espaços em branco (incluindo tabulação horizontal e quebra de linha) são automaticamente concatenados pelo compilador. Isso é útil quando se tem um string constante muito grande e deseja-se escrevê-lo em duas linhas. Por exemplo:

```
"Este e' um string constante muito grande para "
"ser contido numa unica linha do meu programa"
```

Os dois strings constantes do último exemplo serão concatenados pelo compilador para formar um único string constante:

```
"Este e' um string constante muito grande para ser contido numa unica linha do meu
programa"
```

3.6 Propriedades dos Operadores da Linguagem C

Informalmente, um operador representa uma operação elementar da linguagem C. Essa operação é, dependendo dela, aplicada a um ou mais valores denominados operandos.

C é uma linguagem intensivamente baseada no uso de operadores. Portanto o entendimento das propriedades dos operadores da linguagem é fundamental para a aprendizagem da própria linguagem.

As propriedades dos operadores são divididas em duas categorias:

[1] **Propriedades que todos os operadores possuem**. Essas propriedades correspondem exatamente àquelas vistas na **Seção 2.5.1** e serão revistas aqui para facilitar a retenção.

[2] **Propriedades que alguns operadores possuem**. Essas propriedades ainda não foram estudadas e, portanto, requerem maior atenção do leitor.

3.6.1 Propriedades que Todos os Operadores Possuem

Qualquer operador de C possui as propriedades apresentadas a seguir.

Resultado

O **resultado** de um operador é o valor resultante da aplicação do operador sobre seus operandos. Por exemplo: a aplicação do operador `+` na expressão `2 + 3` resulta em `5`.

Aridade

A **aridade** de um operador é o número de operandos sobre os quais o operador atua. De acordo com essa propriedade, os operadores de C são divididos em três categorias:

- **Operadores unários** — são operadores que requerem um operando
- **Operadores binários** — são operadores que requerem dois operandos
- **Operador ternário** — é um operador que requer três operandos

Por exemplo, o operador de soma, representado por `+`, tem aridade dois (i.e., ele é um operador binário).

Precedência

A **precedência** de um operador determina sua ordem de aplicação com relação a outros operadores. Isto é, um operador de maior precedência é aplicado antes de um operador de menor precedência. Em C, operadores são agrupados em **grupos de precedência** que satisfazem as seguintes propriedades:

- Todos os operadores que fazem parte de um mesmo grupo de precedência possuem a mesma precedência.
- Operadores que pertencem a diferentes grupos de precedência possuem precedências diferentes.

Essa propriedade foi discutida em detalhes na **Seção 2.5.1**. Refira-se àquela seção em caso de dúvidas.

Associatividade

Associatividade é um conceito semelhante ao de precedência no sentido de que ambos são utilizados para decidir a ordem de aplicação de operadores numa expressão. Entretanto, diferentemente do que ocorre com precedência, a associatividade é utilizada com operadores de mesma precedência. Existem dois tipos de associatividade em C:

- **Associatividade à esquerda**: o operador da esquerda é aplicado antes do operador da direita.
- **Associatividade à direita**: o operador da direita é aplicado antes do operador da esquerda.

Essa propriedade também foi discutida em maior profundidade na **Seção 2.5.1**. Portanto refira-se a essa seção em caso de dúvidas.

3.6.2 Propriedades que Alguns Operadores Possuem

Além das propriedades já apresentadas, *alguns operadores* da linguagem C possuem as propriedades adicionais apresentadas a seguir.

Efeito Colateral

Essa propriedade é análoga ao efeito colateral (muitas vezes indesejável) provocado por um medicamento. Nessa analogia, o efeito principal do medicamento corresponde ao resultado produzido pelo operador e o efeito colateral é a alteração de valor produzida pelo operador em um de seus operandos. Entretanto, no caso de operadores com efeito colateral, muitas vezes, o efeito colateral produzido pelo operador é o único proveito desejado. Em resumo, o efeito colateral de um operador consiste em alterar o valor de um de seus operandos e, portanto, o operando sujeito a esse efeito colateral deve ser uma variável. Um exemplo de operador com efeito colateral é o operador de atribuição (v. **Seção 3.9**).

Ordem de Avaliação

A ordem de avaliação de um operador indica qual dos operandos sobre os quais ele atua é avaliado primeiro. Logo essa propriedade só faz sentido para operadores com aridade maior do que um (i.e., operadores binários e ternários). Apenas quatro operadores de C possuem essa propriedade e esse fato frequentemente acarreta em expressões capazes de produzir dois resultados válidos possíveis (v. exemplos nas **Seções 3.9** e **3.11**). Os operadores de conjunção **&&** e disjunção lógicas **||**, que serão apresentados na **Seção 3.7.3**, são dois operadores que possuem ordem de avaliação definida.

Curto-circuito

Essa propriedade faz com que, às vezes, apenas um dos operandos de um operador binário seja avaliado. Apenas os operadores lógicos binários de conjunção e disjunção possuem essa propriedade (v. **Seção 3.7.3**).

3.6.3 Uso de Parênteses

Conforme foi enfatizado na **Seção 2.5.6**, o uso de parênteses influencia as propriedades de precedência, associatividade e, por conseguinte, resultado. Nenhuma outra propriedade de operadores sofre influência do uso de parênteses.

3.7 Operadores e Expressões

Uma expressão é uma combinação legal de operadores e operandos. Aquilo que constitui uma combinação *legal* de operadores e operandos é precisamente definido para cada operador da linguagem C.

3.7.1 Operadores Aritméticos

Uma **expressão aritmética** é uma combinação de **operadores aritméticos** e operandos numéricos que, quando avaliada, resulta num valor numérico. Os operandos de uma expressão aritmética podem incluir variáveis, constantes ou chamadas de funções, que resultem em valores numéricos. Os operadores aritméticos básicos de C são apresentados na **Tabela 3–3**.

Símbolo	Operador
-	Menos unário (inversão de sinal)
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto de divisão inteira

Tabela 3–3: Operadores Aritméticos de C

A maioria dos operadores da **Tabela 3–3** funciona como em aritmética convencional e em outras linguagens de programação, mas, em C, existem algumas diferenças.

Usando um compilador que segue um padrão anterior a C99, quando um ou ambos os operandos envolvidos numa divisão inteira são negativos, existem dois possíveis resultados e o mesmo ocorre com resto de divisão inteira. Em compiladores que seguem os padrões mais recentes da linguagem C (C99 e C11), a divisão inteira tem sempre um resultado único possível, que é obtido como se o quociente da divisão real dos respectivos operandos tivesse sua parte fracionária descartada. Por exemplo, o quociente da divisão real de `-17` por `5` é `-3.4`; desprezando-se a parte fracionária desse resultado, obtém-se `-3`, que é o quociente da divisão inteira `-17/5`.

Tendo calculado o resultado de uma divisão inteira, o resto dessa mesma divisão pode ser obtido por meio da fórmula:

$$\textit{resto} = \textit{numerador} - \textit{quociente} \times \textit{denominador}$$

Por exemplo, utilizando essa fórmula e o resultado obtido para `-17/5`, tem-se que:

$$\begin{aligned} -17\%5 &= -17 - -3 \times 5 \\ &= -17 + 15 \\ &= -2 \end{aligned}$$

Operadores aritméticos são agrupados em grupos de precedência conforme mostra a **Tabela 3–4**. Operadores de um mesmo grupo nesta tabela têm a mesma precedência. Dentre os operadores apresentados nessa tabela, apenas o operador unário é associativo à direita; os demais são associativos à esquerda. Por exemplo, a expressão:

```
- -2
```

seria interpretada como `-(-2)`, visto que o operador `-` (unário) é associativo à direita. Note que, na expressão acima, há um espaço em branco entre os dois traços. Isso evita que o compilador interprete-os como operador de decremento (v. **Seção 3.11**).

Grupo de Operadores	Precedência
– (unário)	Mais alta
*, /, %	⬇
+, – (binários)	Mais baixa

Tabela 3–4: Precedências de Operadores Aritméticos de C

É importante salientar que divisão inteira ou resto de divisão inteira por zero é uma operação ilegal que causa o término abrupto (**aborto**) da execução de um programa que tenta realizar tal operação. Por outro lado, divisão real por zero não causa aborto de programa, mas não produz um número real como resultado. Portanto antes de efetuar qualquer operação de divisão ou resto de divisão, deve-se verificar, por meio de uma instrução **if** (v. Seção 4.6.1), se o divisor da operação é zero.

3.7.2 Operadores Relacionais

Operadores relacionais são operadores binários utilizados em expressões de comparação. Os operandos de um operador relacional podem ser de qualquer tipo numérico. Os operadores relacionais de C são apresentados, com seus possíveis resultados, na Tabela 3–5, enquanto a Tabela 3–6 exibe seus grupos de precedência. Os quatro primeiros operadores na Tabela 3–5 têm a mesma precedência, que é menor do que as precedências dos operadores aritméticos vistos antes. Os dois últimos operadores estão uma classe de precedência abaixo dos quatro primeiros operadores.

Símbolo	Operador	Aplicação	Resultado
`>`	maior do que	`a > b`	`1` se `a` é maior do `b`; `0`, caso contrário
`>=`	maior ou igual	`a >= b`	`1` se `a` é maior do que ou igual a `b`; `0`, caso contrário
`<`	menor do que	`a < b`	`1` se `a` é menor do `b`; `0`, caso contrário
`<=`	menor ou igual	`a <= b`	`1` se `a` é menor do que ou igual a `b`; `0`, caso contrário
`==`	igualdade	`a == b`	`1` se `a` é igual a `b`; `0`, caso contrário
`!=`	diferente	`a != b`	`1` se `a` é diferente de `b`; `0`, caso contrário

Tabela 3–5: Operadores Relacionais de C

Grupo de Operadores	Precedência
>, >=, <, <=	Mais alta
==, !=	Mais baixa

Tabela 3–6: Precedências dos Operadores Relacionais de C

A linguagem C e a linguagem algorítmica apresentada no Capítulo 2 possuem operadores relacionais equivalentes. Entretanto, uma diferença fundamental é que a linguagem algorítmica possui o tipo de dados booleano e expressões de comparação resultam em verdadeiro ou falso. Em C, o resultado de qualquer expressão relacional corresponde a um valor inteiro (`0` ou `1`). Além disso, alguns operadores relacionais correspondentes nessas linguagens usam símbolos diferentes.

O programador deve tomar cuidado para não usar inadvertidamente o símbolo `=`, que representa o operador de igualdade em linguagem algorítmica, em substituição ao símbolo `==`, que representa esse operador em C. Esse tipo de erro, muito comum em programação em C, é muitas vezes difícil de ser localizado, pois o uso por engano

do símbolo = em vez de == é, na maioria das vezes, *sintaticamente* legal, embora, frequentemente, resulte numa interpretação diferente da pretendida. Um exemplo concreto dessa situação será apresentado na **Seção 4.6.1**.

Conforme foi visto na **Seção 3.4.3**, números reais podem ser representados aproximadamente no computador. Por isso, operadores relacionais não devem ser usados para comparar números reais da mesma maneira que eles são usados para comparar números inteiros. Na **Seção 5.11.6**, será mostrado como números reais podem ser comparados.

3.7.3 Operadores Lógicos

Operadores lógicos em C correspondem aos operadores de **negação**, **conjunção** e **disjunção** da linguagem algorítmica (v. **Seção 2.5.4**). Entretanto, diferentemente do que ocorre na linguagem algorítmica, os operadores lógicos em C podem receber como operandos quaisquer valores numéricos. Os operadores lógicos de C são apresentados em ordem decrescente de precedência na **Tabela 3–7**.

Operador	Símbolo	Precedência
Negação	!	Mais alta
Conjunção	&&	↓
Disjunção	||	Mais baixa

Tabela 3–7: Operadores Lógicos de C em Ordem Decrescente de Precedência

A aplicação de um operador lógico sempre resulta em `0` ou `1`, dependendo dos valores dos seus operandos. Os valores resultantes da aplicação desses operadores de acordo com os valores de seus operandos são resumidos na **Tabela 3–8** e na **Tabela 3–9**.

X	!X
0	1
diferente de 0	0

Tabela 3–8: Resultados do Operador de Negação (!) de C

X	Y	X && Y	X || Y
0	0	0	0
0	diferente de 0	0	1
diferente de 0	0	0	1
diferente de 0	diferente de 0	1	1

Tabela 3–9: Resultados dos Operadores de Conjunção (&&) e Disjunção (||) de C

Para memorizar a **Tabela 3–9**, você precisa apenas considerar que a aplicação de **&&** resulta em `1` apenas quando os dois operandos são diferentes de zero; caso contrário, o resultado é `0`. De modo semelhante, `X || Y` resulta em `0` apenas quando ambos os operandos são iguais a `0`.

O operador lógico **!** tem a mesma precedência do operador aritmético unário, (representado por **–**), visto na **Seção 3.7.1**. Os operadores lógicos **&&** e **||** têm precedências mais baixas do que operadores aritméticos e relacionais, mas não fazem parte de um mesmo grupo de precedência: o operador **&&** tem precedência mais alta do que o operador **||**. A **Tabela 3–10** apresenta as precedências relativas entre todos os operadores vistos até aqui.

A propósito, uma informação importante que merece ser memorizada é a seguinte:

Recomendação *Todos os operadores unários de C fazem parte de um mesmo grupo de precedência. Os operadores desse grupo possuem a segunda maior precedência dentre todos os operadores da linguagem C e a associatividade deles é à direita.*

Apenas os operadores de acesso, que serão apresentados no Capítulo 10, possuem maior precedência do que os operadores unários.

Os operadores **&&** e **||** têm ordem de avaliação de operandos especificada como sendo da esquerda para a direita. Isto é, o primeiro operando é sempre avaliado em primeiro lugar.

Grupo de Operadores	Precedência
!, – (unários)	Mais alta
*, /, %	
+, – (binários)	
>, >=, <, <=	
==, !=	
&&	
||	Mais baixa

Tabela 3–10: Precedências de Operadores Aritméticos, Relacionais e Lógicos de C

Curto-circuito do Operador de Conjunção (&&)

Quando o primeiro operando do operador **&&** é zero, seu segundo operando não é avaliado e o resultado da operação é zero. Por exemplo, quando o valor de `a` for zero na expressão:

```
(a != 0) && (b/a > 4)
```

o operando `a != 0` resulta em `0` e o operando `b/a > 4` não é avaliado, pois considera-se que, para que toda a expressão resulte em `0`, basta que um dos operandos seja `0`.

Curto-circuito do Operador de Disjunção (||)

Quando o primeiro operando do operador **||** é diferente de zero, seu segundo operando não é avaliado e o resultado da operação é `1`. Por exemplo, se o valor de `x` for `5` na expressão:

```
(x > 0) || (y < 0)
```

o operando `x > 0` resulta em `1` e o operando `y < 0` não é avaliado.

3.8 Definições de Variáveis

Em programação, uma **definição de variável** tem três objetivos: (1) prover uma interpretação para o espaço em memória ocupado pela variável, (2) fazer com que seja alocado espaço suficiente para conter a variável e (3) associar esse espaço a um identificador (i.e., o nome da variável).

Em C, toda variável precisa ser definida antes de ser usada. Uma definição de variável em C consiste num identificador representando o tipo da variável seguido do nome da variável. Variáveis de um mesmo tipo podem ainda ser definidas juntas e separadas por vírgulas. Por exemplo, a definição:

```
int minhaVar, i, j;
```

é responsável pela alocação em memória das variáveis `minhaVar`, `i` e `j` como sendo do tipo **int**.

É recomendado que identificadores que representem categorias diferentes de componentes de um programa sigam notações diferentes de composição para facilitar a identificação visual de cada componente. No caso de variáveis, as recomendações são as seguintes:

- Inicie o nome de uma variável com letra minúscula.
- Se o nome da variável for composto, utilize letra maiúscula no início de cada palavra seguinte, inclusive palavras de ligação (p. ex., preposições e conjunções)
- Não utilize subtraço.

Seguem mais algumas recomendações para a escolha de identificadores para variáveis:

- Escolha nomes que sejam significativos (p. ex., `matricula` é mais significativo do que `x` ou `m`).
- Evite utilizar nomes de variáveis que sejam muito parecidos ou que difiram em apenas um caractere (p. ex., `primeiraNota` e `segundaNota` são melhores do que `nota1` e `nota2`).
- Evite utilizar `l` (*ele*) e `o` (*ó*) como nomes de variáveis pois são facilmente confundidos com `1` (*um*) e `0` (*zero*). Esse problema também pode ser evitado com uma escolha adequada de fonte para edição de programas, como aquela usada neste livro. (A propósito, o nome dessa fonte é *Adobe Source Code Pro*.)

Outra recomendação que melhora a legibilidade de programas é o uso de espaços horizontais para alinhar identificadores de variáveis numa seção de declaração. Por exemplo:

```
int         umaVariavel;
static int  outraVariavel;
```

3.9 Operador de Atribuição

Uma das instruções mais simples em C é a instrução de **atribuição**. Uma instrução desse tipo preenche o espaço em memória representado por uma variável com um valor determinado e sua sintaxe tem a seguinte forma geral:

***variável* = *expressão*;**

A interpretação para uma instrução de atribuição é a seguinte: a expressão (lado direito) é avaliada e o valor resultante é armazenado no espaço de memória representado pela variável (lado esquerdo).

Quando um espaço em memória é alocado para conter uma variável, o conteúdo desse espaço (i.e., o valor da própria variável) *pode ser* indeterminado (v. **Seção 5.9.3**). Isso significa que não se deve fazer nenhuma suposição sobre o valor de uma variável antes que ela assuma um valor *explicitamente* atribuído. Às vezes, é desejável que uma variável assuma certo valor no instante de sua definição. Essa **iniciação** pode ser feita em C combinando-se a definição da variável com a atribuição do valor desejado. Por exemplo, suponha que se deseje atribuir o valor inicial `0` a uma variável inteira denominada `minhaVar`. Então, isso poderia ser feito por meio da seguinte iniciação:

```
int  minhaVar = 0;
```

Uma instrução de atribuição é uma expressão e o sinal de igualdade utilizado em atribuição representa o operador principal dessa expressão. Esse operador faz parte de um grupo de operadores que têm uma das mais baixas precedências dentre todos os operadores de C. De fato, apenas o operador vírgula (v. **Seção 4.9**) possui precedência menor do que esse grupo de operadores.

O operador de atribuição possui efeito colateral, que consiste exatamente na alteração de valor causada na variável (operando) que ocupa o lado esquerdo da expressão. Esse operador possui associatividade à direita e o resultado da aplicação desse operador é o valor recebido pela variável no lado esquerdo da expressão de atribuição. Em virtude das suas propriedades, o operador de atribuição pode ser utilizado para a execução de múltiplas atribuições numa única linha de instrução. Por exemplo, se `x`, `y` e `z` são variáveis do tipo **int**, a atribuição composta:

```
x = y = z = 1;
```

resulta na atribuição de `1` a `z`, `z` a `y` e `y` a `x`, nessa ordem. Nesse caso, `x`, `y` e `z` terão, no final da execução da instrução, o mesmo valor, mas isso nem sempre acontece numa atribuição múltipla, pois podem ocorrer conversões implícitas, como mostra o seguinte exemplo:

```
int    i;
double d;

i = d = 2.5;
```

Nesse exemplo, `d` recebe o valor `2.5`, mas `i` recebe o valor `2` (o valor de `d` convertido em **int**).

Exercício: Que valores receberiam `i` e `d` na atribuição `d = i = 2.5`?

Como ocorre com a maioria dos operadores de C, o operador de atribuição não possui ordem de avaliação de operandos definida. Isso, aliado ao fato de o operador de atribuição possuir efeito colateral, pode dar origem a expressões capazes de produzir dois resultados diferentes, mas aceitáveis. Por exemplo, no trecho de programa:

```
int x = 0, y = 0;

(x = 1)*(x + 2);
```

a última expressão pode produzir dois resultados diferentes, dependendo de qual dos operandos [`(x = 1)` ou `(x + 2)`] é avaliado primeiro. Isto é, se o primeiro operando for avaliado antes do segundo, o resultado será `3` (`1` vezes `3`); enquanto, se o segundo operando for avaliado antes do primeiro, o resultado será `2` (`1` vezes `2`). Qualquer dos dois resultados é válido porque a padrão da linguagem C não especifica qual dos dois operandos deve ser avaliado primeiro. Portanto a expressão `(x = 1)*(x + 2)` não é portável em C.

Examinando detalhadamente a expressão do último exemplo, podem-se descobrir quais são os ingredientes de uma expressão capaz de produzir dois resultados distintos e legítimos:

- **Uso de operador sem ordem de avaliação de operandos especificada**. No último exemplo, o operador de multiplicação não tem ordem de avaliação de operandos definida, mas se esse operador for substituído, por exemplo, pelo operador **&&**, a expressão resultante `(x = 1) && (x + 2)` será portável.
- **Uso de um operador com efeito colateral**. No exemplo em questão, o operador de atribuição possui efeito colateral; se esse operador for substituído, por exemplo, pelo operador de subtração, a expressão resultante `(x - 1)*(x + 2)` será portável.
- **Uma variável que sofre efeito colateral faz parte dos dois operandos de um operador sem ordem de avaliação de operandos definida**. No exemplo em questão, a variável `x` aparece nos dois operandos do operador `*`. Se uma das ocorrências dessa variável for substituída por outra variável, a expressão resultante torna-se portável. Por exemplo, `(x = 1)*(y + 2)` é portável.

3.10 Conversões de Tipos

3.10.1 Conversões Implícitas

C permite que operandos de tipos aritméticos diferentes sejam misturados em expressões. Entretanto, para que tais expressões façam sentido, o compilador executa **conversões implícitas** (ou **automáticas**). Essas conversões

não incluem arredondamento e muitas vezes são responsáveis por resultados inesperados. Portanto o programador deve estar absolutamente ciente das transformações feitas implicitamente pelo compilador antes de misturar valores de tipos diferentes numa expressão; caso contrário, é melhor evitar definitivamente essa mistura.

Diz-se que dois tipos são **compatíveis** entre si quando eles são iguais ou quando qualquer valor de um tipo pode ser convertido implicitamente num valor do outro tipo. Assim, todos os tipos aritméticos discutidos neste livro são compatíveis entre si. Entretanto, isso não significa que não ocorre perda de informação durante tal conversão (v. adiante).

Existem cinco situações nas quais conversões são feitas implicitamente por um compilador de C. Três delas serão discutidas a seguir, enquanto as demais serão discutidas no **Capítulo 5**.

Conversão de Atribuição

Nesse tipo de conversão, o valor do lado direito de uma atribuição é convertido no tipo da variável do lado esquerdo. Um problema que pode ocorrer com esse tipo de conversão surge quando o tipo do lado esquerdo é *mais curto* do que o tipo do lado direito da atribuição. Por exemplo, se `c` é uma variável do tipo **char**, a atribuição:

```
c = 936;
```

pode resultar, na realidade, na atribuição do valor `168` a `c`. Por que isso ocorre? Primeiro, o valor `936` é grande demais para caber numa variável do tipo **char** (que, relembrando, ocupa apenas um byte). Segundo, pode-se observar que `936` é representado em 16 bits pela sequência:

```
00000011  10101000
```

Mas, como valores do tipo **char** devem ser contido em apenas 8 bits, o compilador *pode* considerar apenas os 8 bits de menor ordem:

```
10101000
```

que correspondem a `168` em base decimal.

Conversões entre números reais e inteiros também podem ser problemáticas. Claramente, a atribuição de um número real a uma variável inteira pode causar perda de informação, pois números reais são usualmente capazes de acomodar valores bem maiores do que o maior valor permitido para uma variável inteira. Por exemplo, se `i` é uma variável do tipo **int**, a seguinte atribuição:

```
i = 1.85E100;
```

faz com que a `i` seja atribuído o valor `2147483647` numa dada implementação de C. O estranho valor atribuído à variável `i` surgiu em virtude da ocorrência de overflow (v. **Seção 4.11.7**); i.e., uma variável do tipo **int** não é capaz de armazenar um valor tão grande quanto aquele que lhe foi atribuído.

Os erros decorrentes de overflow nos exemplos apresentados acima são fáceis de entender. Mas, numa atribuição de um valor real a uma variável inteira podem ocorrer erros decorrentes de truncamento que são bem mais difíceis de detectar e corrigir.

Truncar um número significa desprezar alguns de seus algarismos e, quando um número real é convertido em número inteiro, sua parte fracionária é truncada. Em virtude da relativa complexidade envolvida com erros de truncamento, a esse tópico será dedicada uma seção especial (**Seção 7.5** do **Capítulo 7**).

Apesar de os tipos reais serem usualmente capazes de incluir valores maiores do que os tipos inteiros em termos de ordem de grandeza, a atribuição de um número inteiro a uma variável de um tipo real pode causar perda de precisão, pois o tipo real poderá não ser capaz de representar todos os algarismos significativos do número

inteiro, que, nesse caso, será arredondado. Entretanto, problemas dessa natureza são improváveis com os tipos **double** e **int** usados neste livro.

Conversão Aritmética Usual

Quando um operador que não é de atribuição possui operandos de tipos diferentes, um deles é convertido no tipo do outro, de modo que eles passem a ter o mesmo tipo. Essa categoria de conversão é denominada **conversão aritmética usual**, porque ela afeta operandos aritméticos, mas ela se aplica a operadores de qualquer natureza (com exceção de atribuição) e obedece as seguintes regras:

- Quando um operando é um valor inteiro (i.e., do tipo **int** ou **char**) e outro operando do mesmo operador é do tipo **double**, o operando inteiro é convertido em **double** antes de a expressão ser avaliada. Precisamente, isso significa que o valor inteiro usado pelo programador será substituído por um valor do tipo **double** quando o programa for compilado.
- Quando um operando é do tipo **int** e o outro operando do mesmo operador é **char**, o operando do tipo **char** é convertido em **int** antes de a expressão ser avaliada.

Conversão de Alargamento

Mesmo quando uma variável do tipo **char** (ou parâmetro ou retorno de função desse tipo — v. **Capítulo 5**) não é misturada com operandos de outros tipos, por uma questão de eficiência, seu valor é convertido, num valor do tipo **int**. Essa categoria de conversão é denominada **conversão de alargamento** e nunca traz nenhum dano para um programa. É por causa dela que programadores conscientes usam o tipo **int** em declarações de variáveis e parâmetros mesmo que eles armazenem apenas valores que caberiam numa variável do tipo **char**.

3.10.2 Conversão Explícita

O programador pode especificar conversões explicitamente em C antepondo ao valor que deseja transformar o nome do tipo desejado entre parênteses. Por exemplo, suponha que se tenham as seguintes linhas de programa em C:

```
int    i1 = 3, i2 = 2;
double d;

d = i1/i2;
```

Nesse caso, em virtude de conversões implícitas, `d` receberá o valor `1.0`, o que talvez não seja o esperado. Entretanto, se um dos inteiros for promovido explicitamente a **double** o resultado será `1.5`, o que pode ser obtido do seguinte modo:

```
d = (double) i1/i2;
```

A construção (*tipo*), como **(double)**, trata-se de mais um operador da linguagem C, denominado **operador de conversão explícita**. Esse operador tem a mesma precedência e associatividade dos outros operadores unários. Logo a expressão anterior é interpretada como:

```
((double) i1)/i2
```

Se a expressão `i1/i2` for colocada entre parênteses, a conversão será feita, de forma redundante, sobre o resultado dessa expressão e o valor atribuído a `d` será novamente `1.0`.

O uso do operador de conversão explícita, mesmo quando ele é desnecessário, é capaz de melhorar a legibilidade de programas. Por exemplo, suponha que `d` é do tipo **double** e `i` é do tipo **int**. Então, na atribuição:

```
i = (int) d;
```

o operador **(int)** é utilizado apenas para indicar que o programador está ciente de que ocorre uma conversão para **int**. Funcionalmente, esse operador é redundante, uma vez que essa transformação ocorreria implicitamente se o operador não fosse incluído. No entanto, o uso do operador **(int)** torna clara essa conversão.

3.11 Incremento e Decremento

Os operadores de **incremento** e **decremento** são operadores unários que, quando aplicados a uma variável, adicionam-lhe ou subtraem-lhe `1`, respectivamente. Esses operadores aplicam-se a variáveis numéricas ou ponteiros (v. **Seção 8.6**). Os operadores de incremento e decremento são representados respectivamente pelos símbolos `++` e `--`. Eles têm a mesma precedência e associatividade de todos os operadores unários.

Existem duas versões para cada um desses operadores: (1) **prefixa** e (2) **sufixa**. Sintaticamente, essa classificação refere-se à posição do operador em relação ao seu operando. Se o operador aparece antes do operando (p. ex., `++x`), ele é um operador prefixo; caso contrário (p. ex., `x++`), ele é sufixo. Todos esses operadores produzem efeitos colaterais nas variáveis sobre as quais atuam. Esses efeitos colaterais correspondem exatamente ao incremento (i.e., acréscimo de `1` ao valor) ou decremento (i.e., subtração de `1` do valor) da variável.

Com relação a efeitos colaterais, não existe diferença quanto ao uso da forma prefixa ou sufixa de cada um desses operadores. Isto é, qualquer versão do operador de incremento produz o mesmo efeito de incremento e qualquer versão do operador de decremento produz o mesmo efeito de decremento.

A diferença entre as versões prefixa e sufixa de cada um desses operadores está no resultado da operação. Os operadores sufixos produzem como resultado o *próprio valor da variável* sobre a qual atuam; por outro lado, operadores prefixos resultam no valor da variável *após o efeito colateral* (i.e., incremento ou decremento) ter ocorrido. Portanto se o resultado de uma operação de incremento ou de decremento não for utilizado, não faz nenhuma diferença se o operador utilizado é prefixo ou sufixo. Por exemplo, no trecho de programa a seguir:

```
int  y, x = 2;
y = 5*x++;
```

`y` recebe o valor `10`, enquanto se a instrução contendo incremento fosse escrita como:

```
y = 5*++x;
```

`y` receberia o valor `15`. Em ambos os casos, entretanto, a variável `x` seria incrementada para `3` (i.e., o efeito colateral seria o mesmo).

Supondo que `x` é uma variável numérica ou um ponteiro, os resultados e efeitos colaterais dos operadores de incremento e decremento são resumidos na **Tabela 3–11**.

Operação	Denominação	Valor da Expressão	Efeito Colateral
`x++`	incremento sufixo	o mesmo de `x`	adiciona `1` a `x`
`++x`	incremento prefixo	o valor de `x` mais `1`	adiciona `1` a `x`
`x--`	decremento sufixo	o mesmo de `x`	subtrai `1` de `x`
`--x`	decremento prefixo	o valor de `x` menos `1`	subtrai `1` de `x`

Tabela 3–11: Operadores de Incremento e Decremento

Muitas vezes, o programador está interessado apenas no efeito colateral do operador de incremento ou decremento. Nesse caso, a escolha da versão de operador utilizada é absolutamente irrelevante. Ou seja, o programador deve preocupar-se com a diferença entre as versões desses operadores apenas quando os valores resultantes de expressões formadas por esses operadores forem utilizados.

Do mesmo modo como ocorre com o operador de atribuição (v. **Seção 3.9**), não é recomendado o uso de uma variável afetada por um operador de incremento ou decremento na mesma expressão em que ocorre tal incremento ou decremento. Por exemplo, considere o seguinte trecho de programa:

```
int  i, j = 4;
i = j * j++;
```

Conforme já foi visto, a linguagem C não especifica qual dos operandos da multiplicação é avaliado primeiro e, portanto o resultado a ser atribuído a `i` irá depender da livre interpretação do compilador. Nesse caso, há dois resultados possíveis: `16` ou `20` (verifique isso).

O programador também deve evitar (ou, pelo menos, tomar bastante cuidado com) o uso de operadores com efeitos colaterais, como os operadores de incremento e decremento, com os operadores lógicos **&&** e **||**. O problema agora é que, conforme visto na **Seção 3.7.3**, algumas expressões lógicas nem sempre são completamente avaliadas. Como exemplo, considere a expressão:

```
(a > b) && (c == d++)
```

Nessa situação, a variável `d` seria incrementada apenas quando `a` fosse maior do que `b` e, talvez, o programador desejasse que ela fosse incrementada *sempre* que essa expressão fosse avaliada.

3.12 Comentários

Comentários em C são quaisquer sequências de caracteres colocadas entre os **delimitadores de comentários** `/*` e `*/`. Caracteres entre delimitadores de comentários são totalmente ignorados por um compilador de C e, portanto, não existe nenhuma restrição em relação a esses caracteres. Por exemplo:

```
/* Isto é um comentário acentuado em bom português */
```

De acordo com o padrão ISO não é permitido o uso aninhado de comentários desse tipo; i.e., um comentário no interior de outro comentário. Entretanto, algumas implementações de C aceitam isso, o que pode ser bastante útil na depuração de programas, pois permite excluir, por meio de comentários, trechos de programa que já contenham comentários (v. **Seção 7.4.3**).

O padrão C99 introduz o delimitador de comentário `//`. Caracteres que seguem esse delimitador até o final da linha que o contém são ignorados pelo compilador. Exemplo:

```
// Outro comentário acentuado em bom português
```

Esse tipo de comentário pode ser aninhado em comentários do tipo anterior:

```
/*
x = 10; // Isto é um comentário
*/
```

E vice-versa:

```
x = 10; // Isto /* é outro */ comentário
```

O objetivo principal de comentários é explicar o programa para outros programadores e para você mesmo quando for lê-lo algum tempo após sua escrita. Um programa sem comentários força o leitor a fazer inferências para tentar entender o que e como o programa faz. Aqui, serão apresentados alguns conselhos sobre *quando*, *como* e *onde* comentar um programa a fim de clarificá-lo.

O melhor momento para comentar um trecho de um programa ou algoritmo é exatamente quando esse trecho está sendo escrito. Isso se aplica especialmente àqueles trechos de programa contendo sutilezas, inspirações momentâneas ou coisas do gênero, que o próprio programador terá dificuldade em entender algum tempo depois. Alguns comentários passíveis de ser esquecidos (p. ex., a data de início da escrita do programa) também devem ser acrescentados no momento da escrita do programa. Outros comentários podem ser acrescentados quando o programa estiver pronto, embora o ideal continue sendo comentar todo o programa à medida que ele é escrito. Se você deixar para comentar um programa algum tempo após tê-lo concluído, pode ser que não seja capaz de descrever tudo que fez. Ou seja, o que provavelmente era óbvio quando você escreveu o programa pode ter se tornado indecifrável quando estiver escrevendo os comentários algum tempo depois.

Comentários devem ser claros e dirigidos para programadores com alguma experiência na linguagem de programação utilizada. Isto é, comentários devem incluir tudo aquilo que um programador precisa saber sobre um programa e nada mais. Eles não têm que ser didáticos como muitos comentários apresentados aqui e em outros textos de ensino de programação. Esses comentários didáticos são úteis nesses textos que têm exatamente o objetivo de ensinar, mas comentários num programa real têm o objetivo de explicar o programa para programadores e não o de ensinar a um leigo na linguagem o que está sendo feito. Por exemplo, o comentário na seguinte linha de instrução

```
x = 2; /* O conteúdo de x passa a ser 2 */
```

é aceitável num texto que se propõe a ensinar programação, mas não faz sentido num programa real.

A melhor forma de adquirir prática na escrita de comentários é lendo programas escritos por programadores profissionais e observando os vários estilos de comentários utilizados.

Existem dois formatos básicos de comentários: comentário de bloco e comentário de linha, que devem ser utilizados conforme é sugerido aqui ou de acordo com a necessidade. Em qualquer caso, você não precisa seguir os *formatos* dos modelos sugeridos. Isto é, você é livre para usar sua criatividade para criar seu próprio estilo de comentário; o que se espera é que o *conteúdo* corresponda àquele sugerido nos formatos de comentários apresentados a seguir.

Blocos de comentário são utilizados no início do programa [i.e., no início do arquivo contendo a função **main()**] com o objetivo informativo de apresentar o propósito geral do programa, data de início do projeto, nome do programador, versão do programa, nota de direitos autorais (*copyright*) e qualquer outra informação pertinente. Por exemplo:

```
/****
 *
 * Título do Programa: MeuPrograma
 *
 * Autor: José da Silva
 *
 * Data de Início do Projeto: 10/11/2012
 * Última modificação: 19/11/2012
 *
 * Versão: 2.01b
 *
 * Descrição: Este programa faz isto e aquilo.
```

```
*
* Dados de Entrada: Este programa espera os seguintes dados...
*
* Dados de Saída: Este programa produz como saída o seguinte...
*
* Uso: Este programa deve ser usado assim...
*
* Copyright © 2012 José da Silva Software Ltda.
*
****/
```

Outros itens comumente usados em comentários iniciais de programas incluem:

- Propósito do programa (i.e., para que o programa foi construído?).
- Crédito para alguma porção do programa desenvolvido por outrem.
- Como devem ser os formatos dos arquivos que seu programa lê ou escreve.
- Restrições que se aplicam ao programa (p. ex., *Este programa não checa erros de entrada de dados*).
- Histórico de revisões: alterações efetuadas no programa, quem as fez e quando.
- Tratamento de erros: quando o programa detecta um erro, o que ele faz?
- Observações: inclua qualquer comentário relevante sobre o programa que ainda não tenha sido apresentado.

Comentários de bloco dessa natureza fazem parte dos exemplos de programação obtidos via download no site do livro (***www.ulysseso.com/ip***).

3.13 Bibliotecas

Em programação, **biblioteca** é uma coleção de componentes que o programador pode incluir em seus programas. Em C, tais componentes incluem, por exemplo, constantes simbólicas (v. **Seção 3.15**), tipos de dados derivados (v. **Seção 10.2**) e funções. **Função** é o nome que se dá a um subprograma na linguagem C e é assunto a ser discutido em profundidade no **Capítulo 5**. O uso de componentes prontos para ser incorporados em programas facilita sobremaneira o trabalho dos programadores.

3.13.1 A Biblioteca Padrão de C

A **biblioteca padrão de C** é aquela preconizada pelo padrão ISO dessa linguagem. Isso significa que todo compilador de C que adere ao padrão ISO deve ser distribuído acompanhado dessa biblioteca. A biblioteca que acompanha um compilador pode incluir componentes adicionais além daqueles recomendados pelo padrão de C. O uso de componentes específicos de uma determinada biblioteca num programa prejudica sua portabilidade.

A biblioteca padrão de C é dividida em grupos de componentes que têm alguma afinidade entre si. Por exemplo, existem componentes para entrada e saída, processamento de strings, operações matemáticas etc. Cada grupo de componentes, denominado **módulo**, possui dois arquivos associados:

- Um arquivo-objeto que contém as implementações dos componentes do módulo previamente compiladas.
- Um arquivo de texto, denominado **cabeçalho**, que contém *definições parciais* (**alusões**) legíveis das funções implementadas no arquivo-objeto correspondente. Um arquivo de cabeçalho pode ainda conter, entre outros componentes, definições de tipos (v. **Seção 10.2**) e constantes simbólicas (v. **Seção 3.15**). Arquivos de cabeçalho normalmente têm a extensão `.h` (que é derivada de *header*; i.e., cabeçalho em inglês).

Para utilizar algum componente de um módulo de biblioteca num programa, deve-se incluir o arquivo de cabeçalho desse módulo usando uma **diretiva #include** (usualmente, no início do programa) com o formato:

```
#include <nome do arquivo>
```

ou:

```
#include "nome do arquivo"
```

Normalmente, o primeiro formato de inclusão é usado para cabeçalhos da biblioteca padrão, enquanto o segundo formato é usado para cabeçalhos de outras bibliotecas.

Linhas de um programa em C que começam com o símbolo # são denominadas **diretivas**. Elas são processadas por um programa, denominado **pré-processador**, cuja execução antecede o processo de compilação propriamente dito. Diretivas possuem sintaxe própria. Por exemplo, diretivas não terminam com ponto e vírgula, como instruções da linguagem C.

Como foi afirmado, um arquivo de cabeçalho contém informações incompletas sobre funções. Mas, apesar de incompletas, essas informações são suficientes para o compilador checar se um determinado uso (chamada) de função é correto. As definições completas das funções aludidas num cabeçalho estão contidas no arquivo-objeto associado ao cabeçalho. Para que um programa-fonte que usa uma função de biblioteca possa ser transformado em programa executável, é necessário que o arquivo-objeto resultante de sua compilação seja ligado ao arquivo-objeto que contém o código compilado da função. Essa ligação é feita por um linker (v. **Seção 3.16.2**).

Atualmente, 29 cabeçalhos fazem parte da biblioteca padrão de C. A **Tabela 3–12** apresenta os cabeçalhos mais comumente utilizados e seus propósitos. Apenas alguns dos vários componentes da biblioteca padrão de C serão apresentados neste livro devido à sua natureza introdutória.

Cabeçalho	Usado com...
`<stdio.h>`	**Entrada e saída em geral, incluindo leitura de dados via teclado** (v. **Capítulo 11**), **escrita de dados na tela** (v. **Seção 3.14.1**) **e processamento de arquivos** (v. **Capítulo 11**)
`<stdlib.h>`	❒ Funções **srand()** e **rand()** usadas em geração de números aleatórios (v. **Seção 4.10**) ❒ Alocação dinâmica de memória (**Capítulo 12**)
`<time.h>`	**Função time() usada com srand() em geração de números aleatórios** (v. **Seção 4.10**).
`<string.h>`	**Processamento de strings** (v. **Capítulo 8**)
`<ctype.h>`	**Classificação e transformação de caracteres** (v. **Capítulo 8**)
`<math.h>`	**Operações matemáticas com números reais** [p. ex., **sqrt()**, **pow()**]

Tabela 3–12: Principais Cabeçalhos da Biblioteca Padrão de C

3.13.2 A Biblioteca LeituraFacil

A biblioteca LeituraFacil foi desenvolvida com o objetivo de facilitar o aprendizado de programação removendo parte da complexidade inerente à linguagem C com respeito a leitura de dados via teclado. O processo de instalação dessa biblioteca foi descrito no **Capítulo 1**. Na **Seção 3.14.2**, será mostrado como utilizar alguns de seus componentes.

3.13.3 Outras Bibliotecas

Um compilador pode vir acompanhado de outras bibliotecas além da biblioteca padrão de C. Essas bibliotecas facilitam ainda mais a criação de programas, mas têm como grande desvantagem o fato de não serem portáveis. A falta de portabilidade aflige particularmente bibliotecas dedicadas à construção de interfaces gráficas

aderentes a uma plataforma. Por exemplo, uma biblioteca específica para construção de interfaces gráficas para sistemas da família Windows não tem nenhuma utilidade para construção de interfaces gráficas (p. ex., KDE, Gnome) para o sistema Linux.

3.14 Entrada via Teclado e Saída via Tela

As funções responsáveis pelas operações de entrada e saída em C fazem parte de um módulo da biblioteca padrão de C denominado *stdio*[1]. Para utilizar essas funções deve-se incluir o cabeçalho `stdio.h`, que contém informações sobre elas, por meio da diretiva:

```
#include <stdio.h>
```

Quando encontra uma diretiva **#include**, o pré-processador de C a substitui pelo conteúdo do arquivo que se deseja incluir. Por exemplo, quando a diretiva acima é processada, o conteúdo do arquivo `stdio.h` é inserido no local do programa onde ela se encontra. Assim, quando encontrar uma chamada de uma função referenciada no arquivo `stdio.h`, o compilador será capaz de verificar se a chamada é correta. Se você não incluir o respectivo arquivo de cabeçalho para uma função de biblioteca que deseja utilizar em seu programa, o nome dela será considerado pelo compilador como, por exemplo, identificador não declarado, visto que nem ele faz parte da linguagem C em si nem você o declarou explicitamente. (Chamar uma função significa fazer com que ela seja executada, como será visto no Capítulo 5. Cada item entre parênteses numa chamada de função é denominado parâmetro. Parâmetros são separados por vírgulas.)

3.14.1 Escrita de Dados na Tela

A função **printf()** é equivalente em C à instrução escreva da linguagem algorítmica apresentada no Capítulo 2. Isto é, ela permite a escrita de constantes, variáveis ou expressões de quaisquer tipos na tela do computador. A função **printf()** também permite a escrita de strings. No entanto, diferentemente da instrução escreva da linguagem algorítmica, a função **printf()** requer que seja especificado o formato de cada item que ela escreve. Essa especificação é feita por meio de um string (tipicamente constante), denominado **string de formatação**, de modo que uma chamada da função **printf()** assume o seguinte formato:

printf(*string de formatação*, e_1, e_2, ..., e_n);

Nesse formato, cada e_i pode ser uma constante, variável, expressão ou um string. Para cada item e_i a ser escrito na tela, deve haver um **especificador de formato** no interior do string de formatação que indica como esse item deve ser escrito.

A Tabela 3–13 enumera os especificadores de formato mais comuns utilizados pela função **printf()**.

Especificador de formato	O item será apresentado como...
%c	Caractere
%s	Cadeia de caracteres (string)
%d ou **%i**	Inteiro em base decimal
%f	Número real em notação convencional
%e (**%E**) ou **%g** (**%G**)	Número real em notação científica

Tabela 3–13: Especificadores de Formato Comuns Utilizados por printf()

[1] Esse nome vem de *standard input/output*; i.e., entrada/saída padrão.

Além de especificadores de formato, um string de formatação pode ainda conter outros caracteres. Quando um caractere de um string de formatação não faz parte de um especificador de formato, ele é escrito exatamente como ele é. Por exemplo, ao final da execução do trecho de programa a seguir:

```
int   n = 3;
printf("O quadrado de %d e' %d.", n, n*n);
```

será escrito o seguinte na tela:

```
O quadrado de 3 e' 9.
```

Na chamada de **printf()** do último exemplo, o string de formatação contém dois especificadores de formato do mesmo tipo: **%d**. Eles indicam que há dois parâmetros além do string de formatação que devem ser escritos em formato de inteiro na base decimal. Esses parâmetros são a variável **n** e a expressão **n*n**.

Especificadores de formato podem ser ainda mais específicos ao indicar como os parâmetros finais de **printf()** devem ser escritos. Por exemplo, pode-se indicar que um número real deve ser exibido com um mínimo de cinco casas, sendo duas delas decimais, por meio do especificador **%5.2f**. O seguinte programa e o respectivo resultado apresentado na tela demonstram os efeitos de alguns especificadores de formato.

```
#include <stdio.h>

int main(void)
{
   int    i1 = 40, i2 = 123;
   double x1 = 123.45, x2 = 123.456789;

   printf("\n\t>>> Especificador %%d: |%d|", i1);
   printf("\n\t>>> Especificador %%5d: |%5d|", i1);
   printf("\n\t>>> Especificador %%-5d: |%-5d|", i1);
   printf("\n\t>>> Especificador %%5.3d: |%5.3d|\n", i1);

   printf("\n\t>>> Especificador %%10.3f: |%10.3f|", x1);
   printf("\n\t>>> Especificador %%10.3e: |%10.3e|", x1);
   printf("\n\t>>> Especificador %%-10g: |%-10g|\n", x1);

   printf("\n\t>>> Formato padrao int (%%d): %d\n", i2);
   printf( "\t>>> Com especificador de precisao (%%2.8d): %2.8d\n", i2 );

   printf("\n\t>>> Formato padrao double (%%f): %f\n", x2);
   printf( "\t>>> Com especificador de precisao (%%-10.2f): %-10.2f\n", x2 );
  return 0;
}
```

Esse programa apresenta o seguinte resultado:

```
        >>> Especificador %d: |40|
        >>> Especificador %5d: |   40|
        >>> Especificador %-5d: |40   |
        >>> Especificador %5.3d: |  040|

        >>> Especificador %10.3f: |   123.450|
        >>> Especificador %10.3e: |1.235e+002|
        >>> Especificador %-10g: |123.45    |

        >>> Formato padrao int (%d): 123
        >>> Com especificador de precisao (%2.8d): 00000123

        >>> Formato padrao double (%f): 123.456789
        >>> Com especificador de precisao (%-10.2f): 123.46
```

Um estudo mais aprofundado de formatação de saída está além do escopo deste livro, mas vários exemplos no decorrer do texto mostram formas alternativas de uso dos especificadores apresentados na Tabela 3–13 (v. exemplo apresentado na Seção 3.19.4). Entretanto, o aprendiz não deve dar muita importância a especificadores de formato. É verdade que seus programas poderão apresentar saídas de dados mais elegantes se você conhecê-los bem. Mas, por outro lado, você não será considerado um bom programador apenas por conhecer uma grande quantidade de especificadores de formato.

Outra função de saída da biblioteca padrão de C, que é menos frequentemente usada do que **printf()**, é a função **putchar()**. Essa função recebe como entrada um valor do tipo **int** e escreve na tela o caractere correspondente a esse valor. Por exemplo, a instrução:

```
putchar('A');
```

resultaria na escrita de `A` na tela.

3.14.2 Leitura de Dados via Teclado Usando LeituraFacil

Leitura de dados introduzidos via teclado é um dos aspectos mais difíceis de programação. Escrever um programa robusto capaz de responder a quaisquer caracteres digitados pelo usuário é uma tarefa que poucos programadores são capazes de realizar. Em particular, dominar completamente as funções de entrada de dados via teclado da biblioteca padrão de C é uma missão complicada para alunos de introdução à programação (v. Seção 11.9).

Para facilitar o aprendizado, este livro usa a biblioteca LeituraFacil desenvolvida com o objetivo de lidar com as complexidades inerentes às funções de entrada via teclado da biblioteca padrão de C, que serão apresentadas no Capítulo 11.

Para utilizar as funções da biblioteca LeituraFacil, você deverá tê-la instalado de acordo com as recomendações da Seção 1.6.4. Se você utilizar CodeBlocks, deverá também ter configurado esse IDE conforme preconizado na Seção 1.7.2. É necessário ainda incluir o arquivo de cabeçalho dessa biblioteca em cada programa que usa qualquer de suas funções por meio da diretiva:

```
#include "leitura.h"
```

As funções da biblioteca LeituraFacil são apresentadas na Tabela 3–14.

Função	O que faz	Exemplo
`LeCaractere()`	Lê um caractere	`int c;` `c = LeCaractere();`
`LeInteiro()`	Lê um número inteiro	`int i;` `i = LeInteiro();`
`LeReal()`	Lê um número real	`double d;` `d = LeReal();`
`LeString()`	Lê um string (v. Seção 9.5.1)	`char ar[5];` `LeString(ar, 5);`
`LeOpcao()`	Lê um caractere que satisfaz certo critério (v. Seção 5.8.2)	`int c;` `c = LeOpcao("ABCabc");`

Tabela 3–14: Funções da Biblioteca LeituraFacil

As três primeiras funções apresentadas na Tabela 3–14 são muito fáceis de usar, conforme mostram os exemplos. As demais funções serão discutidas à medida que seus usos se fizerem necessários.

Além de serem fáceis de usar, as funções da biblioteca LeituraFacil são capazes de tornar um programa robusto do ponto de vista de entrada de dados. Por exemplo, se um programa contém o seguinte fragmento:

```
#include <stdio.h>
#include "leitura.h"
...
int numero;

printf("Digite um numero inteiro: ");
numero = LeInteiro();
```

e, quando esse trecho de programa for executado, o usuário digitar, digamos, **abc**, o programa informará o usuário sobre o erro cometido e solicitará a introdução de um novo valor, como mostrado a seguir:

```
Digite um numero inteiro: abc
>>> O valor digitado e' invalido. Tente novamente
>
```

Obter os mesmos resultados apresentados pelas funções da biblioteca LeituraFacil usando as funções de leitura via teclado da biblioteca padrão de C não é tão trivial, conforme será visto no **Capítulo 11**.

3.14.3 Uso de Prompts

Se o programador deseja obter algum tipo de informação de um usuário de seu programa, o primeiro passo é descrever precisamente para o usuário que tipo de informação o programa espera obter dele. Assim, toda instrução de entrada de dados deve ser precedida por uma informação dirigida ao usuário sobre aquilo que o programa espera que ele introduza. Essa solicitação de dados apresentada ao usuário é denominada **prompt** e deve preceder *qualquer* instrução de entrada de dados via teclado. Normalmente, utiliza-se a função **printf()** para essa finalidade.

Para apresentar o prompt mais adequado ao usuário do programa, o programador deve, com a devida antecedência, conhecer o perfil de usuário de seu programa para ser capaz de comunicar-se na *língua* dele. Quer dizer, se o programa destina-se a usuários de certa área de conhecimento específica, não há problema em usar jargões dessa área, mas, se os usuários podem apresentar níveis diversos de conhecimento, jargões devem ser evitados.

Um engano frequente entre programadores iniciantes é assumir que os usuários de seus programas são versados em computação. Assim, um uso de jargão, que muitas vezes é imperceptível para o programador, diz respeito àqueles jargões usados em sua própria área de conhecimento. Tendo isso em mente, ele usa em prompts jargões típicos de profissionais dessa área, como, por exemplo: *digite um string*. Hoje em dia, *digitar* já se tornou uma expressão comum, mas, provavelmente, um usuário comum não faz a menor ideia do significado de string. Portanto revise seus prompts e substitua ou elimine qualquer jargão exclusivo de sua área.

3.15 Constantes Simbólicas

A linguagem C permite que se associe um identificador a um valor constante. Esse identificador é denominado **constante simbólica** e pode ser definido em C por meio de uma diretiva **#define**. Por exemplo, suponha que se deseje denominar `PI` o valor `3.14`, então a seguinte diretiva seria utilizada:

```
#define  PI  3.14
```

Quando o programa contendo essa definição é compilado, o compilador substitui todas as ocorrências de `PI` por seu valor (i.e., `3.14`).

O uso de constantes simbólicas em um programa tem dois objetivos principais:

[1] Tornar o programa mais legível. Por exemplo, `PI` é mais legível do que o valor `3.14`.

[2] **Tornar o programa mais fácil de ser modificado**. Por exemplo, suponha que o valor constante `3.14` aparece em vários pontos de seu programa. Se você desejasse modificar esse valor (digamos para `3.14159`) você teria de encontrar todos os valores antigos e substituí-los. Entretanto se esse valor fosse representado por uma constante simbólica definida no início do programa, você precisaria fazer apenas uma modificação na própria definição da constante.

O uso prático de constantes simbólicas será discutido em maiores detalhes na **Seção 6.5**.

A linguagem algorítmica apresentada no **Capítulo 2** pode ser expandida para permitir o uso de constantes simbólicas. Nesse caso, para declarar uma constante simbólica, precede-se seu nome com **constante** e termina-se a declaração com o símbolo = seguido do valor que a constante representa, como mostra o seguinte exemplo:

```
constante PI = 3.14
```

3.16 Como Construir um Programa 2: Implementação

A construção de um programa é um processo cíclico no sentido de que se pode retornar a uma etapa anterior quando a etapa corrente não apresenta um resultado satisfatório. A **Figura 3–1** ilustra graficamente o processo cíclico de construção de um programa de pequeno porte. Programas mais complexos do que aqueles discutidos neste texto requerem projetos bem mais sofisticados que constituem assunto de disciplinas da área de Engenharia de Software.

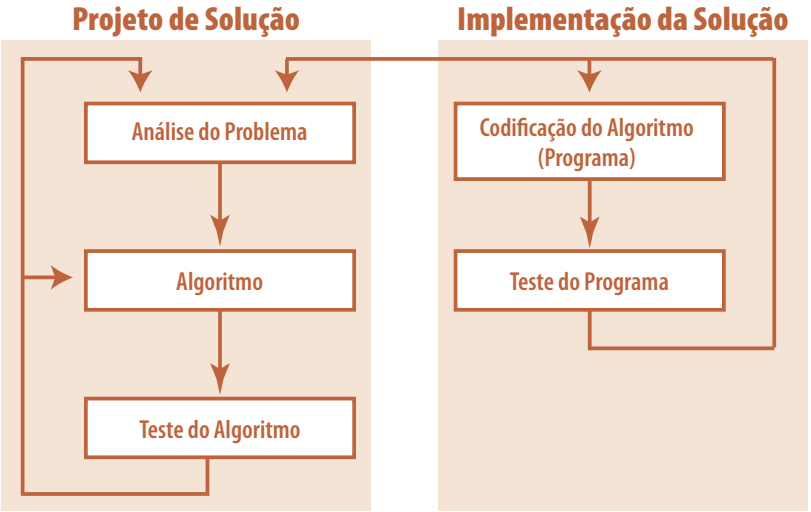

FIGURA 3–1: PROCESSO DE CRIAÇÃO DE UM PROGRAMA

A **Seção 2.9** descreveu em detalhes como deve ser o projeto de solução de um problema simples por meio de um programa. Em resumo, as etapas envolvidas num projeto dessa natureza são:

1. Análise do problema (v. **Seção 2.9.1**)
2. Refinamento do algoritmo (v. **Seção 2.9.2**)
3. Teste do algoritmo (v. **Seção 2.9.3**)

Nas seções a seguir, serão exploradas as etapas implicadas na implementação da solução do problema, que são:

4. Codificação do algoritmo (v. **Seção 3.16.1**)
5. Construção do programa executável (v. **Seção 3.16.2**)
6. Teste do programa (v. **Seção 3.16.3**)

Normalmente, a **Etapa 5** de construção de um programa executável é tão simples para programadores com alguma experiência (v. **Seção 3.16.2**) que não é considerada separada da etapa de codificação. Por isso, a **Etapa 5** não aparece na **Figura 3–1**.

3.16.1 Etapa 4: Codificação do Algoritmo

Após completa a etapa final de projeto, a etapa seguinte consiste em codificar o algoritmo de modo a obter um programa-fonte. Essa última etapa é resumida no seguinte quadro:

4 Escreva o programa-fonte usando um editor de programas.

4.1 Traduza cada instrução do algoritmo numa instrução equivalente seguindo as convenções da linguagem de programação escolhida.

4.2 Edite a tradução resultante do passo anterior. A execução desse passo resulta num programa-fonte.

Se a linguagem algorítmica utilizada na fase de projeto for próxima à linguagem de programação escolhida para codificação, a **Etapa 4.1** não deve oferecer nenhuma dificuldade. Além disso, as **Etapas 4.1** e **4.2** podem ser executadas como uma unidade. Isto é, com um pouco de experiência, pode-se traduzir um algoritmo e editar o programa traduzido num único passo, sem que se tenha que fazer uma tradução manuscrita. A **Seção 3.17** mostra como realizar a **Etapa 4** na prática.

3.16.2 Etapa 5: Construção do Programa Executável

O quadro a seguir resume o que deve ser realizado na Etapa 5:

5 Utilize um compilador e linker compatíveis com a linguagem de programação e o sistema operacional utilizados e construa um programa executável.

Conforme foi visto no **Capítulo 1**, o processo de compilação resulta num arquivo-objeto, mas esse arquivo não é necessariamente um programa executável. O arquivo-objeto resultante da compilação precisa ser ligado a outros arquivos-objeto que porventura contenham funções ou variáveis utilizadas pelo programa. Essa tarefa é realizada por meio de um linker, conforme foi mencionado na **Seção 1.2.2**. A **Figura 3–2** ilustra, de modo simplificado, o processo de transformação de um programa constituído de um único arquivo-fonte em programa executável.

Figura 3–2: Processo de Construção de um Programa Executável

A efetivação dessa etapa pode resumir-se a único clique de mouse ou pode levar algum tempo em virtude de necessárias correções de erros de compilação ou ligação. A **Seção 3.17** apresenta algumas opções disponíveis

para concretização dessa etapa e a **Seção 3.18** ensina como lidar com erros de compilação e ligação que são comuns nessa fase.

3.16.3 Etapa 6: Teste do Programa

O fato de não ter ocorrido erro de compilação ou ligação na **Etapa 5** não significa que o programa executável obtido naquela etapa esteja correto. É preciso testá-lo com casos de entrada qualitativamente diferentes (v. **Seção 2.9.3**). Portanto, na **Etapa 6**, o programa deve ser executado para verificar se ele funciona conforme o esperado (i.e., conforme foi especificado na definição do problema). Assim, deve-se examinar o programa sob várias circunstâncias (casos de entrada) e verificar se, em cada uma delas, o programa funciona adequadamente.

Normalmente, qualquer programa não trivial possui erros de programação. Assim o objetivo maior quando se testa um programa é encontrar erros que impeçam o seu funcionamento normal.

Técnicas utilizadas para testes de programas complexos constituem uma disciplina à parte e uma apresentação completa dessas técnicas está além do escopo deste livro. Logo apenas duas das técnicas mais comuns e fáceis de ser implementadas serão descritas.

A primeira técnica é conhecida como **inspeção de programa** e consiste em ler atenciosamente o programa e responder uma **lista de verificação** contendo questões referentes a erros comuns em programação na linguagem de codificação do programa.

A segunda técnica comum de verificação de programas é conhecida como **teste exaustivo** e consiste em utilizar dados típicos de entrada e simular manualmente a execução do programa, do mesmo modo que é simulada a execução de um algoritmo (v. **Seção 2.9.3**). Num teste exaustivo, é importante que sejam utilizados casos de entrada qualitativamente diferentes. Também, é igualmente importante que sejam testadas não apenas dados de entrada válidos, mas também algumas entradas inválidas.

Se o programa apresenta um comportamento considerado anormal, ele deve ser **depurado**. Isto é, as instruções responsáveis pelos erros devem ser encontradas e corrigidas. A **Seção 7.4** apresenta algumas técnicas elementares de depuração.

3.16.4 Saltando Etapas

É bastante tentador, mesmo para programadores mais experientes, apressar-se em iniciar a escrita de um programa no computador tão logo se tenha uma ideia daquilo que seria a solução para o problema em questão. Mas, do mesmo modo, também é comum gastar-se muito mais tempo removendo erros de um programa do que escrevendo-o.

Parece humanamente impossível construir-se um programa não trivial que não contenha erros (*bugs*), mas um programa escrito às pressas é muito mais susceptível a erros do que um programa que implementa a solução de um problema bem estudado previamente conforme foi preconizado na **Seção 2.9**. Portanto o conselho a ser seguido é:

Recomendação ***Resista à tentação de escrever um programa no computador tão logo você tenha uma vaga ideia de como deve ser a solução do problema que ele resolve.***

Enquanto não adquire experiência suficiente para saltar etapas, siga os procedimentos prescritos na **Seção 2.9** e no presente capítulo. Programadores experientes são capazes de escrever programas-fonte de relativa simplicidade diretamente num editor de programas e, com muito treinamento, você também será capaz de realizar isso. Mas, enquanto não atingir a plenitude de conhecimento em programação, resista a essa tentação.

Se você ainda não consegue saltar etapas do processo de desenvolvimento de um programa, não se sinta frustrado: siga todas as etapas prescritas, pois, com a prática adquirida, você conseguirá escrever cada vez mais um número maior de programas sem ter que escrever previamente seus algoritmos em pseudolinguagem. Nunca esqueça, entretanto, que não se devem escrever programas mais complexos sem planejá-los previamente (**Etapas 1–3** da **Seção 2.9**). O tempo gasto no planejamento de um programa é recuperado em sua depuração. Ou seja, frequentemente, a depuração de um programa mal planejado leva muito mais tempo do que a escrita do programa em si.

3.17 Programas Monoarquivos em C

Existem dois tipos de sistemas de execução de programas escritos em C:

[1] **Sistemas com hospedeiro**. Programas executados em sistemas com hospedeiro são, tipicamente, sujeitos à supervisão e ao suporte de um sistema operacional. Esses programas devem conter uma função **main()**, que é a primeira função chamada quando começa a execução do programa. Todas as características de C discutidas neste livro são voltadas para programas dessa natureza.

[2] **Sistemas livres**. Programas executados em sistemas livres não possuem hospedeiro ou sistema de arquivos e uma implementação de C para tais sistemas não precisa atender a todas as recomendações impostas para sistemas com hospedeiro. Num programa desse tipo, o nome e o tipo da primeira função chamada quando o programa é executado são determinados pela implementação. Este livro não lida com esse tipo de programa.

Este livro dedica-se a ensinar como construir programas simples executados em linha comando (console) de um sistema operacional. Esses programas são denominados **monoarquivos**, pois eles consistem de um único arquivo-fonte. A seguir, mostrar-se-á como editar um programa-fonte e construir um programa executável em C.

3.17.1 Estrutura de um Programa Simples em C

Um programa simples em C, consistindo de um único arquivo-fonte e que usa a biblioteca LeituraFacil, possui o formato apresentado esquematicamente na **Figura 3–3**.

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

/* Inclua aqui outros cabeçalhos padrão utilizados no programa */

#include "leitura.h"  /* Para usar LeituraFacil */

/* Inclua aqui definições de constantes */
/* simbólicas utilizadas no programa    */

int main(void)
{
    /* Inclua aqui definições de variáveis   */
    /* que serão necessárias no seu programa */

    /* Inclua aqui instruções que executem as ações    */
    /* necessárias para funcionamento do seu programa. */
    /* I.e., traduza em C cada passo de seu algoritmo. */

    return 0;  /* Informa o sistema operacional que */
               /* o programa terminou normalmente   */
}
```

Figura 3–3: Programa Monoarquivo Simples em C

A instrução:

```
return 0;
```

que aparece ao final da maioria dos programas escritos em C indica que o programa foi executado sem anormalidades. Em geral, o valor usado com **return** no interior de **main()** indica para o sistema operacional ou outro programa que cause a execução do programa em questão se a execução desse programa foi bem sucedida ou não. Assim, quando o programador prevê algum problema que impeça a execução normal do programa, ele usa um valor diferente de zero. Por exemplo, quando o IDE CodeBlocks invoca o compilador GCC para compilar um programa, ele é capaz de saber se a compilação foi bem sucedida ou não checando o valor retornado pelo GCC.

O uso de zero para representar sucesso na execução de um programa pode parecer um contrassenso. Afinal, quando uma ação é bem sucedida não se costuma lhe atribuir zero. Porém, suponha, por exemplo, que um programa para ser bem sucedido precisa processar um arquivo que deve ter um formato específico. Então, pelo menos, dois fatos podem impedir que o programa seja bem sucedido: (1) o arquivo não é encontrado e (2) o arquivo não tem o formato esperado. Quando um desses fatos ocorre, se o programa desejar indicar por que não foi bem sucedido, ele pode usar um valor diferente para cada categoria de erro. Portanto, se zero indicasse que o programa não foi bem sucedido, ele não poderia discriminar os dois tipos de erro. Por isso, convencionou-se que zero indicaria sucesso e valores diferentes de zero indicariam a ocorrência de alguma anormalidade.

3.17.2 Como Criar um Programa-fonte

Se você estiver usando um editor de texto comum (o que não é recomendado) ou um editor de programas, crie um programa-fonte do mesmo modo como você procede na criação de um arquivo de texto qualquer. Se você estiver usando CodeBlocks, siga o procedimento descrito na **Seção 1.7.3**.

3.17.3 Configurando o Editor do IDE CodeBlocks

Se você usar as recomendações de endentação apresentadas neste livro, provavelmente, seus programas serão legíveis e esteticamente agradáveis de ler. Infelizmente, editores de texto e de programas incorporam um vilão que pode destruir a estética de um programa. Esse vilão é a tabulação.

Se você usar tabulação (i.e., a tecla `[TAB]` na parte superior esquerda do teclado), sofrerá uma grande decepção quando seu programa-fonte for aberto num outro editor de texto e você verificar que parte do seu trabalho em prol do bom estilo de programação foi por água abaixo. E a decepção será ainda maior se você misturar tabulações com espaços em branco obtidos com a barra de espaços. Por exemplo, no trecho de programa a seguir, a definição de variável `int x` e a instrução `x = 10` foram endentadas de maneiras diferentes: a referida definição foi endentada usando quatro espaços (com a barra do teclado), enquanto a instrução foi endentada por meio da tecla `[TAB]` configurada com quatro espaços. Apesar de estas duas linhas aparecem perfeitamente alinhadas no editor de programas no qual o programa-fonte foi editado, aqui (e em outros editores de texto) elas aparecem desalinhadas.

```
int main(void)
{
    int x; /* Endentado com quatro espaços */
       x = 10; /* Tabulação de quatro espaços */
    return 0;
}
```

Bons editores de texto podem ser configurados para usar espaços em branco em vez de tabulações, mas, mesmo assim, são necessários alguns cuidados extras. Para evitar o uso de tabulações no editor do CodeBlocks, siga o seguinte procedimento:

1. Clique na opção *Editor...* do menu *Settings*.
2. Localize a seção denominada *TAB options*.
3. Desmarque a opção *Use TAB charactere*.
4. Marque a opção *TAB indents*.
5. Digite 3 ou 4 na caixa precedida por *TAB size in spaces*.

A **Figura 3–4** mostra o resultado final do último procedimento.

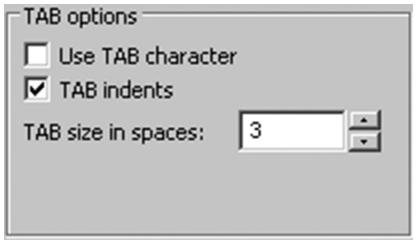

FIGURA 3–4: CONFIGURANDO OPÇÕES DE TABULAÇÃO EM CODEBLOCKS

Para certificar-se que tabulação realmente não é usada na edição de programas, configure as opções de endentação seguindo os seguintes passos:

1. Clique na opção *Editor...* do menu *Settings*.
2. Localize a seção denominada *Indent options*.
3. Marque a opção *Auto indent*. Usando essa opção, quando ocorre uma quebra de linha, a linha seguinte será alinhada com a linha precedente, a não ser que a opção *Smart indent* (a seguir) esteja selecionada.
4. Marque a opção *Smart indent*. Fazendo uso dessa opção o editor reconhece que algumas instruções precisam ser endentadas em relação a outras e efetua endentação levando isso em consideração. Por exemplo, se você não usar essa opção e quebrar a linha após o abre-chaves da função **main()**, a próxima linha será alinhada com o abre-chaves. Usando essa opção, a próxima linha será endentada em relação ao abre-chaves.
5. Se desejar, marque a opção *Brace Completion*. Empregando essa opção, todos os símbolos que têm abertura e fechamento aparecem automaticamente aos pares. Por exemplo, quando você digita abre-parênteses, o fecha-parênteses aparece automaticamente e o mesmo ocorre com chaves, apóstrofos, aspas etc. Essa opção pode ser incômoda para quem está acostumado com editores que não a possuem. De qualquer modo, selecione essa opção e experimente-a; se não a apreciar, desmarque-a novamente.
6. Se desejar, marque a opção *Backspace unindents*. Usando essa opção, quando você pressiona a tecla `[BACKSPACE]` antes do primeiro caractere de uma linha que não é espaço em branco, são apagados tantos espaços quanto forem os espaços de endentação. Se essa opção não estiver selecionada, apenas um espaço é excluído. Normalmente, essa opção é útil.
7. Se você enfrenta problemas com a visualização da estrutura de um programa, marque a opção *Show indentation guides*. Experimente-a e veja como seu programa aparece na janela de edição. Se gostar, permaneça com ela; caso contrário, desmarque-a.
8. Na caixa de seleção identificada por *Show spaces*, selecione (temporariamente) a opção *Always*. Com essa opção, o editor apresenta espaços em branco como pontos e as abomináveis tabulações como setas (→). Use essa opção temporariamente até certificar-se que o editor não está lhe traindo e detonando a formatação de seu programa.

Ao final da execução desse último procedimento, a seção de configuração do editor intitulada *Indent options* deverá apresentar-se como mostrado na **Figura 3–5**.

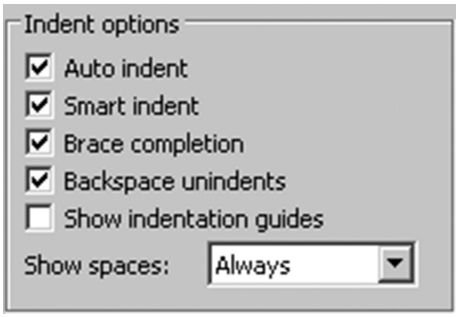

FIGURA 3–5: CONFIGURANDO OPÇÕES DE ENDENTAÇÃO EM CODEBLOCKS

3.17.4 Usando um Molde de Programa em CodeBlocks

Em breve, você irá notar que a maioria dos programas apresentados neste livro compartilham muitos trechos iguais ou semelhantes. Por exemplo:

- Todo programa precisa incluir um cabeçalho de apresentação, como aquele visto como exemplo na Seção 3.12.
- Todo programa hospedado escrito em C possui uma função **main()** que, na maioria das vezes, começa e termina do mesmo modo.
- Todo programa escreve algo na tela. Portanto é sempre necessário incluir o arquivo `stdio.h` que possibilita usar a função **printf()**.
- A maioria dos programas leem dados via teclado. Portanto, na maioria das vezes, é necessário incluir o arquivo `leitura.h`.
- Também, é uma boa ideia incluir um exemplo de execução do programa colocado entre comentários após o final do programa.

Não seria bom começar um programa com todos esses itens comuns já incluídos, em vez de começar com um arquivo vazio? O editor do IDE CodeBlocks permite que se especifique o conteúdo de cada arquivo criado. Por exemplo, suponha que você deseja que cada arquivo-fonte novo contenha o seguinte:

```
/****
 *
 * Título:
 *
 * Autor:
 *
 * Data de Criação: //2014
 * Última modificação: //2014
 *
 * Descrição:
 *
 * Entrada:
 *
 * Saída:
 *
 ****/

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

```
int main(void)
{
   return 0;
}
/******************* Exemplo de Execução ********************
************************************************************/
```

Para obter o resultado desejado, siga o seguinte procedimento:

1. No menu *Settings*, escolha a opção *Editor...*
2. No painel da esquerda da janela que surge em seguida, clique sobre o ícone denominado *Default code*.
3. Certifique-se que, na caixa de seleção intitulada *Type of file* no topo da janela, está selecionada a opção *C/C++ Source File*.
4. No painel da direita da referida janela digite o conteúdo com o qual você deseja iniciar cada arquivo criado pelo editor do CodeBlocks, como mostra a **Figura 3–6**.
5. Para concluir a operação, clique sobre o botão *OK*.

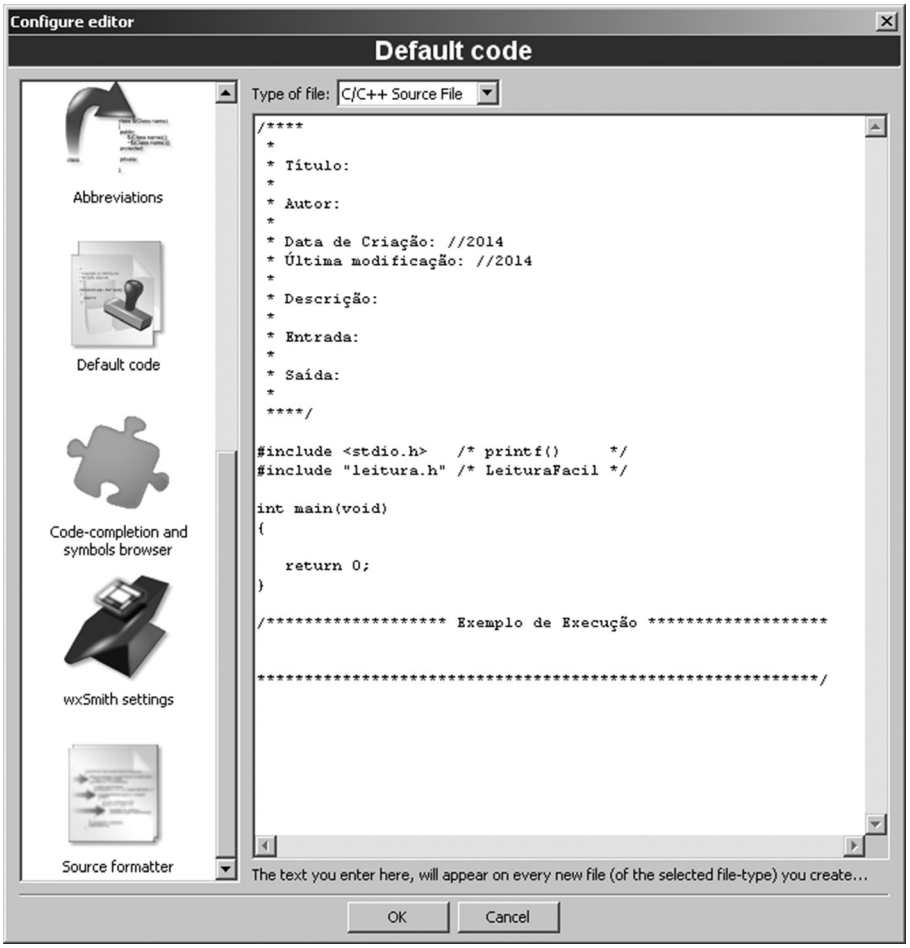

Figura 3–6: Default Code em CodeBlocks

Existe um atalho mais rápido e prático para a tarefa descrita acima, que consiste em copiar o conteúdo do arquivo `Molde.c`, que se encontra no site do livro (*www.ulysseso.com/ip*) e colá-lo no espaço mencionado. O conteúdo desse arquivo é o mesmo que aparece nesta seção.

Para experimentar o efeito do procedimento descrito, crie um arquivo-fonte conforme foi visto na Seção 1.7.3 e observe o resultado.

3.17.5 Criando um Programa Executável

Na Seção 1.7.4, mostrou-se como obter um programa executável a partir de um programa-fonte usando o IDE CodeBlocks. Essa é a opção mais rápida e fácil de se obter o resultado desejado.

Uma opção mais trabalhosa e que requer alguma intimidade com o sistema operacional utilizado é por meio da invocação do compilador e do linker via linha de comando. Nesse caso, em uma janela de terminal aberta no diretório onde se encontra o programa-fonte, deve-se emitir o seguinte comando:

```
gcc -Wall -std=c99 -pedantic arq-fonte -lleitura -o arq-executável
```

Emitindo esse comando, você invocará o compilador e o linker simultaneamente com as opções recomendadas para compilar e ligar o programa-fonte denominado **`arq-fonte`** e o resultado será um arquivo executável denominado **`arq-executável`**. Se, eventualmente, você digitar o comando esquematizado acima de modo incorreto e conhecer razoavelmente a interface utilizada pelo sistema operacional, saberá corrigir os erros sem ter que digitar todo o comando novamente. Caso contrário, você desperdiçará muito tempo com essa opção de compilação e é mais recomendado que você use o programa denominado **`compile`** que se encontra no site dedicado a este livro.

3.18 Lidando com Erros de Sintaxe e Advertências

3.18.1 É C ou C++?

Provavelmente, a primeira frustração com que se depara um iniciante em programação em C é decorrente do uso de compilador errado. Esse problema é oriundo do fato de a maioria dos compiladores de C também serem compiladores de C++ e essas linguagens, apesar de compartilharem diversas semelhanças, são diferentes e seus padrões seguem caminhos independentes. De fato, esses compiladores incorporam dois tradutores: um para C e outro para C++. Além disso, como padrão, esses compiladores assumem que, a priori, arquivos-fonte que representam programas em C usam a extensão **`.c`** e aqueles que representam programas em C++ possuem extensão **`.cpp`**. Portanto, para evitar problemas, não use esta última extensão em seus programas-fonte. Por exemplo, usando GCC, o seguinte programa é compilado normalmente se a extensão do arquivo-fonte for **`.c`**:

```
#include <stdio.h>

int main(void)
{
   const int  x = 5;
   int       *p = &x;

   *p = 10;

   return 0;
}
```

Entretanto, o mesmo programa não é compilado (em virtude de erro de sintaxe) se a extensão do arquivo-fonte for **`.cpp`**.

Mesmo quando um programa consegue ser compilado por compiladores de C e C++ providos por um mesmo fabricante, os resultados apresentados pelos programas executáveis obtidos podem ser diferentes. Por exemplo, quando o seguinte programa:

```
#include <stdio.h>
```

```
int main(void)
{
   printf("sizeof('A') = %d", sizeof('A'));
   return 0;
}
```

é compilado com GCC e, em seguida, executado, ele apresenta como resultado:

```
sizeof('A') = 4
```

quando a extensão do arquivo-fonte é `.c`[2]. Mas, se a extensão desse arquivo for trocada para `.cpp`, o resultado será:

```
sizeof('A') = 1
```

Concluindo, siga sempre o seguinte conselho quando compilar um programa em C:

Recomendação	***Certifique-se que está realmente usando um compilador de C quando compilar seus programas escritos nessa linguagem. Além disso, não use a extensão .cpp para nomes de programas-fonte escritos em C.***

3.18.2 Erros de Compilação

Sintaxe refere-se às regras de uma linguagem de programação que regem a construção de programas escritos nessa linguagem. Um **erro de compilação** (também denominado **erro de sintaxe**) ocorre em virtude de uma violação das regras de sintaxe da linguagem ora em uso e um compilador é capaz de traduzir um programa apenas quando ele está livre de qualquer erro de sintaxe. Isto é, um programa contendo erros de sintaxe não pode ser nem compilado nem executado.

Quando um compilador tenta traduzir um programa e, durante o processo, descobre algum erro de sintaxe, ele apresenta uma ou mais mensagens de diagnóstico que explicam a causa do erro (ou, pelo menos, tentam explicar...).

Qualquer programador, independentemente de sua experiência, comete erros de sintaxe. Programadores inexperientes provavelmente comentem erros dessa natureza por falta de familiaridade com a linguagem de programação utilizada. Por outro lado, programadores mais experientes os comentem em virtude da rapidez com que produzem código ou por mera distração. Nesse aspecto, uma diferença entre programadores experientes e iniciantes é que, para os primeiros, erros dessa natureza não causam preocupação e são corrigidos fácil e rapidamente. Para programadores iniciantes, corrigir erros de compilação pode demandar muito tempo e constituir uma experiência martirizante, principalmente por causa da falta de precisão com que compiladores apontam os erros e à linguagem enigmática que muitas vezes usam.

Erros de sintaxe podem ser provocados por deslizes de digitação decorrentes do pressionamento acidental de uma tecla em vez de outra. Exemplos de erros de digitação corriqueiros decorrentes da proximidade de teclas são descritos abaixo:

- Trocar ' por " ou vice-versa.
- Trocar as teclas {, [, },] entre si.
- Trocar as teclas ponto, vírgula e ponto e vírgula entre si.
- Uso das teclas de acento agudo (´) ou grave (`) em substituição a apóstrofos. Nem acento agudo nem acento grave são usados em programação (a não ser em comentário, o que é raro).

[2] O valor **4** obtido corresponde à largura do tipo **int** na implementação utilizada para testar o programa..

Outros erros de sintaxe frequentemente cometidos são enumerados a seguir:

- Omissão de parênteses depois de **main()**.
- Omissão de abre-chaves ou fecha-chaves.
- Digitação incorreta de uma palavra-chave (p. ex., *doble* em vez de **double**) ou um nome de função (p. ex., *print* em vez de *printf*). No caso de erro de digitação incorreta de palavra-chave, se você estiver usando um editor de programas com coloração de sintaxe, o erro pode ser percebido imediatamente, pois palavras-chaves são dispostas com uma formatação destacada (p. ex., em negrito).
- Não envolver strings constantes entre aspas. Novamente, se você estiver usando um editor com coloração de sintaxe, esse erro é facilmente visualizado porque strings constantes têm um colorido destacado.
- Omissão de ponto e vírgula ao final de uma instrução ou declaração.
- Inclusão de ponto e vírgula ao final de diretivas (i.e., linhas do programa que começam com #). Iniciantes em programação muitas vezes adotam duas abordagens extremas em relação ao uso de ponto e vírgula: ou esquecem com frequência de usá-lo ou o usam em locais em que não são necessários ou são indevidos.
- Esquecer-se de incluir o cabeçalho referente a uma função de biblioteca. Os cabeçalhos mais comumente usados aqui são aqueles apresentados na **Tabela 3–12** (v. **Seção 3.13.1**).
- Usar variáveis que não foram previamente declaradas.
- Chamar uma função com um número errado de parâmetros ou com parâmetros de tipos incompatíveis com a definição da função (v. **Seção 5.5**).
- Aplicação de incremento ou decremento sobre uma expressão. Por exemplo:

```
(x - 2)++ /* Não compila */
```

3.18.3 Mensagens de Erros

Mensagens de erro emitidas por um compilador indicam que seu programa está sintaticamente incorreto e, portanto, não pode ser compilado (e muito menos executado). Uma instrução ou declaração é considerada **ilegal** quando não consegue ser compilada devido a erro de sintaxe.

É importante observar que um único erro pode originar duas ou mais mensagens de erro. Portanto, enquanto você não adquire experiência suficiente para perceber isso, dê atenção a cada mensagem de erro individualmente e na ordem em que elas são apresentadas. Isto é, tente resolver uma mensagem de erro de cada vez e re-compile o programa depois de cada tentativa de correção.

Considere como exemplo o seguinte programa-fonte e suponha que o nome do arquivo que o contém seja `Erro1.c`.

```
#include <stdio.h>

int main(void)
{
   int  i = 10;

   printf( "i = %d\n", i)

   return 0;
}
```

Esse programa contém um erro de sintaxe proposital e, como o programa é diminuto, esse erro é facilmente enxergado por qualquer programador com alguma experiência em C. Mas, um iniciante pode ter alguma dificuldade para perceber que o erro é a falta ponto e vírgula ao final da instrução **printf()**. Se você tentar compilar

o programa acima usando GCC por meio de qualquer método descrito na Seção 3.17.5, receberá como resposta a seguinte mensagem de erro:

```
C:\Programas\Erro1.c In function 'main':
C:\Programas\Erro1.c 9 error: expected ';' before 'return'
```

Se você estiver usando CodeBlocks, essa mensagem será apresentada no painel inferior da janela com algum embelezamento para facilitar o entendimento, mas, em essência, o conteúdo das mensagens de erro obtidas com o compilador GCC executado em linha de comando ou por intermédio de CodeBlocks é o mesmo.

Mensagens de erro apresentadas pelo GCC começam com o nome completo do arquivo (i.e., incluindo seu caminho desde a raiz do sistema de arquivos até o diretório onde ele se encontra) seguido do nome da função na qual a instrução que originou a mensagem de erro se encontra. Por exemplo, a linha:

```
C:\Programas\Erro1.c In function 'main':
```

indica que ocorreu erro na função **main()** do arquivo `Erro1.c` encontrado no diretório (pasta) `C:\Programas`. As linhas seguintes descrevem os erros encontrados nessa função. Essas linhas começam (novamente) com o nome completo do arquivo seguido por *error*, dois pontos e o número da linha em que o erro foi detectado. No exemplo em questão, há apenas um erro e uma linha correspondente a esse erro:

```
C:\Programas\Erro1.c 9 error: expected ';' before 'return'
```

O que essa mensagem de erro quer dizer é que o compilador esperava encontrar um ponto e vírgula antes da instrução **return** que se encontra na linha 9. Talvez, o compilador não tenha indicado o erro no local que você esperava, que seria a linha 7, que contém a instrução **printf()** que não foi devidamente terminada com ponto e vírgula. Mas, embora pareça estranho à primeira vista, o compilador está absolutamente correto na indicação do erro e a raciocínio empregado por ele é o seguinte:

1. É necessário um ponto e vírgula para encerrar a instrução **printf()** antes do início da próxima instrução.
2. Como, nesse caso, o uso de espaços em branco não faz a menor diferença para o compilador, o ponto e vírgula ausente pode ser colocado em qualquer local entre **printf()** e **return**, e não necessariamente ao final da linha contendo **printf()**. Ou seja, embora, do ponto de vista de estilo, seja recomendado que o ponto e vírgula seja colocado realmente nessa posição, para o compilador, isso é irrelevante.
3. O compilador encontra a instrução **return** sem antes ter encontrado o referido ponto e vírgula. Assim, ele indica que houve erro na linha contendo essa última instrução. Portanto o compilador está absolutamente correto quando informa que esperava um ponto e vírgula antes dessa instrução. Para comprovar isso, coloque ponto e vírgula em qualquer local entre **printf()** e **return** e você verá que a mensagem de erro não mais aparecerá.

Diante do que foi exposto, pode-se enunciar a seguinte afirmação em relação a mensagens de erro emitidas por um compilador:

Recomendação	***Raramente, uma mensagem de erro indica a linha onde o programador esperava que o erro fosse indicado. Isto é, o erro pode ter ocorrido algumas linhas antes daquela à qual a mensagem de erro faz referência.***

Mas, se serve de consolo, a seguinte afirmação também é verdadeira:

Recomendação	***Um erro de sintaxe nunca ocorre numa linha posterior àquela referida numa mensagem de erro.***

Apesar de o compilador muitas vezes apresentar mensagens de erro vagas e difíceis de compreender, com a experiência adquirida após encontrar várias situações semelhantes, em pouco tempo, você será capaz de facilmente encontrar e corrigir erros apontados pelo compilador.

3.18.4 Mensagens de Advertência

Mensagens de advertência não impedem que um programa executável seja criado. No entanto, elas indicam situações que podem causar o mau funcionamento do programa. Portanto nunca ignore completamente uma mensagem de advertência. Em vez disso, leia e entenda por que cada mensagem de advertência foi gerada. Se, eventualmente, você decidir desprezar uma dada mensagem de advertência, convença-se de que isso não trará consequências danosas ao seu programa. A observância desse conselho pode lhe resguardar de muita dor de cabeça na depuração de seus programas.

Exemplos de Mensagens de Advertência

A seguir, serão apresentados alguns exemplos de mensagens de advertência com sugestões sobre como lidar com elas.

Exemplo 3.1 Considere o seguinte programa como exemplo e suponha que seu arquivo-fonte é denominado **`Advertencia1.c`**:

```
#include <stdio.h>

int main(void)
{
   int i;

   printf( "i = %d\n", i);

   return 0;
}
```

Esse último programa é absolutamente correto do ponto de vista sintático. Em outras palavras, ele é compilado sem problemas por qualquer compilador aderente a algum padrão ISO de C. Entretanto, um compilador decente que tenha sido invocado com a opção de apresentação do maior número possível de advertências (p. ex., **`-Wall`** no GCC) apresentará uma mensagem de advertência após compilar esse programa. Por exemplo, o compilador GCC emitirá a seguinte mensagem:

```
C:\Programas\Advertencia1.c: In function 'main':
C:\Programas\Advertencia1.c 7 warning: 'i' is used uninitialized in this
function [-Wuninitialized]
```

Compare essa mensagem de advertência com a mensagem de erro apresentada como exemplo na **Seção 3.18.3** e note que, em termos de formato e conteúdo, a diferença entre as duas mensagens é a presença da palavra *warning* nessa última mensagem. A última expressão entre colchetes que acompanha uma mensagem de advertência indica a razão pela qual a mensagem de advertência foi emitida. Nesse exemplo específico, a expressão: **`-Wuninitialized`** indica que a mensagem foi apresentada por causa do uso dessa opção de compilação que foi, implicitamente, incluída em virtude do uso da opção **`-Wall`**, que incorpora todas as opções de advertência.

Nessa situação, o compilador não detectou nenhum erro no seu programa, mas emitiu uma mensagem de advertência que prenuncia que algo grave pode acontecer quando o programa for executado. No caso específico do último exemplo, o compilador informa que o conteúdo da variável **`i`** é usado sem que lhe tenha sido atribuído nenhum valor. Esse tipo de mensagem está sempre associado a um problema que se manifestará quando o programa for executado. Para constatar o que foi afirmado, execute o programa e veja o que acontece.

Numa sessão de execução do programa em discussão o resultado obtido foi:

```
i = 2147332096
```

Mas, o resultado poderia ter sido qualquer outro, visto que, como a variável **i** não foi iniciada, o espaço em memória que ela representa é indefinido.

Exemplo 3.2 Nem toda mensagem de advertência antecipa um problema grave que possa ocorrer durante a execução de um programa, como foi o caso no exemplo precedente. Por exemplo, considere agora o seguinte arquivo-fonte cujo nome é **Advertencia2.c**.

```
#include <stdio.h>

int main(void)
{
   int i = 10, j;
   printf( "i = %d\n", i);
   return 0;
}
```

Esse último programa é semelhante ao anterior, mas a fonte da mensagem de advertência anterior foi corrigida com a iniciação da variável **i** e foi acrescentada mais uma variável denominada **j**. Quando esse último programa é compilado com o compilador GCC usando a opção **-Wall**, ele emite a seguinte mensagem de advertência:

```
C:\Programas\Advertencia2.c: In function 'main':
C:\Programas\Advertencia2.c 5 warning: unused variable 'j'
[-Wunused-variable]
```

Essa mensagem de advertência alerta o programador para o fato de a variável **j** ter sido definida, mas não ter sido usada. Diferentemente do problema indicado pela mensagem de advertência anterior, o problema indicado por meio da última mensagem de advertência não causa danos graves durante a execução do programa (mas, causa um pequeno desperdício de memória).

Exemplo 3.3 Algumas versões antigas do compilador GCC podem apresentar a seguinte mensagem de advertência:

```
no newline at end of file
```

Essa mensagem é rara e significa que o arquivo-fonte não termina com uma linha em branco. Você pode resolver essa peitica acrescentando uma linha em branco ao final do arquivo ou apenas ignorá-la.

Interpretando Mensagens de Advertência

Resumindo o que foi discutido nesta seção, uma mensagem de advertência pode alertar o programador em relação a:

- Um erro grave que ocorrerá quando o programa for executado (primeiro exemplo).
- Uma má utilização de recursos computacionais (segundo exemplo).
- Um fato absolutamente irrelevante (terceiro exemplo).
- Uma situação normal, mas que pode constituir um erro grave num contexto diferente. Por exemplo, o uso de um operador de atribuição num local onde, tipicamente, deveria ser utilizado um operador de igualdade (exemplos concretos dessa natureza serão apresentados no **Capítulo 9**).

Como se pode ver, há várias interpretações para mensagens de advertências, mas, independentemente desse fato, as recomendações a seguir devem ser seguidas rigorosamente:

Recomendações

- ***Use sempre a opção que faz com que o compilador emita o maior número possível de mensagens de advertência. No caso do compilador GCC, essa opção é –Wall.***
- ***Nunca deixe de dar a devida atenção a uma mensagem de advertência.***

3.18.5 Mensagens de Erro e Advertência Combinadas

Existem mais quatro observações importantes em relação a mensagens de erro e advertência:

Recomendações

- ***Um único erro de sintaxe pode originar mais de uma mensagem de erro.***
- ***Um erro de sintaxe pode originar, além de uma mensagem de erro, uma mensagem de advertência.***
- ***Uma instrução sintaticamente correta que pode originar uma mensagem de advertência nunca é capaz de dar origem a uma mensagem de erro.***
- ***O primeiro erro apontado pelo compilador é sempre um erro real, mas mensagens de erro subsequentes podem não corresponder a outros erros legítimos. Isto é, esses eventuais erros podem ser apontados em consequência da ocorrência do primeiro erro detectado pelo compilador.***

Portanto, baseado nessas premissas, quando deparado com uma combinação de mensagens de erro, o programador deve primeiro tentar corrigir a instrução que deu origem à primeira mensagem de erro. Então, em vez de tentar resolver as demais mensagens de erro, ele deve recompilar o programa para verificar se as demais mensagens de erro ainda persistem. Se ocorrerem combinações de mensagens de erro e advertência, deve-se empregar o mesmo raciocínio, sempre dando atenção a mensagens de erro antes das mensagens de advertência.

Considere o seguinte programa como mais um exemplo:

```
#include <stdio.h>

int main(void)
{
   int i = 10

   printf( "i = %d\n", i);

   return 0;
}
```

Esse programa contém apenas um erro de sintaxe, que é a ausência de ponto e vírgula ao final da definição da variável `i`. Quando se tenta compilar esse programa usando GCC com a opção `-Wall`, esse compilador apresenta as seguintes mensagens (o nome do programa-fonte é `Erro2.c`):

```
C:\Programas\Erro2.c: In function 'main':
C:\Programas\Erro2.c 7 error: expected ',' or ';' before 'printf'|
C:\Programas\Erro2.c 5 warning: unused variable 'i' [-Wunused-variable]
```

Assim, de acordo com o diagnóstico apresentado pelo compilador há uma mensagem de erro e outra de advertência. A mensagem de erro assinala corretamente o erro de sintaxe, mas a mensagem de advertência não faz jus ao programa, pois ela informa que a variável `i` foi definida, mas não foi usada. Porém, o que ocorreu na realidade foi que a instrução **printf**(), que usa a variável `i`, não foi compilada em virtude do erro de sintaxe. Se

for efetuada a correção da instrução que originou a mensagem de erro, quando o programa for recompilado, nenhuma das duas mensagens será emitida.

3.18.6 Erros de Ligação

Erros de ligação são frequentemente confundidos com erros de compilação. Mas, um **erro de ligação** é apontado pelo linker e não pelo compilador. Os erros de ligação mais comuns são causados pelas seguintes situações:

[1] O linker conseguiu efetuar todas as ligações necessárias, mas, em virtude de alguma restrição imposta pelo sistema operacional, não conseguiu criar o arquivo executável que deveria resultar do processo de ligação.

[2] Um programa usa uma função de biblioteca que o linker não consegue encontrar porque o programador não lhe informou onde encontrá-la.

Impossibilidade de Criação de Programa Executável

O primeiro tipo de erro de ligação mencionado é o mais comum e mais fácil de ser resolvido. Ele ocorre principalmente durante a fase de testes de um programa quando se realiza alguma alteração necessária no programa e tenta-se reconstruir o programa executável esquecendo-se que ele ainda está sendo executado. Nesse caso, o linker GCC emite a seguinte mensagem:

```
Permission denied
ld returned 1 exit status
```

Observe que o formato dessa mensagem de erro é bem diferente daquelas apresentadas quando o compilador encontra um erro de sintaxe (v. **Seção 3.18.3**). A primeira linha dessa mensagem:

```
Permission denied
```

informa o programador que o linker não obteve permissão para criar o arquivo executável, muito provavelmente porque um arquivo executável com o mesmo nome daquele que se tenta criar encontra-se correntemente em uso. A ocorrência desse tipo de erro é pouco provável quando se usa ligação via linha de comando, mas é muito comum quando se utiliza um IDE. Para simular esse tipo de erro, abra o arquivo-fonte a seguir usando CodeBlocks:

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

int main(void)
{
   int  i;

   printf("\nDigite um inteiro: ");
   i = LeInteiro();

   printf("\nNumero inteiro digitado: %d", i);

   return 0;
}
```

Então, construa um programa executável (v. **Seção 3.17.5**), abra uma janela de terminal no diretório onde se encontra o programa e comece a executá-lo. Suponha que, no instante em que o programa apresenta o prompt:

```
Digite um inteiro:
```

você decide que seria melhor que o prompt incluísse a palavra *numero* antes de *inteiro*. Então, você retorna ao editor de programas, altera a primeira instrução **printf()** que contém esse prompt, salva o arquivo e tenta

reconstruir o programa executável. Acontece que você esqueceu que a última versão do arquivo executável está em execução e, quando o linker tenta sobrescrevê-lo, o sistema operacional impede que isso aconteça. Enfim, você é presenteado com a referida mensagem de erro.

A segunda linha da aludida mensagem:

```
ld returned 1 exit status
```

pode ser um tanto redundante, mas é bastante instrutiva em virtude dos seguintes aspectos:

- Essa mensagem começa com *ld*, que é o nome oficial do linker GCC. Assim habitue-se ao fato de toda mensagem de erro contendo *ld* estar associada ao uso do linker e ser, assim, um erro de ligação, e não de compilação.
- O restante da linha informa que *ld* (i.e., o linker) retornou `1`. Conforme foi visto na **Seção 3.17.1**, um programa retorna um valor diferente de zero quando ocorre algum problema que impede seu funcionamento normal. Nesse caso específico, o funcionamento normal de um linker é produzir um programa executável.

Linker Não Encontra Biblioteca

Observação: Se você configurou seu ambiente de desenvolvimento conforme recomendado no **Capítulo 1** e usa as prescrições para obtenção de um arquivo executável apresentadas naquele e no presente capítulo, o tipo de erro descrito nesta seção provavelmente não ocorrerá. Portanto continue lendo esta seção apenas se estiver interessado em conhecer o funcionamento de um linker em maior profundidade.

Para ilustrar o segundo tipo de erro comum de ligação mencionado, considere o seguinte programa, cujo nome do arquivo-fonte é `ErroLigacao.c`:

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

int main(void)
{
   int  i;

   printf("\nDigite um numero inteiro: ");
   i = LeInteiro();

   printf("\nNumero inteiro digitado: %d", i);

   return 0;
}
```

Do ponto de vista sintático, esse programa está perfeitamente correto e você poderá comprovar essa afirmação compilando-o (literalmente) com a seguinte invocação do compilador GCC digitada na linha de comando do sistema operacional:

```
gcc -c ErroLigacao.c
```

Invocado com a opção `-c`, o compilador GCC apenas compila (literalmente) o programa-fonte, criando o respectivo arquivo-objeto, mas não invoca o linker para efetuar as devidas ligações e, assim, criar o arquivo executável correspondente. Como o programa não contém nenhum erro sintático, com a invocação acima, o compilador é bem sucedido em sua tarefa e cria o arquivo-objeto `ErroLigacao.o`.

Quando o compilador GCC é invocado com a opção `-o`, ele entende que a intenção do programador é criar um programa executável. Nesse caso, se o arquivo a ser processado for um arquivo-fonte, ele será compilado e, em seguida, terá suas ligações efetuadas. Por outro lado, se o arquivo a ser processado for um arquivo-objeto,

apenas as ligações necessárias serão efetuadas. Assim, quando o linker GCC é invocado com a opção **-o**, como a seguir, para efetuar as ligações do arquivo-objeto **ErroLigacao.o**:

```
gcc ErroLigacao.o -o ErroLigacao.exe
```

ele apresenta as seguintes mensagens de erro:

```
ErroLigacao.o:ErroLigacao.c:(.text+0x1b): undefined reference to 'LeInteiro'
collect2: ld returned 1 exit status
```

A primeira dessas mensagens informa que, no código objeto **ErroLigacao.o**, que teve origem no arquivo-fonte **ErroLigacao.c**, foi encontrada uma chamada da função **LeInteiro()**. Entretanto, o linker não conseguiu encontrar a biblioteca (arquivo-objeto) que contém tal função nem ele foi programado com essa informação. Quer dizer, o linker não precisa ser informado sobre o local onde se encontram os arquivos-objeto que fazem parte da biblioteca padrão, desde que tanto essa biblioteca quanto o compilador e o linker façam parte de uma mesma cadeia de ferramentas de desenvolvimento (v. **Seção 1.6.2**). Em qualquer outra circunstância, o linker precisa ser informado sobre o paradeiro das bibliotecas necessárias. A forma mais fácil de prover essa informação é por meio da opção **-l** seguida do nome do arquivo-objeto de biblioteca sem extensão, como no seguinte comando:

```
gcc ErroLigacao.c -lleitura -o ErroLigacao.exe
```

Entretanto, para esse último comando funcionar, as seguintes regras devem ser satisfeitas:

- O nome da biblioteca deve começar com **lib** e ter extensão **.a**, como, por exemplo, **libleitura.a**.
- O arquivo de biblioteca deve jazer num diretório onde o linker costuma encontrar arquivos de biblioteca (p. ex., o diretório **/usr/local/lib/** no Linux).

Existem outras maneiras de fazer com que um linker encontre uma biblioteca necessária para a construção de um programa executável. Mas, esse tópico está além do escopo deste livro.

3.18.7 Prática: Cometendo Erros Voluntariamente

Uma ótima maneira de aprender a lidar com erros de programação é cometê-los intencionalmente e, para cada erro, verificar como o compilador responde. Para tal aprendizado, use programas bem simples, como os exemplificados a seguir e acrescente erros um-a-um. Familiarizando-se desse modo com erros, você estará mais apto a lidar com eles na vida real; i.e., quando eles forem acidentais e não propositais.

Prática de Erros 1

Considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int x = 10, y = 2;

   printf("x/y = %d", x/y);
}
```

Esse programa é bastante simples e livre de erros de sintaxe. Assim, a prática consistirá em introduzir erros nesse programa e, para cada erro inserido, preencher as seguintes informações:

- **Erro**: [descreva aqui o erro introduzido voluntariamente no programa]
- **Mensagem**: [copie aqui a mensagem de erro ou advertência apresentada pelo compilador. Para melhorar o entendimento acrescente (**E**) se a mensagem for de erro e (**A**) se a mensagem for de advertência]

- **Consequência**: [apresente aqui qual é a consequência do erro. Evidentemente, se o erro introduzido no programa for de sintaxe, a consequência é que ele não compila. Se a mensagem emitida pelo compilador for de advertência execute o programa e verifique qual é o resultado.]

Por exemplo, usando o programa acima e o modelo de experimento proposto, pode-se inserir um ponto e vírgula indevido ao final da diretiva `#include` e preencher as respectivas informações como:

- **Erro**: Ponto e vírgula ao final da diretiva `#include`.
- **Mensagem**: (**A**):

```
C:\Programas\Erros1.c 1 warning: extra tokens at end of
#include directive [enabled by default]
```

- **Consequência**: nenhuma

Depois de satisfeitos os propósitos da introdução de um erro e antes de passar para o erro seguinte, o programa deve ser corrigido de volta ao seu estado inicial.

Exercícios. Seguindo o modelo de atividade prática proposta, introduza os erros descritos a seguir e faça as devidas anotações.

1. Remova os parênteses da função **main**().
2. Remova o ponto e vírgula que acompanha a definição de variáveis.
3. Substitua vírgula por ponto e vírgula na definição de variáveis.
4. Remova as aspas da direita do string de **printf**().
5. Remova as duas aspas do string de **printf**().
6. Substitua as aspas do string de **printf**() por apóstrofos.
7. Substitua **%d** por **%f** em **printf**().
8. Remova o abre-chaves na linha seguinte àquela contendo **main**().
9. Substitua { por } na linha seguinte àquela contendo **main**().
10. Remova o fecha-chaves ao final da função **main**().
11. Remova a linha contendo **return**.

Prática de Erros 2

Para uma segunda atividade prática, considere o seguinte programa:

```
#include <stdio.h>

#define PI 3.14

int main(void)
{
   double raio = 1.0, area;

      /* Calcula a área do círculo */
   area = 2*PI*raio;

   printf("Area = %f", area);

   return 0;
}
```

Exercício: Siga o modelo da atividade prática anterior e introduza nesse programa os erros descritos a seguir:

1. Remova o fechamento de comentário (i.e., `*/`).
2. Acrescente ponto e vírgula ao final da diretiva **#define**.

3. Coloque um fecha-parênteses adicional na chamada de **printf**().
4. Coloque um abre-parênteses adicional na chamada de **printf**().
5. Troque o especificador de formato **%f** por **%d** em **printf**().
6. Troque o especificador de formato **%f** por **%s** em **printf**().
7. Remova a iniciação da variável `raio`.

Desafio: O último programa, em seu estado original, não contém erro de sintaxe, visto que ele é compilado normalmente, nem erro de execução, pois ele é executado normalmente sem ser abortado. No entanto, esse programa contém um erro de lógica. Isto é, ele apresenta um resultado que não satisfaz o bom senso. Qual é esse erro?

3.19 Exemplos de Programação

3.19.1 Um Programa Robusto

Problema: Escreva um programa que lê um número inteiro, um número real e um caractere, nessa ordem, usando as funções da biblioteca LeituraFacil. Após cada leitura, o programa deve exibir o valor lido na tela.

Solução: O algoritmo aparece na Figura 3–7 e o programa bem logo em seguida.

Algoritmo LêEEscreveNúmeros

```
inteiro i, c
real f

escreva("Digite um numero inteiro:")
leia(i)

escreva("Numero inteiro digitado:", i)

escreva("Digite um numero real:")
leia(f)

escreva("Numero real digitado:", f)

escreva("Digite um caractere")
leia(c)

escreva("Caractere digitado:", c)
```

Figura 3–7: Algoritmo de Leitura e Escrita de Números

```
#include <stdio.h>    /* Entrada e saída */
#include "leitura.h"  /* LeituraFacil    */

/****
 * main(): Lê corretamente um número inteiro, um número real
 *         e um caractere e apresenta-os na tela.
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int    i, c;
   double f;
```

```
      /* Apresenta o programa */
   printf( "\n\t>>> Este programa mostra como deve ser um programa robusto.\n" );

   printf("\n\t>>> Digite um numero inteiro > ");
   i = LeInteiro();

   printf("\n>> Numero inteiro digitado: %d\n", i);

   printf("\n\t>>> Digite um numero real > ");
   f = LeReal();
   printf("\n>> Numero real digitado: %f\n", f);

   printf("\n\t>>> Digite um caractere > ");
   c = LeCaractere();

   printf("\n>> Caractere digitado: %c\n", c);

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

   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa mostra como deve ser um programa robusto.

        >>> Digite um numero inteiro > abc

        >>> O valor digitado e' invalido. Tente novamente
        > -12

>> Numero inteiro digitado: -12

        >>> Digite um numero real > x2.5
        >>> O valor digitado e' invalido. Tente novamente
        > 2.5x
        >>> 1 caractere foi descartado

>> Numero real digitado: 2.500000

        >>> Digite um caractere > xyz
        >>> 2 caracteres foram descartados

>> Caractere digitado: x

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

3.19.2 Um Programa Melindroso

Problema: Escreva um programa que lê um número inteiro, um número real e um caractere, nessa ordem, usando as funções do módulo stdio da biblioteca padrão de C. Após cada leitura, o programa deve apresentar o valor lido na tela.

Solução: A solução para esse problema segue o mesmo algoritmo do exemplo apresentado na Seção 3.19.1.

Programa

```
#include <stdio.h>  /* Entrada e saída */

/****
 *
 * main(): Tenta ler usando scanf() um número inteiro,
 *         um número real e um caractere e apresenta
 *         o resultado da leitura na tela.
 *
```

```
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int    i, c;
   double f;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa e' fragil como alfeni. Trate-o com carinho.\n" );

   printf("\n\t>>> Digite um numero inteiro > ");
   scanf("%d", &i);

   printf("\n>> Numero inteiro digitado: %d\n", i);

   printf("\n\t>>> Digite um numero real > ");
   scanf("%lf", &f);

   printf("\n>> Numero real digitado: %f\n", f);

   printf("\n\t>>> Digite um caractere > ");
   c = getchar();

   printf("\n>> Caractere digitado: %c\n", c);

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

   return 0;
}
```

Exemplo de execução do programa 1 [*o usuário é mal comportado ou deficiente cognitivo*]:

```
        >>> Este programa e' fragil como alfeni. Trate-o com carinho.

        >>> Digite um numero inteiro > abc

>> Numero inteiro digitado: 0

        >>> Digite um numero real >
>> Numero real digitado: 0.000000

        >>> Digite um caractere >
>> Caractere digitado: a

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

Exemplo de execução do programa 2 [*o usuário é bem comportado e esperto*]:

```
        >>> Este programa e' fragil como alfeni. Trate-o com carinho.

        >>> Digite um numero inteiro > -12

>> Numero inteiro digitado: -12

        >>> Digite um numero real > 2.54

>> Numero real digitado: 2.540000

        >>> Digite um caractere >
>> Caractere digitado:

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

Análise: Compare esse programa com aquele apresentado na Seção 3.19.1 e note que esse programa não funciona adequadamente nem quando o usuário é bem comportado. Você entenderá melhor os problemas do programa acima na Seção 11.9.

3.19.3 Separando um Inteiro em Centena, Dezena e Unidade

Problema: Escreva um programa que lê um número inteiro de três dígitos e separa-o em centena, dezena e unidade.

Solução: O algoritmo aparece na Figura 3–8 e o programa vem logo em seguida.

ALGORITMO SEPARAINTEIRO

```
inteiro n, resto

escreva("Digite um numero inteiro de três dígitos:")
leia(n)

escreva("Centena:", n/100)

resto ← n%100

escreva("Dezena:", resto/10)

escreva("Unidade:", resto%10)
```

FIGURA 3–8: ALGORITMO DE SEPARAÇÃO DE INTEIRO EM CENTENA, DEZENA E UNIDADE

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

/****
 * main(): Separa um número inteiro introduzido via
 *         teclado em centena, dezena e unidade
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int n, resto;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa le um numero inteiro de tres digitos e"
           "\n\t>>> separa-o em centena, dezena e unidade.\n" );

      /* Lê o número */
   printf("\nDigite um numero inteiro de tres digitos: ");
   n = LeInteiro();

   printf("\n\t>>> Centena: %d", n/100); /* Exibe a centena */

      /* Descarta a centena */
   resto = n%100;

      /* Exibe a dezena */
   printf("\n\t>>> Dezena: %d", resto/10);

      /* Exibe a unidade */
   printf("\n\t>>> Unidade: %d\n", resto%10);

   return 0;
}
```

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

Exemplo de execução do programa:

```
        >>> Este programa le um numero inteiro de tres digitos e
        >>> separa-o em centena, dezena e unidade.

Digite um numero inteiro de tres digitos: 543

        >>> Centena: 5
        >>> Dezena: 4
        >>> Unidade: 3
```

3.19.4 Alinhamento de Inteiros na Tela

Problema: Escreva um programa que mostra como alinhar inteiros à direita ou à esquerda na tela usando **printf()** com os especificadores de formato **%*n*d** e **%–*n*d**, sendo ***n*** um valor inteiro positivo.

Solução: A solução desse problema envolve apenas chamadas da função **printf()**. Assim o algoritmo a ser seguido é tão trivial que torna-se desnecessário apresentá-lo. O programa é apresentado a seguir.

```
#include <stdio.h> /* Entrada e saída */

/****
 * main(): Alinha inteiros à esquerda e à direita na tela
 *         usando os especificadores %nd e %-nd de printf()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int n1 = 1,
       n2 = 12,
       n3 = 123,
       n4 = 1234;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa alinha numeros inteiros na tela\n" );

      /* Alinhamento à direita */
   printf("\n%5d\n", n1);
   printf("%5d\n", n2);
   printf("%5d\n", n3);
   printf("%5d\n", n4);

      /* Alinhamento à esquerda */
   printf("\n%-5d\n", n1);
   printf("%-5d\n", n2);
   printf("%-5d\n", n3);
   printf("%-5d\n", n4);

      /* Alinhamento à direita e à esquerda */
   printf("\n%5d\t%-5d\n", n1, n1);
   printf("%5d\t%-5d\n", n2, n2);
   printf("%5d\t%-5d\n", n3, n3);
   printf("%5d\t%-5d\n", n4, n4);

   return 0;
}
```

Quando executado o programa apresenta o seguinte na tela:

```
        >>> Este programa alinha numeros inteiros na tela

    1
   12
  123
 1234

1
12
123
1234

    1   1
   12   12
  123   123
 1234   1234
```

Análise: O alinhamento de números apresentado pelo programa é obtido por meio do especificador **%*n*d**, sendo **n** um inteiro positivo ou negativo. Quando o valor de ***n*** é positivo, o alinhamento é à direita; quando **n** é negativo, o alinhamento dá-se à esquerda.

3.19.5 Multiplicando Duas Frações

Problema: Escreva um programa que multiplica duas frações. Cada fração é lida como dois números inteiros: o primeiro representa o numerador e o segundo o denominador da função. Então, o programa calcula o produto e apresenta o resultado.

Solução: O algoritmo aparece na Figura 3–9 e o programa bem logo em seguida.

Algoritmo MultiplicaFrações

```
inteiro n1, d1, n2, d2

escreva("Digite o numerador da primeira fração:")
leia(n1)

escreva("Digite o denominador da primeira fração:")
leia(d1)

escreva("Digite o numerador da segunda fração:")
leia(n2)

escreva("Digite o denominador da segunda fração:")
leia(d2)

escreva("Resultado:", n1, "/", d1, " * ", n2, "/", d2, "=",
     n1*n2, "/", d1*d2, "=", (n1*n2)/(d1*d2) );
```

Figura 3–9: Algoritmo de Multiplicação de Frações

```
/****
 * main(): Lê e multiplica duas frações
 *
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
```

```
int main(void)
{
   int n1, d1, /* Numerador e denominador da    */
       n2, d2; /* primeira e da segunda frações */

      /* Apresenta o programa */
   printf("\n\t>>> Este programa multiplica duas fracoes.\n");

   printf("\nDigite o numerador da primeira fracao: ");
   n1 = LeInteiro();

   printf("\nDigite o denominador da primeira fracao: ");
   d1 = LeInteiro();

   printf("\nDigite o numerador da segunda fracao: ");
   n2 = LeInteiro();

   printf("\nDigite o denominador da segunda fracao: ");
   d2 = LeInteiro();

   printf( "\nResultado: %d/%d * %d/%d = %d/%d = %3.2f\n",
           n1, d1,  n2, d2,  n1*n2, d1*d2,
           (double) (n1*n2)/(d1*d2) );

   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa multiplica duas fracoes

Digite o numerador da primeira fracao: 2

Digite o denominador da primeira fracao: 3

Digite o numerador da segunda fracao: 5

Digite o denominador da segunda fracao: 7

Resultado: 2/3 * 5/7 = 10/21 = 0.48
```

Análise: O último programa é bastante simples, mas merece um comentário importante. Se qualquer dos números que representam os denominadores das frações for zero, o programa será abortado. No próximo capítulo, você aprenderá como precaver-se contra esse percalço.

3.20 Exercícios de Revisão

A Linguagem C (Seção 3.1)

1. (a) O que é padronização de uma linguagem de programação? (b) Por que um padrão de linguagem de programação é desejável?
2. (a) O que é o padrão ISO da linguagem C? (b) Como é popularmente conhecido o padrão corrente da linguagem C?
3. O significa dizer que uma característica de C é *dependente de implementação*?
4. Uma questão de reconhecimento de valor: quem criou a linguagem C?

Identificadores (Seção 3.2)

5. Quais são as regras para formação de identificadores da linguagem C?
6. (a) O que são palavras-chave? (b) Uma variável pode ter o nome de uma palavra-chave?

7. (a) O que são palavras reservadas da linguagem C? (b) Uma variável usada num programa escrito em C pode ter o mesmo nome de uma palavra reservada?
8. Um programa em C pode ser escrito usando apenas letras maiúsculas?
9. Quais dos seguintes nomes não podem ser utilizados como identificadores em C? Explique por que.

   (a) var (c) int (d) $a e) a$
   (g) double (i) VOID (j) void (l) _10

Códigos de Caracteres (Seção 3.3)

10. Em linhas gerais, qual é o conteúdo do conjunto básico de caracteres da linguagem C?
11. (a) O que é um código de caracteres? (b) Apresente três exemplos de códigos de caracteres.
12. Como caracteres são representados na memória de um computador?
13. Seja `a` uma variável do tipo **char**. Suponha que o código de caracteres ASCII seja utilizado. Que valor inteiro (em base decimal) será armazenado em `a` após a execução de cada uma das seguintes instruções:

    (a) `a = 'G';` (b) `a = 9;` (c) `a = '9';` (d) `a = '1' + '9';`

    [**Sugestão**: Consulte uma tabela do código ASCII, que é fartamente encontrada na internet.]
14. Suponha que `c` seja uma variável do tipo **char**. As atribuições `c = 'A'` e `c = 65` são equivalentes?

Tipos de Dados Primitivos (Seção 3.4)

15. (a) O que é um tipo de dado? (b) O que é um tipo de dado primitivo de uma linguagem de programação?
16. Descreva os seguintes tipos primitivos da linguagem C:

    (a) **int**
    (b) **char**
    (c) **double**

Constantes (Seção 3.5)

17. Quais são os dois formatos de escrita de constantes reais?
18. (a) O que é uma sequência de escape? (b) Em que situações sequências de escape se fazem necessárias? (c) Apresente três exemplos de sequências de escape.
19. As sequências de escape `'\n'` e `'\t'` são as mais usadas em escrita na tela. Qual é o efeito de cada uma delas?
20. (a) O que é um string constante? (b) Qual é a diferença entre string constante e caractere constante?
21. (a) Quando um string constante é muito extenso, como ele pode ser separado em partes? (b) Qual é a vantagem que essa facilidade oferece para o programador?
22. Uma sequência de escape pode ser inserida num string constante?
23. (a) A expressão `'A' + 'B'` é legal? Explique. (b) Em caso afirmativo, essa expressão tem algum significado prático? (c) **Pegadinha**: qual é o resultado dessa expressão?
24. Outra pegadinha: qual é o resultado de `'Z' - 'A'`? [**Dica**: Há garantia de que todas as letras sejam contíguas num código de caracteres?]
25. (a) Qual é o resultado de `'9' - '0'` (não é pegadinha)? (b) Qual é o resultado de `'2' - '0'`? (c) Em geral, o que significa uma expressão do tipo `dígito - '0'`?
26. Como é classificada cada uma das seguintes constantes:

    (a) `10` (b) `3.14` (c) `1.6e10` (d) `'\n'` (e) `"A"`

Propriedades dos Operadores da Linguagem C (Seção 3.6)

27. (a) Dentre os operadores apresentados neste capítulo, qual deles tem a maior precedência? (b) Qual deles tem a menor precedência?

28. (a) O que é efeito colateral de um operador? (b) Quais são os operadores que possuem efeito colateral apresentados neste capítulo?

29. Dentre os operadores apresentados neste capítulo, quais deles possuem ordem de avaliação de operandos definida?

30. A ordem de avaliação de um operador pode ser alterada por meio de parênteses?

31. Suponha que `i` e `j` sejam variáveis do tipo **int**. Explique por que a instrução abaixo não é portável (apesar de ser sintaticamente legal em C):

```
j = (i + 1) * (i = 1);
```

32. Quais são as precedências relativas entre os operadores aritméticos, relacionais e lógicos? Em outras palavras, considerando cada categoria de operadores como uma unidade, qual dessas categorias tem maior, intermediária e menor precedência?

Operadores e Expressões (Seção 3.7)

33. (a) O que é um operador? (b) O que é um operando? (c) O que é uma expressão?

34. Calcule o resultado de cada uma das seguintes operações inteiras:

(a) `-20/3`
(b) `-20%3`
(c) `-20/-3`
(d) `-20%-3`
(e) `20/-3`
(f) `20%-3`

35. (a) O que é um operador relacional? (b) Quais são os operadores relacionais de C?

36. Em que diferem os operadores relacionais apresentados no **Capítulo 2** e os operadores relacionais de C?

37. Quais são os possíveis resultados de aplicação de um operador relacional?

38. Os operadores relacionais de C têm a mesma precedência?

39. Assuma a existência das seguintes definições de variáveis num programa em C:

```
int     m = 5, n = 4;
double  x = 2.5, y = 1.0;
```

Quais serão os valores das seguintes expressões?

(a) `m + n + x - y`
(b) `m + x - (n + y)`
(c) `x - y + m + y / n`

40. Aplique parênteses nas expressões abaixo que indiquem o modo como um programa executável traduzido por um compilador de C efetuaria cada operação:

(a) `a = b*c == 2`
(b) `a = b && x != y`
(c) `a = b = c + a`

41. (a) Quais são os operadores lógicos de C? (b) Quais são os possíveis resultados da aplicação de um operador lógico?

42. (a) Quais são as semelhanças entre os operadores lógicos da pseudolinguagem apresentada no **Capítulo 2** e os operadores lógicos de C? (b) Quais são as diferenças entre essas categorias de operadores?

43. (a) O que é curto-circuito de um operador? (b) Que operadores apresentam essa propriedade?

44. Sejam `x` e `y` duas variáveis do tipo **int** previamente definidas. As expressões (1) e (2) a seguir são equivalentes? Explique seu raciocínio.

(1) `x > 0 || ++y < 10`

(2) `++y < 10 || x > 0`

45. Sejam `x` e `y` duas variáveis do tipo **int** previamente definidas. Por que um programa contendo a expressão (1) a seguir pode ser abortado em decorrência de sua avaliação, mas o mesmo não ocorre se essa expressão for substituída pela expressão (2) abaixo?

(1) `y%x && x > 0`

(2) `x > 0 && y%x`

46. (a) Por que a expressão `1/3 + 1/3 + 1/3` não resulta em `1`? (b) Qual é o resultado dessa expressão?

47. Suponha que `x` seja uma variável do tipo **int**. Escreva expressões lógicas em C que denotem as seguintes situações:

(a) `x` é maior do que `2`, mas menor do que `10`

(b) `x` é divisível por `2` ou `3`

(c) `x` é par, mas é diferente de `6`

(d) `x` é igual a `2` ou `3` ou `5`

48. Suponha que `x` e `y` sejam variáveis do tipo **int**. (a) É possível determinar o resultado da expressão `(x = 0) && (y = 10)` sem saber quais são os valores correntes de `x` e `y`? (b) Se for possível calcular esse resultado, qual é ele?

49. Por que o programa a seguir escreve na tela: `O valor de 2/3 e' 0.000000`?

```
#include <stdio.h>

int main(void)
{
   double produto = 2/3;

   printf("O valor de 2/3 e' %f\n", produto);

   return 0;
}
```

Definições de Variáveis (Seção 3.8)

50. O que é uma definição de variável?

51. Que informação é provida para o compilador por uma definição de variável?

52. Por que é importante usar nomenclaturas diferentes para escrita de identificadores que pertencem a categorias diferentes?

Operador de Atribuição (Seção 3.9)

53. Em que diferem os operadores `=` e `==`?

54. Qual é o valor atribuído à variável `z` na instrução de atribuição abaixo?

```
int x = 2, y = 5, z = 0;

z = x*y == 10;
```

55. Suponha que `i` seja uma variável do tipo **int** e `d` seja uma variável do tipo **double**. Que valores serão atribuídos a `i` e `d` nas instruções a seguir:

(a) `d = i = 2.5;`

(b) `i = d = 2.5;`

Conversões de Tipos (Seção 3.10)

56. (a) O que é conversão implícita? (b) Por que às vezes ela é necessária?

57. Em que situações um compilador efetua conversões implícitas?

58. Cite alguns problemas decorrentes de conversões implícitas.

59. (a) Um computador é capaz de realizar a operação `2 + 2.5` (i.e., a soma de um número inteiro com um número real)? (b) É permitido a escrita dessa expressão num programa em C? (c) O resultado dessa operação é inteira ou real?

60. (a) Para que serve o operador **(double)** na instrução de atribuição abaixo? (b) Ele é estritamente necessário? (c) Existe alguma justificativa para seu uso?

```
double x;
x = (double)2;
```

61. Suponha que `x` seja uma variável do tipo **double**. Que valor é atribuído a `x` em cada uma das seguintes atribuições:

(a) `x = 5/2;`

(b) `x = (double)5/2;`

(c) `x = (double)5/(double)2;`

(d) `x = (double)(5/2);`

Incremento e Decremento (Seção 3.11)

62. (a) O que é incremento? (b) O que é decremento? (c) Quais são os operadores de incremento e decremento?

63. Explique a diferença entre os operadores prefixo e sufixo de incremento.

64. Dadas as seguintes iniciações:

```
int  j = 0, m = 1, n = -1;
```

Quais serão os resultados de avaliação das seguintes expressões em C?

(a) `m++ - --j`

(b) `m * m++` [O resultado dessa expressão é dependente do compilador; apresente os dois resultados possíveis.]

65. O que exibe na tela cada um dos seguintes trechos de programa?

(a)

```
int x = 1;
printf("x++ = %d", x++);
```

(b)

```
int x = 1;
printf("++x = %d", ++x);
```

66. Seja `x` uma variável do tipo **int**. O que há de errado com a expressão: `++(x + 1)`?

67. Considere o seguinte trecho de programa:

```
int  i, j = 4;

i = j * j++;
```

Mostre que, se a variável `j` for avaliada primeiro, a expressão `j * j++` resultará em `16` e, se a expressão `j++` for avaliada primeiro, o resultado será `20`.

68. Suponha que `soma` e `x` sejam variáveis do tipo **int** iniciadas com `0`. A instrução a seguir é portável? Explique.

```
soma = (x = 2) + (++x);
```

Comentários (Seção 3.12)

69. (a) Como comentários podem ser inseridos num programa? (b) Como um compilador lida com comentários encontrados num programa?

70. Por que comentários são necessários num programa?

71. Por que um programa sem comentários compromete seu entendimento?

72. Qual é o melhor momento para comentar um programa? Por quê?

73. Em que sentido comentários num programa escrito por um programador profissional não devem imitar comentários encontrados em livros de ensino de programação?

74. Por que comentários redundantes são quase tão prejudiciais a um programa quanto a ausência de comentários?

75. Que informações mínimas deve conter um comentário de bloco incluído no início de um programa?

Bibliotecas (Seção 3.13)

76. No contexto de programação, o que é uma biblioteca?

77. É possível programar utilizando uma linguagem de programação que não possui biblioteca?

78. Como a biblioteca de uma linguagem de programação auxilia o programador?

79. (a) O que é a biblioteca padrão de C? (b) Por que funções dessa biblioteca não são estritamente consideradas partes integrantes da linguagem C?

80. Por que se diz que a biblioteca padrão de C é um apêndice dessa linguagem?

81. O que é um arquivo de cabeçalho?

82. (a) Para que serve uma diretiva **#include**? (b) Como uma diretiva **#include** é processada?

83. Quando se faz necessário incluir cada um dos seguintes cabeçalhos?

(a) `stdio.h`

(b) `math.h`

(c) `string.h`

(d) `leitura.h`

84. Por que a inclusão do cabeçalho `stdio.h` da biblioteca padrão de C é escrita como:

```
#include <stdio.h>
```

enquanto a inclusão do cabeçalho `leitura.h` da biblioteca LeituraFacil é escrita como:

```
#include "leitura.h"
```

Entrada via Teclado e Saída via Tela (Seção 3.14)

85. Teste seu conhecimento sobre a língua portuguesa: qual é a diferença entre os significados de *interação* e *iteração*?

86. O que é um programa robusto do ponto de vista de entrada de dados?

87. Descreva o funcionamento da função **printf()**.

88. O que é um especificador de formato?

89. (a) O que é string de formatação? (b) O que pode conter um string de formatação da função **printf()**?

90. A que deve corresponder um especificador de formato encontrado num string de formatação numa chamada de **printf()**?

91. Suponha que você tenha num programa a seguinte iniciação de variável:

```
int c = 'A';
```

Usando a variável `c`, como você exibiria na tela: (a) o caractere `'A'` e (b) o inteiro associado a esse caractere no código de caracteres utilizado.

92. Descreva o efeito de cada um dos seguintes especificadores de formato de **printf()**:

(a) **%f**
(b) **%5.2f**
(c) **%5d**
(d) **%-5d**
(e) **%d**
(f) **%c**
(g) **%s**

93. Como você produziria na tela: (a) uma quebra de linha e (b) um espaço de tabulação?

94. Quando um número real é escrito na tela com o uso de **printf()** com o especificador **%f** quantas casas decimais aparecem?

95. Suponha que o conteúdo de uma variável `x` do tipo **double** seja `2.4569`. O que será apresentado na tela quando cada uma das seguintes chamadas de **printf()** for executada?

(a) `printf("%f", x);`
(b) `printf("%d", x);` [Pense bem antes de responder esse item]
(c) `printf("%3.2f", x);`
(d) `printf("%4.2f", x);`

96. O que apresenta na tela o seguinte programa?

```
#include <stdio.h>

int main(void)
{
   int i = 43;

   printf("%d\n", printf("%d", printf("%d",i)));

   return 0;
}
```

97. Por que o programa a seguir não exibe o resultado esperado?

```
#include <stdio.h>

int main(void)
{
   int resultado = 2 + 2;

   printf("O resultado e': %d\n");

   return 0;
}
```

98. O que programa a seguir exibe na tela?

```
#include <stdio.h>

int main(void)
{
   int i = 6, j;

   j = ++i < 7 && i++/6 || ++i <= 9;

   printf("\ni = %d, j = %d\n", i, j);

   return 0;
}
```

99. O que programa a seguir escreve na tela?

```
#include <stdio.h>

int main(void)
{
    printf("1/3 = %f\n", 1/3);
    return 0;
}
```

100. O que é prompt e qual é sua importância num programa interativo?

101. Critique os seguintes prompts apresentados para um usuário comum de um programa:

(a) *Digite um string*

(b) *Tecle ^C para encerrar o programa*

102. Por que a ausência de prompt ou a apresentação de um prompt impreciso pode ser responsável por erros num programa?

103. Por que o programa a seguir não exibe o resultado esperado?

```
#include <stdio.h>

int main(void)
{
   double resultado;

   resultado = 21.0 / 7.0;

   printf("Resultado da divisao: %d\n", resultado);

   return 0;
}
```

Constantes Simbólicas (Seção 3.15)

104. (a) O que é uma constante simbólica? (b) Como uma constante simbólica é definida?

105. (a) Por que não se deve terminar uma definição de constante simbólica com ponto e vírgula? (b) Por que um programador iniciante tem dificuldade em detectar esse tipo de erro? (c) Como o uso de convenções diferentes para escrita de identificadores que pertencem a categorias diferentes facilita a detecção desse tipo de erro?

106. Que vantagens são obtidas com o uso de constantes simbólicas num programa?

107. Como se declara uma constante simbólica em linguagem algorítmica?

Como Construir um Programa 2: Implementação (Seção 3.16)

108. Quais são os passos envolvidos na construção de um programa de pequeno porte numa linguagem algorítmica?

109. Qual é a diferença entre teste e depuração de um programa?

Programas Monoarquivos em C (Seção 3.17)

110. É verdade que todo programa em C deve necessariamente ter uma função denominada *main*?

111. (a) O que é um sistema de execução com hospedeiro? (b) O que é um sistema livre?

112. O que é um programa de console?

113. Como é o esboço de um programa monoarquivo simples em C?

114. (a) O que significa *main* num programa em C? (b) Por que todo programa com hospedeiro em C precisa incluir uma função **main()**? (c) Esse nome pode ser alterado?

115. Qual é o significado da instrução `return 0;` encontrada em programas escritos em C?

Lidando com Erros de Sintaxe e Advertências (Seção 3.18)

116. Qual é a diferença entre mensagem de erro e mensagem de advertência emitida por um compilador?

117. (a) O que é um erro sintático ou de compilação? (b) O que é um erro de execução? (c) O que é um erro lógico? (d) Qual deles é o mais fácil de ser consertado? (e) Qual deles é o mais difícil de ser corrigido? [Você aprenderá mais sobre esse tópico no **Capítulo 7**.]

118. Apresente cinco erros sintáticos comuns.

119. Por que não é recomendado ignorar uma mensagem de advertência?

120. O que há de errado com o seguinte programa?

```
#include <stdio.h>
#include "leitura.h"

int main(void)
{
   int x;

   printf("\nDigite um numero inteiro: ");
   x = LeInteiro();

   printf("\n100/%d = %d\n", x, 100/x);

   return 0;
}
```

3.21 Exercícios de Programação

3.21.1 Fácil

EP3.1 Escreva um programa em C que exibe o seguinte na tela:

```
Meu nome e' [apresente aqui seu nome]
Sou [aluno ou aluna] do curso [nome do curso]
Digite um caractere seguido de ENTER para encerrar o programa:
```

Esse programa deverá apresentar os conteúdos entre colchetes substituídos adequadamente e só deverá ter sua execução encerrada quando o usuário digitar algum caractere seguido de `[ENTER]`. [**Sugestão**: Use a função `LeCaractere()` para ler o caractere que corresponde à tecla digitada pelo usuário.]

EP3.2 Escreva um programa que leia valores inteiros representando numeradores e denominadores de duas funções, calcula a soma das duas frações e apresenta o resultado na tela. [**Sugestão**: v. exemplo apresentado na **Seção 3.19.5**.]

EP3.3 Escreva um programa que lê dois valores reais representando os lados de um retângulo numa dada unidade de comprimento e exibe na tela a área, o perímetro e a diagonal do retângulo. [**Sugestão**:

O perímetro é dado por $P = 2 \cdot (L_1 + L_2)$; a área é dada por $A = L_1 \cdot L_2$ e a diagonal é dada por $D = sqrt(L_1 \cdot L_1 + L_2 L_2)$, sendo que L_1 e L_2 são os lados do retângulo.]

EP3.4 Escreva um programa que lê três números inteiros digitados pelo usuário. Então, o programa apresenta na tela a soma, o produto e a média dos números introduzidos.

EP3.5 Escreva um programa que lê um número inteiro com quatro dígitos introduzido pelo usuário e apresenta na tela, em linhas separadas, a milhar, a centena, a dezena e a unidade do número. [**Sugestão**: v. exemplo apresentado na **Seção 3.19.3**.]

EP3.6 Escreva um programa que calcula o gasto mensal de gasolina do usuário e apresenta o resultado. A interação desse programa com o usuário é exemplificada a seguir:

```
Quantos quilometros por dia voce dirige? 35
Quantos dias por semana voce dirige? 6
Quantos quilometros por litro seu carro faz? 10
Qual e' o preco do litro da gasolina (R$)? 2.71

>>> Gasto mensal de gasolina: R$227.64
```

EP3.7 Escreva um programa que calcula o volume de um cone circular reto de acordo com a fórmula:

$$V = \frac{\pi \cdot r^2 \cdot h}{3}$$

Nessa fórmula, V é o volume, r é o raio e h é a altura do cone. A seguir, um exemplo de execução desse programa:

```
Este programa calcula o volume de um cone circular reto

Introduza o raio (cm): 2.5
Introduza a altura (cm): 9.2

>>> Volume do cone: 60,18 cm3
```

3.21.2 Moderado

EP3.8 O programa a seguir contém erros sintáticos e resulta em mensagens de advertência quando se tenta compilá-lo com o compilador GCC ou no ambiente CodeBlocks usando a opção `-Wall`:

```
#include <stdio.h>
#include "leitura.h"

int main(void)
{
    int x, y, z;  /* Declaração das variáveis inteiras x, y e z */

    x = 10;  /* Atribui 10 à variável x */
    Y = 0;   /* Atribui 0 à variável y  */

    z = LeInteiro();  /* Lê um valor para a variável z */
    PRINTF("%d", y);  /* Escreve o valor da variável y */

    return 0
}
```

Sua tarefa consistirá do seguinte:

(a) Usando o ambiente CodeBlocks, edite o programa acima *exatamente* como ele é apresentado aqui e salve o arquivo.

(b) Compile o programa conforme descrito na **Seção 1.7.4**. Você deverá obter como resultado no painel inferior do CodeBlocks três mensagens de erro e três mensagens de advertência.

(c) Descreva o significado e a causa de cada mensagem de erro e de advertência.

(d) Para cada mensagem de erro, descreva uma forma de contornar o respectivo problema.

(e) Para cada mensagem de advertência, descreva uma forma de evitar que o compilador continue a emitir a referida advertência.

(f) Implemente no programa as alterações sugeridas em (d) e (e) e certifique-se de que o programa obtido realmente é compilado sem nenhuma mensagem de erro ou advertência.