Capítulo 6

Estilo de Programação

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia no contexto de programação:
  - ❒ Endentação
  - ❒ Jargão
  - ❒ Notação camelo
  - ❒ Número mágico
  - ❒ Código espaguete
  - ❒ Comentário de bloco
  - ❒ Comentário de linha
  - ❒ Despedida graciosa
- Explicar por que são usadas convenções diferentes para escrita de identificadores que pertencem a categorias diferentes
- Descrever a convenção utilizadas para escrita de cada uma das seguintes categorias de identificadores:
  - ❒ Constantes simbólicas
  - ❒ Variáveis
  - ❒ Funções
  - ❒ Definições de tipos
- Pormenorizar o cuidado especial que se deve ter com o uso de expressões reais como expressões condicionais
- Descrever as recomendações de estilo que devem ser observadas quando se escrevem strings constantes
- Discernir quando uma constante é considerada um número mágico
- Estar convencido de que incluir comentários num programa é estritamente necessário
- Não inserir num programa comentários irrelevantes, redundantes ou demasiadamente didáticos
- Inserir espaços horizontais e verticais num programa para melhorar sua legibilidade
- Praticar princípios básicos de interação com o usuário

OBJETIVOS

STE CAPÍTULO preconiza várias práticas de estilo que, se seguidas, certamente melhorarão o código produzido por programadores iniciantes.

6.1 Expressões e Instruções

Escreva instruções e expressões que sejam razoavelmente curtas. Também, quando for inevitável escrever instruções que ocupam mais de uma linha, endente as linhas seguintes em relação a linha inicial da instrução. Por exemplo, escreva:

```
resultado = (minhaVariavel1 + minhaVariavel2)*minhaVariavel3 -
             minhaVariavel4 + minhaVariavel5;
```

em vez de:

```
resultado = (minhaVariavel1 + minhaVariavel2)*minhaVariavel3 -
minhaVariavel4 + minhaVariavel5;
```

Evite o uso de expressões condicionais muito longas e complexas. Uma forma de verificar se uma expressão condicional é muito complexa é utilizar o chamado **teste do telefone**: leia a expressão em voz alta; se você conseguir entender a expressão à medida que a lê, a expressão passa no teste; caso contrário, se você se deixar de acompanhá-la, é melhor dividir a expressão em subexpressões.

Tente escrever blocos de instruções que não sejam muito longos e que caibam inteiramente na tela, pois isso facilita a leitura deles. Além disso, evite o aninho de blocos em mais de dois níveis (v. **Seção 4.2**).

6.2 String Constantes

Evite escrever strings constantes que sejam muito longos numa mesma linha. Ou seja, utilize o fato de os compiladores concatenarem strings constantes separados por espaços em branco e divida-os em múltiplas linhas. Por exemplo:

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

6.3 Jargões

Um **jargão** é um padrão de codificação frequentemente encontrado em programas escritos em linguagens da família de C (C++, Java etc.). Por exemplo:

```
while (1) {
   ...
}
```

é um jargão usado com frequência na escrita de laços de repetição infinitos (v. **Seção 4.5.6**), enquanto:

```
while (5) {
   ...
}
```

não é um jargão, apesar de funcionar exatamente do mesmo modo que o laço anterior.

O uso de `i`, `j` e `k` como nomes de variáveis de contagem (v. **Seção 4.5.3**) também pode ser considerado um jargão que persiste há muito tempo em programação. Adotando essa convenção, tal variável não precisa ter um nome mais longo, como `contador`, por exemplo.

Este livro está repleto de jargões utilizados com frequência pela comunidade de programadores de C. Utilize-os também e você obterá passaporte para essa comunidade.

6.4 Estilos de Escrita de Identificadores

O uso consistente de **convenções** para a criação de identificadores das diversas categorias que compõem um programa facilita bastante sua legibilidade, pois permite discriminar visualmente aquilo que um identificador representa (i.e., se ele representa uma variável, função, constante simbólica etc.). As convenções utilizadas na escrita de categorias de identificadores são descritas oportunamente ao longo do texto à medida que essas categorias forem discutidas. Por exemplo, as convenções usadas para escrita de variáveis foram descritas na **Seção 3.8** e aquelas usadas com constantes simbólicas serão descritas na próxima seção.

Identificadores que exercem papéis importantes num programa devem ter nomes representativos em relação aos papéis exercidos por eles. Por exemplo, uma variável denominada `idadeDoAluno` é muito melhor do que uma variável denominada simploriamente `x`. Por outro lado, identificadores com importância menor (p. ex., variáveis de contagem), não precisam ter nomes representativos.

Finalmente, não utilize identificadores nem muito longos nem muito abreviados: encontre um meio-termo que seja sensato.

6.5 Constantes Simbólicas e Números Mágicos

Como norma geral de estilo com respeito a constantes simbólicas (v. **Seção 3.15**), o programador deve adotar a seguinte política:

Recomendação ***Sempre que se pode atribuir uma interpretação a um valor constante, ele deve ser associado a uma constante simbólica.***

Por exemplo, suponha que `2*3.14*r` seja uma expressão que calcula o perímetro de um círculo de raio `r`. Então, a constante `3.14` é naturalmente interpretada como π (i.e., a relação entre o perímetro e o diâmetro de uma circunferência). Portanto, nessa expressão, a constante `3.14` seria substituída por uma constante simbólica denominada `PI`, de modo que a instrução:

```
perimetro = 2*3.14*r;
```

deveria ser reescrita como:

```
perimetro = 2*PI*r;
```

tendo a constante `PI` definida no início do programa como:

```
#define PI 3.14
```

Note que o valor `3.14`, que tem uma interpretação, foi substituído pela constante simbólica `PI` na instrução, mas o valor `2` permaneceu, visto que ele tem significado próprio; i.e., ele apenas faz parte de uma fórmula. Nesse caso, seria um equívoco tentar dotar essa constante de um nome representativo, como, por exemplo:

```
#define DOIS 2 /* Algum dia DOIS terá outro valor? */
```

Para facilitar o reconhecimento visual, as recomendações para escrita de identificadores usados em constantes simbólicas devem ser diferentes daquelas utilizadas para variáveis (v. **Seção 3.8**). Usualmente, recomenda-se que nomes de constantes simbólicas sejam escritos de acordo com as seguintes normas:

- Utilizam-se apenas letras maiúsculas.
- Se o nome da constante for composto, utilizam-se subtraços para separar as partes (p. ex., `VALOR_DE_TESTE`).

Empregando as sugestões preconizadas para formação de nomes de variáveis e constantes simbólicas, um programador que leia a expressão:

```
2*PI*r
```

imediatamente, concluirá, com uma rápida inspeção visual, que `PI` é uma constante simbólica e `r` é uma variável.

Números mágicos são valores numéricos constantes desprovidos de significado próprio e que dificultam o entendimento de um programa. Como exemplos de números mágicos, considere as seguintes linhas de programa:

```
y = 2.54*x; /* 2.54 é um número mágico */
while (i < 10)  /* 10 é um número mágico */
char c = 65; /* 65 é um número mágico */
```

Em geral, qualquer valor numérico que faz parte de uma expressão ou declaração e ao qual se possa atribuir uma interpretação subjetiva é considerado um número mágico. Números mágicos obrigam o leitor de um programa a fazer inferências, muitas vezes imprecisas, para tentar desvendar o significado deles. Em nome da boa legibilidade, é sempre recomendado que valores numéricos sejam representados por constantes simbólicas ou constantes que façam parte de uma enumeração (v. **Seção 10.9**). Por exemplo, os números mágicos que aparecem no último exemplo poderiam ser representados por constantes simbólicas como:

```
#define CENTIMETROS_POR_POLEGADA 2.54
#define LIMITE_SUPERIOR 10
...
y = CENTIMETROS_POR_POLEGADA*x;
while (i < LIMITE_SUPERIOR)
char c = 'A';
```

Uma situação comum na qual muitos iniciantes em C teimam em utilizar números mágicos é no processamento de caracteres. Por exemplo, frequentemente, programadores inexperientes utilizam o seguinte fragmento de programa para processar todos os caracteres compreendidos entre `'A'` e `'Z'`:

```
char c;
...
for (c = 65; c <= 90; ++c)
   ...
```

Nesse mau exemplo, os números mágicos `65` e `90` não apenas tornam o programa difícil de entender, como também o tornam dependente de implementação. Ou seja, o programa não terá portabilidade, pois se está assumindo implicitamente que `'A'` é representado por `65` e `'Z'` é representado por `90`, o que não é o caso em qualquer código de caracteres (v. **Seção 3.3**).

6.6 Estruturas de Controle

Estruturas de controle são fontes potenciais de erros de programação mesmo para programadores experientes. Em especial, erros em laços de repetição e desvios condicionais são comuns. Portanto, para precaver-se contra o surgimento de erros em seus programas, o programador precisa adotar uma rígida disciplina na escrita de estruturas de controle. A seguir serão revistas algumas sugestões apresentadas para uso mais seguro dessas estruturas:

- Use endentação para indicar subordinação de uma instrução em relação a outra. Por exemplo, as chamadas de **printf()** e `LeInteiro()` no trecho de programa abaixo estão subordinadas à instrução **if**:

```
if (x <= 0) {
   printf("Digite um valor inteiro positivo: ");
   x = LeInteiro();
}
```

- Sempre envolva o corpo de uma estrutura de controle entre chaves, mesmo que ele seja composto de apenas uma instrução. Isso previne o isolamento acidental de uma instrução acrescentada posteriormente ao corpo da estrutura (v. **Seção 4.5.1**).
- Coloque o abre-chaves do corpo de uma estrutura de controle na linha inicial da estrutura de controle (i.e., na linha que contém **while**, **do**, **if**, **else** e **switch**). Isso previne que uma instrução vazia acidental encerre prematuramente a estrutura de controle (v. **Seção 4.5.1**). Isto é, o uso de abre-chaves na mesma linha inicial de uma estrutura de controle tem como vantagem reduzir o risco de se encerrar acidentalmente a estrutura de controle por meio de um ponto e vírgula colocado nessa posição (um erro muito frequente em programação em C). Por exemplo, é muito mais fácil cometer o erro:

  ```
  while (x); /* O ponto e vírgula encerra o laço while */
  {
       ...
  ```

  do que o erro:

  ```
  while (x);{ /* Cometer este erro é mais difícil */
       ...
  ```

  Além disso, o erro a seguir é inócuo:

  ```
  while (x) {; /* Este ponto e vírgula não causa dano */
       ...
  ```

- Uma instrução **if-else** aninhada na parte **else** de outra instrução **if-else** deve ser iniciada na mesma linha da referida parte **else**. Isso facilita a leitura das instruções **if-else** e resultam em economia de endentações (v. **Seção 4.6.1**).
- Não utilize expressões reais como expressões condicionais, exceto para testar se um valor real é positivo, negativo ou zero (v. **Seção 5.11.6**).
- Sempre que usar o operador relacional de igualdade para comparar uma variável com uma constante, use a constante do lado esquerdo da comparação. Assim, se você, acidentalmente, trocar igualdade por atribuição, o compilador indicará o erro (v. **Seção 4.6.1**).
- Evite o uso de expressões condicionais muito longas e complexas. Uma forma de se verificar se uma expressão condicional é muito complexa é utilizar o teste do telefone exposto na **Seção 6.1**.
- Sempre inclua a parte **default** numa instrução **switch**. É recomendado ainda que a parte **default** apareça por último numa instrução **switch-case**. Se você acha que essa parte nunca será atingida, coloque uma instrução **printf()** que apresente uma mensagem informando que o programa não esperava apresentá-la se estivesse correto (v. **Seção 4.6.3**).

Para endentação, use três (suficiente) ou quatro (máximo) espaços em branco. Espaços menores do que três podem não ser suficientemente legíveis, enquanto espaços maiores do que quatro farão com que as linhas de instrução atinjam rapidamente a largura da tela. Além das sugestões para endentação já discutidas, outras serão apresentadas ao longo do texto. Você não precisa seguir exatamente todas essas sugestões, mas, qualquer que seja sua escolha de endentação, seja consistente (i.e., use-a coerentemente em todos os seus programas).

No tocante a desvios incondicionais, nos finais das seções que introduzem as instruções **break**, **continue** e **goto**, foram apresentadas recomendações de uso parcimonioso dessas instruções. Alguns programadores advogam

que essas instruções devem terminantemente ser abolidas de qualquer programa. Mas, esse ponto de vista é antiquado e exacerbado, conforme se tentará mostrar na Seção 6.7.

6.7 Usar ou Não Usar goto, Eis a Questão

Linguagens de programação muito antigas (p. ex., Fortran) eram carentes de estruturas de controle e requeriam o uso extensivo de instruções goto para implementar as estruturas de controle ausentes nessas linguagens. Acontece, porém, que o uso indiscriminado de goto conduzia ao chamado **código espaguete**, que tem essa denominação porque, se forem traçadas linhas seguindo o fluxo de execução de um programa que usa goto abusivamente, elas tenderão a se cruzar diversas vezes como se formassem um prato de espaguete. Evidentemente, um programa com essa característica é difícil de ler ou modificar.

A aversão ao uso de goto demonstrada por muitos programadores é oriunda de um movimento denominado **programação estruturada**, que emergiu no início da década de 1960. Naquela época, o uso de laços de repetição era escasso e o uso de goto era profuso em virtude de deficiências encontradas nas linguagens de programação existentes. Mas, mesmo com o surgimento de linguagens ricas em estruturas de controle (p. ex., Algol), o uso de goto já estava disseminado entre programadores. Assim, adeptos de programação estruturada passaram a advogar, radicalmente, que, como qualquer programa poderia ser escrito sem o uso de goto numa linguagem rica em estruturas de controle (**linguagem estruturada**), o uso dessa instrução deveria ser terminantemente abolido. Entretanto, o fato é que, mesmo que o uso de **goto** não seja estritamente necessário, existem situações nas quais seu uso é naturalmente recomendado, pois, além de ser mais eficiente, não produz código espaguete. Isto é, abolir o uso de **goto** em certas situações requer variáveis e instruções extras e o resultado obtido não tem legibilidade sensivelmente melhor do que seria o caso se o uso de **goto** não fosse evitado. Por exemplo, no esboço de programa adiante, uma instrução **goto** é usada para encerrar dois laços de repetição a partir do laço mais interno:

```
while (1) {
   for (int i = 0;  i < n;  i++) {
      ...
      if (...)
         goto final; /* Encerra os dois laços */
      ...
   }
   ...
}

final:
   ;
```

Sem o uso de **goto**, seria necessário usar uma variável extra e algumas instruções adicionais, como mostra o esboço de programa equivalente:

```
int encerra = 0;

while (1) {
   for (int i = 0;  i < n;  i++) {
      ...
      if (...) {
         encerra = 1;
         break; /* Encerra o laço for */
      }
      ...
   }
```

CONTINUA

```
    if (encerra)
       break; /* Encerra o laço while */
    ...
}
```

Outra situação na qual o uso de **goto** é justificável é em tratamento de exceção, mas discutir esse tema está além do escopo deste livro.

Por causa da mencionada repulsa em relação a **goto**, algumas linguagens (p. ex., Java) até mesmo excluem essa estrutura de controle. Na realidade, raramente essa instrução precisa ser utilizada em linguagens estruturadas como C e não existe situação geral na qual o uso de **goto** seja indicado. Entretanto, como foi mostrado acima, existem algumas raras situações nas quais seu uso pode melhorar a eficiência e a legibilidade de um programa.

Hoje em dia, não há mais restrições impostas por linguagens de programação e programas que usam desvios condicionais judiciosamente não resultam em código espaguete. Note, contudo, que não se está preconizando aqui o uso indiscriminado de desvios incondicionais. Apenas tenta-se exorcizar um estigma.

6.8 Documentação de Funções

Documentar uma função significa prover comentários que tornem claros diversos aspectos da função.

Comentários podem ser divididos em duas categorias:

- **Comentário de Bloco**. Esse é um tipo de comentário destinado a documentar ou esclarecer um conjunto de linhas de uma função.
- **Comentário de Linha**. Esse é um tipo de comentário que se dedica a clarificar uma única linha de instrução ou o papel de uma variável ou qualquer outro componente de uma função. O fato de ele ser denominado comentário de linha não significa que ele deve ocupar exatamente uma única linha.

6.8.1 Comentários de Bloco

Toda função bem documentada deve ter logo acima de seu cabeçalho um comentário de bloco de **apresentação da função** contendo as seguintes informações sobre ela:

- O nome da função acompanhado de uma **descrição geral** e sucinta de seu propósito.
- **Descrição de cada parâmetro da função**, informando, inclusive, o modo de cada um deles. Não é necessário informar qual é o tipo de cada parâmetro porque essa informação está prontamente disponível no cabeçalho da função, que se encontra logo abaixo.
- **Descrição dos possíveis valores retornados pela função**. Novamente, não é preciso informar o tipo de cada valor retornado, porque ele aparece no cabeçalho da função.

Essas são as informações mínimas que devem constar num comentário de apresentação de uma função, mas outras informações relevantes podem constar num comentário dessa natureza:

- Se a função usa um algoritmo com certo grau de complexidade, deve-se incluir uma descrição sucinta do algoritmo ou uma referência bibliográfica que possibilite dirimir dúvidas a respeito do algoritmo.
- Quando houver alguma restrição de uso da função, devem-se descrever precisamente quais são os pressupostos assumidos pela função e como ela deve ser utilizada apropriadamente.
- Qualquer outra informação que o programador julgar pertinente.

Considere o seguinte exemplo esquemático de comentário de apresentação de uma função:

```
/****
 *
 * UmaFuncao(): [descreva o propósito geral da função]
 *
 * Parâmetros:
 *      x (entrada): [descrição do parâmetro x]
 *      y (saída): [descrição do parâmetro y]
 *      z (entrada/saída): [descrição do parâmetro z]
 *
 * Retorno: [os possíveis valores retornados pela função]
 *
 * Observações: [Pressuposições que a função faz sobre os parâmetros,
 *              limitações da função etc.]
 *
 ****/
```

Observe que os parâmetros da função são descritos separadamente e seus modos apresentados entre parênteses. Quando a função não possui parâmetros, escreve-se *Nenhum* após *Parâmetros:* no comentário de bloco.

Seguindo palavra *Retorno:* deve-se descrever aquilo que a função retorna (i.e., o *significado* do valor retornado). Se a função não retorna nada (i.e., se seu tipo de retorno é **void**), escreve-se *Nenhum* ou *Nada* após *Retorno:*. Um engano frequentemente cometido por iniciantes é descrever o *tipo de retorno* da função (ao invés do significado do valor retornado). Essa última informação não merece nenhum comentário, já que o tipo de retorno de uma função é evidente no cabeçalho da mesma.

Como exemplo mais concreto de uso de cabeçalho de apresentação de função considere:

```
/****
 *
 * DoubleEmString(): Converte um valor do tipo double em string
 * Parâmetros:
 *      numero (entrada) - número a ser convertido
 *      decimais (entrada) - número de casas decimais
 *
 * Retorno: Endereço do string que contém a conversão se esta for bem
 *          sucedida; NULL, se a conversão não for possível.
 *
 * Observações:
 *    1. Cada nova chamada bem sucedida desta função
 *       sobrescreve o conteúdo do array.
 *    2. Esta função não faz arredondamento para o número
 *       de casas decimais especificado. I.e., o valor da
 *       arte fracionária é simplesmente truncado.
 *    3. A técnica utilizada pode causar overflow de
 *       inteiro. Quando isso ocorre, a função retorna NULL.
 *
 ****/
```

Outro local onde comentários de bloco se fazem necessários é logo antes de trechos de programas difíceis de ser entendidos por conter truques, sutilezas etc. Um comentário desse gênero deve ser endentado em relação às instruções a que ele se refere para não prejudicar a visualização dessas instruções. Por exemplo:

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

6.8.2 Comentários de Linha

Comentários de linha são utilizados para comentar uma única linha de instrução ou declaração, mas, em si, eles podem ocupar mais de uma linha por razões de estética ou legibilidade. Como o propósito de tais comentários é explicar uma instrução ou declaração que não é clara para um programador de C, não se deve comentar aquilo que é óbvio para um programador versado nessa linguagem. Por exemplo, o seguinte comentário não deve ser incluído num programa, pois seu conteúdo é evidente para qualquer programador com experiência mínima em C:

```
x = ++y; /* x recebe o valor de y incrementado de 1 */
```

Comentários como o do último exemplo não são apenas irrelevantes ou redundantes: pior, eles tornam o programa mais difícil de ler pois desviam a atenção do leitor para informações inúteis. A respeito, muitos comentários apresentados neste livro introdutório podem parecer óbvios para um programador com alguma experiência em C, mas não são tão evidentes para um aprendiz de programação. Assim, um comentário como o desse exemplo é aceitável neste texto. O incorreto é imitá-lo na prática profissional de programação.

Algumas instruções ou declarações mais complexas ou confusas do que a anterior devem ser comentadas não apenas para facilitar a leitura como também para que o programador verifique se elas correspondem exatamente aquilo que os comentários informam. Por exemplo:

```
int nDiasAteAntesDoMes; /* Número de dias decorridos desde o dia de referência até */
                        /* o final do mês que antecede aquele cujo calendário será */
                        /* exibido                                                 */
```

Formatar comentários de modo que eles sejam legíveis e, ao mesmo tempo, não interrompam o fluxo de escrita do programa é muitas vezes difícil e requer alguma habilidade. Quando sobra pouco espaço à direita da instrução para a escrita de um comentário de linha, pode-se colocá-lo precedendo a instrução e sem espaço vertical entre ambos, como, por exemplo:

```
   /* Determina em que dia o mês começa */
diaInicial = InicioDoMes(nDiasAteAntesDoMes);
```

Use espaços horizontais para alinhar comentários. Assim, os comentários serão mais fáceis e menos cansativos de ler. Por exemplo:

```
            /* Este comentário é */
/* desagradável de ler */

      /* Mas este é um comentário */
      /* bem mais aprazível       */
```

Cada variável que tem papel significativo na função ou programa deve ter um comentário associado à sua definição que clarifica o papel desempenhado por ela:

```
double pesoDoAluno; /* O peso do aluno em quilogramas */
```

Variáveis que não sejam significativas *não precisam ser comentadas*. O comentário a seguir, por exemplo, é dispensável:

```
int i;  /* Variável usada como contador no laço for */
```

Um breve comentário de linha é útil para indicar a quem pertence um dado fecha-chaves em blocos longos ou aninhados. Esses comentários são dispensáveis quando um bloco é suficientemente curto. Por exemplo:

```
while (x){
     ... /* Um longo trecho de programa */
     if  (y > 0){
         ... /* Outro longo trecho de programa */
     } /* if */
     ... /* Outro longo trecho de programa */
}  /* while */
```

No exemplo acima, os comentários `/* if */` e `/* while */` que acompanham os fecha-chaves indicam que eles pertencem, respectivamente, às instruções **if** e **while**.

Existem outras situações nas quais comentários de linha são recomendados:

- Para chamar atenção que uma instrução é deliberadamente vazia (v. **Seção 4.3**).
- Idem para um corpo de função temporariamente vazio (v. **Seção 5.4.2**).
- Para explicar uma instrução **switch** na qual instruções pertencentes a mais de um **case** poderão ser executadas; i.e., na ausência deliberada de uma instrução **break**. Na prática, isso é raríssimo de acontecer e não há exemplo de tal instrução **switch-case** neste livro.
- Quando o bom senso indicar...

Quando alterar alguma instrução com um comentário associado, lembre-se de atualizar o respectivo comentário de modo a refletir a alteração da instrução. Um comentário que não corresponde àquilo que está sendo descrito é pior do que não haver nenhum comentário.

Finalmente, na dúvida entre o que é e o que não é óbvio para ser comentado, comente, pois, nesse caso, é melhor pecar por redundância do que por omissão. Mas, lembre-se que comentários não devem ser utilizados para compensar um programa mal escrito.

6.8.3 Modelos de Comentários

Esta seção apresenta alguns modelos de comentários de bloco e linha. Você pode usá-los como eles são ou se inspirar neles para criação de seus próprios modelos.

```
/*******************************************************
*******************************************************
***********  Este modelo de comentário pode  ***********
***********  ser usado para chamar atenção   ***********
********** sobre um detalhe muito importante ***********
*******************************************************
*******************************************************/
```

```
/*******************************************************
 *                                                     *
 * Um comentário longo contendo informações importantes *
 * pode ser inserido numa caixa como esta.             *
 *                                                     *
 *******************************************************/
```

```
/*---------------------------------------------*\
 * Este é um formato alternativo para inserção *
 * de comentário numa caixa.                   *
\*---------------------------------------------*/
```

```
/*
 * Um comentário longo que não é colocado
 * numa caixa pode ser escrito assim.
 */
```

```
/*
 * Um comentário sem caixa, mas capaz de chamar atenção
 * ^^ ^^^^^^^^^^ ^^^ ^^^^^^ ^^^ ^^^^^ ^^ ^^^^^^ ^^^^^^^
 *
 * Este tipo de comentário é simpático. O problema é que
 * se a configuração de teclado for ABNT, você terá que
 * digitar duas vezes cada caractere '^'.
 */
```

```
/*-------> Um comentário curto, mas importante <-------*/

/*>>>>>>>>> Outro comentário curto importante <<<<<<<<<*/

/* Comentário simples que explica a próxima instrução */
```

6.9 Uso de Espaços em Branco Verticais

O uso de espaços verticais em branco não apenas facilita a leitura de um programa como também a torna menos cansativa. A seguir, são apresentados alguns conselhos úteis sobre o uso de espaços verticais.

- Use espaços verticais para separar funções e blocos com alguma afinidade lógica.
- Use espaços verticais em branco para separar seções logicamente diferentes de uma função.
- Use espaços verticais em branco entre declarações ou entre definições e instruções.

Exemplos de bom uso de espaços verticais são abundantes nos exemplos de programação apresentados neste livro.

6.10 Interagindo com o Usuário

O estudo de boas práticas de interação com o usuário constitui em si uma disciplina à parte. Entretanto, é conveniente que se apresente para o programador iniciante, que talvez desconheça a disciplina de **interação humano-computador**, um conjunto mínimo de recomendações básicas que devem ser seguidas enquanto ele não aprofunda seu conhecimento sobre o assunto. Essas recomendações serão apresentadas a seguir[1].

- A maioria dos programas interativos deve iniciar sua execução apresentando ao usuário informações sobre o que o programa faz e, se não for óbvio, como o programa deve ser encerrado. Talvez, seja ainda conveniente incluir nessa apresentação informações adicionais sobre o programa, tais como seu autor, versão e qualquer outra informação pertinente para o usuário. Essa recomendação não se aplica a programas que recebem parâmetros via linha de comando (v. **Seção 9.6**).
- Toda entrada de dados deve ser precedida por um prompt (v. **Seção 3.14.3**) informando o usuário sobre o tipo e o formato dos dados de entrada que o programa espera que ele introduza.
- Se for o caso, o programa deve informar o usuário qual ação ele deve executar para introduzir certos dados se essa ação não for óbvia. Por exemplo, você não precisa solicitar ao usuário para *pressionar uma tecla* para introduzir um caractere, basta solicitá-lo que *digite um caractere*. Mas, você precisa informar o usuário como executar uma operação não usual, tal como *pressione simultaneamente as teclas* `ALT`, `SHIFT` *e* `A`, em vez de *digite* `ALT-SHIFT-A`.
- Toda saída de dados deve ser compreensível para o usuário. Lembre-se que o usuário não é necessariamente versado em computação ou programação. Portanto não utilize jargões de sua área de conhecimento.
- O programa deve informar o usuário como ele deve proceder para encerrar uma dada iteração (i.e., entradas repetidas de dados).

[1] Leve em consideração que muitos programas apresentados neste livro não são dirigidos para usuários comuns e não seguem todas essas recomendações.

- Se um determinado processamento for demorado, informe o usuário que o programa está executando uma tarefa. Não deixe o usuário sem saber o que está acontecendo (v. **Seção 4.11.7**).
- Não use sons de alerta (representados por `'\a'`) abusivamente. Uma situação na qual o uso de alerta é recomendado é quando o usuário não atende uma solicitação de dados conforme esperado pelo programa. Mas, mesmo nesse caso, não use mais de um caractere `'\a'` de cada vez para não aborrecer o usuário e aqueles que o cercam.
- Tenha sempre em mente que o programa deverá ser usado por *usuários comuns*. Portanto não faça suposições sobre o nível cognitivo dos usuários.
- Ao encerrar o programa, despeça-se graciosamente do usuário. Uma **despedida graciosa** não tem intenção e fazer o usuário rir ao encerramento do programa. Essa expressão é usada em contraposição a programas que terminam abruptamente, deixando o usuário sem saber o que aconteceu. Essa recomendação nem sempre se aplica a programas que recebem parâmetros via linha de comando (v. **Seção 9.6**).

6.11 Exercícios de Revisão

Expressões e Instruções (Seção 6.1)

1. (a) O que é endentação? (b) Por que algumas linhas de um programa são endentadas em relação a outras?
2. (a) Endentação é absolutamente necessária num programa em C? (b) Cite três situações nas quais endentações são usadas.
3. Por que se deve tomar cuidado especial com o uso de expressões reais como expressões condicionais?
4. O programa abaixo mostra por que não se devem comparar números reais por meio do operador relacional de igualdade. Qual é o resultado deste programa? Explique-o.

```
#include <stdio.h>

int main(void)
{
   double d = 0.0;
   int    i;

   for(i = 0; i < 10; i++) {
      d = d + 0.1;
   }

   printf("\nd = %f\n", d);
   printf( "\nd e' %s 1.0\n", d == 1.0 ? "igual a" : "diferente de" );

   return 0;
}
```

String Constantes (Seção 6.2)

5. Que recomendações de estilo devem ser observadas quando se escrevem strings constantes?

Jargões (Seção 6.3)

6. (a) O que é um jargão de uma linguagem de programação? (b) Que vantagem o programador obtém usando jargões?
7. Reescreva as seguintes construções de C utilizando formas mais convencionais (i.e., jargões):

(a)
```
int i = 0;

while (i <= 10) {
   printf("i = %d", i);
   ++i;
}
```

(b)
```
int i = 0;

while (i <= 100)
   printf("i = %d", i++);
```

(c)
```
for (int i = 0; i++ < 15; )
   printf("i = %d", i - 1);
```

(d)
```
while (-5) {
   ...
}
```

(e)
```
if (x < 0) {
    return 0;
} else
    if (y > 0 || x == 25) {
        return -1;
    } else
        if (z >= 0) {
            return -2;
        } else
            return -3;
```

(f)
```
for (int i = 0; i++ < 10; x = x + delta) {
   printf("\nDigite o valor de delta: ");
   delta = LeReal();
}
```

(g)
```
if (valor != OK)
    return valor;
return OK;
```

Estilos de Escrita de Identificadores (Seção 6.4)

8. O que é notação camelo?

9. Por que são usadas convenções diferentes para escrita de identificadores que pertencem a categorias diferentes?

10. Descreva a convenção utilizadas para escrita de cada uma das seguintes categorias de identificadores:

(a) Constantes simbólicas

(b) Variáveis

(c) Funções

(d) Definições de tipos

Constantes Simbólicas e Números Mágicos (Seção 6.5)

11. (a) O que é um número mágico? (b) Qual é a relação existente entre números mágicos e constantes simbólicas?

12. Quando uma constante numérica não é considerada um número mágico?

Estruturas de Controle (Seção 6.6)

13. Reescreva o trecho de programa a seguir utilizando um estilo de programação mais convencional:

```
while(valor != 0)
{
   valor = -1;

   printf("Introduza o proximo numero: ");
   valor = LeInteiro();

   if(valor == 0)
   {
      continue;
   }

  if(valor < 0)
   {
      printf("%d nao e' um valor valido.\n",valor);
      continue;
   }
   else
   {
      menor == 0 ? menor = valor : 0;
      n += 1;
      media += valor;
      maior < valor ? maior = valor : 0;
      valor < menor ? menor = valor : 0;
   }
}
```

Usar ou Não Usar goto, Eis a Questão (Seção 6.7)

14. Por que o uso de **goto** deve ser comedido?

15. (a) O que é código espaguete e por que ele deve ser evitado? (b) Que relação existe entre o uso de **goto** e código espaguete? (c) Quando o uso de **goto** não é danoso a um programa?

Documentação de Funções (Seção 6.8)

16. O que são comentários de bloco e como eles devem ser utilizados?

17. O que são comentários de linha e onde eles devem ser utilizados?

18. Que informações devem estar contidas no comentário de bloco que deve preceder cada definição de função?

19. Em que situações o uso de comentários de linha é recomendado?

20. Qual é a melhor ocasião para escreverem-se comentários num programa e por quê?

21. Por que o uso de comentários irrelevantes pode prejudicar a legibilidade de um programa?

22. Critique a seguinte reflexão: *um comentário ruim é pior do que nenhum comentário*.

Uso de Espaços em Branco Verticais (Seção 6.9)

23. Para que servem espaços verticais em branco num programa?

24. Em que situações o uso de espaços verticais em branco é recomendado?

Interagindo com o Usuário (Seção 6.10)

25. Qual é a diferença entre *interação* e *iteração*?

26. Quais são os princípios básicos de interação com o usuário apresentados neste capítulo?

27. Qual é o meio de entrada para programas baseados em console?

28. Por que é importante apresentar um prompt preciso antes de ler dados introduzidos por um usuário de um programa?

29. O que deve ser evitado em comunicação com o usuário?

30. O que é uma despedida graciosa de programa?