Capítulo 4

Fluxo de Execução

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia relacionada à linguagem C:
  - ❒ Instrução
  - ❒ Bloco de instruções
  - ❒ Instrução vazia
  - ❒ Fluxo de execução
  - ❒ Estrutura de controle
  - ❒ Laço de repetição
  - ❒ Laço de contagem
  - ❒ Terminal de instrução
  - ❒ Desvio condicional
  - ❒ Desvio incondicional
  - ❒ Expressão condicional
  - ❒ Condição de parada
  - ❒ Leis de De Morgan
  - ❒ Operador condicional
  - ❒ Operador vírgula
- Descrever e usar as seguintes palavras-chave da linguagem C:
  - ❒ **while** ❒ **for** ❒ **else** ❒ **case** ❒ **break** ❒ **goto**
  - ❒ **do** ❒ **if** ❒ **switch** ❒ **default** ❒ **continue**
- Identificar as construções que são consideradas instruções em C
- Descrever fluxo natural de execução de um programa e como ele pode ser alterado
- Dizer quando uma instrução constituída por uma expressão faz sentido prático
- Classificar estruturas de controle
- Determinar a condição de parada de um laço de repetição
- Implementar laço de contagem
- Esclarecer por que o uso de desvios incondicionais deve ser comedido
- Explicar como uma instrução for pode ser substituída por uma sequência de instruções equivalente
- Citar os operadores da linguagem C que possuem ordem de avaliação de operandos definida
- Descrever o funcionamento das funções **rand()**, **srand()** e **time()**

OBJETIVOS

4.1 Introdução

FLUXO DE EXECUÇÃO de um programa diz respeito à ordem e ao número de vezes com que instruções do programa são executadas. Um programa que segue seu **fluxo natural de execução** é executado sequencialmente da primeira à última instrução, sendo cada uma delas executada uma única vez.

O fluxo natural de execução de um programa pode ser alterado por meio de **estruturas de controle**, que são instruções capazes de alterar a sequência e a frequência com que outras instruções são executadas. Estruturas de controle que alteram a frequência (i.e., o número de vezes) com que outras instruções são executadas são denominadas **laços de repetição**, enquanto aquelas que alteram a sequência (i.e., a ordem) de execução de outras instruções são denominadas **desvios**. Esses desvios podem ainda ser classificados em **desvios condicionais** e **desvios incondicionais**.

O estudo das estruturas de controle da linguagem C é o tópico principal deste capítulo. Mas, antes de apresentar as estruturas de controle de C, serão apresentadas algumas considerações importantes sobre instruções dessa linguagem.

4.2 Sequências de Instruções

Em C, uma instrução pode consistir de qualquer uma das alternativas abaixo:

- Expressão
- Instrução **return** (que será estudada no Capítulo 5)
- Estrutura de controle
- Chamada de função[1] (que será estudada em detalhes no Capítulo 5)

Quando uma instrução consiste numa expressão, ela só fará sentido se a expressão contiver operadores com efeito colateral. Como exemplo, considere o seguinte programa:

```
#include <stdio.h>

int main(void)

{
   int x = 0, y = 0;

      /* A seguinte instrução é legal, mas não faz sentido */
   2*(x + 1);

      /* A seguinte instrução é legal e faz sentido */
   y = 2*(y + 1);

   printf("\nx = %d, y = %d\n", x, y);

   return 0;
}
```

Nesse programa, que é perfeitamente correto em C, a instrução constituída pela expressão:

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

não tem absolutamente nenhum efeito sobre o resultado do programa. Aliás, se você compilar esse programa no compilador GCC (ou outro bom compilador), ele apresentará uma mensagem de advertência alertando o programador sobre esse fato.

[1] Na realidade, chamadas de função são consideradas expressões. Mas, para manter a simplicidade deste texto introdutório, elas não serão consideradas como tais no presente contexto.

Por outro lado, se você substituir a expressão `2*(x + 1)` por uma expressão contendo um operador com efeito colateral, como, por exemplo, `x++;` ou `x = 10;` o programa não apenas continuará sendo legal, como também fará sentido.

Em C, uma instrução pode aparecer apenas dentro de uma função (v. **Capítulo 5**). Quer dizer, fora de uma função podem existir definições de variáveis, constantes simbólicas e outros componentes de um programa que ainda serão discutidos, mas nunca pode existir uma instrução fora de uma função. Por exemplo, você não será capaz de compilar o seguinte programa:

```
#include <stdio.h>
int x = 0; /* Definições de variáveis podem aparecer fora de funções */
x = 2*x; /* Mas, uma instrução NÃO pode aparecer fora de uma função */
int main(void)
{
   printf("x = %d", x);
   return 0;
}
```

Toda instrução em C deve conter um ponto e vírgula ao final. Isto é, ponto e vírgula é considerado **terminal de instrução**. Declarações e definições também devem ser encerradas com ponto e vírgula. Mas, as seguintes linhas de programa não requerem ponto e vírgula e sua inclusão pode causar erro de sintaxe ou de lógica:

- Diretivas de pré-processamento (i.e., linhas começando com #), como **#include** e **#define**.
- Comentários.
- Uma linha que continua na linha seguinte.
- Instruções ou declarações que terminam com fecha-chaves.

Uma **sequência** (ou **bloco**) **de instruções** consiste em uma ou mais instruções confinadas entre chaves (i.e., entre { e }) e pode ser inserida em qualquer local de um programa onde uma única instrução é permitida. Uma sequência de instruções pode conter ainda definições de variáveis e não deve terminar com ponto e vírgula.

Variáveis definidas dentro de um bloco de instruções são conhecidas como **variáveis locais** e têm validade apenas no interior do bloco. Além disso, conforme já foi visto (v. **Seção 3.8**), cada variável deve ser definida antes de ser utilizada pela primeira vez. Ou seja, pode-se colocar uma definição em qualquer local de um bloco, desde que seja antes da primeira instrução que use a variável. Mas, por uma questão de estilo de programação, recomenda-se que todas variáveis locais a um bloco sejam definidas no início do bloco (i.e., na linha seguinte ao abre-chaves do bloco).

Blocos de instruções podem ser aninhados dentro de outros blocos, como mostrado esquematicamente abaixo:

```
{
   ... /* Primeiro bloco */
   {
      ... /* Segundo bloco aninhado dentro do primeiro */
      {
         ... /* Terceiro bloco aninhado dentro do segundo */
      }
      ...
   }
   ...
}
```

O número de níveis de aninho de blocos permitido por um compilador aderente ao padrão ISO de C vai além da imaginação de qualquer programador equilibrado. Porém, o aninho de mais de um nível (como esquematizado acima) não é recomendado, pois tal construção tem sua legibilidade comprometida.

4.3 Instruções Vazias

Um aspecto interessante da linguagem C é que ela permite a escrita de **instruções vazias** (i.e., que não executam nenhuma tarefa) em qualquer local onde se pode colocar uma instrução normal. Uma instrução vazia em C é representada por um ponto e vírgula, desde que ele não seja considerado terminal de instrução ou declaração. Por exemplo, se um programa contém a linha:

```
x = 2*y;;
```

o primeiro ponto e vírgula é considerado terminal de instrução, enquanto o segundo ponto e vírgula é uma instrução vazia. Isso ocorre porque, depois do primeiro ponto e vírgula era esperada outra instrução ou um fecho de bloco (i.e., }).

Ainda considerando o último exemplo, o ponto e vírgula que representa instrução vazia provavelmente é decorrente de um acidente de trabalho. Ou seja, talvez, o programador tenha se distraído ou o teclado utilizado estivesse defeituoso. Como, isoladamente, ponto e vírgula representa uma instrução vazia, a inserção acidental desse símbolo num local onde se espera uma instrução normal é interpretada pelo compilador como uma construção válida.

Instruções vazias acidentais podem ser deletérias ou não. No exemplo em questão, a instrução vazia acidental é inócua (i.e., ela não causa nenhum dano ao programa). Mas, por mais surpreendente que possa parecer neste instante, uma instrução vazia pode corresponder exatamente àquilo que o programador deseja obter. Quando uma instrução vazia é proposital, recomenda-se que o programador coloque-a numa linha separada e acompanhada de comentário explicativo para deixar claro que essa instrução é realmente proposital (e não acidental).

Mais adiante, neste capítulo, serão apresentados exemplos de instruções vazias acidentais e sugestões para o programador prevenir-se de efeitos danosos decorrentes dessas instruções. Serão apresentados ainda exemplos de situações que requerem instruções vazias propositais.

4.4 Estruturas de Controle

As estruturas de controle de C podem ser classificados em três categorias:

- **Repetições** (ou **iterações**) que permitem a execução de uma ou mais instruções repetidamente.
- **Desvios condicionais** que permitem decidir, com base no resultado da avaliação de uma expressão, qual trecho de um programa será executado.
- **Desvios incondicionais** que indicam, incondicionalmente, qual instrução será executada em seguida.

As próximas seções explorarão detalhadamente as estruturas de controle da linguagem C.

4.5 Laços de Repetição

Laços de repetição permitem controlar o número de vezes que uma instrução ou sequência de instruções é executada. A linguagem C possui três laços de repetição: **while**, **do-while** e **for**. Essas estruturas serão detalhadas em seguida.

4.5.1 while

A instrução **while** (ou **laço while**) é uma estrutura de repetição que tem o seguinte formato:

while (*expressão*)
instrução;

A expressão entre parênteses pode ser aritmética, relacional ou lógica, mas, tipicamente, esses dois últimos tipos de expressões predominam. Uma constante ou variável também pode ocupar o lugar da expressão. A instrução endentada na linha seguinte no esquema acima é denominada **corpo do laço** e pode ser representada por uma sequência de instruções entre chaves. Em geral, qualquer instrução cuja execução está subordinada a uma estrutura de controle constitui o corpo dessa estrutura.

A instrução **while** é interpretada conforme descrito a seguir.

1. A expressão entre parênteses é avaliada.
2. Se o resultado da avaliação da expressão for diferente de zero, o corpo do laço é executado. Então, o fluxo de execução retorna ao passo 1.
3. Se o resultado da avaliação da expressão for igual a zero, a instrução **while** é encerrada e a execução do programa continua na próxima instrução que segue o laço **while**.

O diagrama da **Figura 4–1** ilustra o funcionamento do laço de repetição **while**. Note, nessa figura, a formação de um laço quando o resultado da expressão é diferente de zero. É esse laço que dá origem à expressão *laço de repetição*. Note ainda que, se inicialmente a expressão resultar em zero, o corpo do laço não será executado nenhuma vez.

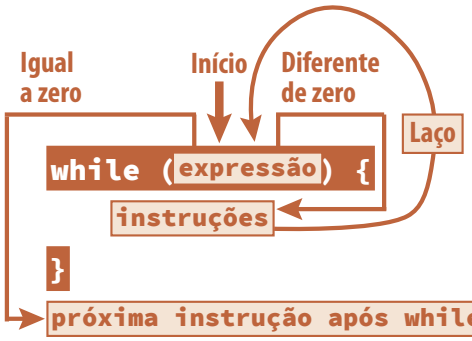

FIGURA 4–1: DIAGRAMA DE FUNCIONAMENTO DO LAÇO WHILE

Como exemplo de uso da instrução **while** considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 10;

   while (x < y) {
      x++;
      y--;
   }

   printf("x = %d, y = %d", x, y);

   return 0;
}
```

No laço **while** desse exemplo, a expressão é `x < y` e o corpo do laço é constituído pela sequência de instruções composta por `x++` e `y--`. Como o corpo do laço é uma sequência de instruções, essas instruções estão envolvidas por chaves. Além disso, o abre-chaves do bloco que constitui o corpo do laço aparece na primeira linha da

instrução **while**, e não na linha seguinte como talvez fosse mais esperado. A adoção dessa prática tem justificativa, que será exposta mais adiante.

Exercício: Antes de prosseguir, responda às seguintes questões referentes ao último exemplo:

[1] Quantas vezes o corpo do laço **while** é executado?

[2] Quais serão os valores das variáveis `x` e `y` ao final do laço?

[3] Se as versões sufixas dos operadores de incremento e decremento (i.e., `x++` e `y--`) forem substituídas por versões prefixas (i.e., `++x` e `--y`) , as respostas às questões 1 e 2 serão diferentes?

Depois de tentar responder analiticamente (i.e., sem o auxílio do computador) às questões acima, você pode obter as respostas das questões 1 e 2, compilando e, então, executando o programa. A resposta da questão 2 é exatamente aquilo que a função **printf()** exibe na tela. Para responder à primeira questão, note que a variável `x` funciona como uma **variável de contagem**, pois ela é iniciada com zero e, a cada passagem do laço, seu valor é acrescido de um. Portanto o valor final dessa variável corresponde exatamente ao número de vezes que o corpo do laço é executado. Para responder a questão 3, você não precisará alterar o programa, basta notar que nas duas expressões em que aparecem os operadores de incremento e decremento, apenas os efeitos colaterais desses operadores são utilizados (i.e., os resultados das expressões são desprezados). Se você ainda não consegue responder a questão 3, mesmo após essa dica, estude novamente a **Seção 3.11**.

Esquecer de colocar as chaves em torno de uma sequência de instruções faz com que apenas a primeira delas seja considerada como corpo de um laço **while**. Por exemplo, remova as chaves em torno das instruções que fazem parte do laço **while** do último programa, de modo a obter o seguinte programa resultante:

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 10;

   while (x < y)
      x++; /* Esta instrução faz parte do while */
      y--; /* Apesar da endentação, esta instrução não faz parte do while */

   printf("x = %d, y = %d", x, y);

   return 0;
}
```

Se você editar, compilar e executar os dois programas anteriores, verá que eles apresentam resultados distintos[2]. Isso ocorre porque, diferentemente do programa anterior que tinha duas instruções no corpo do laço **while**, aqui o corpo do laço contém apenas uma instrução, que é `x++`. Ou seja, apesar de a endentação da instrução `y--` sugerir que ela faz parte do corpo do **while**, na realidade, ela é considerada a instrução que segue a instrução **while**. Isso ocorre porque, em programação moderna, endentação faz sentido apenas entre programadores. Ou seja, ela é completamente ignorada por compiladores, que desprezam quaisquer espaços em branco adicionais.

Para prevenir o esquecimento de chaves em torno de sequências de instruções, recomenda-se que elas sejam sempre usadas, mesmo quando o corpo da instrução **while** é constituído por uma única instrução. Isso evita que você esqueça de inserir as chaves se, por acaso, quiser acrescentar mais alguma instrução ao corpo do **while**.

Deve-se ainda tomar cuidado para não escrever ponto e vírgula após a primeira linha de uma instrução **while**, pois, assim, o corpo do laço será interpretado como uma instrução vazia. Como exemplo, considere o seguinte programa:

[2] O resultado escrito na tela pelo primeiro programa é: `x = 5, y = 5`, enquanto o segundo programa escreve: `x = 10, y = 9`.

```
#include <stdio.h>

int main(void)
{
   int x = 10;

   while (x);
      x--;

   printf("x = %d", x);

   return 0;
}
```

Nesse exemplo, o programador, muito provavelmente, pretendia que o corpo do laço fosse `x--` (conforme sugerido pela endentação), mas na realidade o corpo será a instrução vazia (i.e., o ponto e vírgula ao final da primeira linha). Se você editar, compilar e executar esse programa, verá que ele nunca termina nem apresenta nada na tela. Isso ocorre porque o programa contém um laço de repetição infinito, acidentalmente causado pelo ponto e vírgula ao final da primeira linha do laço **while**. Analisando mais detalhadamente essa situação, observa-se que o valor inicial de `x` é `10` e que essa variável deve eventualmente assumir zero para que o laço **while** termine. Acontece que o corpo do laço é uma instrução vazia que, obviamente, não altera o valor de `x`. Assim, o laço nunca encerra. (Se você deseja executar esse programa para confirmar o que se está afirmando aqui, poderá encerrá-lo com a combinação de teclas `[CTRL]+[C]`.)

Novamente, o uso de chaves envolvendo o corpo de um laço **while**, mesmo quando o corpo é constituído de apenas uma instrução, previne o efeito nocivo de instruções vazias acidentais. Mas, o uso de chaves só é eficiente se o abre-chaves for colocado após os parênteses da expressão de uma instrução **while**. Por exemplo, suponha que um programador segue apenas parcialmente essa recomendação e escreve o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int x = 10;

   while (x);
   {
      x--;
   }

   printf("x = %d", x);

   return 0;
}
```

O código em linguagem de máquina gerado pelo compilador para esse programa é exatamente igual àquele gerado para o programa anterior. Ou seja, esse último programa apresenta o mesmo defeito do programa anterior causado pelo ponto e vírgula na linha contendo **while**. Se o programador tivesse seguido à risca a recomendação sugerida no parágrafo anterior, mesmo que ele tivesse digitado um ponto e vírgula acidental, seu programa apareceria assim:

```
#include <stdio.h>

int main(void)
{
   int x = 10;

   while (x) {;
      x--;
   }
```

```
   printf("x = %d", x);

   return 0;
}
```

Nesse último programa, assim como nos dois programas anteriores, há um ponto e vírgula acidental que representa uma instrução vazia. A diferença é que, aqui, a instrução vazia não causa nenhum dano ao programa. Se ainda não o fez, edite, compile e execute os três últimos programas para confirmar o que foi afirmado.

4.5.2 do-while

Um laço de repetição **do-while** tem o seguinte formato:

```
do
   instrução;
while (expressão);
```

Como na instrução **while** vista antes, *instrução* representa o corpo do laço e *expressão* é uma expressão que, quando resulta em zero, encerra o laço. Em termos de interpretação, a única diferença entre as instruções **while** e **do-while** é o ponto onde cada instrução inicia, como mostra a Figura 4–2, que ilustra o funcionamento da instrução **do-while**:

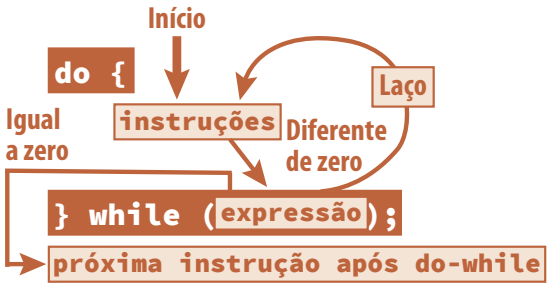

FIGURA 4–2: DIAGRAMA DE FUNCIONAMENTO DO LAÇO DO-WHILE

Se você comparar detidamente a Figura 4–1, que mostra o funcionamento de uma instrução **while**, com a Figura 4–2, referente ao funcionamento de um laço **do-while**, poderá concluir que o que diferencia essas duas instruções é como cada uma começa. Ou seja, uma instrução **while** começa com a avaliação da expressão que a acompanha, enquanto uma instrução **do-while** começa com a execução do corpo do respectivo laço. Como consequência, é assegurado que o corpo do laço de uma instrução **do-while** é executado pelo menos uma vez. Assim, a instrução **do-while** é indicada para situações nas quais se deseja que o corpo do laço seja sempre executado.

Considere o seguinte programa como exemplo de uso da instrução **do-while**:

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 10;

   do {
      x++;
      y--;
   } while (x < y);
   printf("x = %d, y = %d", x, y);

   return 0;
}
```

Se você comparar o primeiro exemplo apresentado na Seção 4.5.1 como esse último exemplo, concluirá que a única diferença entre eles é o tipo de laço de repetição usado por cada um deles. Além disso, os dois programas apresentam os mesmos resultados e, por isso, são considerados funcionalmente equivalentes. Quer dizer, se você considerar para esse último exemplo, as mesmas questões formuladas para o primeiro exemplo da Seção 4.5.1,concluirá que as respostas às respectivas questões são as mesmas.

Nem sempre os laços **while** e **do-while** são equivalentes. Por exemplo, considere os dois programas a seguir:

Programa 1

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 0;

   while (x < y) {
      x++;
      y--;
   }

   printf("x=%d, y=%d", x, y);

   return 0;
}
```

Programa 2

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 0;

   do {
      x++;
      y--;
   } while (x < y);

   printf("x=%d, y=%d", x, y);

   return 0;
}
```

Apesar das semelhanças entre os dois últimos programas, o primeiro programa exibe `x = 0, y = 0` na tela, enquanto o segundo escreve: `x = 1, y = -1`. Além disso, o laço **while** do primeiro programa não é executado nenhuma vez, mas o laço **do-while** do segundo programa é executado uma vez.

As mesmas recomendações de uso e posicionamento de chaves apresentadas para o laço **while** (v. Seção 4.5.1) são válidas aqui.

4.5.3 for

O laço **for** é um pouco mais complicado do que os outros dois laços de repetição de C e sua sintaxe segue o seguinte esquema:

for (*expressão*$_1$**;** *expressão*$_2$**;** *expressão*$_3$**)**
    instrução;

Qualquer das expressões entre parênteses é opcional, mas, usualmente, todas as três são utilizadas. Uma instrução **for** é interpretada conforme a seguinte sequência de passos:

1. *expressão*$_1$ é avaliada. Tipicamente, essa é uma expressão de atribuição.
2. *expressão*$_2$, que é a expressão condicional da estrutura **for**, é avaliada.
3. Se *expressão*$_2$ resultar em zero, a instrução **for** é encerrada e o controle do programa passa para a próxima instrução que segue o laço **for**. Se *expressão*$_2$ resultar num valor diferente de zero, o corpo do laço (representado por *instrução* no quadro esquemático) é executado.
4. Após a execução do corpo do laço, *expressão*$_3$ é avaliada e retorna-se ao passo 2.

A Figura 4–3 ilustra o funcionamento da instrução **for**.

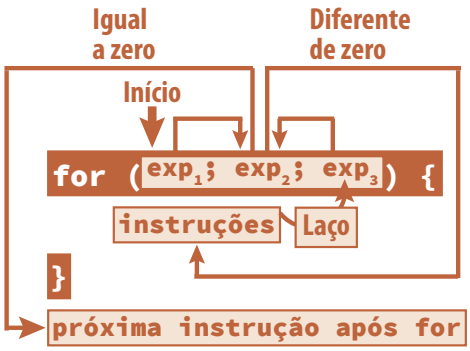

FIGURA 4–3: DIAGRAMA DE FUNCIONAMENTO DO LAÇO FOR

Considere o seguinte programa como exemplo de uso de laço **for**:

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 1; i <= 10; ++i) {
      printf("C e C++ sao linguagens diferentes\n");
   }
   return 0;
}
```

Esse programa escreve dez vezes na tela a frase:

```
C e C++ sao linguagens diferentes
```

Comparando-se os diagramas que ilustram os funcionamentos das estruturas **while** e **for**, pode-se facilmente concluir que a instrução **for** é equivalente em termos funcionais à seguinte sequência de instruções (existe uma exceção para essa equivalência que será explorada na Seção 4.7.2):

```
expressão1;

while (expressão2) {
   instrução;
   expressão3;
}
```

Utilizando-se desse conhecimento, o programa do último exemplo poderia ser reescrito como:

```
#include <stdio.h>

int main(void)
{
   int i;

   i = 1;

   while (i <= 10) {
      printf("C e C++ sao linguagens diferentes\n");
      ++i;
   }

   return 0;
}
```

Apesar da mencionada equivalência entre um laço **for** e um conjunto de instruções envolvendo **while**, por questões de legibilidade e concisão, esse conjunto de instruções não é indicado para substituir instruções **for** em situações nas quais o uso dessa instrução parece ser a escolha mais natural.

A instrução **for** é mais frequentemente utilizada em **laços de contagem**; i.e., quando se deseja executar uma instrução ou sequência de instruções um número específico de vezes. Um laço de contagem é caracterizado por:

- Uma **variável de contagem** ou **contador**, tipicamente iniciada com `1` ou `0` na primeira expressão de um laço **for**. Alguns poucos programadores gostam de denominar essa variável como `contador` ou `cont`, mas é mais comum que ela seja denominada como `i`, `j` ou `k`.
- Uma expressão relacional, usada como segunda expressão do aludido laço **for**, que indica implicitamente quantas vezes o corpo do laço será executado. Por exemplo, se variável de contagem `i` for iniciada com `1` e deseja-se executar o corpo do laço `n` vezes, essa expressão relacional deve ser `i <= n`. Se a variável `i` for iniciada com zero, a expressão relacional deve ser `i < n` ou `i <= n - 1`.
- Um incremento da variável de contagem na terceira expressão do mesmo laço **for** usado como laço de contagem.

A instrução **for** do primeiro programa apresentado na corrente subseção constitui um exemplo de laço de contagem:

```
for (i = 1; i <= 10; ++i) {
   printf("C e C++ sao linguagens diferentes\n");
}
```

Um erro comum em laços de contagem é executar o corpo do laço **for** um número de vezes diferente do pretendido em virtude do uso de um operador relacional inadequado (p. ex., `<` em vez de `<=` ou vice-versa). Por exemplo, se a condição de teste utilizada no exemplo anterior fosse `i < 10`, em vez de `i <= 10`, o corpo do **for** seria executado apenas 9 vezes (e não 10 vezes, como é o caso). Esse tipo de erro não é indicado pelo compilador e é, muitas vezes, difícil de ser detectado. A melhor forma de prevenir esse tipo de erro é testar cada laço de contagem até convencer-se de que ele realmente funciona conforme o esperado. Enfim, para certificar-se que o corpo de um laço de contagem é executado o número desejado de vezes, o conselho básico a ser seguido é:

Recomendação ***Num laço de contagem for, dedique atenção redobrada ao valor inicial e ao maior valor assumido pela variável de contagem.***

Conforme foi discutido na **Seção 3.4.3**, números reais são armazenados de modo aproximado. Portanto deve-se evitar o uso de variáveis do tipo **double** para controlar laços de contagem.

É importante ainda salientar que programadores de C apresentam grande predileção por iniciar contagens com zero por uma razão exposta no **Capítulo 8**, mas pessoas normais não começam a contar a partir de zero. Por isso, é muito frequente errar a escrita da expressão relacional que indica quantas vezes o corpo de um laço de contagem será executado quando a contagem começa em zero.

Conforme foi mencionado acima, qualquer das expressões entre parênteses pode ser omitida numa instrução **for**. Entretanto, os dois pontos e vírgulas devem sempre ser incluídos. Na prática, é comum omitir-se *expressão*$_1$ ou *expressão*$_3$, mas não ambas ao mesmo tempo porque, omitir simultaneamente *expressão*$_1$ e *expressão*$_3$ torna a expressão **for** equivalente a uma única instrução **while** (v. relação entre **for** e **while** apresentada acima). Normalmente, *expressão*$_2$ é incluída, pois trata-se da condição de teste. Quando essa condição de teste é omitida, ela é considerada igual a `1` e o laço **for** é considerado um laço infinito (v. **Seção 4.5.6**). Em termos de estilo de programação, se você precisa omitir alguma expressão de um laço **for**, é melhor usar um laço **while** em substituição ao laço **for**.

É relativamente comum utilizar uma instrução vazia como corpo de um laço **for** quando a tarefa desejada é executada pelas expressões entre parênteses do laço, como mostra o seguinte exemplo:

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 10;

   for ( ; x < y; x++, y--) {
      ; /* Instrução vazia */
   }

   printf("x = %d, y = %d", x, y);

   return 0;
}
```

Esse último programa é funcionalmente equivalente àquele apresentado no início da **Seção 4.5.1**. Compare os dois programas e observe que o corpo do laço **while** do programa da **Seção 4.5.1** é constituído pelas instruções `x++` e `y--`, que são expressões. No presente exemplo, essas expressões são reunidas numa única expressão por meio do operador vírgula, que será formalmente apresentado na **Seção 4.9**. Portanto, se a intenção do programador é apenas incrementar a variável `x` e decrementar a variável `y`, isso é realizado na terceira expressão do laço **for** e nada mais precisa ser feito no corpo desse laço. Por isso, o corpo do laço é representado por uma instrução vazia.

De acordo com o padrão C99, é permitido o uso de iniciações de variáveis no lugar da primeira expressão de um laço **for**. Variáveis declaradas dessa maneira têm validade apenas no laço **for** correspondente. Por exemplo, considere o programa a seguir:

```
#include <stdio.h>

int main(void)
{
   for (int i = 1; i <= 10; ++i) {
      printf("Como e' divertido programar em C\n");
   }

      /* A instrução a seguir impede o programa de ser */
      /* compilado pois a variável i não é válida aqui */
   printf("\nValor de i = %d", i);

   return 0;
}
```

Esse programa não consegue ser compilado porque a função **printf()** faz referência à variável `i` num local em que ela é inválida.

4.5.4 Laços Aninhados

Quando um laço de repetição faz parte do corpo de outro laço, diz-se que ele é **aninhado**. O exemplo a seguir mostra o uso de dois laços de repetição **for**, sendo o segundo laço aninhado no primeiro.

```
#include <stdio.h>

int main(void)
{
   int i, j;

      /* Laço externo */
   for (i = 1; i <= 2; ++i) {
```

```
      printf( "\nIteracao do laco for externo no. %d\n", i );

         /* Laço interno (aninhado) */
      for (j = 1; j <= 3; ++j) {
         printf( "\tIteracao do laco for interno no. %d\n", j );
      }
   }

   return 0;
}
```

Nesse programa, laço externo é executado duas vezes, enquanto o laço interno é executado três vezes cada vez que o corpo do laço externo é executado. Portanto, no total, o corpo do laço interno é executado seis vezes e o resultado desse programa é o seguinte:

```
Iteracao do laco for externo no. 1
        Iteracao do laco for interno no. 1
        Iteracao do laco for interno no. 2
        Iteracao do laco for interno no. 3

Iteracao do laco for externo no. 2
        Iteracao do laco for interno no. 1
        Iteracao do laco for interno no. 2
        Iteracao do laco for interno no. 3
```

4.5.5 Expressões Condicionais e Condições de Parada

Uma **condição de parada** de um laço de repetição é uma expressão tal que, quando ela resulta num valor diferente de zero, o laço é encerrado. Quando uma condição de parada resulta num valor diferente de zero, diz-se que ela foi **satisfeita** ou **atingida**.

De acordo com o que foi visto em relação aos três laços de repetição de C, cada um deles possui uma expressão condicional que controla a execução do corpo do laço, de modo que o corpo do laço é executado quando ela é diferente de zero e encerra em caso contrário. Portanto pode-se concluir que uma expressão que controla a execução de um laço de repetição deve ser a negação da condição de parada do respectivo laço e vice-versa. Por exemplo, no laço **while**:

```
while (x < y) {
   x++;
   y--;
}
```

a expressão que controla o laço é `x < y` e sua negação e condição de parada do laço é `!(x < y)` ou `(x >= y)`.

Uma discussão sobre condições de parada é importante porque, enquanto planeja escrever um laço de repetição, muitas vezes, o programador pensa em termos de condição de parada e, baseado nela, escreve a expressão de controle do laço. Infelizmente, essa passagem de condição de parada para expressão de controle causa muita confusão entre programadores iniciantes e esse é especialmente o caso quando uma condição de parada consiste numa conjunção ou disjunção de subexpressões.

Suponha, por exemplo, que um programador deseja escrever um laço **while** que termine quando `x` é maior do que `5` e `y` é menor do que ou igual a `10`, sendo `x` e `y` variáveis inteiras devidamente definidas. Então, a condição de parada que ele deseja que seja satisfeita pode ser escrita como[3]: `(x > 5) && (y <= 10)`. Como foi salientado, a expressão condicional que deve acompanhar o laço **while** desejado pelo programador deve ser a negação dessa última expressão. Agora, o problema é que programadores que não são proficientes em Lógica

[3] Devido às precedências dos operadores envolvidos nessa expressão, não há necessidade de uso de parênteses, mas eles facilitam a leitura da expressão.

Matemática são compelidos a escrever a negação dessa última expressão como: `!(x > 5) && !(y <= 10)`, o que não é correto. Isto é, de acordo com uma lei de equivalência, denominada **Lei de De Morgan**, a negação de uma conjunção é a disjunção das negações dos operandos da conjunção. Essa lei parece complicada, mas não é.

Suponha que `A` e `B` são operandos de uma conjunção, então a referida lei de De Morgan afirma que:

`!(A && B)` ***é equivalente a*** `!A || !B`

Ciente dessa lei, o programador saberia que a negação da condição de parada:

```
(x > 5) && (y <= 10)
```

é a expressão:

```
!(x > 5) || !(y <= 10)
```

ou a expressão equivalente:

```
(x <= 5) || (y > 10)
```

Existe outra lei de De Morgan que trata da negação de disjunções, que afirma que:

`!(A || B)` ***é equivalente a*** `!A && !B`

Em palavras, essa segunda lei de De Morgan assegura que a negação de uma disjunção é a conjunção das negações dos operandos da disjunção.

As leis de De Morgan podem ser generalizadas para conjunções e disjunções de um número arbitrário de operandos como:

`!(A1 && A2 && ... && An)` ***é equivalente a*** `!A1 || !A2 || ... || !An`
`!(A1 || A2 || ... || An)` ***é equivalente a*** `!A1 && !A2 && ... && !An`

Outro ponto importante que causa certa confusão entre programadores iniciantes e que precisa ser ressaltado é que expressões condicionais que comparam valores com zero não precisam usar explicitamente os operadores relacionais `==` e `!=`. Por exemplo, suponha que `x` seja uma variável que se deseja verificar se é igual a ou diferente de zero. Então, é fácil verificar que as seguintes equivalências são válidas:

`x != 0` ***é equivalente a*** `x`
`x == 0` ***é equivalente a*** `!x`

4.5.6 Laços de Repetição Infinitos

Um **laço de repetição infinito** é aquele cuja condição de parada nunca é atingida ou é atingida apenas em decorrência de overflow (v. **Seção 4.11.7**). Mais precisamente, apesar de a denominação sugerir que um laço infinito nunca termina, ele pode terminar. Isto é, o que a definição afirma é que um laço infinito não termina em decorrência da avaliação *normal* da expressão de controle do laço (i.e., sem levar em conta overflow), mas há outros meios de encerrar a execução de um laço de repetição, como será visto adiante.

Algumas vezes, um laço de repetição infinito é aquilo que realmente o programador deseja, mas, outras vezes, tal instrução é decorrente de um erro de programação que a impede de terminar apropriadamente.

Laços de repetição infinitos podem ser divididos em duas categorias:

[1] Laço de repetição que **não contém uma condição de parada** (ou, equivalentemente, que contém uma **condição de parada que é sempre zero**).

[2] Laço de repetição que contém uma **condição de parada que nunca é atingida** (sem ocorrência de overflow).

Como exemplos de laços de repetição infinitos da primeira categoria, têm-se, os laços **while** e **for** dos seguintes programas:

Programa 1

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 10;

   while (1) {
      if (x >= y) {
         break;
      }

      x++;
      y--;
   }
   printf("x = %d, y = %d", x, y);

   return 0;
}
```

Programa 2

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 10;

   for ( ; ; ) {
      if (x >= y) {
         break;
      }

      x++;
      y--;
   }
   printf("x = %d, y = %d", x, y);

   return 0;
}
```

Os dois programas apresentados acima são funcionalmente equivalentes entre si e são também equivalentes ao primeiro exemplo apresentado na **Seção 4.5.1**, sendo que, em termos de estilo, aquele da **Seção 4.5.1** é o melhor deles.

Nos dois últimos programas, as linhas:

```
while (1) {
```

e

```
for ( ; ; ) {
```

constituem jargões comumente utilizados para indicar repetição infinita intencional.

Os laços **while** e **for** dos programas do último exemplo encerram quando a instrução **break** é executada e isso ocorre quando a expressão `x >= y` da instrução **if** resulta em `1`. As instruções **if** e **break** serão respectivamente examinadas nas **Seções 4.6.1** e **4.7.1**. Se você sentir alguma dificuldade para entender os dois últimos programas, retorne à presente seção após estudar essas duas seções.

Como exemplo de laço de repetição infinito do segundo tipo mencionado acima, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 1; i <= 10; i++) {
      printf("Valor de i: %d\n", i);

           /* Algumas instruções que impedem o programador de enxergar o erro */
      /* ... */

      i = 2;
   }

   return 0;
}
```

Certamente, a instrução **for** do último exemplo contém um erro de programação porque ela possui uma expressão condicional (`i <= 10`) que *sempre* resulta em `1`, pois, sempre que essa condição é testada, o valor de `i` é diferente de zero (i.e., na primeira avaliação dessa expressão, `i` vale `1` e, nas demais avaliações, `i` vale `3`). Portanto a condição de parada (`i > 10`) jamais será atingida.

4.6 Desvios Condicionais

Instruções de **desvios condicionais** permitem desviar o fluxo de execução de um programa dependendo do resultado da avaliação de uma expressão (**condição**). Essas instruções serão examinadas a seguir.

4.6.1 if-else

A principal instrução condicional em C é **if-else**, que tem a seguinte sintaxe:

if *(expressão)*
    instrução$_1$*;*
else
    instrução$_2$*;*

A interpretação de uma instrução **if** é a seguinte: a expressão entre parênteses é avaliada; se o resultado dessa avaliação for diferente de zero, *instrução*$_1$ será executada; caso contrário, *instrução*$_2$ será executada. As instruções *instrução*$_1$ e *instrução*$_2$ podem, naturalmente, ser substituídas por sequências de instruções. A **Figura 4–4** ilustra o funcionamento da instrução **if-else**:

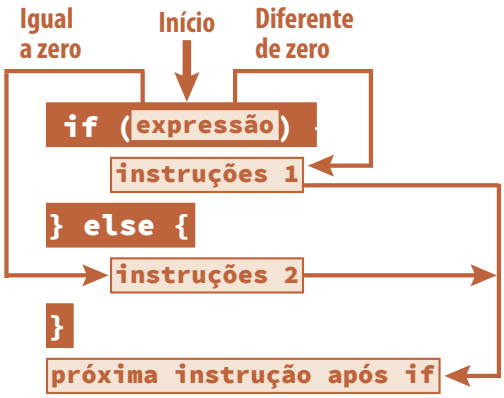

FIGURA 4–4: DIAGRAMA DE FUNCIONAMENTO DA INSTRUÇÃO IF-ELSE

O seguinte programa contém um exemplo de uso de um desvio **if-else**:

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

int main(void)
{
   int x = 0;

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

   if (x % 2 == 0) {
      printf("\nO numero introduzido e' par\n");
   } else {
      printf("\nO numero introduzido e' impar\n");
   }
```

```
   printf("\nPrograma encerrado\n");

   return 0;
}
```

A parte **else** é opcional e, quando ela é omitida, não há desvio quando o resultado da expressão é igual a zero e o fluxo de execução é resumido na próxima instrução após **if**. A Figura 4–5 ilustra o funcionamento da instrução **if-else** com a parte **else** omitida.

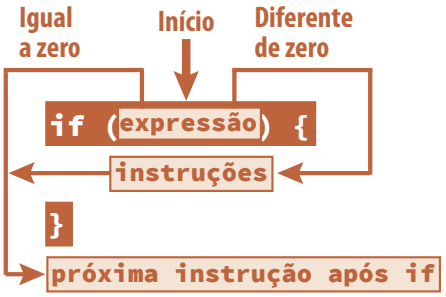

FIGURA 4–5: DIAGRAMA DE FUNCIONAMENTO DA INSTRUÇÃO IF SEM ELSE

O programa abaixo contém um exemplo de uso de um desvio **if-else** sem a parte **else**:

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

int main(void)
{
   double x;

   printf("Digite um valor real: ");
   x = LeReal();

      /* Calcula a raiz quadrada do número se ele não for negativo */
   if (x >= 0) {
      printf("\nA raiz quadrada do numero e' %f", sqrt(x));
   }

   printf("\n\nPrograma encerrado");

   return 0;
}
```

As endentações das instruções subordinadas aos desvios **if** nesses exemplos refletem relações de dependência entre instruções. Por exemplo, no último programa, a instrução **printf**() que segue **if** pertencem ao **if** e é endentada em relação a essa última instrução, enquanto a última instrução **printf**() é independente do **if** e é escrita no mesmo nível dessa instrução. Endentação serve ao único propósito de melhorar a legibilidade do programa e não faz a menor diferença para o compilador. Para ilustrar esse ponto, suponha, por exemplo, que você deseja executar as instruções `x++` e `printf("x = %d", x)` apenas quando `x` for diferente de zero na seguinte instrução **if**:

```
if (x)
   x++;  /* Executada quando x != 0 */
   printf("x = %d", x); /* Executada sempre  */
```

Apesar de a endentação indicar uma ilusória dependência da instrução **printf**() com a instrução **if**, na realidade, a instrução **printf**() será sempre executada, independentemente do valor de `x`. Para executar ambas as instruções quando `x` for diferente de zero, você terá que uni-las numa sequência de instruções como:

```
if (x) {
   x++;  /* Executada quando x != 0 */
   printf("x = %d", x); /* Idem  */
}
```

Deve-se ainda chamar a atenção para o perigo representado pela troca, por engano, do operador relacional de igualdade (representado por `==`) pelo operador de atribuição (representado por `=`). Como exemplo, considere o seguinte trecho de programa:

```
int x = 0;

...

if  (x = 10) /* O resultado da expressão é sempre 10 */
   y = 2*x;   /* Esta instrução será sempre executada */
```

Essa instrução **if** contém com certeza um erro de programação, pois, como a expressão `x = 10` resulta sempre em `10` (v. **Seção 3.9**), a instrução `y = 2*x` será sempre executada. Assim, se essa fosse realmente a intenção do programador, não faria sentido o uso de uma instrução **if**.

Para precaverem-se contra esse tipo de erro, alguns programadores adotam a disciplina de escreverem sempre expressões de igualdade, como aquela do último exemplo, com a constante do lado esquerdo. Logo, se o operador de igualdade for acidentalmente trocado pelo operador de atribuição, o compilador indicará o erro. Por exemplo:

```
int x = 0;

...

if  (10 = x)  /* O compilador indicará um erro nesta expressão */
   y = 2*x;
```

4.6.2 Instruções if Aninhadas

Instruções **if** podem ser aninhadas de modo a representar **desvios múltiplos**. Uma instrução **if** é aninhada quando ela compõe o corpo de uma parte **if** ou **else** de um desvio **if-else**. Por exemplo, o programa adiante, que exibe na tela o menor dentre três números inteiros introduzidos pelo usuário, contém duas instruções **if** aninhadas:

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

int main(void)
{
   int x, y, z, menor;

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

   printf("\nDigite outro numero inteiro: ");
   y = LeInteiro();

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

   if (x <= y)
      if (x <= z)
         menor = x;
      else
         menor = z;
   else
      if (y <= z)
         menor = y;
```

```
      else
         menor = z;
   printf("\nO menor numero e': %d\n", menor);
   return 0;
}
```

Esse programa funciona perfeitamente bem (verifique isso), mas quando uma instrução **if** segue imediatamente a parte **else** de um desvio **if-else**, é recomendado colocar o **if** aninhado na mesma linha do **else** (i.e., nesse caso, não há endentação como no programa acima). Obviamente, adotar essa recomendação não é obrigatório, mas segui-la facilita a leitura de instruções **if** aninhadas. Usando essa recomendação, os desvios **if-else** do programa anterior seriam formatados como:

```
if (x <= y)
   if (x <= z)
      menor = x;
   else
      menor = z;
else if (y <= z)
   menor = y;
else
   menor = z;
```

Um aspecto importante em instruções aninhadas e que acarreta algumas vezes em erro de programação é o casamento de cada **else** com o respectivo **if**. Especificamente, no último exemplo, o primeiro **else** casa com o segundo **if**, o segundo **else** casa com o primeiro **if** e o terceiro **else** casa com o terceiro **if**, conforme ilustrado na Figura 4–6.

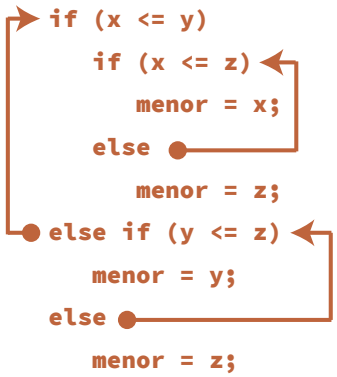

FIGURA 4–6: CASAMENTO DE ELSE COM IF 1

Em geral, deve-se adotar a seguinte regra de casamento entre partes **if** e partes **else** aninhadas:

Um else está sempre associado ao if mais próximo que não tenha ainda um else associado.

Essa regra deixa de ser válida se o **if** mais próximo do **else** estiver isolado entre chaves. Os trechos de programa na Figura 4–7 ilustram esse ponto.

Na ilustração da esquerda, **else** refere-se ao segundo **if**, enquanto, na ilustração da direita, **else** refere-se ao primeiro **if**. A instrução da direita também poderia ser escrita colocando-se uma instrução vazia num **else** pertencente ao **if** mais interno, como ilustrado na Figura 4–8.

Na ilustração da esquerda, **else** refere-se ao segundo **if**, enquanto, na ilustração da direita, **else** refere-se ao primeiro **if**. A instrução da direita também poderia ser escrita colocando-se uma instrução vazia num **else** pertencente ao **if** mais interno, como ilustrado na Figura 4–8.

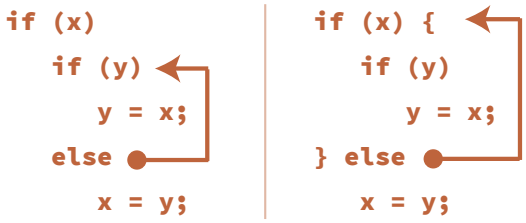

FIGURA 4–7: CASAMENTO DE ELSE COM IF 2

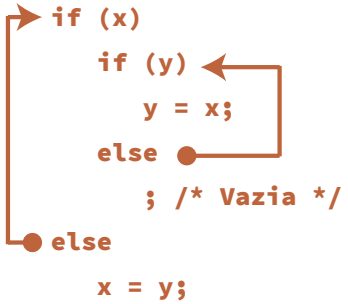

FIGURA 4–8: CASAMENTO DE ELSE COM IF 3

Lembre-se que é sempre boa prática de programação indicar por meio de comentários quando uma instrução vazia está sendo deliberadamente utilizada, como no último exemplo. Além disso, a colocação da parte **else** no mesmo nível de endentação de sua parte **if** associada, como nos exemplos acima, ajuda a identificar possíveis casamentos equivocados entre essas partes.

4.6.3 switch-case

A instrução **switch-case** é uma instrução de **desvios múltiplos** (ou instrução de **seleção**) que é útil quando existem várias ramificações possíveis a ser seguidas num trecho de programa. Nesse caso, o uso de instruções **if** aninhadas poderia tornar o programa mais difícil de ser lido. O uso de instruções **switch-case** em tal situação não apenas melhora a legibilidade, como também a eficiência do programa. Infelizmente, nem sempre uma instrução **switch-case** pode substituir instruções **if** aninhadas. Mas, quando é possível ser utilizada, uma instrução **switch-case** permite que caminhos múltiplos sejam escolhidos com base no valor de uma expressão.

A sintaxe que deve ser seguida por uma instrução **switch-case** é:

```
switch (expressão) {
    case expressão-constante1:
        instrução1;
    case expressão-constante2:
        instrução2;
    ...
    case expressão-constantenn:
        instruçãon;
    default:
        instruçãod;
}
```

A expressão entre parênteses que segue imediatamente a palavra **switch** deve resultar num valor inteiro (**char** ou **int**); essa expressão não pode resultar num valor do tipo **double**, por exemplo. As expressões constantes que acompanham cada palavra-chave **case** devem resultar em valores inteiros. Tipicamente, em vez de uma expressão, uma simples constante acompanha cada **case**.

A instrução **switch** é interpretada da seguinte maneira:

1. *expressão* é avaliada.
2. Se o resultado da avaliação de *expressão* for igual a alguma expressão constante seguindo uma ramificação **case**, *todas* as instruções que seguem essa expressão serão executadas.
3. Se o resultado da avaliação de *expressão* não for igual a nenhuma instrução constante seguindo um **case** e houver uma parte **default**, a instrução seguindo essa parte será executada.

A Figura 4–9 ilustra o funcionamento da instrução **switch-case**.

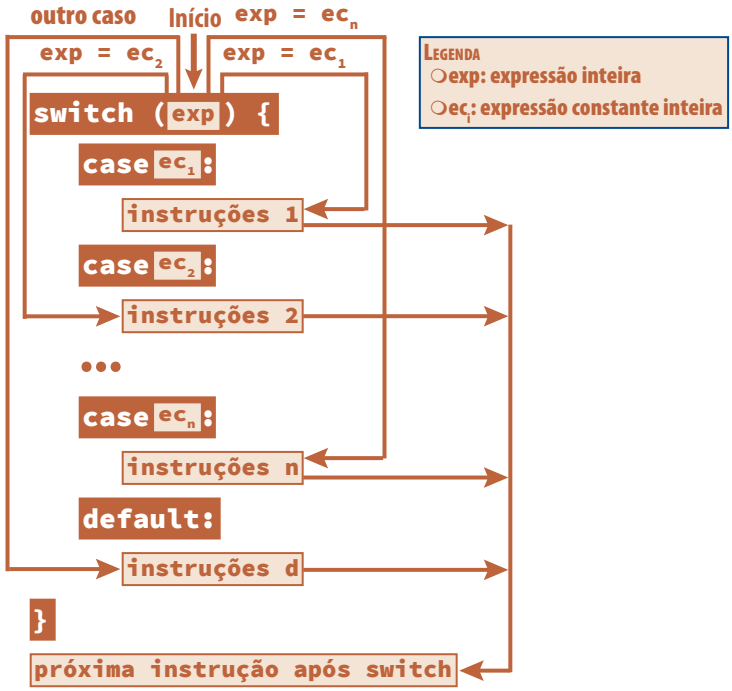

Figura 4–9: Diagrama de Funcionamento da Instrução switch-case

A parte **default** de uma instrução **switch** é opcional e, quando ela é omitida e não ocorre casamento do resultado da expressão com nenhuma expressão constante que acompanha cada **case**, não ocorre desvio. Quando a parte **default** está presente, recomenda-se que ela seja a última parte de uma instrução **switch**, pois a adoção dessa prática melhora a legibilidade da instrução.

Uma diferença importante entre a instrução **switch** de C e instruções análogas em outras linguagens é que, na instrução **switch**, *todas* as instruções seguindo a ramificação **case** selecionada são executadas, mesmo que algumas dessas instruções façam parte de outras ramificações **case**. Esse comportamento é evitado (e usualmente é o que se deseja) por meio do uso de desvios incondicionais: **break**, **goto** ou **return**. Comumente, instruções **break** são utilizadas para fazer com que uma instrução **switch** seja encerrada e o controle passe para a instrução seguinte a essa instrução. Portanto, na grande maioria das vezes, deve-se utilizar **break** no final de cada instrução (ou sequência de instruções) seguindo cada **case**.

Para compreender melhor o que foi exposto no último parágrafo, considere o seguinte programa:

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

int main(void)
{
   char opcao;

   printf( "\nA - Opcao A"
           "\nB - Opcao B"
           "\nC - Opcao C"
           "\nD - Opcao D" );

   printf("\n\nEscolha uma das opcoes disponiveis: ");
   opcao = LeCaractere();

   switch (opcao) {
      case 'A':
         printf("\nVoce escolheu a opcao A");
      case 'B':
         printf("\nVoce escolheu a opcao B");
      case 'C':
         printf("\nVoce escolheu a opcao C");
      case 'D':
         printf("\nVoce escolheu a opcao D");
      default:
         printf("\nVoce nao escolheu uma opcao valida");
   }
   return 0;
}
```

Esse último programa apresenta um menu de opções para o usuário e solicita que ele escolha uma das opções válidas. Suponha que, quando instado, o usuário escolhe a opção `A`. Então, o resultado da execução do programa apresenta-se como:

```
A - Opcao A
B - Opcao B
C - Opcao C
D - Opcao D

Escolha uma das opcoes disponiveis: A

Voce escolheu a opcao A
Voce escolheu a opcao B
Voce escolheu a opcao C
Voce escolheu a opcao D
Voce nao escolheu uma opcao valida
```

Evidentemente, esse não é o resultado esperado pelo programador, mas ele é decorrente do efeito cascata inerente à instrução **switch-case**, que pode ser suprimido usando-se instruções **break** (v. Seção 4.7.1), como demonstra o seguinte programa.

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

int main(void)
{
   char opcao;

   printf( "\nA - Opcao A"
           "\nB - Opcao B"
```

```
            "\nC - Opcao C"
            "\nD - Opcao D" );

   printf("\n\nEscolha uma das opcoes disponiveis: ");
   opcao = LeCaractere();

   switch (opcao) {
      case 'A':
         printf("\nVoce escolheu a opcao A");
         break;
      case 'B':
         printf("\nVoce escolheu a opcao B");
         break;
      case 'C':
         printf("\nVoce escolheu a opcao C");
         break;
      case 'D':
         printf("\nVoce escolheu a opcao D");
         break;
      default:
         printf("\nVoce nao escolheu uma opcao valida");
         break;
   }

   return 0;
}
```

Se o último programa for executado e, novamente, o usuário escolher a opção **A**, o resultado da execução desse programa será:

```
A - Opcao A
B - Opcao B
C - Opcao C
D - Opcao D

Escolha uma das opcoes disponiveis: A

Voce escolheu a opcao A
```

Nesse exemplo, existe a possibilidade de execução de cinco sequências de instruções referentes ao valor correntemente assumido pela variável **opcao**: uma para cada um dos valores **'A'**, **'B'**, **'C'** e **'D'** e a última para um valor de **opcao** diferente desses valores (representado pela parte **default**). A instrução **break** seguindo a instrução associada à parte **default** não é realmente necessária, pois não existe mais nenhuma instrução que segue a parte **default**; entretanto, o uso dessa instrução **break** constitui boa prática de programação porque previne o esquecimento quando ela é estritamente necessária.

Agora, o último programa apresenta um resultado que agrada ao programador. Entretanto, esse último programa não deverá agradar muito ao usuário que se sentirá frustrado ao perceber que a entrada esperada pelo programa é sensível ao uso de letras maiúsculas e minúsculas. Por exemplo, se o usuário tivesse digitado **'a'** na última interação apresentada, o programa exibiria na tela: `Voce nao escolheu uma opcao valida`. Quer dizer, seria bem mais razoável que o programa permitisse que o usuário tivesse liberdade para escolher **A** ou **a**, **B** ou **b** etc.

Quando mais de uma opção **case** num **switch** corresponde a uma mesma instrução, pode-se colocar essas opções juntas e seguidas pela instrução comum, como mostra o programa abaixo:

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

int main(void)
```

```
{
   char opcao;

   printf( "\nA - Opcao A"
           "\nB - Opcao B"
           "\nC - Opcao C"
           "\nD - Opcao D" );

   printf("\n\nEscolha uma das opcoes disponiveis: ");
   opcao = LeCaractere();

   switch (opcao) {
      case 'A':
      case 'a':
         printf("\nVoce escolheu a opcao A");
         break;
      case 'B':
      case 'b':
         printf("\nVoce escolheu a opcao B");
         break;
      case 'C':
      case 'c':
         printf("\nVoce escolheu a opcao C");
         break;
      case 'D':
      case 'd':
         printf("\nVoce escolheu a opcao D");
         break;
      default:
         printf("\nVoce nao escolheu uma opcao valida");
         break;
   }

   return 0;
}
```

O último programa é mais amigável ao usuário, pois permite-lhe digitar sua opção sem preocupar-se com o uso de letras maiúsculas ou minúsculas.

4.7 Desvios Incondicionais

Desvios incondicionais permitem o desvio do fluxo de execução de um programa independentemente da avaliação de qualquer condição (como ocorre com os desvios condicionais). Essas instruções serão examinadas a seguir.

4.7.1 break

A instrução **break** foi apresentada na Seção 4.6.3 como um meio de impedir a passagem de uma instrução referente a uma ramificação **case** para outras instruções pertencentes a outras ramificações **case** de uma instrução **switch**. Pode-se, entretanto, interpretar o comportamento de **break** mais genericamente como uma forma de causar o término imediato de uma estrutura de controle. Além de **switch**, as demais estruturas de controle afetadas pela instrução **break** são os laços de repetição (i.e., **while**, **do-while** e **for**). Considere como exemplo prático de uso de **break** o seguinte programa.

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

int main(void)
{
```

```
   char opcao;

   while (1) {
      printf( "\n\n*** Opcoes ***\n"
              "\nA - Opcao A"
              "\nB - Opcao B"
              "\nC - Opcao C"
              "\nD - Opcao D"
              "\nE - Encerra o programa" );

      printf("\n\nEscolha uma das opcoes disponiveis: ");
      opcao = LeCaractere();

         /* Se o usuário escolher a opção E, encerra o laço */
      if (opcao == 'E' || opcao == 'e') {
         break; /* Encerra o laço while */
      }

      switch (opcao) {
         case 'A':
         case 'a':
            printf("\nVoce escolheu a opcao A");
            break;
         case 'B':
         case 'b':
            printf("\nVoce escolheu a opcao B");
            break;
         case 'C':
         case 'c':
            printf("\nVoce escolheu a opcao C");
            break;
         case 'D':
         case 'd':
            printf("\nVoce escolheu a opcao D");
            break;
         default:
            printf("\nVoce nao escolheu uma opcao valida");
            break;
      } /* switch */
   } /* while */

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

   return 0;
}
```

O programa acima representa o esboço de um programa que implementa uma clássica **interação dirigida por menus** (v. **Seção 5.8**. Em linhas gerais, esse programa funciona da seguinte maneira:

1. Um menu de opções é apresentado ao usuário. Essa apresentação do menu de opções é implementada no programa pela instrução **printf**() que segue imediatamente a linha contendo `while (1)`. Não é necessário usar várias instruções **printf**() para apresentar as várias linhas do menu. Isto é, os caracteres `'\n'` que aparecem no string da única instrução **printf**() garantem as quebras de linha necessárias na apresentação do menu. Essa instrução **printf**() contém um único string que é dividido em várias linhas para facilitar a visualização.
2. O programa solicita ao usuário que escolha uma das opções oferecidas. Isso é realizado pelo prompt representado pela instrução **printf**() que segue a apresentação do menu.

3. O programa lê a opção escolhida pelo usuário. A função `LeCaractere()` que segue o prompt mencionado no passo anterior realiza essa tarefa.
4. Se a opção escolhida pelo usuário corresponder àquela especificada como saída do programa, o programa apresenta uma mensagem de despedida e o programa é encerrado. Isso é representado no programa pela instrução **if** que segue a leitura da opção do usuário. Na realidade, a instrução **break** no corpo da instrução **if** não encerra imediatamente o programa. O que **break** faz é encerrar o laço (infinito) **while**, mas, como a instrução que segue **while** é `return 0`, o programa será encerrado logo em seguida.
5. Se a opção escolhida pelo usuário não for aquela que representa saída do programa, verifica-se se ela corresponde a alguma outra opção oferecida pelo programa. A instrução **switch** do programa é responsável por essa tarefa. Ou seja, as partes **case** da instrução **switch** correspondem às opções válidas do programa, enquanto a parte **default** representa a escolha de uma opção inválida. Na prática, quando a opção escolhida pelo usuário é válida as instruções correspondentes à parte **case** com a qual a opção casa são responsáveis pela realização da operação correspondente à opção. Esse exemplo simulado apenas apresenta na tela a opção escolhida pelo usuário.

O uso indiscriminado de desvios incondicionais, como **break**, pode tornar os programas difíceis de ser lidos. Assim esses desvios devem ser usados com cautela. Usualmente, sempre existe uma maneira mais elegante de se escrever um trecho de programa sem o uso de **break**. Exceções a essa regra são o uso de **break** em instruções **switch-case**, conforme visto na Seção 4.6.3, e para encerrar laços de repetição infinitos, como foi visto na Seção 4.5.6 e na presente seção.

4.7.2 continue

Instruções **continue** provocam desvios incondicionais apenas em laços de repetição. Uma instrução **continue** faz com que as instruções que a seguem num laço de repetição sejam saltadas. Mais especificamente, o efeito de uma instrução **continue** em cada laço de repetição é o seguinte:

- Laço **while**. O fluxo de execução continua com a avaliação da expressão condicional do laço **while** e prossegue conforme descrito na Seção 4.5.1 e ilustrado na Figura 4–1.
- Laço **do-while**. O fluxo de execução continua com a avaliação da expressão condicional do laço **do-while** e prossegue conforme descrito na Seção 4.5.2 e ilustrado na Figura 4–2.
- Laço **for**. O fluxo de execução continua com a avaliação da terceira expressão do laço **for** e prossegue conforme descrito na Seção 4.5.3 e ilustrado na Figura 4–3.

Considere o seguinte programa como exemplo da instrução **continue**:

```
#include <stdio.h>

int main(void)
{
   int cont = 0;

   while (cont < 5) {
      cont++;

      if (cont == 3)
         continue;

      printf( "\nExecucao numero %d do corpo do laco", cont );
   }

   return 0;
}
```

O programa acima exibe uma mensagem na tela a cada execução do corpo do laço, exceto na terceira passagem pelo corpo do laço. Isto é, quando o valor da variável `cont` é `3`, a instrução **continue** é executada, o que faz com que a chamada da função **printf()** seja saltada. Assim, o resultado da execução do programa é:

```
Execucao numero 1 do corpo do laco
Execucao numero 2 do corpo do laco
Execucao numero 4 do corpo do laco
Execucao numero 5 do corpo do laco
```

No caso do laço **for**, a instrução **continue** causa o desvio para a avaliação da terceira expressão do laço (e não para a segunda, como se poderia supor). Por exemplo, considere o programa a seguir:

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 1; printf("\nExpressao 2"); printf("\nExpressao 3")){
         /* Na primeira avaliação, i++ vale 1; */
         /* na segunda avaliação, i++ vale  2  */
      if (i++ < 2) {
         continue;
      } else {
         break;
      }
   }
   return 0;
}
```

Quando é executado, esse programa exibe o seguinte na tela:

```
Expressao 2
Expressao 3
Expressao 2
```

Note que a segunda e terceira expressões do laço **for** são chamadas da função **printf()**. Essa função foi apresentada na Seção 3.14.1, mas, naquela ocasião, não foi informado que cada chamada de **printf()** resulta num valor inteiro que representa o número de caracteres escritos na tela. Entretanto, esse valor resultante de uma chamada de **printf()** muito raramente é usado. No último programa, não é necessário saber exatamente qual é o valor resultante de cada chamada de **printf()**, mas, nos dois casos, claramente os valores resultantes são diferentes de zero. Portanto o laço **for** é um laço infinito (pois a segunda expressão é sempre diferente de zero) e seu encerramento ocorre por meio da instrução **break**.

Na Seção 4.5.3, apresentou-se uma sequência de instruções que poderia ser usada em substituição a uma instrução **for**. Naquela ocasião, também alertou-se para o fato de aquela equivalência não ser válida quando o corpo do laço **for** contivesse uma instrução **continue**. Para comprovar o que foi afirmado naquela seção, considere o seguinte programa que foi obtido convertendo-se a instrução **for** do programa anterior de acordo com as recomendações da Seção 4.5.3.

```
#include <stdio.h>

int main(void)
{
   int i;

   i = 1;
   while ( printf("\nExpressao 2") ) {
```

```
         /* Na primeira avaliação, i++ vale 1, na segunda avaliação, i++ vale  2 */
      if (i++ < 2) {
         continue;
      } else {
         break;
      }
      printf("\nExpressao 3");
   }
   return 0;
}
```

Quando o último programa é executado, ele produz como resultado:

```
Expressao 2
Expressao 2
```

Note que os dois programas anteriores produzem resultados diferentes, o que confirma a referida afirmação apresentada na Seção 4.5.3.

A mesma recomendação sugerida ao final da Seção 4.7.1 com relação à instrução **break** é válida para a instrução **continue**: seja comedido no uso dessa instrução, pois seu uso excessivo tende a prejudicar a legibilidade de um programa.

4.7.3 goto

A instrução **goto** é um desvio incondicional mais poderoso do que os demais e assume a seguinte forma:

goto *rótulo;*

Nessa representação esquemática, *rótulo* é um identificador que indica a instrução para a qual o fluxo do programa será desviado. Em outras palavras, para usar **goto**, é necessário ter uma instrução rotulada para onde o desvio será efetuado.

Rotular uma instrução consiste em precedê-la por um identificador seguido de dois pontos. Por exemplo:

```
umRotulo: x = 2*y + 1; /* Esta instrução é rotulada */
```

Em termos de estilo de programação, as convenções usadas para criação de rótulos são as mesmas usadas para nomes de variáveis (v. Seção 3.8). Além disso, recomenda-se manter a endentação da instrução rotulada e retroceder a endentação do rótulo colocando-o na linha anterior à instrução. Por exemplo:

```
while  (x) {
   --x;
   ...
umRotulo: /* O rótulo é recuado em relação à instrução */
   x = 2*y + 1; /* A endentação da instrução é mantida */
   ...
}
```

Qualquer instrução pode ser rotulada, mas só faz sentido rotular uma instrução se ela for usada como alvo de uma instrução **goto**. Outrossim, as seguintes regras devem ser obedecidas:

- Duas instruções não podem possuir o mesmo rótulo numa mesma função (v. Seção 5.10).
- Uma instrução rotulada deve fazer parte do corpo da mesma função que contém a instrução **goto** que a referencia. Em outras palavras, uma instrução **goto**, não pode causar desvio de uma função para outra.

Do mesmo modo que ocorre com os demais desvios incondicionais, o uso abusivo de **goto** é considerado uma má prática de programação (v. Seção 6.7).

4.7.4 Uso de break e continue em Laços Aninhados

Um laço de repetição é **aninhado** quando ele faz parte do corpo de outro laço de repetição. No caso de laços aninhados, as instruções **break** e **continue** têm efeito apenas no laço que *imediatamente* as contém. Isso que dizer que, se um laço interno contém uma instrução **break** (ou **continue**), essa instrução não tem nenhum efeito sobre o laço externo. Considere, por exemplo, o seguinte trecho de programa:

```
while (x > 10) {
   do {
      ...
      if (z == 0)
         break;
      ...
   } while (y <= 15);
   ...
}
```

Nesse exemplo, a execução da instrução **break** termina o laço **do-while** interno, mas não encerra o laço **while** externo. Raciocínio análogo aplica-se ao uso de **continue** em laços aninhados. Idem para aninho entre laços de repetição e **switch-case**, no caso de **break**.

A instrução **goto** não sofre influência de aninhos de quaisquer estruturas de controle em qualquer profundidade. Isto é, mesmo que uma instrução **goto** faça parte do corpo de uma estrutura de controle aninhada, pode-se desviar para qualquer instrução que esteja na mesma função que contém a instrução **goto**. Não custa ainda repetir que nenhuma instrução é capaz de causar desvio entre funções.

4.7.5 Instruções Inacessíveis

Por definição, um desvio incondicional causa alteração na ordem de execução de instruções *independentemente* da avaliação de qualquer expressão. Entretanto, na prática, desvios incondicionais devem sempre fazer parte do corpo de um desvio condicional. Além disso, um desvio incondicional não deve ser seguido por nenhuma instrução que tenha o mesmo nível de subordinação do mesmo desvio (ou, visualmente, nenhuma instrução com a mesma endentação de um desvio incondicional deve segui-lo). Se essas duas regras não forem seguidas, as instruções que seguem um desvio incondicional nunca serão executadas e, portanto, podem ser removidas do programa que as contém. Considere, por exemplo, o seguinte programa:

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

int main(void)
{
   int opcao;

   while (1) {
      printf("\nDigite 'E' para encerrar o programa");
      opcao = LeCaractere();

      if (opcao == 'E' || opcao == 'e') {
         break;
         printf("\nAdeus"); /* Instrução inacessível */
      }

      printf("\nO programa continua");
   }

   return 0;
}
```

Se você compilar esse programa, o compilador poderá emitir a seguinte mensagem de advertência:

```
Warning: Unreachable code in function main
```

e indicar a instrução:

```
printf("\nAdeus");
```

como fonte dessa mensagem.

Essa mensagem de advertência significa que a instrução `printf("\nAdeus")` jamais será executada, sendo, assim, considerada **código morto**. Alguns compiladores até mesmo assumem a liberdade de não incluir a tradução de instruções que nunca serão executadas no programa-objeto resultante de uma compilação.

A discussão apresentada nesta seção também se aplica ao caso de instruções **return**, que serão estudadas no Capítulo 5.

4.8 Operador Condicional

O operador condicional é o único **operador ternário** da linguagem C e é representado pelos símbolos **?** e **:**. Esse operador aparece no seguinte formato:

$$operando_1 \; ? \; operando_2 \; : \; operando_3$$

O primeiro operando representa tipicamente uma expressão condicional, enquanto o segundo ou o terceiro operando representa o resultado do operador condicional, dependendo do valor do primeiro operando. Mais precisamente, o resultado da expressão será $operando_2$ quando $operando_1$ for diferente de zero e $operando_3$ quando $operando_1$ for zero.

O operador condicional representa uma abreviação para instruções **if-else** de um formato específico, como mostrado no seguinte exemplo:

```
if (x)
   w = y;
else
   w = z;
```

A instrução **if** desse exemplo pode ser substituída por:

```
w = x ? y : z;
```

Existem muitos exemplos de uso prático do operador condicional neste livro — o primeiro deles é apresentado na Seção 4.11.8.

Assim como os operadores **&&** e **||**, a ordem de avaliação de operandos do operador condicional é bem definida: o primeiro operando é *sempre* avaliado em primeiro lugar, em seguida é avaliado o segundo ou o terceiro operando, de acordo com o resultado da avaliação do primeiro operando.

A precedência do operador condicional é maior apenas do que a do operador de atribuição (v. Seção 3.9) e do operador vírgula (v. Seção 4.9) e sua associatividade é à direita. Como esse é o único operador em seu grupo de precedência, associatividade só é utilizada quando existe mais de uma ocorrência desse operador numa mesma expressão. Nesse caso, a expressão torna-se difícil de ser compreendida. Por exemplo, você seria capaz descobrir qual é o valor atribuído à variável `x` no trecho de programa a seguir?

```
int i = 5, j = 1, k = 2, x;
x = i < j ? k >= i ? j - 1 : j : i ? j : k;
```

Se você não for capaz de responder a questão acima, deve se convencer que é melhor evitar mais de um uso do operador condicional numa mesma expressão. (A propósito, o valor assumido por `x` no trecho de programa em questão é `1`.)

4.9 Operador Vírgula

O operador vírgula é um operador que permite a inclusão de mais de uma expressão em locais onde uma única expressão seria naturalmente permitida. O resultado desse operador é o operando da direita; i.e., o operando esquerdo não contribui para o resultado. Por causa disso, o uso desse operador só faz sentido quando o primeiro operando é uma expressão contendo um operador que causa efeito colateral. Vírgulas também podem ser usadas em chamadas de funções e em declarações e definições de componentes de um programa em C. Mas, nesses casos, a vírgula é considerada um **separador**, e não um operador.

O operador vírgula tem a mais baixa precedência dentre todos os operadores de C e sua associatividade é à esquerda. Esse operador possui ordem de avaliação de operandos bem definida: primeiro o operando da esquerda é avaliado e, em seguida, o mesmo ocorre com o operando da direita.

O uso mais comum do operador vírgula ocorre quando se precisa utilizar mais de uma expressão no lugar da primeira ou terceira expressões de um laço **for** (v. Seção 4.5.3). Por exemplo:

```
#include <stdio.h>

int main(void)
{
   int x, y;

   for (x = 0, y = 10; x < y; x++, y--) {
      ; /* Instrução vazia */
   }

   printf("x = %d, y = %d", x, y);

   return 0;
}
```

A primeira expressão do laço **for** desse programa é constituída pelas subexpressões `x = 0` e `y = 10` unidas pelo operador vírgula. Nesse mesmo programa, o operador vírgula também une as subexpressões `x++` e `y--` para formar a terceira expressão do laço **for**. Nos dois casos de uso do operador vírgula no último exemplo, os resultados das aplicações desse operador são desprezadas. Tipicamente, na prática, o resultado de aplicação desse operador é abandonado.

Uma dúvida que pode afligir alguns leitores nesse instante é a seguinte: *se o resultado da expressão não interessa na maioria das vezes, por que não usar outro operador qualquer?* Existem duas justificativas principais para uso do operador vírgula nessas situações:

[1] O fato de o operador vírgula possuir a menor precedência da linguagem C implica em raramente haver necessidade de uso de parênteses.

[2] O fato de o operador vírgula ter ordem de avaliação de operandos definida permite que ele seja usado com operandos que sofrem efeitos colaterais sem o risco de incorrer numa expressão com resultado dependente de implementação (v. Seção 3.9). Por exemplo, após a avaliação da expressão

`(i = 1) , (j = ++i)`, sendo `i` e `j` variáveis do tipo **int**, ambas as variáveis receberão `2` como valor (e não há outro resultado possível).

Em situações diferentes daquela sugerida acima, é melhor evitar o uso do operador vírgula e, ao invés disso, escrever as expressões em instruções separadas. O uso desse operador muitas vezes prejudica a legibilidade do programa e seu uso deve ser evitado na maioria dos casos.

4.10 Geração de Números Aleatórios

Números aleatórios são valores gerados ao acaso, tais como os números de um sorteio. Geração de números aleatórios é útil em diversas situações práticas de programação, conforme será visto em vários exemplos apresentados ao longo deste livro (o primeiro deles é apresentado na **Seção 4.11.8**).

Em C, números aleatórios são gerados utilizando-se a função **rand()**, que faz parte do módulo stdlib da biblioteca padrão. Portanto, para usar essa função, é necessário incluir o cabeçalho `<stdlib.h>`.

A função **rand()** gera números aleatórios inteiros entre zero e um valor definido pela implementação da biblioteca de C utilizada. Para geração de números aleatórios entre `M` e `N`, sendo `M < N`, deve-se usar a fórmula:

```
rand()%(N - M + 1) + M
```

Um gerador de número aleatório, como é o caso da função **rand()**, usa um valor inicial, denominado **semente**, do qual dependem todos os valores gerados. Assim, se uma mesma semente for usada em duas ocasiões, os mesmos números serão gerados em cada ocasião e, por isso, os valores gerados deixam de ser aleatórios. Logo, para gerar sequências diferentes de números em diferentes ocasiões, o gerador de números aleatórios precisa usar sementes diferentes.

A função **srand()** é usada para **alimentar** o gerador de números aleatórios atribuindo-lhe uma semente diferente daquela usada como padrão. A maneira mais comum de alimentar o gerador com um valor que dificilmente é repetido e, consequentemente, garantir que uma nova sequência de números aleatórios é sempre gerada, é por meio de uso da função **time()**.

A função **time()** é declarada no cabeçalho `<time.h>` da biblioteca padrão e retorna o número de segundos decorridos desde uma época de referência, que, usualmente, é 00:00:00 (GMT) do dia 1° de janeiro de 1970.

Para alimentar o gerador de números aleatórios com o valor retornado pela função **time()**, a função **srand()** é chamada como:

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

Na **Seção 4.11.8**, será apresentado um exemplo de uso das funções **rand()** e **srand()**.

4.11 Exemplos de Programação

4.11.1 Inteiros Divisíveis por 2 ou 3

Problema: Escreva um programa que exibe na tela os números inteiros entre `1` e `20` que são divisíveis por `2` ou `3`.

Solução: O algoritmo aparece na **Figura 4–10** e o programa bem logo em seguida.

Algoritmo DivisíveisPor2Ou3

Entrada: Não há

Saída: Mensagem informando se um número inteiro é par ou ímpar

```
inteiro i

i ← 1

enquanto (i ≤ 20) faça
  se (i%2 = 0 ou i%3 = 0) então
    escreva(i)

  i ← i + 1
```

Figura 4–10: Algoritmo de Divisão por 2 ou 3

```
#include <stdio.h>

/****
 *
 * main():Exibe na tela os valores entre 1 e 20 que são divisíveis por 2 ou 3
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int i;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa exibe os valores entre 1 e"
           "\n\t>>> 20 que sao divisiveis por 2 ou 3.\n\n" );

      /* Verifica se cada valor entre 1 e 20 é divisível por 2 ou por 3 */
   for (i = 1; i <= 20; ++i) {
      if ( i%2 == 0 || i%3 == 0 )
         printf("\t%d\n", i);
   }

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

   return 0;
}
```

Análise: Um número inteiro é divisível por `2` ou por `3` quando o resto da divisão dele por `2` ou o resto da divisão dele por `3` é igual a `0`. Ou seja, um número inteiro `n` é divisível por `2` ou `3` quando a seguinte expressão lógica é satisfeita (i.e., resulta em `1`):

```
i%2 == 0 || i%3 == 0
```

Portanto, para apresentar na tela os números inteiros entre `1` e `20`, basta testar quais deles satisfazem essa expressão, e, quando for o caso, exibir cada um deles.

Resultado de execução do programa:

```
>>> Este programa exibe os valores entre 1 e
>>> 20 que sao divisiveis por 2 ou 3.

2
3
4
6
8
9
10
12
14
15
16
18
20

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

Exercício: O programa acima contém diversos números mágicos (v. **Seção 6.5**). Identifique-os e substitua-os adequadamente por constantes simbólicas.

4.11.2 Pares ou Ímpares?

Problema: Escreva um programa que apresenta na tela os números inteiros pares ou ímpares entre `1` e `20` em formato de tabela.

Solução: O algoritmo aparece na **Figura 4–11** e o programa vem logo em seguida.

ALGORITMO PARESOUÍMPARES

ENTRADA: Não há

SAÍDA: Mensagem informando se um número inteiro é par ou ímpar

```
inteiro i

escreva("Par   Impar\n")

i ← 1

enquanto (i ≤ 20) faça
  se (i%2 = 0) então
    escreva(" ", i) /* Escreve na primeira coluna */
  senão
    escreva("    ", i) /* Escreve na segunda coluna */

  i ← i + 1
```

FIGURA 4–11: ALGORITMO DE DETERMINAÇÃO DE PARES OU ÍMPARES

```
#include <stdio.h>

/****
 * main(): Escreve na tela os números pares e ímpares entre 1 e 20 em forma de tabela
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
```

```
int main(void)
{
   int i;

      /* Apresenta o programa */
   printf("\n\t>>> Este programa exibe os numeros pares e impares entre 1 e 20.\n");

      /* Apresenta o cabeçalho da tabela */
   printf("Par\tImpar\n");
   printf("===\t=====\n\n"); /* Embelezamento */

      /* Verifica se cada número entre 1 e 20 é par */
      /* ou ímpar, e apresenta-o na devida coluna   */
   for (i = 1; i <= 20; ++i) {
      if (i%2 == 0) { /* O número é par */
         printf(" %2d\n", i);
      } else { /* O número é ímpar */
         printf("\t  %2d\n", i);
      }
   }

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

   return 0;
}
```

Análise: Esse programa é tão simples que não requer mais comentários além daqueles que se encontram no próprio programa.

Resultado de execução do programa:

```
        >>> Este programa exibe os numeros
        >>> pares e impares entre 1 e 20.

Par     Impar
===     =====

           1
  2
           3
[Trecho removido]
 18
          19
 20
```

Exercício: Identifique e substitua os números mágicos (v. **Seção 6.5**) do programa acima por constantes simbólicas.

4.11.3 Conversão de Celsius para Fahrenheit

Problema: Escreva um programa que exiba na tela uma tabela de conversão de graus Celsius para Fahrenheit. A temperatura em Celsius deve variar de cinco em cinco graus entre 0^{o} e 100^{o}.

Solução: O algoritmo aparece na **Figura 4–12** e o programa vem logo em seguida.

Algoritmo ConverteCelsiusEmFahrenheit

Entrada: Não há
Saída: Tabela de conversão de Celsius para Fahrenheit

```
inteiro c
escreva("Celsius   Fahrenheit")
c ← 0
enquanto (c ≤ 100) faça
  escreva("  ", c , "       ", 1.8*c + 32)
  c ← c + 5
```

Figura 4–12: Algoritmo de Conversão de Celsius para Fahrenheit

```
#include <stdio.h>

/****
 * main(): Apresenta na tela uma tabela de conversão de graus Celsius para Fahrenheit
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 * ****/
int main(void)
{
   int c; /* Temperatura em Celsius */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa cria uma tabela de conversao"
           "\n\t>>> de Celsius para Fahrenheit\n" );

      /* Apresenta o cabeçalho da tabela com uma linha de embelezamento */
   printf("\nCelsius\tFahrenheit"
          "\n=======\t==========\n");

      /* A fórmula de transformação de graus Celsius em graus Fahrenheit é: */
      /* F = 1.8*C + 32, na qual F é a temperatura em Fahrenheit e C é      */
      /* a temperatura em Celsius.                                          */
   for (c = 0; c <= 100; c = c + 5) {
         /* Observe o efeito dos especificadores */
         /* de formato de printf() na exibição   */
      printf( "\n  %3d\t  %5.1f", c, 1.8*c + 32 );
   }

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

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

   return 0;
}
```

Análise: Leia os comentários inserido no programa, pois eles devem ser suficientes para seu entendimento.

Resultado de execução do programa:

```
        >>> Este programa cria uma tabela de conversao
        >>> de Celsius para Fahrenheit

Celsius Fahrenheit
======= ==========

    0       32.0
    5       41.0

[Trecho removido]

   95      203.0
  100      212.0

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

4.11.4 Cálculo de Dívidas

Problema: Escreva um programa que calcula dívidas usando juros compostos decorrentes de um empréstimo. A fórmula utilizada deve ser:

$$dívida = quantia \times (1 + juros)^{período}$$

Solução: O algoritmo aparece na Figura 4–13 e o programa vem logo em seguida.

Algoritmo CálculoDeDívidas

Entrada: Não há

Saída: Tabela de conversão de Celsius para Fahrenheit

```
inteiro período
real    juros, quantia, dívida

escreva("Quanto você tomou emprestado (R$)?")
leia(quantia)

escreva("Quanto e' a taxa de juros (%)?")
leia(juros)

juros ← juros/100

período ← 1

enquanto (período ≤ 10) faça
  divida = quantia*pow(1 + juros, período)

  escreva("Apos", período ,"anos você deverá: R$", divida)

  período ← período + 1
```

Figura 4–13: Algoritmo de Cálculo de Dívidas

Análise: Esse algoritmo usa a função pow que efetua operações de potenciação (v. Seção 2.5.5).

```
#include <stdio.h>   /* printf() */
#include <math.h>    /* pow()    */
#include "leitura.h" /* LeReal() */

/****
 * main(): Calcula dívidas usando juros compostos
 *
 * Parâmetros: Nenhum
```

```
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int    periodo;
   double juros, quantia, divida;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa calcula a divida do usuario que fez"
           "\n\t>>> um emprestimo usando a formula de juros compostos.\n\n" );

      /* Lê a quantia que usuário tomou emprestado */
   printf("\nQuanto voce tomou emprestado (R$)? ");
   quantia = LeReal();

      /* Lê a taxa de juros do empréstimo em percentagem. Note */
      /* como o símbolo '%' é escrito no string de formatação. */
   printf("\nQuanto e' a taxa de juros (%%)? ");
   juros = LeReal();

      /* Calcula os juros sem percentagem */
   juros = juros/100;

      /* Apresenta o prejuízo do usuário a cada ano */
   for (periodo = 1; periodo <= 5; ++periodo) {
      divida = quantia*pow(1 + juros, periodo);
      printf( "\nApos %d anos voce devera': R$%5.2f", periodo, divida );
   }

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

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

   return 0;
}
```

Análise: Leia os comentários do programa, pois eles são suficientes para seu entendimento.

Exemplo de execução do programa:

```
        >>> Este programa calcula a divida do usuario que fez
        >>> um emprestimo usando a formula de juros compostos.

Quanto voce tomou emprestado (R$)? 1000

Quanto e' a taxa de juros (%)? 31

Apos 1 anos voce devera': R$1310.00
Apos 2 anos voce devera': R$1716.10
Apos 3 anos voce devera': R$2248.09
Apos 4 anos voce devera': R$2945.00
Apos 5 anos voce devera': R$3857.95

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

4.11.5 Triângulos

Problema: Escreva um programa que verifica se três valores introduzidos pelo usuário podem constituir os lados de um triângulo.

Solução: O algoritmo aparece na Figura 4–14 e o programa vem logo em seguida.

ALGORITMO VERIFICATRIÂNGULOS

ENTRADA: Não há

SAÍDA: Tabela de conversão de Celsius para Fahrenheit

```
real x, y, z

escreva("Digite o primeiro valor:")
leia(x)

escreva("Digite o segundo valor:")
leia(y)

escreva("Digite o terceiro valor:")
leia(z)

se ( x ≤ 0 ou y ≤ 0 ou z ≤ 0 ) então
  escreva("Pelo menos um valor não é positivo")
senão se ( x < y + z e y < x + z e z < y + x ) então
  escreva("Os valores constituem os lados de um triangulo")
senão
  escreva("Os valores NÃO constituem os lados de um triangulo")
```

FIGURA 4–14: ALGORITMO DE VERIFICAÇÃO DE TRIÂNGULOS

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

/****
 * main(): Verifica se três valores reais constituem os lados de um triângulo
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   double x, y, z;

   printf( "\n>>> Este programa verifica se tres "
           "\n>>> valores reais podem constituir "
           "\n>>> os lados de um triangulo.\n\n" );

   /*** Lê os valores candidatos a lados de um triângulo ***/

   printf("\nDigite o primeiro valor: ");
   x = LeReal();

   printf("\nDigite o segundo valor: ");
   y = LeReal();

   printf("\nDigite o terceiro valor: ");
   z = LeReal();

      /* Verifica se os valores introduzidos satisfazem a desigualdade triangular */
   if (x <= 0 || y <= 0 || z <= 0) {
      printf("\nPelo menos um valor nao e' positivo.\n");
   } else if ( x < y + z && y < x + z && z < y + x ) {
      printf( "\nOs valores constituem os lados de um triangulo.\n" );
   } else {
```

```
        printf( "\nOs valores NAO constituem os lados de um triangulo.\n" );
    }

    return 0;
}
```

Análise: Três números podem constituir os lados de um triângulo se qualquer um deles é menor do que a soma dos demais (desigualdade triangular). Ou seja, se os números forem representados por `x`, `y` e `z`, eles formarão um triângulo se a expressão lógica:

```
x < y + z && y < x + z && z < y + x
```

resultar em `1`.

Exemplo de execução do programa:

```
>>> Este programa verifica se tres
>>> valores reais podem constituir
>>> os lados de um triangulo.

Digite o primeiro valor: 3
Digite o segundo valor: 4
Digite o terceiro valor: 5
Os valores constituem os lados de um triangulo.
```

4.11.6 Anos Bissextos

Preâmbulo: Um ano é bissexto quando:

- É maior do que ou igual a `1753`, que é o ano no qual anos bissextos começaram a ser levados em consideração e
- Ele é múltiplo de `400` (p. ex., `1600`, `2000`) ou
- Ele é múltiplo de `4`, mas não é múltiplo de `100` (p. ex., `1992`, `1996`)

Problema: Escreva um programa que lê valores inteiros introduzidos pelo usuário, supostamente representando anos, e verifica se trata-se de um ano bissexto.

Solução: O algoritmo aparece na Figura 4–15 e o programa vem logo em seguida.

ALGORITMO ANOSBISSEXTOS

ENTRADA: Não há
SAÍDA: Tabela de conversão de Celsius para Fahrenheit

```
inteiro a

enquanto (verdadeiro) faça
  escreva("Digite um ano maior do que ou igual a 1753 (0 encerra o programa)")
  leia(a)

  se (a = 0)
    pare
  se (a < 1753) então
    escreva("O ano deve ser maior do que ou igual a 1753")
  senão se (a%400 = 0 ou (a%4 = 0 e a%100 ≠ 0))
    escreva("O ano é bissexto")
  senão
    escreva("O ano não é bissexto")
```

FIGURA 4–15: ALGORITMO DE CÁLCULO DE ANOS BISSEXTOS

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

   /* Anos bissextos tais como são conhecidos hoje */
   /* começaram a ser contados a partir de 1753    */
#define MENOR_ANO 1753
/****
 * main(): Verifica se cada número inteiro introduzido
 *         pelo usuário representa um ano bissexto
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int a;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa verifica se cada numero inteiro introduzido"
           "\n\t>>> pelo usuario representa um ano bissexto.\n"
           "\n\t>>> Digite zero para encerrar o programa.\n" );

      /* Lê cada valor introduzido pelo usuário e informa se trata-se de um  */
      /* ano bissexto ou não.  O laço encerra quando o usuário digitar zero. */
   while (1) {
         /* Faz a leitura */
      printf("\n\t>>> Digite um ano maior do que ou igual\n"
             "\t>>> a %d (0 encerra o programa) > ", MENOR_ANO);
      a = LeInteiro();

      if (!a) { /* Se a == 0, encerra o laço */
         break;
      }

      if (a < MENOR_ANO) {
         printf("\n\t>>> O ano deve ser maior do que ou igual a %d\n", MENOR_ANO);
      } else {
            /* Informa se o ano é bissexto ou não */
         printf("\n\t>>> O ano %se' bissexto\n",
                !(a%400) || (!(a%4) && a%100) ? "" : "NAO ");
      }
   }

   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Análise:

- A variável **ano** representa o valor que se deseja testar se trata-se de um ano bissexto. Logo:
  1. O fato de o ano ser múltiplo de **400** é representado pela 'expressão:

     ```
     ano%400 == 0 ano%400 == 0
     ```

     que equivale a:

     ```
     !(ano%400)
     ```

  2. O fato de o ano ser múltiplo de **4**, mas não ser múltiplo de **100** é representado pela expressão:

```
ano%4 == 0 && ano%100 != 0
```

que equivale a:

```
!(ano%4) && ano%100
```

3. A expressão que determina se um ano é bissexto é a disjunção das duas últimas expressões:

```
!(ano%400) || (!(ano%4) && ano%100)
```

- O programa usa uma constante simbólica (v. **Seção 3.15**) definida como:

```
#define MENOR_ANO 1753
```

Essa constante representa o ano a partir do qual os anos bissextos tais como são conhecidos hoje começaram a ser levados em consideração.

- O restante do programa é auto-explicativo.

Exemplo de execução do programa:

```
>>> Este programa verifica se cada numero inteiro introduzido
>>> pelo usuario representa um ano bissexto.

>>> Digite zero para encerrar o programa.

>>> Digite um ano maior do que ou igual
>>> a 1753 (0 encerra o programa) > 1500

>>> O ano deve ser maior do que ou igual a 1753

>>> Digite um ano maior do que ou igual
>>> a 1753 (0 encerra o programa) > 1996
>>> O ano e' bissexto

>>> Digite um ano maior do que ou igual
>>> a 1753 (0 encerra o programa) > 0

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

4.11.7 Maior Valor do Tipo int

Preâmbulo: Em programação, overflow de um determinado tipo de dado ocorre quando se tenta armazenar um valor grande demais para ser contido numa variável desse tipo. Por exemplo, o maior valor que pode ser armazenado numa variável do tipo **int** é determinado pela constante **INT_MAX**, definida no cabeçalho `<limits.h>` da biblioteca padrão de C. Assim, somando-se `1` a esse valor máximo resulta em overflow do tipo **int**.

Problema: Escreva um programa que determina qual é o valor da constante **INT_MAX** descrita no preâmbulo.

Solução: O algoritmo aparece na **Figura 4–16** e o programa vem logo em seguida.

ALGORITMO MAIORVALORINT

ENTRADA: Não há
SAÍDA: O maior valor do tipo **int**

```
inteiro maiorInt
maiorInt ← 0
enquanto (maiorInt + 1 > maiorInt) faça
  maiorInt ← maiorInt + 1
escreva("Maior valor do tipo int = ", maiorInt)
```

FIGURA 4–16: ALGORITMO DE MAIOR VALOR DO TIPO INT

```
#include <stdio.h>   /* Entrada e saída  */
#include <limits.h>  /* Limites inteiros */
/****
 * main(): Determina qual é o maior valor do tipo int e mostra
 *         como detectar overflow numa operação inteira
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int maiorInt = 0;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa determina qual e' o maior"
           "\n\t>>> valor do tipo int e mostra como detectar"
           "\n\t>>> overflow numa operacao inteira.\n" );

      /* Apresenta uma mensagem para o usuário saber que o programa está vivo */
   printf("\nEncontrando o maior valor do tipo int...");

      /* Encontra o maior valor do tipo int. Seja paciente e não digite */
      /* [CTRL] + [C], esta operação leva um tempo considerável.        */
   while (maiorInt + 1 > maiorInt) {
      maiorInt++;
   }

      /* Os valores exibidos a seguir devem ser iguais. */
      /* Caso contrário, o programa está incorreto.     */
   printf("\n\nMaior valor do tipo int = %d", maiorInt);
   printf("\n                INT_MAX = %d\n", INT_MAX);

   return 0;
}
```

Análise: Esse programa é bem curto, se comparado aos demais apresentados neste capítulo, mas merece algumas explicações.

- Esse programa demora um tempo considerável entre sua apresentação inicial até que ele exiba o resultado esperado na tela (no exemplo de interação abaixo, o corpo do laço **while** foi executado mais de duas bilhões de vezes!). Portanto é de bom tom que o programa informe o usuário que processará algo durante um longo tempo. É isso que faz a segunda chamada de **printf()** do programa.
- Em Matemática, se `x` é um número inteiro positivo, é obvio que `x + 1` sempre será maior do que `x`. Mas, em computação, isso só ocorre se o valor resultante da expressão `x + 1` couber numa variável do tipo da variável `x`. Quando, na avaliação dessa expressão, o resultado não é maior do que `x`, pode-se concluir que ocorreu overflow. O que o laço **while** do programa em discussão faz é incrementar a variável `maiorInt` até que a expressão `maiorInt + 1 > maiorInt` deixe de ser válida. Quando isso ocorre, a variável `maiorInt` terá atingido seu limite máximo, que é o valor procurado.
- Além de exibir na tela o valor da variável `maiorInt`, o programa apresenta o valor da constante **INT_MAX**, definida no cabeçalho `<limits.h>`. O objetivo da apresentação dessa constante é apenas conferir o resultado (i.e., verificar se o programa realmente produz o resultado esperado).
- O objetivo desse programa é meramente pedagógico. Ou seja, na prática, se precisar usar o valor máximo que uma variável do tipo **int** pode assumir, inclua o cabeçalho `<limits.h>` e use a constante **INT_MAX** definida nele.

Exemplo de execução do programa: Quando compilado usando GCC no Windows, o resultado do programa é o seguinte:

```
        >>> Este programa determina qual e' o maior
        >>> valor do tipo int e mostra como detectar
        >>> overflow numa operacao inteira.

Encontrando o maior valor do tipo int...

Maior valor do tipo int = 2147483647
                INT_MAX = 2147483647
```

4.11.8 Sorteando Bônus para o Cliente

Problema: Escreva um programa que sorteia um bônus entre `1` e `10` reais para o usuário, de acordo com um valor inteiro introduzido por ele.

Solução: O algoritmo aparece na Figura 4–17 e o programa vem logo em seguida.

Algoritmo SorteioDeBônus

Entrada: Não há
Saída: Tabela de conversão de Celsius para Fahrenheit

```
inteiro sorte, bonus

constante MENOR_BONUS = 1
constante MAIOR_BONUS = 10

escreva("Digite seu numero de sorte")
leia(sorte)

srand

bonus ← rand%MAIOR_BONUS + MENOR_BONUS

escreva("Parabens! Voce acaba de ganhar R$", bonus)
```

Figura 4–17: Algoritmo de Sorteio de Bônus

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

#define MENOR_BONUS 1
#define MAIOR_BONUS 10

/****
 * main(): Sorteia para o usuário um bônus em reais entre MENOR_BONUS e MAIOR_BONUS
 *         de acordo com um valor inteiro introduzido como entrada
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int sorte, bonus;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa sorteia um premio"
           "\n\t>>> entre R$%d e R$%d. Boa sorte!\n", MENOR_BONUS, MAIOR_BONUS );
```

```
   printf("\n\t>>> Digite seu numero de sorte > ");
   sorte = LeInteiro();

      /* Alimenta o gerador de números aleatórios com o número de sorte do usuário */
   srand(sorte);

      /* Efetua o sorteio */
   bonus = rand()%(MAIOR_BONUS - MENOR_BONUS + 1) + MENOR_BONUS;

   printf( "\a\n\t>>> Parabens! Voce acaba de ganhar %d "
           "rea%s!!!\n", bonus, bonus == 1 ? "l" : "is");

   return 0;
}
```

Análise: O programa funciona da seguinte maneira:

1. Um valor inteiro introduzido pelo usuário é lido pela função `LeInteiro()`.
2. O número introduzido pelo usuário é usado para alimentar o gerador de números aleatórios [que é a função **rand**()]. A chamada da função **srand**() realiza essa tarefa.
3. Um número aleatório é gerado e transformado num valor entre `MENOR_BONUS` e `MAIOR_BONUS` que será o bônus ganho pelo usuário. A fórmula utilizada para cálculo do bônus é aquela apresentada na Seção 4.10.
4. O programa apresenta o resultado do sorteio para o usuário. Isso é realizado pela última instrução **printf**() do programa.

Na última instrução **printf**(), aparece o uso do operador condicional como:

```
bonus == 1 ? "l" : "is"
```

De acordo com o que foi apresentado na descrição do operador condicional o resultado dessa expressão será o string `"l"` quando o resultado da expressão `bonus == 1` for diferente de zero e será o string `"is"` quando esse resultado for zero. Agora, o resultado da expressão composta pelo operador condicional é o string que será exibido em substituição ao especificador de formato **%s** que aparece no string de formatação de **printf**(). Assim, quando o valor da variável `bonus` for `1`, a chamada de **printf**() escreverá `"real"`; caso contrário, ela escreverá `"reais"`. Sem o uso do operador condicional, seria necessário o uso de uma instrução **if-else** e duas chamadas de **printf**().

Exemplo de execução do programa:

```
>>> Este programa sorteia um premio
>>> entre R$1 e R$10. Boa sorte!

>>> Digite seu numero de sorte > 21

>>> Parabens! Voce acaba de ganhar 8 reais!!!
```

4.11.9 Números Iguais às Somas dos Cubos de seus Dígitos

Problema: Escreva um programa que encontra os números entre `100` e `999` que são iguais às somas dos cubos de seus dígitos.

Solução: O algoritmo aparece na Figura 4–18 e o programa vem logo em seguida.

Algoritmo SomaDeCubosDeDígitos

Entrada: Não há
Saída: Tabela de conversão de Celsius para Fahrenheit

```
inteiro numero, centena, dezena, unidade, somaCubos

centena ← 1

enquanto (centena ≤ 9) faça
  dezena ← 0

  enquanto (dezena ≤ 9) faça
    unidade ← 0

    enquanto (unidade ≤ 9) faça
      somaCubos ← centena*centena*centena +
                  dezena*dezena*dezena +
                  unidade*unidade*unidade
      numero ← 100*centena + 10*dezena + unidade

      se (numero = somaCubos) então
        escreva(numero)
      unidade ← unidade + 1
    dezena ← dezena + 1
  centena ← centena + 1
```

Figura 4–18: Algoritmo de Soma de Cubos de Dígitos

```
#include <stdio.h>

/****
 * main(): Encontra os números entre 100 e 999 que são
 *         iguais às somas dos cubos de seus dígitos
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int numero,  /* O número a ser testado */
       centena, /* A centena do número */
       dezena,  /* A dezena do número  */
       unidade, /* A unidade do número */
       somaCubos; /* A soma dos cubos de centena, dezena e unidade */

      /* Apresenta o programa */
   printf( "\n\t>>> Este encontra os numeros entre 100"
           "\n\t>>> e 999 que sao iguais as somas dos"
           "\n\t>>> cubos de seus digitos:\n" );

      /* Considera cada trinca constituída de centena, dezena e unidade que */
      /* compõe um número inteiro entre 100 e 999 e verifica se a soma dos  */
      /* componentes da trinca é igual ao número formado por ela.           */

   for(centena = 1; centena <= 9; centena++) {
      for(dezena = 0; dezena <= 9; dezena++) {
         for(unidade = 0; unidade <= 9; unidade++) {
```

```
            /* Calcula a soma dos cubos dos componentes da trinca */
            somaCubos = centena*centena*centena + dezena*dezena*dezena +
                        unidade*unidade*unidade;

            /* Determina qual é o número formado pela trinca */
            numero = 100*centena + 10*dezena + unidade;

            /* Verifica se o número é igual à soma dos componentes da trinca */
            if(numero == somaCubos) {
                printf("\n\t>>> %d", numero);
            } /* if */
        } /* for */
    } /* for */
} /* for */

return 0;
}
```

Resultado de execução do programa:

```
>>> Este encontra os numeros entre 100
>>> e 999 que sao iguais as somas dos
>>> cubos de seus digitos:

>>> 153
>>> 370
>>> 371
>>> 407
```

Exercício: Identifique e substitua os números mágicos (v. **Seção 6.5**) do programa acima por constantes simbólicas.

4.11.10 Exponenciação

Problema: Escreva um programa que calcula potenciações de números reais elevados a números inteiros. O programa deve encerrar quando um valor indefinido for produzido.

Solução: O algoritmo aparece na **Figura 4–19** e o programa vem logo em seguida.

ALGORITMO EXPONENCIAÇÃO

ENTRADA: Não há
SAÍDA: Tabela de conversão de Celsius para Fahrenheit

```
inteiro exp, i
real base, resultado

enquanto (verdadeiro)
  escreva("Digite o valor da base:")
  leia(base)

  escreva("Digite o valor do expoente:")
  leia(exp)

  resultado ← 1

  se (base = 0 e exp ≤ 0) então
    escreva("Valor indefinido")
    pare
```

CONTINUA

FIGURA 4–19: ALGORITMO DE EXPONENCIAÇÃO

Algoritmo Exponenciação (Continuação)


```
se (exp = 0) então
  escreva("Resultado: 1")
se (exp > 0) então
  i ← 1

  enquanto (i ≤ exp) faça
    resultado ← resultado*base

    i ← i + 1

 escreva("Resultado:", resultado)
senão
  exp ← -exp

  i ← 1
  enquanto (i ≤ exp) faça
    resultado ← resultado*base
    i ← i + 1

  escreva("Resultado:", 1/resultado)
```

Figura 4–19 (Cont.): Algoritmo de Exponenciação

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

/****
 * main(): Calcula potenciações de números reais elevados a números inteiros
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int    exp, i;
   double base, resultado;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa calcula potenciacoes de numeros reais"
           "\n\t>>> elevados a numeros inteiros. O programa encerra"
           "\n\t>>> quando o resultado for indefinido.\n" );

      /* O laço encerra quando o resultado for indefinido */
   while (1) {
         /* Lê o valor da base */
      printf( "\n\n\t>>> Digite o valor da base: " );
      base = LeReal();

         /* Lê o valor do expoente */
      printf( "\t>>> Digite o valor do expoente: " );
      exp = LeInteiro();

      resultado = 1; /* Inicia a variável que armazenará o resultado */

         /* Se a base for zero e o expoente for menor do  */
         /* que ou igual a zero, o resultado é indefinido */
```

```
      if (!base && exp <= 0) {
         printf("\n\t>>> Resultado indefinido");
         break; /* Encerra o laço */
      } else if (!exp) {
            /* Se a base não for zero e o expoente for zero o resultado é 1 */
         printf("\n\t>>> Resultado: 1");
      } else if (exp > 0) {
            /* Se a base não for zero e o expoente for positivo, multiplica */
            /* a base o número de vezes representado pelo expoente          */
         for (i = 1; i <= exp; i++) {
            resultado = resultado*base;
         }

         printf("\n\t>>> Resultado: %f", resultado);
      } else {

         /* Se a base não for zero e o expoente for negativo, calcula o resultado */
         /* como se o expoente fosse positivo e depois inverte o resultado        */

         exp = -exp;

         for (i = 1; i <= exp; i++) {
            resultado = resultado*base;
         }

         printf("\n\t>>> Resultado: %f", 1/resultado);
      }
   } /* while */

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

   return 0;
}
```

Exemplo de execução do programa:

```
>>> Este programa calcula potenciacoes de numeros reais
>>> elevados a numeros inteiros. O programa encerra
>>> quando o resultado for indefinido.

>>> Digite o valor da base: 2
>>> Digite o valor do expoente: -3

>>> Resultado: 0.125000

>>> Digite o valor da base: 0
>>> Digite o valor do expoente: 0

>>> Resultado indefinido

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

4.11.11 Somando Positivos e Contando Negativos

Problema: Escreva um programa que lê um número indeterminado de valores inteiros, soma aqueles que são positivos e conta quantos valores negativos foram introduzidos. A entrada de dados deve encerrar quando zero for introduzido.

Solução: O algoritmo aparece na Figura 4–20 e o programa vem logo em seguida.

```
ALGORITMO POSITIVOSENEGATIVOS
ENTRADA: Não há
SAÍDA: Tabela de conversão de Celsius para Fahrenheit

inteiro valor, soma, negativos

soma ← 0
negativos ← 0

enquanto (verdadeiro) faça
    escreva("Digite um valor inteiro: ")
    leia(valor)

  se (valor = 0) então
    pare
  se (valor < 0) então
    negativos ← negativos + 1
  senão
    soma ← soma + valor

escreva("Soma dos valores positivos:", soma)
escreva("Valores negativos introduzidos:", negativos)
```

FIGURA 4–20: ALGORITMO DE SOMA DE NÚMEROS POSITIVOS E CONTAGEM DE NÚMEROS NEGATIVOS

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

/****
 * main(): Lê um número indeterminado de valores inteiros, soma aqueles que
 *         são positivos e conta quantos valores negativos foram introduzidos.
 *         A entrada de dados encerra quando zero for introduzido.
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int valor, /* Valor inteiro recentemente lido */
       soma = 0, /* Soma dos valores positivos */
       negativos = 0; /* Quantidade de valores negativos */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa soma os valores positivos e a"
           " quantidade de valores \n\t>>> negativos digitados."
           "\n\t>>> Zero encerra a entrada de dados.\n" );

      /* O laço encerra quando for lido zero */
   while (1) {
      printf("\n\t>>> Digite um valor inteiro: ");
      valor = LeInteiro();

      if (valor == 0) { /* Se o usuário digitar zero, encerra o laço */
         break;
      }
         /* Se o valor for negativo, acrescenta 1 à */
         /* contagem. Se ele for positivo, soma-o.  */
```

```
      if (valor < 0) {
         ++negativos;
      } else {
         soma = soma + valor;
      }
   }

      /* Apresenta o resultado */
   printf("\n\t>>> Soma dos valores positivos: %d", soma);
   printf( "\n\t>>> Valores negativos: %d\n", negativos );

   return 0;
}
```

Exemplo de execução do programa:

```
>>> Este programa soma os valores positivos e a quantidade de valores
>>> negativos introduzidos. Zero encerra a entrada de dados.

>>> Digite um valor inteiro: 2
>>> Digite um valor inteiro: 1
>>> Digite um valor inteiro: -1
>>> Digite um valor inteiro: -3
>>> Digite um valor inteiro: 4
>>> Digite um valor inteiro: -4
>>> Digite um valor inteiro: 0

>>> Soma dos valores positivos: 7
>>> Valores negativos introduzidos: 3
```

4.11.12 Fatorando Números Inteiros Positivos

Problema: Escreva um programa que decompõe números inteiros positivos introduzidos via teclado. O programa deve encerrar quando o usuário digitar zero ou um valor negativo.

Solução: O algoritmo aparece na Figura 4–21 e o programa vem logo em seguida.

Algoritmo Fatoração

Entrada: Não há

Saída: Tabela de conversão de Celsius para Fahrenheit

```
inteiro numero, possivelFator, faltaFatorar

enquanto (verdadeiro) faça
  escreva("Digite o numero positivo que será fatorado: ")
  leia(numero)

  se (numero ≤ 0) então
    escreva("O numero deveria ser positivo\n")
    pare

  possivelFator ← 2
  faltaFatorar ← numero

  escreva("Fatores de  ", numero)

  enquanto (possivelFator*possivelFator ≤
        faltaFatorar) faça
```

CONTINUA

Figura 4–21: Algoritmo de Fatoração de Números Inteiros Positivos

```
    se (faltaFatorar%possivelFator = 0) então
      escreva(possivelFator, " * ")

      faltaFatorar ← faltaFatorar/possivelFator
    senão
      se (possivelFator = 2) então
        possivelFator ← 3
      senão
        possivelFator ← possivelFator + 2

escreva(faltaFatorar)
```

FIGURA 4–21 (CONT.): ALGORITMO DE FATORAÇÃO DE NÚMEROS INTEIROS POSITIVOS

Análise: Esse algoritmo requer a inclusão de comentários para facilitar seu entendimento. Esses comentários não foram incluídos nesse algoritmo para evitar repetição, pois eles são os mesmos que acompanham o programa adiante.

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

/****
 * main(): Determina os fatores de cada número
 *         inteiro positivo recebido como entrada
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int numero,
       possivelFator, /* Um candidato a fator do número */
       faltaFatorar; /* Parte do número que falta fatorar */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa apresenta fatoracoes"
           "\n\t>>> de numeros inteiros positivos.\n" );

      /* O laço encerra quando for introduzido um número que não é positivo */
   while (1) {
      printf("\n\t>>> Numero positivo que sera' fatorado: ");
      numero = LeInteiro();

      if (numero <= 0) {
         printf("\n\t>>> O numero deveria ser positivo\n");
         break;
      }

         /* O primeiro candidato a fator é 2 pois 1 é fator de qualquer número */
      possivelFator = 2;

         /* O número ainda não foi fatorado */
      faltaFatorar = numero;

      printf("\n\t>>> Fatores de %d:\n\n\t\t> ", numero);

         /* No máximo, o divisor de um número é igual à raiz quadrada do      */
         /* número. Portanto, se o quadrado de um possível divisor for maior  */
```

```
        /* do que o número, ele não tem mais divisores e o laço é encerrado. */
     while(possivelFator*possivelFator <= faltaFatorar) {
           /* Verifica se o candidato a fator é realmente um fator */
        if(faltaFatorar%possivelFator == 0) {
              /* Encontrado um fator */
           printf("%d * ", possivelFator);

              /* Atualiza o valor que falta fatorar, considerando-o como o   */
              /* resultado da divisão do número pelo último fator encontrado */
           faltaFatorar = faltaFatorar/possivelFator;

           continue; /* O resto do laço não será executado */
        }

        /* O último candidato não é um fator. Então, tenta-se outro candidato. */

           /* Se o último candidato foi 2, o próximo candidato será 3. */
           /* Caso contrário,  soma-se 2 ao último candidato.          */
        if(possivelFator == 2) {
           possivelFator = 3;
        } else {
           possivelFator = possivelFator + 2;
        }
     }

        /* O último fator é o que faltou fatorar */
     printf("%d\n", faltaFatorar);
  }

  printf( "\n\t>>> Obrigado por usar este programa.\n");

  return 0;
}
```

Exemplo de execução do programa:

```
>>> Este programa apresenta fatoracoes
>>> de numeros inteiros positivos.

>>> Numero positivo que sera' fatorado: 1000

>>> Fatores de 1000:

        > 2 * 2 * 2 * 5 * 5 * 5

>>> Numero positivo que sera' fatorado: 0

>>> O numero deveria ser positivo

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

4.12 Exercícios de Revisão

Introdução (Seção 4.1)

1. O que é fluxo de execução de um programa?
2. (a) Como é o fluxo natural de execução de um programa? (b) Como ele pode ser alterado?

Sequências de Instruções (Seção 4.2)

3. O programa a seguir é legal em C?

```
int main(void)
{
    (1 + 2) * 4;

    return  0;
}
```

4. A instrução a seguir é legal em C? (b) Qual é o efeito dela num programa?

```
(1 + 2) * 4;
```

5. Que tipos de construções são consideradas instruções em C?
6. O que é um bloco de instruções?
7. Por que quando uma expressão constitui uma instrução, ela faz sentido apenas se contiver operadores com efeito colateral?
8. Por que o seguinte programa não consegue ser compilado?

```
#include <stdio.h>

int x = 10;

x++;

int main(void)
{
   printf("x = %d", x);

   return 0;
}
```

9. Por que, apesar de ser semelhante ao programa da questão anterior, o seguinte programa é compilado normalmente?

```
#include <stdio.h>

int x = 10;

int main(void)
{
   x++;

   printf("x = %d", x);

   return 0;
}
```

10. (a) Qual é o símbolo usado como terminal de instrução? (b) Toda linha de um programa em C termina com esse símbolo?

Instruções Vazias (Seção 4.3)

11. (a) Apresente dois exemplos de instruções vazias acidentais. (b) Apresente dois exemplos de instruções vazias intencionais.

Estruturas de Controle (Seção 4.4)

12. Como estruturas de controle são categorizadas?
13. (a) O que são desvios condicionais? (b) Quais são os desvios condicionais da linguagem C?
14. (a) O que são desvios incondicionais? (b) Quais são os desvios incondicionais da linguagem C?
15. (a) O que são laços de repetição? (b) Quais são os laços de repetição da linguagem C?

Laços de Repetição (Seção 4.5)

16. Descreva a sintaxe (i.e., o formato) e a semântica (i.e., o funcionamento) dos laços de repetição de C:
(a) **while**
(b) **do-while**
(c) **for**

17. (a) Quantas vezes o corpo do laço **while** do trecho de programa abaixo será executado? (b) Quais serão os valores de `x` e `y` imediatamente após a saída desse laço? (c) As versões sufixas dos operadores `++` e `--` utilizadas nesse laço podem ser trocadas pelas respectivas versões prefixas desses operadores sem alterar as respostas às questões (a) e (b)?

```
int  x = 0, y = 10;

while (x < y) {
   x++;
   y--;
}
```

18. (a) Existe alguma diferença entre a instrução **while** do exercício anterior e a instrução **do-while** a seguir? (b) Quantas vezes o corpo desse laço **do-while** será executado? (c) Quais serão os valores de `x` e `y` imediatamente após a saída desse laço?

```
int  x = 0, y = 10;

do {
  x++;
  y--;
} while (x < y);
```

19. (a) Quantas vezes o corpo do seguinte laço **for** será executado? (b) Quais serão os valores de `x` e `y` imediatamente após a saída desse laço? (c) As versões sufixas dos operadores `++` e `--` utilizadas podem ser trocados pelas respectivas versões prefixas desses operadores sem alterar as respostas às questões (a) e (b)?

```
int  x = 0, y = 10;

for ( ; x++ < y--; ) {
       ;  /* Instrução vazia */
}
```

20. Na instrução **for** a seguir, a expressão `j++` pode ser substituído por `++j` sem alterar a funcionalidade da instrução? Justifique sua resposta.

```
for  (j = 1; j <= 10; j++) {
   printf("Como e' divertido programar em C\n");
}
```

21. O que há de errado com o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int i;

   while (i < 100) {
      printf("%d\n", i);
      ++i;
   }

   return 0;
}
```

22. O que há de errado com o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int i = 1;

   while (i > 0) {
      printf("%d\n", i);
      ++i;
   }
   return 0;
}
```

23. O que o seguinte programa escreve na tela?

```
#include <stdio.h>

int main( void )
{
   int i, soma = 0;

   for (i = 2; i <= 100; i = i + 2) {
      soma = soma + i;
   }

   printf("Soma = %d\n", soma);

   return 0;
}
```

24. (a) O laço **while** abaixo encerra? (b) Se for o caso, o que será escrito na tela por este trecho de programa? [**Dica**: Esta questão não tão trivial quanto parece. Consulte a **Seção 4.11.7** antes de tentar respondê-la.]

```
int i = 10;

while (i++ > 0) {
   ;
}

printf("Valor final de i: %d\n", i);
```

25. (a) O laço **while** abaixo encerra? (b) Se for o caso, o que será exibido na tela por este trecho de programa?

```
int i = 10;

while (--i > 0) {
   ;
}

printf("Valor final de i: %d\n", i);
```

26. Reescreva, sem usar **for**, um trecho de programa equivalente ao seguinte:

```
int i;

for (i = 0; i < 10; ++i) {
   printf("i = %d\n", i);
}
```

27. O que há de errado com o laço **for** a seguir:

```
int i;

for (i = 0; i < 10; i + 2) {
   printf("i = %d\n", i);
}
```

28. Por que o laço **for** abaixo nunca encerra?

```
int i, j;

for (i = 0; j = 10, i < j, ++i; --j) {
   printf("\ni = %d\tj = %d\n", i, j);
}
```

29. O que é condição de parada de um laço de repetição?

30. Por que o programa abaixo nunca encerra?

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

int main(void)
{
   int valor;

   do {
      printf( "\nDigite um valor inteiro "
              "(1 ou 2 encerra o programa): " );
      valor = LeInteiro();

      printf("Voce digitou: %d\n", valor);
   } while (valor != 1 || valor != 2);

   return 0;
}
```

31. Qual é a condição de parada do seguinte laço **while**?

```
int i = 100;

while (i > 0) {
   ...
   --i;
}
```

32. (a) O que o programador que escreveu o seguinte programa pretendia escrever na tela? (b) Por que o objetivo do programador não é satisfeito? (c) Como o erro apresentado por este programa pode ser corrigido? (d) Como um programador pode precaver-se contra erros dessa natureza?

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 0; i < 10; ++i);
      printf("Passagem no. %d", i);

   return 0;
}
```

33. Supondo que o usuário do programa a seguir introduz os dados solicitados corretamente, o que este programa escreve na tela após receber os dados?

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

int main(void)
{
   int  x, y, i, j;
   char car;

   printf("Digite um caractere: ");
   car = LeCaractere();

   printf("Digite um numero inteiro entre 5 e 15: ");
   x = LeInteiro();

   printf("Digite outro numero inteiro entre 5 e 15: ");
   y = LeInteiro();

   putchar('\n');

   for ( i = 1; i <= y; i++ ) {
      for ( j = 1; j <= x; j++ ) {
         putchar(car);
      }
      putchar('\n');
   }

   return 0;
}
```

34. (a) O que é um laço de contagem? (b) Que laço de repetição é usado mais frequentemente para implementar laços de contagem?

35. O que há de errado com o seguinte trecho de programa?

```
int i;

for (i = 1, i < 10, i++)
   printf("i = %d", i);
```

36. Suponha que `i` é uma variável do tipo **int**. Quantas vezes cada um dos seguintes laços de contagem será executado?

(a)
```
for (i = 1; i < 11; ++i) {
   /* Corpo do laço */
}
```

(b)
```
for (i = 1; i <= 10; ++i) {
   /* Corpo do laço */
}
```

(c)
```
for (i = 0; i < 11; ++i) {
   /* Corpo do laço */
}
```

(d)
```
for (i = 0; i <= 10; ++i) {
   /* Corpo do laço */
}
```

(e)
```
for (i = 10; i > 0; --i) {
   /* Corpo do laço */
}
```

(f)
```
for (i = 10; i >= 0; --i) {
   /* Corpo do laço */
}
```

(g)
```
for (i = 0; i < 11; --i) {
   /* Corpo do laço */
}
```

(h)
```
for (i = 10; i >= 0; ++i) {
   /* Corpo do laço */
}
```

37. Quantas linhas cada programa a seguir escreve?

(a)
```
#include <stdio.h>

int main(void)
{
   int i, j;

   for (i = 1; i <= 10; ++i) {
     for (j = 1; j <= 10; ++j) {
       printf("Linha %d\n", j);
      }
   }

   return 0;
}
```

(b)
```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 1; i <= 10; ++i) {
     for (i = 1; i <= 10; ++i) {
       printf("Linha %d\n", i);
      }
   }

   return 0;
}
```

38. O programa a seguir foi criado com o objetivo de construir uma tabela de conversão entre graus centígrados e Fahrenheit para os 100 primeiros valores inteiros de graus centígrados.

```
#include <stdio.h>

int main(void)
{
    int cent;

    printf("Centigrados\t\tFahrenheit\n");

    for (cent = 0; cent <= 100; ++cent);
        printf("%d\t\t\t%d\n", cent, (9*cent)/5 + 32);

    return 0;
}
```

No entanto, o programa consegue escrever apenas:

```
Centigrados             Fahrenheit
101                     213
```

Explique por que o programa não funciona conforme deveria e apresente uma maneira de corrigi-lo. [**Dica**: O erro apresentado por esse programa seria evitado se o programador seguisse as recomendações apresentadas na **Seção 4.5.1**.]

39. O que exibe na tela o seguinte programa? [**Sugestão**: Consulte a tabela de sequências de escape apresentada na **Seção 3.5.3**.]

```
#include <stdio.h>

int main()
{
   int i;

   printf("\nFlamengo");

   for(i = 0; i < 5; i++) {
      putchar('\b');
   }

   printf("afo");
   printf("\rBot\n");

   return 0;
}
```

40. Qual é a relação entre expressões condicionais e condições de parada?

41. Enuncie as leis de De Morgan.

Desvios Condicionais (Seção 4.6)

42. Para que serve o uso de instruções **break** no corpo de uma instrução **switch-case**?

43. Suponha que `x` seja uma variável do tipo **int**. Por que a chamada de **printf()** no trecho de programa a seguir é sempre executada, independentemente do valor assumido por `x`?

```
if (0 < x < 10)
   printf("Valor de x: %d", x);
```

44. O trecho de programa a seguir contém um erro de programação muito comum em C. (a) Qual é esse erro? (b) Como o programador poderia precaver-se contra a ocorrência desse tipo de erro? (c) Compile este programa e verifique se o compilador utilizado emite alguma mensagem de advertência alertando o programador com relação ao referido erro.

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

int main(void)
{
   int x;

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

   if (x = 0) {
      printf("\nVoce digitou zero\n");
   } else {
     printf("\nVoce digitou um valor diferente de zero\n");
   }

   return 0;
}
```

45. O programa abaixo apresenta um erro de digitação (i.e., a palavra-chave **default** é escrita como *defalut*). (a) Explique por que este programa consegue ser compilado. (b) O que este programa apresenta na tela?

```
#include<stdio.h>

int main(void)
{
   int x = 5;

   switch(x) {
      case 1:
         printf("Um");
         break;
      case 2:
         printf("Dois");
         break;
      defalut:
         printf("Outro valor");
   }

   return 0;
}
```

46. Suponha que `x` seja uma variável do tipo **int** cujo valor seja `5` logo antes da execução da instrução **if** abaixo. O que será exibido na tela?

```
if (x > 0 && ++x < 10)
   printf("Valor de x: %d", x);
```

47. Suponha que `x` seja uma variável do tipo **int** cujo valor seja `5` logo antes da execução da instrução **if** a seguir. O que será exibido na tela?

```
if (x > 0 || ++x < 10)
   printf("Valor de x: %d", x);
```

48. Qual é a saída do seguinte programa quando o usuário introduz: (a) Um número maior do que `50`? (b) Um número par menor do que `50`? (c) Um número ímpar menor do que `50`?

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

int main(void)
{
   int i , numero;

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

   if (numero > 50) {
      printf("Tchau");
      return 1;
   }

   if (!(numero%2)) {
      for (i = 1; i <= numero; ++i) {
         printf("+\n");
      }
   } else {
      for (i = 1; i < numero; ++i) {
         putchar('-');
      }
   }
   return 0;
}
```

49. O que o programa a seguir exibe na tela?

```
#include <stdio.h>

int main(void)
{
   int linha,  /* Linha na tela  */
       coluna; /* Coluna na tela */

   putchar('\n');

   for (linha = 1; linha <= 10; ++linha) {
      for (coluna = 1; coluna <= 10; ++coluna) {
         putchar(linha%2 ? '>' : '<');
      }
      putchar('\n');
   }
   return 0;
}
```

50. Suponha que `x`, `y` e `z` sejam variáveis do tipo **int** devidamente iniciadas. O que há de errado com o seguinte trecho de programa?

```
if (x) {
   printf("x = %d", x);
} else if (!x) {
   printf("y = %d", y);
} else {
   printf("z = %d", z);
}
```

51. Por que o seguinte programa acredita que zero não é zero? [**Dica**: Leia atentamente o programa, pois o erro que ele incorpora é de difícil visualização.]

```
#include <stdio.h>

int main(void)
{
   int  numero;

   printf("Introduza um numero inteiro: ");

   numero = LeInteiro();

   if (numero =! 0)
      printf("O numero NAO e' zero\n");
   else
      printf("O numero e' zero\n");

   return 0;
}
```

52. Por que é recomendado o uso de uma parte **default** numa instrução **switch-case** mesmo quando essa parte não é estritamente necessária?

53. Qual é o resultado exibido na tela pelo seguinte programa?

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

int main(void)
{
   int x, y, z;

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

   printf("Digite outro numero inteiro: ");
   y = LeInteiro();

   printf("Digite outro numero inteiro: ");
   z = LeInteiro();

   if(x == y) {
      if( y == z) {
         printf("%d\n", x);
      } else {
         printf("%d %d\n", x, z);
      }
   } else if (x == z || y == z) {
      printf("%d %d\n", x, y);
   } else {
      printf("%d %d %d\n", x, y, z);
   }
   return 0;
}
```

54. Por que o programa abaixo escreve sempre a mesma mensagem: *Voce tem credito de 0*?

```
#include <stdio.h>

int main(void)
{
   int dividaInicial, pago, debito;

   printf("Divida inicial: ");
   dividaInicial = LeInteiro();

   printf("Quanto voce pagou? ");
   pago = LeInteiro();

   debito = pago - dividaInicial;

   if (debito = 0)
      printf("Voce nao deve nada\n");
   else if (debito < 0)
      printf("Voce deve %d\n", -debito);
   else
      printf("Voce tem credito de %d\n", debito);

   return 0;
}
```

55. Por que o programa a seguir acha que qualquer número inteiro positivo é ímpar?

```
#include <stdio.h>

int main(void)
{
   int numero;

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

   if (numero <= 0) {
      printf("\nEu pedi um numero positivo\n");
      return 1;
   }

   if (!numero%2) {
      printf("\n%d e' par\n", numero);
   } else {
      printf("\n%d e' impar\n", numero);
   }

   return 0;
}
```

Desvios Incondicionais (Seção 4.7)

56. (a) Compare as instruções **break** e **continue**. (b) Em quais estruturas de controle essas instruções podem ser incluídas?

57. (a) Descreva o funcionamento da instrução **goto**. (b) Por que o uso frequente de **goto** não é incentivado?

58. O que o seguinte programa exibe na tela?

```
#include<stdio.h>
int main(void)
{
   int i = 1;

   do {
      printf("%d\n", i);
      i++;

      if(i < 10)
         continue;
   } while (0);
   return 0;
}
```

59. No seguinte fragmento de programa, suponha que não haja instrução de escrita nas instruções representadas por três pontos e que a instrução **break** seja executada. O que será exibido na tela após a execução dessa instrução?

```
while (1) {
   for (i = 1; i < 10; i++) {
      ...
      break;
   }
   printf("Final da instrucao for");
}

printf("Final da instrucao while");
```

60. O que apresenta na tela o seguinte programa?

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 1; i <= 10; i++) {
      if (i%2) {
         continue;
      }
      printf("%d ", i);
   }

   return 0;
}
```

61. Usando **goto** é possível desviar o fluxo de execução para qualquer instrução de um programa?

62. Por que a sequência de instruções equivalente a uma instrução **for** apresentada na Seção 4.5.3 nem sempre é válida?

63. Supondo que o cabeçalho `<stdio.h>` é incluído em cada um dos programas a seguir, qual é o resultado apresentado na tela por cada um deles?

(a)

```
int main(void)
{
   int x = 0, i = 0;

   while (i < 20) {
      if (i%5 == 0) {
         x = x + i;
         printf("%d\t", x);
      }

      ++i;
   }

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

   return 0;
}
```

(b)

```
int main(void)
{
   int x = 0, i = 0;

   do {
      if (i%5 == 0) {
         x++;
         printf("%d\t", x);
      }

      ++i;
   } while (i < 20);

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

   return 0;
}
```

(c)

```
int main(void)
{
   int x = 0, i = 0;

   for (i = 1; i < 10; i = 2*i) {
      ++x;
      printf("%d\t", x);
   }

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

   return 0;
}
```

(d)

```
int main(void)
{
   int x = 0, i = 0;

   for (i = 1; i < 10; ++i) {
      if (i%2 == 1) {
         x = x + i;
      } else {
         --x;
      }

      printf("%d\t", x);
   }

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

   return 0;
}
```

(e)

```
int main(void)
{
   int x = 0, i = 0;

   for (i = 1; i < 10; ++i) {
      if (i%2 == 1) {
         x = x + i;
      } else {
         --x;
      }

      printf("%d\t", x);

      break;
   }

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

   return 0;
}
```

(f)

```
int main(void)
{
   int x = 0, i = 0;

   for (i = 1; i < 10; ++i) {
      if (i%2 == 1) {
         x = x + i;
         break;
      } else {
         --x;
      }

      printf("%d\t", x);
   }
   printf("\ni = %d\n", i);

   return 0;
}
```

(g)

```
int main(void)
{
   int x = 0, i = 0, j = 0;

   for (i = 1; i < 5; ++i) {
      for (j = 0; j < i; ++j) {
         x = i + j;
         printf("%d\t", x);
      }
   }
   printf("\ni = %d, j = %d\n", i, j);
   return 0;
}
```

(h)

```
int main(void)
{
   int x = 0, i = 0, j = 0;

   for (i = 1; i < 5; ++i) {
      for (j = 0; j < i; ++j) {
         switch (i + j) {
            case 0:
               ++x;
               break;
             case 1:
               ++x;
               break;
            case 2:
               x = x + 2;
               break;
            case 3:
               --x;
               break;
            default:
               x = 0;
               break;
         }
         printf("%d\t", x);
      }
   }
```

CONTINUA

(h)

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

    return 0;
}
```

64. Considere o uso de **goto** no programa a seguir. (a) Em termos de estilo, que há de errado com esse programa? (b) Reescreva esse programa sem usar **goto** e sem incluir nenhuma variável adicional.

```
#include <stdio.h>

int main(void)
{
   int contador = 1;

   inicio:
      if (contador > 10) {
         goto fim;
      }

      printf("%d  ", contador);
      contador++;

      goto inicio;

   fim:
      putchar('\n');

   return 0;
}
```

65. Por que o seguinte programa não consegue ser compilado?

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

int main(void)
{
   int n;

   printf("Escolha: '1' = continue; '2' = break: ");
   n = LeCaractere();

   switch(n) {
      case '1':
         printf("\nUsando continue");
         continue;
      case '2':
         printf("\nUsando break");
         break;
   }

   printf("\nTchau.");

   return 0;
}
```

Operador Condicional (Seção 4.8)

66. Descreva o funcionamento do operador condicional.

67. (a) Qual é a precedência relativa do operador condicional? (b) Qual é a associatividade desse operador?

68. Qual é o resultado da seguinte expressão?

```
1 <= -2 ? 1 < 0 ? 1 : 0 : 1 ? 2 : 3
```

69. Por que não é recomendado usar expressões formadas com o operador condicional como operandos desse mesmo operador? [**Dica**: A resposta encontra-se no item anterior.]

70. Suponha que `x` seja uma variável do tipo **int**. Substitua as duas chamadas da função **printf()** no trecho de programa adiante por uma única chamada dessa função. [**Sugestão**: Use o operador condicional.]

```
if (x > 0) {
   printf("x e' positivo");
} else {
   printf("x nao e' positivo");
}
```

Operador Vírgula (Seção 4.9)

71. Qual é a precedência do operador vírgula?

72. (a) Em que situação o operador vírgula é tipicamente utilizado? (b) Por que o resultado desse operador normalmente é desprezado?

73. Considerando as seguintes definições de variáveis:

```
int i, x = 1, y = 2, z = 3;
```

Que valor será atribuído à variável `i` em cada uma das instruções a seguir?

(a) `i = (x, y);`

(b) `i = x, y;`

(c) `i = x, y, z;`

(d) `i = (x, y, z);`

74. Supondo que `i` e `j` sejam variáveis do tipo **int** e, considerando a expressão:

```
(i = 0) , (j = i--)
```

responda as seguintes questões:

(a) Que valores assumirão as variáveis `i` e `j` após a avaliação da expressão?

(b) Qual é o valor da expressão?

(c) Se o resultado da expressão não interessar, o operador vírgula pode ser substituído pelo operador de soma?

(d) Se o resultado da expressão não interessar, o operador vírgula pode ser substituído pelo operador de conjunção?

(e) Se o resultado da expressão não interessar, o operador vírgula pode ser substituído pelo operador de disjunção?

75. Quais são os operadores da linguagem C que possuem ordem de avaliação de operandos definida?

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

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

77. Suponha que `soma` e `x` sejam variáveis do tipo **int** iniciadas com `0`. (a) A seguinte instrução é portável? (b) Que valor é atribuído à variável `soma`? (c) Se os parênteses dessa instrução forem removidos, que valor será atribuído à variável `soma`?

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

78. O que escreve na tela o seguinte programa? [**Dica**: Essa parece ser uma pergunta bastante boba, mas não é. Leia o programa com bastante atenção e você descobrirá que a resposta não é tão trivial.]

```
#include <stdio.h>

int main(void)
{
   double d;

   d = 2,5;
   printf("\nd = %f\n", d);

   return 0;
}
```

79. Por que o seguinte programa não pode ser compilado?

```
#include <stdio.h>

int main(void)
{
   double d = 2,5;

   printf("\nd = %f\n", d);

   return 0;
}
```

Geração de Números Aleatórios (Seção 4.10)

80. Por que, rigorosamente falando, números aleatórios gerados pela função **rand()** devem ser denominados *pseudoaleatórios*?

81. (a) Que números podem ser sorteados no programa abaixo? (b) Por que este programa sempre sorteia o mesmo valor a cada execução?

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

int main(void)
{
   int sorteio;

   sorteio = rand()%10 + 1;

   printf("\nNumero sorteado: %d\n", sorteio);

   return 0;
}
```

82. Para que serve a chamada da função **srand()** no seguinte programa?

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

int main(void)
{
   int  sorteio, semente;

   printf("\nDigite um numero inteiro: ");
   scanf("%d", &semente);

   srand(semente);
   sorteio = rand()%10 + 1;
   printf("\nNumero sorteado: %d\n", sorteio);

   return 0;
}
```

83. (a) Para que serve a chamada da função **time()** no programa a seguir? (b) Qual é a vantagem apresentada por este programa em relação àquele da questão anterior?

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

int main( void )
{
   int    sorteio;

   srand(time(NULL));

   sorteio = rand()%10 + 1;

   printf("\nNumero sorteado: %d", sorteio);

   return 0;
}
```

4.13 Exercícios de Programação

4.13.1 Fácil

EP4.1 Escreva um programa que escreve na tela cem vezes, alternadamente, cada frase a seguir:

- *Só aprende a programar quem escreve programas*
- *Quem não escreve programas não aprende a programar*

[**Sugestões**: (1) Use um laço **for** com uma variável de contagem `i` que varia entre `1` e `200`. (2) No corpo do laço, verifique se o valor de `i` é par e, se for o caso, escreva uma das frases; se não for o caso, escreva a outra frase.]

EP4.2 Escreva um programa que conta de `100` a `200` de cinco em cinco e apresenta os valores resultantes da contagem, como mostra o seguinte exemplo de execução:

```
100
105
110
...
200
```

EP4.3 Escreva um programa que apresenta na tela uma tabela de quadrados de valores inteiros. O número de linhas da tabela deve ser introduzido pelo usuário.

EP4.4 Escreva um programa que lê valores reais que representam raios de círculos via teclado, calcula as áreas dos círculos respectivos e exibe os resultados. O programa deve encerrar quando for lido um raio igual a zero. [**Sugestão**: O algoritmo a ser seguido por esse programa é o mesmo do exercício **EP2.7**.]

EP4.5 Escreva um programa que usa um laço **for** para exibir na tela a soma dos números pares compreendidos entre `1` e `30`. [**Sugestões**: (1) Use uma variável para armazenar o valor da soma e inicie-a com zero. (2) Inicie a variável de contagem do laço **for** com `2` e acrescente `2` a ela a cada passagem no corpo do laço. (3) Acrescente a variável de contagem à variável que acumula a soma. (4) Ao final do laço, apresente na tela o valor da variável que armazena a soma.]

EP4.6 Sabendo que os números pares formam uma progressão aritmética, escreva um programa que apresenta a soma dos números pares compreendidos entre `1` e `30` e que seja mais eficiente do que aquele apresentado como solução para o problema **EP4.5**. [**Sugestão**: Use a fórmula para cálculo da soma dos termos de uma progressão aritmética.]

EP4.7 Escreva um programa que sorteia `10000` valores entre `0` e `9` e determina quantos desses valores são pares ou ímpares. [**Sugestão**: Estude a **Seção 4.10** e o exemplo apresentado na **Seção 4.11.8**.]

EP4.8 Escreva um programa que exibe na tela uma régua com duas escalas, como mostrado a seguir:

```
0         1         2         3         4         5
01234567890123456789012345678901234567890123456789
```

Sugestões: (1) Use um laço de contagem para cada escala da régua. Em cada laço a variável de contagem `i` varia entre `0` e `50`. (2) No primeiro laço, quando `i` for divisível por `10`, escreva na tela o resultado dessa divisão. Caso contrário, escreva um espaço em branco. (3) No segundo laço, escreva na tela o resto da divisão de `i` por `10`.]

EP4.9 Escreva um programa para gerenciamento de finanças pessoais do usuário. O programa deverá solicitar o saldo inicial do usuário e, então, pedir que ele introduza, continuamente, valores de despesas e ganhos. A entrada de dados deve encerrar quando o usuário digitar zero.

EP4.10 Escreva um programa que solicita ao usuário para introduzir um número inteiro. Se esse número estiver entre `1` e `7`, o programa escreve na tela o dia da semana correspondente (Domingo corresponde a `1`, Segunda corresponde a `2` e assim por diante); caso contrário, o programa escreve na tela uma mensagem informando que não existe dia da semana correspondente. [**Sugestão**: Utilize uma instrução **switch-case** em seu programa.]

EP4.11 Escreva um programa que solicita ao usuário para introduzir dois inteiros positivos e calcula a soma dos inteiros compreendidos entre esses dois números. O programa deve permitir que o usuário introduza primeiro o maior ou o menor valor.

EP4.12 Escreva um programa que apresenta na tela uma tabela de conversão de Fahrenheit para Celsius como mostrado a seguir:

```
Fahrenheit      Celsius
==========      =======

     0           -17.8
    10           -12.2
    20            -6.7
    40             4.4
   ...             ...
    70            21.1
    80            26.7
```

[**Sugestão**: Utilize a fórmula $C = (F - 32)/1.8$, sendo C a temperatura em Celsius e `F` a temperatura em Fahrenheit.]

EP4.13 Escreva um programa que lê uma temperatura em Fahrenheit introduzida pelo usuário, converte-a em graus Celsius (t) e escreve o seguinte na tela:

- *Cuidado com a hipotermia*, se $-40 \leq t < -10$
- *Esta' congelante*, se $-10 \leq t < 0$
- *Esta' muito frio*, se $0 \leq t < 10$
- *Esta' frio*, se $10 \leq t < 15$
- *Esta' ameno*, se $15 \leq t < 25$
- *Esta' morno*, se $25 \leq t < 35$
- *Voce vai derreter*, se $35 \leq t < 45$
- *Voce deve estar morto*, para qualquer outro valor de t.

[**Sugestão**: Utilize a fórmula do exercício **EP4.12**.]

EP4.14 Escreva um programa que lê três valores reais positivos introduzidos pelo usuário e informe se eles podem ser constituir os lados de um triângulo retângulo. [**Sugestão**: Verifique se o maior valor introduzido constitui a hipotenusa de um triângulo retângulo usando o teorema de Pitágoras.]

EP4.15 Escreva um programa que lê via teclado uma lista de números inteiros positivos e calcula a média dos valores lidos. A leitura deve encerrar quando o usuário digitar um valor menor do que ou igual a zero.

EP4.16 Um triângulo pode ser classificado como:

- **Equilátero**, que possui todos os lados de tamanhos iguais.
- **Isósceles**, que possui, pelo menos, dois lados de tamanhos iguais.
- **Escaleno**, que possui três lados de tamanhos diferentes.

Escreva um programa que lê três valores reais e verifica se eles podem constituir os lados de um triângulo. Se esse for o caso, o programa deve classificar o triângulo em equilátero, isósceles ou escaleno. Como todo triângulo equilátero também é isósceles, o programa não precisa apresentar essa informação de modo redundante. [**Sugestão**: Estude o exemplo apresentado na **Seção 4.11.5**.]

EP4.17 Escreva um programa em C que exiba na tela os dígitos de `'0'` a `'9'` e seus respectivos valores decimais. [**Observação**: Inevitavelmente, o resultado desse programa não é portável (v. **Seção 3.3**).]

EP4.18 Escreva um programa em C que receba um número inteiro como entrada e determine se o mesmo é par ou ímpar. O programa deve terminar quando um número inteiro negativo for introduzido. A saída do programa deve ser algo como: `O numero introduzido e' par`. [**Sugestão**: Estude o exemplo apresentado na **Seção 4.11.2**.]

EP4.19 Escreva um programa em C que solicita ao usuário que introduza uma nota de avaliação, cujo valor pode variar entre `0.0` e `10.0` e exibe o conceito referente a essa nota de acordo com a seguinte tabela:

Nota	Conceito
Entre `9.0` (inclusive) e `10.0` (inclusive)	A
Entre `8.0` (inclusive) e `9.0`	B
Entre `7.0` (inclusive) e `8.0`	C
Entre `6.0` (inclusive) e `7.0`	D
Entre `5.0` (inclusive) e `6.0`	E
Menor do que `5.0`	F

O programa deve ainda apresentar mensagens de erro correspondentes a entradas fora do intervalo de valores permitido.

EP4.20 Escreva um programa em C que solicita ao usuário para introduzir uma série de valores inteiros positivos e lê esses números até que o usuário introduza o valor `0`. Então, o programa deve apresentar o menor, o maior e a média dos valores introduzidos (sem levar em consideração `0`). Caso o usuário introduza um número negativo, o programa deve informá-lo de que o valor não é válido e não deve levar esse valor em consideração. Exemplo de interação com o programa (**`negrito`** corresponde a entrada do usuário):

```
[Apresentação do programa]

Introduza uma série de números inteiros positivos
terminando a série com zero.
Introduza o próximo número: 5
Introduza o próximo número: -2
-2 não é um valor válido.
Introduza o próximo número: 1
Introduza o próximo número: 6
Introduza o próximo número: 0

Menor valor introduzido: 1
Maior valor introduzido: 6
Média dos valores introduzidos: 4.0
```

[**Sugestão**: Utilize parte do programa resultante da resolução do exercício **EP4.15**.]

EP4.21 Escreva um programa em C que solicita ao usuário para introduzir `n` valores inteiros, lê esses números e apresenta o menor, o maior e a média dos valores introduzidos. O valor `n` deve ser o primeiro dado introduzido pelo usuário. Exemplo de interação com o programa (**`negrito`** corresponde a entrada do usuário):

```
[Apresentação do programa]

Quantos números você irá introduzir? 3
Introduza o próximo número: 5
Introduza o próximo número: -2
Introduza o próximo número: 0

Menor valor introduzido: -2
Maior valor introduzido: 5
Média dos valores introduzidos: 1.0
```

[**Sugestão**: Este exercício é semelhante ao exercício **EP4.20**. Portanto descubra as diferenças entre os dois exercícios e utilize a solução do exercício **EP4.20** como ponto de partida.]

EP4.22 Escreva um programa que lê um caractere e um número inteiro positivo. Defina uma constante simbólica, denominada `CARACTERES_POR_LINHA`, que representa o número de caracteres escritos numa linha. Então, o programa deve escrever na tela o caractere introduzido o número de vezes especificado pelo valor inteiro introduzido pelo usuário. A escrita deve ser tal que, quando o número de caracteres escritos numa linha atingir o valor da mencionada constante, a escrita passe para a próxima linha. [**Sugestões**: (1) Use `LeCaractere()` para ler o caractere e `LeInteiro()` para ler o número inteiro. (2) Teste por meio de uma instrução **if** se o número inteiro introduzido é positivo. Se não for o caso, encerre o programa. (3) Use um laço de contagem no qual a variável contadora varie entre `1` e o valor do inteiro introduzido. (4) No corpo desse laço use **putchar()** para escrever cada caractere. (5) Quando o valor da variável de contagem for divisível pela constante simbólica, escreva um caractere de quebra de linha `'\n'`.]

EP4.23 Escreva um programa para calcular comissões de venda de um vendedor. O percentual de comissão é 10% e a leitura de dados deve terminar quando o usuário digitar um número negativo como valor de uma venda. Antes de encerrar, o programa deve informar o usuário quais foram os totais de vendas e de comissões. A seguir, um exemplo de execução do programa:

```
Valor da venda (R$): 250
==> Comissao: R$25.00

Valor da venda (R$): 55
==> Comissao: R$5.50

Valor da venda (R$): 320
==> Comissao: R$32.00

Valor da venda (R$): -1

Total de vendas: R$625.00
Total de comissoes: R$62.50
```

[**Sugestões**: (1) Defina uma constante simbólica que represente o valor da comissão de vendas. (2) Defina as seguintes variáveis do tipo **double**: `venda`, `comissao`, `totalVendas` e `totalComissoes`. (3) Inicie essas duas últimas variáveis com `0.0`. (4) Use um laço **while** infinito e, no corpo desse laço, leia o valor de cada venda. (5) Se o usuário introduzir um valor de venda negativo, encerre o laço. Caso contrário, calcule o valor da comissão e a apresente o resultado na tela. Atualize também os valores das variáveis `totalVendas` e `totalComissoes`.]

EP4.24 **Preâmbulo**: O **Índice de Massa Corpórea** (IMC) de uma pessoa é um valor usado para determinar se ela se encontra em seu peso ideal ou abaixo ou acima desse. Para calcular o IMC de um indivíduo, divide-se seu peso em quilogramas pelo quadrado de sua altura em metros. Após o IMC de um indivíduo ter sido calculado, seu valor deve ser comparado com os valores da tabela abaixo para que se possa determinar em que condição física o indivíduo se encontra.

IMC (kg/m²)	Situação
Abaixo de `18.5`	Você está abaixo do peso ideal
Entre `18.5` e `24.9`	Parabéns, você está em seu peso ideal
Entre `25.0` e `29.9`	Você está acima de seu peso
Entre `30.0` e `34.9`	Obesidade grau 1
Entre `35.0` e `39.9`	Obesidade grau 2
Acima de `40.0`	Obesidade grau 3

Problema: Escreva um programa que lê o peso e a altura de uma pessoa, calcula seu IMC e apresenta sua situação de acordo com a tabela acima. [**Dica**: Este problema é semelhante àquele do exercício **EP4.13**.]

EP4.25 Escreva um programa que lê dois valores inteiros e informa se um deles é múltiplo do outro. [**Sugestões**: (1) Defina três variáveis inteiras no seu programa denominadas: `maior`, `menor` e `aux`. (2) Leia o primeiro valor e atribua-o à variável `maior`. (3) Leia o segundo valor e atribua-o à variável `menor`. (4) Se o valor da variável `menor` for maior do que o valor da variável `maior`, use a variável `aux` para trocar os valores das variáveis `maior` e `menor`. (5) Verifique se a variável `menor` divide a variável `maior`.]

EP4.26 Escreva um programa que cria a seguinte tabela de equivalência entre polegadas e centímetros, sabendo que *1 pol* corresponde a *2.54 cm*. [**Sugestão**: Use os especificadores **%2d** e **%8.2f** com **printf()** para exibir na tela os valores em polegadas e centímetros, respectivamente.]

```
Polegadas        Centimetros
=========        ===========

    1                2.54
    2                5.08
   ...               ...
    9               22.86
   10               25.40
```

4.13.2 Moderado

EP4.27 **Preâmbulo**: Dados dois números inteiros positivos `m` e `n`, com `m > n`, uma **tripla pitagórica** consiste em três números inteiros positivos `a`, `b` e `c` que satisfazem as seguintes fórmulas:

- $a = m^2 - n^2$
- $b = 2mn$
- $c = m^2 + n^2$

Problema: Escreva um programa que apresenta as triplas de Pitágoras resultantes quando os valores de `m` e `n` variam entre `1` e `5`. O resultado do programa deverá ser o seguinte:

```
      >>> Triplas Pitagoricas <<<

a =  3, b =  4, c =  5 (m = 2, n = 1)
a =  8, b =  6, c = 10 (m = 3, n = 1)
a =  5, b = 12, c = 13 (m = 3, n = 2)
a = 15, b =  8, c = 17 (m = 4, n = 1)
a = 12, b = 16, c = 20 (m = 4, n = 2)
a =  7, b = 24, c = 25 (m = 4, n = 3)
a = 24, b = 10, c = 26 (m = 5, n = 1)
a = 21, b = 20, c = 29 (m = 5, n = 2)
a = 16, b = 30, c = 34 (m = 5, n = 3)
a =  9, b = 40, c = 41 (m = 5, n = 4)
```

[**Sugestões**: (1) Use um laço **for** aninhado em outro laço **for**. Use `m` como variável de contagem de um laço e `n` como variável de contagem do outro laço. (2) No corpo do laço **for** interno, quando `m > n`, calcule os valores de `a`, `b` e `c` e exiba os resultados.]

EP4.28 Escreva um programa que verifica se uma sequência de `n` valores inteiros introduzidos via teclado constitui uma progressão aritmética. Se esse for o caso, o programa deve apresentar a razão e a soma dos termos da progressão aritmética. [**Sugestão**: Verifique se a diferença entre cada número introduzido e seu antecessor é constante. Se for o caso, trata-se de uma PA e a razão é exatamente essa diferença.]

EP4.29 Escreva um programa em C que brinque de adivinhar números com o usuário. O jogo consiste em gerar um número aleatório entre `1` e `100` e solicitar ao usuário que tente adivinhar o número gerado. Cada partida termina quando o usuário acerta o número, faz o máximo de cinco tentativas ou introduz um número negativo. Números não negativos fora do intervalo de `1` a `100` não devem ser levados em consideração. Ao final de cada partida, informe o usuário se ele acertou o número gerado ou qual era esse número. Permita que o usuário jogue quantas partidas ele desejar e, antes de encerrar o programa, informe ao usuário quantas partidas foram jogadas e quantas vezes ele acertou o número. Exemplo de interação do programa:

```
Vou pensar num numero entre 1 e 100.
Tente adivinhar este numero em no máximo 5 tentativas.
Se você digitar um numero negativo, desistirá da partida.

*** Nova partida ***

Apresente seu chute entre 1 e 100: 5
Infelizmente voce errou.
Apresente seu chute entre 1 e 100: 16
Infelizmente voce errou.
Apresente seu chute entre 1 e 100: 41
Infelizmente voce errou.
Apresente seu chute entre 1 e 100: 6
Infelizmente voce errou.
Apresente seu chute entre 1 e 100: 3
Partida encerrada. Parabéns!!! Voce acertou!!!
Deseja jogar uma nova partida? s

*** Nova partida ***

Apresente seu chute entre 1 e 100: 12
Infelizmente voce errou.
Apresente seu chute entre 1 e 100: 0
Numero invalido!!! O numero deve estar entre 1 e 100.
Apresente seu chute entre 1 e 100: 21
...
Partida encerrada. Infelizmente, voce não acertou. Meu numero era 36.
Deseja jogar uma nova partida ('n' ou 'N' encerra o jogo)? n

*** Resumo do jogo ***

*** Numero de partidas jogadas: 2
*** Numero de vezes que você acertou: 1

Obrigado por ter jogado comigo. Digite qualquer tecla para
sair do programa.
```

[**Sugestões**: (1) Para saber como gerar os números aleatórios dentro do requerido intervalo, estude a **Seção 4.10**. (2) É muito trabalhoso testar esse programa com números no intervalo entre **1** e **100**, porque a probabilidade de acerto é muito pequena. Portanto teste seu programa com números entre **1** e **5** e, após estar satisfeito com os resultados, retorne aos valores originais.]