Elementos Básicos da Linguagem C

objetivos

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia:
  - ❒ Operador
  - ❒ Operando
  - ❒ Expressão
  - ❒ Tipo primitivo
  - ❒ Palavra-chave
  - ❒ Palavra reservada
  - ❒ Biblioteca
  - ❒ Cabeçalho
  - ❒ Enumeração
  - ❒ Comentário
  - ❒ Definição de variável
  - ❒ Iniciação de variável
  - ❒ Constante simbólica
  - ❒ Instrução
  - ❒ Bloco de instruções
  - ❒ Instrução vazia
  - ❒ Programa de console
  - ❒ Ponteiro nulo
  - ❒ Desvio condicional
  - ❒ Desvio incondicional
  - ❒ Fluxo de execução
  - ❒ Estrutura de controle
  - ❒ Laço de repetição
  - ❒ Laço de contagem
  - ❒ Conversões de tipos
  - ❒ Especificador de formato
  - ❒ Diretivas #include e #define
  - ❒ Terminal de instrução
  - ❒ Hospedeiro
  - ❒ Sistema livre
- Descrever e usar os seguintes operadores:
  - ❒ Vírgula
  - ❒ De endereço
  - ❒ De indireção
  - ❒ Condicional
  - ❒ Aritméticos
  - ❒ Lógicos
  - ❒ Relacionais
  - ❒ Incremento
  - ❒ Decremento
  - ❒ De conversão
- Explicar todas as propriedades dos operadores da linguagem C
- Usar printf() para escrita na tela
- Definir um ponteiro e usá-lo para acessar o conteúdo de uma variável
- Explicar como a função scanf() pode ser usada incorretamente

ESTE CAPÍTULO APRESENTA as construções básicas da linguagem C, mas não pretende ser um mini-curso dessa linguagem. Ao contrário, como o título dessa parte do livro anuncia, este e os dois próximos capítulos constituem apenas uma revisão da linguagem C que se espera que seja útil no acompanhamento dos tópicos centrais deste livro.

1.1 A Linguagem C Padrão

A linguagem C foi desenvolvida no início da década de 1970 num laboratório da AT&T por Dennis Ritchie como uma linguagem de alto nível dispondo de facilidades de baixo nível. A padronização de C foi inicialmente realizada por um comitê ANSI em 1989 e foi denominada **ANSI C** (ou **C89**). A partir de 1990, um comitê ISO assumiu a responsabilidade pela padronização de C e, em dezembro de 2011, esse comitê ratificou o padrão mais recente da linguagem, conhecido como **C11**. Entretanto, até o final da escrita deste livro, esse último padrão ainda recebia pouco suporte por parte dos fabricantes de compiladores, de modo que este livro é aderente ao padrão **C99** estabelecido em 1999.

1.2 Identificadores, Tipos Primitivos e Constantes

1.2.1 Identificadores

As regras para criação de identificadores em C são as seguintes:

- Um identificador em C deve ser constituído apenas de letras (sem acentuação), dígitos e _ (subtraço).
- O primeiro caractere de um identificador não pode ser um dígito.
- O número de caracteres permitido num identificador depende da versão do padrão de C utilizada. Os padrões C99 e C11 requerem que compiladores de C aceitem, pelo menos, 63 caracteres.

A linguagem C faz distinção entre letras maiúsculas e minúsculas. Por exemplo, as variáveis denominadas `umaVar` e `UmaVar` são consideradas diferentes.

Palavras que possuem significado especial numa linguagem de programação (p. ex., **while** e **for** em C) são denominadas **palavras chaves** da linguagem. Essas palavras não podem ser redefinidas pelo programador. Além disso, identificadores associados a componentes disponibilizados pela biblioteca padrão de C (v. **Seção 1.9**) são conhecidos como **palavras reservadas** (p. ex., *printf*) e o programador deve evitar redefini-los em seus programas.

1.2.2 Tipos de Dados Primitivos

Existem muitos **tipos de dados primitivos** em C, mas apenas dois tipos inteiros (**int** e **char**) e um real (**double**) serão utilizados na maior parte deste livro. O tipo **int** foi escolhido para uso neste livro por simplicidade, mas ele apresenta um sério problema de portabilidade, pois sua largura não é precisamente especificada. Portanto esse tipo pode ser usado sem problemas apenas com objetivos didáticos.

O espaço ocupado em memória por valores do tipo **char** é sempre um byte. Esse tipo é comumente usado para representar caracteres, mas ele pode representar inteiros quaisquer que requerem apenas um byte de memória. Neste livro, esse tipo será usado apenas em processamento de caracteres.

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

1.2.3 Constantes

Existem cinco tipos de constantes em C: inteiras, reais, caracteres, **strings** e de enumeração. Constantes desse último tipo serão vistas na **Seção 1.13**, enquanto as demais serão apresentadas a seguir.

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

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

Constantes reais podem ser escritas em duas notações:

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

  ```
  3.1415
  .5  [a parte inteira é zero]
  7.  [a parte fracionária é zero]
  ```

- **Notação científica**. Nessa notação, um número real consiste em duas partes: **(1) mantissa** e **(2) expoente**. Essas partes são separadas por `e` ou `E` e a segunda parte representa uma potência de `10`. Por exemplo, o número `2E4` deve ser interpretado como `2×104` e lido como dois vezes dez elevado à quarta potência. A mantissa de um número real pode ser inteira ou fracionária, enquanto o expoente pode ser positivo ou negativo, mas deve ser inteiro. Por exemplo, `2.5E-3` representa o número `2.5×10-3`; ou seja, `0.0025`.

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

- Escrevendo-se o caractere entre apóstrofos (p. ex., `'A'`). Apesar de essa forma de representação ser a mais legível e portável, ela é factível apenas quando o caractere possui representação gráfica e o programador possui meios para introduzi-lo (p. ex., uma configuração adequada de teclado).
- **Por meio de uma sequência de escape**, que consiste do caractere \ (barra inversa) seguido por um caractere com significado especial, sendo ambos envolvidos entre apóstrofos. Sequências de escape devem ser usadas para representar certos caracteres que não possuem representação gráfica (p. ex., quebra de linha) ou que têm significados especiais em C (p. ex., apóstrofo). Essa forma de representação de caracteres também é portável, apesar de ser menos legível do que a forma de representação anterior. A **Tabela 1–1** apresenta as sequências de escape mais comuns que podem ser utilizadas num programa escrito em C.

Sequência de Escape	Descrição
`'\t'`	Tabulação
`'\n'`	Quebra de linha
`'\0'`	Caractere nulo

Tabela 1–1: Sequências de Escape

As sequências de escape `'\t'` e `'\n'` são usadas com muita frequência em formatação de dados na tela usando **printf()** (v. **Seção 1.10**). A sequência de escape `'\0'` é usada em processamento de strings (v. **Capítulo 3**).

Uma constante composta de um caractere ou uma sequência de escape entre apóstrofos apenas informa o compilador que ele deve considerar o valor correspondente ao caractere ou à sequência de escape no código de caracteres que ele usa. Mas não é uma boa ideia representar caracteres num programa por constantes inteiras por duas razões. A primeira razão é legibilidade e a segunda razão é que o programa pode ter sua portabilidade comprometida, pois o padrão de C não especifica qual código de caracteres deve ser utilizado por um compilador.

Um string constante consiste em uma sequência de caracteres constantes. Em C, um string constante pode conter caracteres constantes em quaisquer dos formatos apresentados e deve ser envolvido entre aspas. **Strings** constantes separados por um ou mais espaços em branco (incluindo tabulação horizontal e quebra de linha) são automaticamente concatenados pelo compilador. Isso é útil quando se tem um string constante muito grande e deseja-se escrevê-lo em duas linhas. Por exemplo, os strings:

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

serão concatenados pelo compilador para formar um único string constante:

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

1.3 Operadores Básicos

Um **operador** representa uma operação elementar da linguagem C que é aplicada a um ou mais valores denominados **operandos**. C é uma linguagem intensivamente baseada no uso de operadores. Portanto o entendimento das propriedades dos operadores da linguagem é fundamental para a aprendizagem da própria linguagem.

1.3.1 Propriedades de Operadores

As propriedades dos operadores são divididas em duas categorias:

[1] **Propriedades que todos os operadores possuem**. Essas propriedades são as seguintes:

- **Resultado**, que é o valor resultante da aplicação do operador sobre seus operandos.
- **Aridade** que é o número de operandos sobre os quais o operador atua. De acordo com essa propriedade, os operadores de C são divididos em três categorias: **operadores unários** (requerem um operando), **operadores binários** (requerem dois operandos) e **operador ternário** (requer três operandos).
- A **precedência** de um operador determina sua ordem de aplicação com relação a outros operadores. Isto é, um operador de maior precedência é aplicado antes de um operador de menor precedência. Em C, operadores são agrupados em **grupos de precedência**, de modo que todos os operadores que fazem parte de um mesmo grupo de precedência possuem a mesma precedência e operadores que pertencem a diferentes grupos de precedência possuem precedências diferentes.
- **Assim como** precedência, associatividade também é utilizada para decidir a ordem de aplicação de operadores numa expressão. Mas associatividade é utilizada com operadores de mesma precedência. Existem dois tipos de associatividade: à esquerda (o operador da esquerda é aplicado antes do operador da direita) e à direita (o operador da direita é aplicado antes do operador da esquerda)

[2] **Propriedades que alguns operadores possuem**. Essas propriedades são as seguintes:

- O **efeito colateral** de um operador consiste em alterar o valor de um de seus operandos, que, nesse caso, deve ser uma variável.

- A **ordem de avaliação** de um operador indica qual dos operandos sobre os quais ele atua é avaliado primeiro e só faz sentido para operadores binários e ternários. Apenas quatro operadores de C possuem essa propriedade e esse fato frequentemente acarreta em expressões capazes de produzir dois resultados válidos possíveis.
- **Curto-circuito** faz com que, às vezes, apenas um dos operandos de um operador binário seja avaliado. Apenas os operadores lógicos de conjunção e disjunção possuem essa propriedade.

O uso de parênteses influencia as propriedades de precedência, associatividade e, por conseguinte, resultado. Nenhuma outra propriedade de operadores sofre influência do uso de parênteses.

1.3.2 Expressões

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

1.3.3 Operadores Aritméticos

Uma **expressão aritmética** é uma combinação de **operadores aritméticos** e operandos numéricos que, quando avaliada, resulta num valor numérico. Os operandos de uma expressão aritmética podem incluir variáveis, constantes ou chamadas de funções, que resultem em valores numéricos. Os operadores aritméticos básicos de C são: soma (representado por **+**), subtração (–), multiplicação (*), divisão (/), resto de divisão inteira (%) e inversão de sinal (–). Esses operadores são agrupados em grupos de precedência conforme mostrado no **Apêndice A**.

É importante salientar que divisão inteira ou resto de divisão inteira por zero é uma operação ilegal que causa o término abrupto (**aborto**) da execução de um programa que tenta realizar tal operação. Por outro lado, divisão real por zero não causa aborto de programa, mas não produz um número real como resultado.

1.3.4 Operadores Relacionais

Operadores relacionais são operadores binários utilizados em **expressões de comparação** (ou **relacionais**). Os operandos de um operador relacional podem ser de qualquer tipo numérico. Os operadores relacionais de C são: maior do que (representado por >), maior ou igual (>=), menor do que (<), menor ou igual (<=), igual (==) e diferente (!=). O resultado da aplicação de qualquer operador relacional é 0 ou 1. O **Apêndice A** exibe as precedências e associatividades desses operadores.

Como números reais podem ser representados aproximadamente no computador, operadores relacionais não devem ser usados para comparar números reais da mesma maneira que eles são usados para comparar números inteiros.

1.3.5 Operadores Lógicos

Operadores lógicos em C podem receber como operandos quaisquer valores numéricos. A aplicação de um operador lógico sempre resulta em `0` ou `1`, dependendo dos valores dos seus operandos. Quando o operando do operador de **negação** (representado por **!**) é `0`, o resultado é `1`; caso contrário, o resultado é `0`. A aplicação do operador de **conjunção** (representado por **&&**) resulta em `1` apenas quando seus dois operandos são diferentes de zero. Por outro lado, a aplicação do operador de **disjunção** (representado por **||**) resulta em `0` apenas quando seus dois operandos são iguais a zero.

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

Quando o primeiro operando do operador **&&** é zero, seu segundo operando não é avaliado e o resultado da operação é zero. Quando o primeiro operando do operador **||** é diferente de zero, seu segundo operando não é avaliado e o resultado da operação é `1`.

O **Apêndice A** apresenta as precedências relativas entre todos os operadores da linguagem C. Uma informação importante que merece ser memorizada é que todos os operadores unários de C fazem parte de um mesmo grupo de precedência. Os operadores desse grupo possuem a segunda maior precedência dentre todos os operadores da linguagem C e a associatividade deles é à direita. Apenas os operadores de acesso (v. **Seção 3.11**) possuem maior precedência do que os operadores unários.

1.4 Definições de Variáveis

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

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

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

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

1.5 Operadores de Atribuição

Uma instrução de **atribuição** preenche o espaço em memória representado por uma variável com um valor determinado e sua sintaxe tem a seguinte forma geral:

variável = expressão;

Numa instrução de atribuição, a expressão é avaliada e o valor resultante é armazenado no espaço de memória representado pela variável.

Às vezes, é desejável que uma variável assuma certo valor no instante de sua definição. Essa **iniciação** pode ser feita em C combinando-se a definição da variável com a atribuição do valor desejado. Como exemplo de iniciação, considere:

```
int  minhaVar = 0;
```

Uma instrução de atribuição é uma expressão e o sinal de igualdade utilizado em atribuição representa o operador principal dessa expressão. Esse operador faz parte de um grupo de operadores que têm uma das mais baixas precedências dentre todos os operadores de C. De fato, apenas o operador vírgula (v. **Seção 1.16**) possui precedência menor do que esse grupo de operadores.

O operador de atribuição possui efeito colateral, que consiste exatamente na alteração de valor causada na variável (operando) que ocupa o lado esquerdo da expressão. Esse operador possui associatividade à direita e o resultado da aplicação desse operador é o valor recebido pela variável no lado esquerdo da expressão de atribuição. Em virtude das suas propriedades, o operador de atribuição pode ser utilizado para a execução de múltiplas atribuições numa única linha de instrução, como, por exemplo:

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


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

```
int x = 0, y = 0;
(x = 1)*(x + 2);
```

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

Os ingredientes de uma expressão capaz de produzir dois resultados distintos e legítimos são os seguintes:

- ❒ Uso de operador sem ordem de avaliação de operandos especificada
- ❒ Uso de um operador com efeito colateral
- ❒ Uma variável que sofre efeito colateral faz parte dos dois operandos de um operador sem ordem de avaliação de operandos definida

Existem cinco outros operadores de atribuição que combinam operações aritméticas com atribuição em operações únicas. Esses operadores, denominados **operadores de atribuição aritmética** e seus equivalentes funcionais são apresentados na **Tabela 1–2**.

Operador	Uso	Equivalente a
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`

Tabela 1–2: Operadores de Atribuição Aritmética

Todos os operadores de atribuição fazem parte de um mesmo grupo de precedência e a associatividade deles é à direita.

1.6 Conversões de Tipos

1.6.1 Conversões Implícitas

C permite que operandos de tipos aritméticos diferentes sejam misturados em expressões. Entretanto, para que tais expressões façam sentido, o compilador executa **conversões implícitas** (ou **automáticas**). Essas conversões não incluem arredondamento e muitas vezes são responsáveis por resultados inesperados.

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

Existem cinco situações nas quais conversões são feitas implicitamente por um compilador de C. Três delas serão discutidas a seguir, enquanto as demais serão discutidas no **Capítulo 2**.

- **Conversão de Atribuição**. Nesse tipo de conversão, o valor do lado direito de uma atribuição é convertido no tipo da variável do lado esquerdo. Um problema que pode ocorrer com esse tipo de conversão surge quando o tipo do lado esquerdo é mais curto do que o tipo do lado direito da atribuição. Conversões entre números reais e inteiros também podem causar perda de informação devido a overflow ou truncamento que são bem mais difíceis de detectar e corrigir.
- **Conversão Aritmética Usual**. Essa categoria de conversão ocorre quando um operador que não é de atribuição possui operandos de tipos diferentes. Nesse caso, um deles é convertido no tipo do outro, de modo que eles passem a ter o mesmo tipo. Essa categoria de conversão é denominada *aritmética* porque ela afeta *operandos aritméticos*, mas ela se aplica a operadores de qualquer natureza (com exceção de atribuição). A conversão aritmética usual segue uma hierarquia de tipos, mas como neste livro são utilizados apenas os tipos primitivos **char**, **int**, e **double**, quando um valor do tipo **double** é misturado com um valor do tipo **int** ou **char**, esse último valor é convertido em **double**.
- **Conversão de alargamento**. Mesmo quando uma variável do tipo **char** (ou parâmetro ou retorno de função desse tipo) não é misturada com operandos de outros tipos, por uma questão de eficiência, seu valor é convertido, num valor do tipo **int**. Essa categoria de conversão é denominada conversão de alargamento e nunca traz nenhum dano para um programa.

1.6.2 Conversão Explícita

O programador pode especificar conversões explicitamente em C antepondo ao valor que deseja transformar o nome do tipo desejado entre parênteses, como por exemplo:

```
int    i1 = 3, i2 = 2;
d = (double) i1/i2;
```

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

1.7 Incremento e Decremento

Os operadores de **incremento** e **decremento** são operadores unários que, quando aplicados a uma variável, adicionam-lhe ou subtraem-lhe `1`, respectivamente. Esses operadores aplicam-se a variáveis numéricas ou ponteiros (v. **Seção 1.18**). Os operadores de incremento e decremento são representados respectivamente pelos símbolos `++` e `--`. Eles têm a mesma precedência e associatividade de todos os operadores unários.

Existem duas versões para cada um desses operadores: **(1) prefixa** e **(2) sufixa**. Sintaticamente, essa classificação refere-se à posição do operador em relação ao seu operando. Se o operador aparece antes do operando (p. ex., `++x`), ele é um operador prefixo; caso contrário (p. ex., `x++`), ele é sufixo. Todos esses operadores produzem efeitos colaterais nas variáveis sobre as quais atuam. No caso de incremento, o efeito colateral corresponde ao acréscimo de `1` ao valor do operando; no caso de decremento, efeito colateral corresponde à subtração de `1` do valor do operando. Qualquer versão do operador de incremento produz o mesmo efeito de incremento e qualquer versão do operador de decremento produz o mesmo efeito de decremento.

A diferença entre as versões prefixa e sufixa de cada um desses operadores está no resultado da operação. Os operadores sufixos produzem como resultado o próprio valor da variável sobre a qual atuam. Por outro lado, operadores prefixos resultam no valor da variável após o efeito colateral (i.e., incremento ou decremento) ter ocorrido.

Supondo que `x` é uma variável numérica ou um ponteiro, os resultados e efeitos colaterais dos operadores de incremento e decremento são resumidos na **Tabela 1–3**.

Operação	Denominação	Valor da Expressão	Efeito Colateral
`x++`	Incremento sufixo	O mesmo de `x`	Adiciona `1` a `x`
`++x`	Incremento prefixo	O valor de `x` mais `1`	Adiciona `1` a `x`
`x--`	Decremento sufixo	O mesmo de `x`	Subtrai `1` de `x`
`--x`	Decremento prefixo	O valor de `x` menos `1`	Subtrai `1` de `x`

Tabela 1–3: Operadores de Incremento e Decremento

1.8 Comentários

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

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

De acordo com o padrão ISO, não é permitido o uso aninhado de comentários desse tipo, mas algumas implementações de C aceitam isso, o que pode ser bastante útil na depuração de programas, pois permite excluir, por meio de comentários, trechos de programa que já contenham comentários.

Outro delimitador de comentário é `//`. Caracteres que seguem esse delimitador até o final da linha que o contém são ignorados pelo compilador. Exemplo:

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

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

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

E vice-versa:

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

O objetivo principal de comentários é explicar o programa para outros programadores e para você mesmo quando for lê-lo algum tempo após sua escrita. Um programa sem comentários força o leitor a fazer inferências para tentar entender o que e como o programa faz.

1.9 A Biblioteca Padrão de C

Em programação, **biblioteca** é uma coleção de componentes que o programador pode incluir em seus programas. Em C, tais componentes incluem, por exemplo, constantes simbólicas (v. **Seção 1.12**), tipos de dados derivados (v. **Seção 3.9**) e funções (v. **Seção 2.1**). O uso de componentes prontos para ser incorporados em programas facilita sobremaneira o trabalho dos programadores.

A **biblioteca padrão de C** é aquela preconizada pelo padrão ISO dessa linguagem, de modo que todo compilador de C que adere ao padrão ISO deve ser distribuído acompanhado dessa biblioteca.

A biblioteca padrão de C é dividida em grupos de componentes que têm alguma afinidade entre si. Cada grupo de componentes, denominado **módulo**, possui dois arquivos associados:

- Um arquivo-objeto que contém as implementações dos componentes do módulo previamente compiladas. Para que um programa-fonte que usa uma função de biblioteca possa ser transformado em programa executável, é necessário que o arquivo-objeto resultante de sua compilação seja ligado ao arquivo-objeto que contém o código compilado da função. Essa ligação é feita por um linker.
- Um arquivo de texto, denominado **cabeçalho**, que contém definições parciais (alusões) legíveis das funções implementadas no arquivo-objeto correspondente. Um arquivo de cabeçalho pode ainda conter, entre outros componentes, definições de tipos e constantes simbólicas. Arquivos de cabeçalho normalmente têm a extensão `.h`.

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

#include <*nome-do-arquivo*>

ou:

#include "*nome-do-arquivo*"

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

A **Tabela 1–4** apresenta os cabeçalhos mais comumente utilizados e seus propósitos. Apenas alguns dos vários componentes da biblioteca padrão de C serão usados neste livro.

Cabeçalho	Propósito
`<stdio.h>`	*Entrada e saída em geral, incluindo leitura de dados via teclado, escrita de dados na tela (v.* **Seção 1.10***) e processamento de arquivos*
`<stdlib.h>`	❒ *Funções* **srand**() *e* **rand**() *usadas em geração de números aleatórios* ❒ *Funções de alocação dinâmica de memória (***Capítulo 9***)*
`<time.h>`	*Função* **time**() *usada com* **srand**() *em geração de números aleatórios como na chamada:* `srand(time(NULL))`
`<string.h>`	*Processamento de strings (v.* **Seção 3.7***)*
`<ctype.h>`	*Classificação e transformação de caracteres (v.* **Seção 3.7***)*
`<math.h>`	*Operações matemáticas com números reais [p. ex.,* **sqrt**(), **pow**()*]*

Tabela 1–4: Principais Cabeçalhos da Biblioteca Padrão de C

1.10 Escrita de Dados na Tela

A função **printf()** permite a escrita na tela do computador de constantes, variáveis, expressões ou strings. Essa função requer que seja especificado o formato de cada item que ela escreve. Essa especificação é feita por meio de um string (tipicamente constante), denominado **string de formatação**, de modo que uma chamada da função **printf()** assume o seguinte formato:

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

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

A **Tabela 1–5** enumera os especificadores de formato mais comuns utilizados pela função **printf()**.

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

Tabela 1–5: Especificadores de Formato Comuns Utilizados por printf()

Além de especificadores de formato, um string de formatação pode ainda conter outros caracteres. Quando um caractere do string de formatação não faz parte de um especificador de formato, ele é escrito exatamente como ele é.

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

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

resultaria na escrita de `A` na tela.

1.11 Leitura de Dados via Teclado 1: Frágil

Leitura de dados introduzidos via teclado é um dos aspectos mais difíceis de programação. Escrever um programa robusto capaz de responder a quaisquer caracteres digitados pelo usuário é uma tarefa que poucos programadores são capazes de realizar.

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

Nesta seção serão abordados tópicos elementares relacionados à entrada de dados via teclado usando funções declaradas no cabeçalho `<stdio.h>` da biblioteca padrão de C. Aqui, serão mostrados os problemas encontrados pelo programador quando ele usa essas funções sem a devida precaução. Mas a solução para esses problemas será adiada para a **Seção 2.2**. Enquanto isso, infelizmente, seus programas serão frágeis; i.e., eles não serão à prova de usuário.

1.11.1 Função getchar()

A função **getchar()** lê um caractere introduzido via teclado e seu protótipo é:

```
int getchar(void)
```

Quando a função **getchar()** é chamada e o buffer associado ao teclado está vazio (v. **Seção 2.2**), a execução do programa que a chama é interrompida até que o usuário digite um caractere seguido de `[ENTER]`. Essa função retorna um valor inteiro correspondente a um caractere apenas quando ela consegue realmente ler um caractere. Quando isso não é possível, essa função retorna o valor da constante **EOF**, que não pode ser contido numa variável do tipo **char**. Por isso, o tipo da variável que recebe um valor retornado por essa função deve ser do tipo **int**.

1.11.2 Função scanf()

A função **scanf()** permite a leitura de um número arbitrário de valores de vários tipos ao mesmo tempo no meio de entrada padrão. O primeiro parâmetro de **scanf()**, que é obrigatório, é um string de formatação semelhante àquele usado pela função **printf()**. No entanto, normalmente, no caso da função **scanf()**, constam no string de formatação apenas especificadores de formato e espaços em branco. Assim, com **scanf()**, não é recomendado o uso de caracteres que não fazem parte de especificadores de formato nem são considerados espaços em branco.

Os parâmetros que seguem o string de formatação de **scanf()** são endereços de variáveis nas quais os dados lidos serão armazenados. Esses últimos parâmetros são todos de saída, o que justifica o uso de endereços de variáveis (v. **Seção 1.18**). Um erro muito comum de programação em C é esquecer de preceder com o símbolo & cada variável usada como parâmetro de **scanf()**.

O protótipo da função **scanf()** é:

```
int scanf(const char *string, ...)
```

O valor retornado por **scanf()** representa o número de variáveis que tiveram seus valores alterados em virtude de uma chamada dessa função, a não ser que ocorra erro de leitura ou sinalização de final de arquivo. Nesse último caso, a função **scanf()** retorna **EOF**. O valor retornado por **scanf()** é usado para determinar se a chamada foi bem-sucedida ou não, como será visto na **Seção 2.2**.

O string de formatação de **scanf()** especifica qual é o formato esperado dos dados que serão lidos e atribuídos às variáveis cujos endereços constituem os parâmetros que seguem o referido string. A **Tabela 1–6** enumera os especificadores de formato mais comuns utilizados pela função **scanf()**.

Especificador de formato	O que scanf() espera ler?
%c	Um caractere
%s	Um string
%d	Um número inteiro em base decimal do tipo **int**
%lf	Um número real do tipo **double**

Tabela 1–6: Especificadores de Formato Comuns Utilizados por scanf()

A despeito do que alguns programadores de C imaginam, os especificadores **%d** e **%i** (que não aparece na **Tabela 1–6**) são equivalentes quando usados com **printf()**, mas esse não é o caso quando eles são usados com **scanf()**.

Apesar de **scanf()** permitir a leitura de vários valores a cada chamada, é recomendável fazer a leitura de apenas um dado de cada vez, pois, quando vários valores são lidos simultaneamente e detecta-se que um deles foi introduzido incorretamente, talvez o usuário tenha que reintroduzir todos os valores novamente, mesmo aqueles que ele digitou corretamente.

Diferentemente de saída de dados, que depende exclusivamente do programador, entrada de dados depende do comportamento do usuário do programa. Para sentir melhor esse problema, digite e compile o programa abaixo.


```
#include <stdio.h>

int main(void)
{
   int x;

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

   printf("\nO numero inteiro introduzido foi: %d", x);
   printf("\nO dobro desse numero e': %d", 2*x);
   printf("\nO quadrado desse numero e': %d\n", x*x);

   return 0;
}
```

Depois de compilar o programa, exerça o papel de um usuário desatento (ou mal-intencionado) e, quando instado a digitar um número inteiro, digite algumas letras. O resultado que você obtém na tela depende de compilador e sistema utilizados, mas, qualquer que seja o resultado, ele não faz absolutamente nenhum sentido porque você não digitou um número com o qual o programa pudesse calcular seu dobro ou quadrado.

A solução do último problema exposto não é muito trivial e será apresentada na **Seção 2.2**. Por enquanto, recomenda-se que você seja o único usuário de seus próprios programas (e seja bem-comportado...).

1.12 Constantes Simbólicas

Uma **constante simbólica** consiste em um identificador associado a um valor constante. Constantes simbólicas podem ser definidas por meio de uma diretiva **#define**, como, por exemplo:

```
#define  PI  3.14
```

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

O uso de constantes simbólicas em um programa tem dois objetivos principais: **(1)** tornar o programa mais legível e **(2)** tornar o programa mais fácil de ser modificado.

1.13 Enumerações

Uma variável de um tipo enumeração deve assumir apenas um dos valores constantes especificados na definição do seu tipo. Uma definição de variável de um tipo enumeração consiste da palavra-chave **enum** seguida de uma lista de identificadores entre chaves seguida, finalmente, pelo nome da variável. Além disso, a palavra-chave **enum** pode, opcionalmente, ser seguida por um identificador (**rótulo**). Mais precisamente, uma declaração de enumeração tem o seguinte formato:

enum *rótulo {lista-de-nomes-de-constantes} variável*$_1$*, ..., variável*$_N$*;*

Quando o rótulo é utilizado, não é necessário declarar nenhuma variável junto com a declaração da própria enumeração.

A mesma recomendação de nomenclatura para constantes simbólicas é seguida para constantes que fazem parte de uma enumeração; i.e., elas são escritas usando apenas letras maiúsculas.

As constantes simbólicas que fazem parte de uma enumeração são consideradas do tipo **int** e valores inteiros são associados a elas. Caso não haja indicação em contrário, esses valores são baseados nas posições das constantes na lista, sendo que, como padrão, a primeira constante vale `0`, a segunda constante vale `1` e assim por diante.

Tipicamente, os valores atribuídos aos identificadores de constantes de uma enumeração não são importantes, mas permite-se que eles sejam explicitamente especificados. Além disso, se o valor de uma determinada constante de enumeração não for definido explicitamente, ele assumirá o valor da constante anterior na sequência acrescido de `1`. Como exemplo, considere:

```
enum { AZUL = -3, VERMELHO, PRETO = 20, BRANCO } umaCor;
```

Nesse exemplo, as constantes `VERMELHO` e `BRANCO` que não tiveram valores explicitamente atribuídos, receberão implicitamente os valores `-2` e `21`, nessa ordem.

1.14 Instruções

1.14.1 Tipos de Instruções

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

- Expressão (incluindo chamada de função)
- Instrução **return**
- Estrutura de controle

Em C, uma instrução pode aparecer apenas dentro de uma função (v. **Capítulo 2**). Além disso, 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, como **#include** e **#define**
- Comentários
- Uma linha que continua na linha seguinte
- Instruções ou declarações que terminam com fecha-chaves

1.14.2 Sequências de Instruções

Uma **sequência** (ou **bloco**) **de instruções** consiste em uma ou mais instruções confinadas entre chaves e pode ser inserida em qualquer local de um programa onde uma única instrução é permitida. Blocos de instruções também podem ser aninhados dentro de outros blocos. 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.

1.14.3 Instruções Vazias

Uma **instrução vazia** é uma instrução que não executa nenhuma tarefa e pode ser inserida 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.

Um ponto e vírgula que representa instrução vazia pode ser acidental (i.e., decorrente de um acidente de trabalho) ou proposital (quando ela corresponde exatamente àquilo que o programador deseja obter). Instruções vazias acidentais podem ser deletérias ou não. Quando uma instrução vazia é proposital, recomenda-se que o programador a coloque numa linha separada e acompanhada de comentário explicativo para deixar claro que essa instrução é realmente proposital (e não acidental).

1.15 Estruturas de Controle

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. As estruturas de controle de C podem ser classificadas 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.

1.15.1 Laços de Repetição

Laços de repetição permitem controlar o número de vezes que uma instrução ou sequência de instruções é executada. Em C, existem três laços de repetição: **while**, **do-while** e **for**.

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 1–1** ilustra o funcionamento do laço de repetição **while**. Note que, se inicialmente a expressão resultar em zero, o corpo do laço não será executado nenhuma vez.

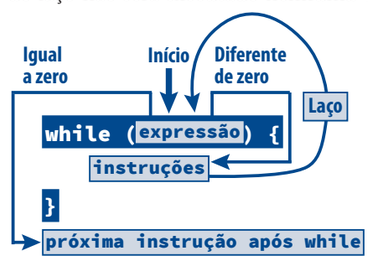

FIGURA 1–1: DIAGRAMA DE FUNCIONAMENTO DO LAÇO WHILE

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**. 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.

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 1–2**, que ilustra o funcionamento da instrução **do-while**:

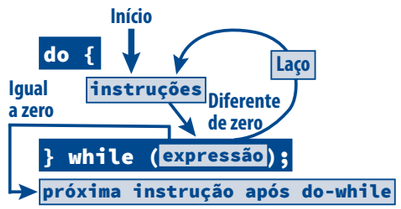

FIGURA 1–2: DIAGRAMA DE FUNCIONAMENTO DO LAÇO DO-WHILE

Comparando a **Figura 1–1**, que mostra o funcionamento de uma instrução **while**, com a **Figura 1–2**, referente ao funcionamento de um laço **do-while**, pode-se 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.

As mesmas recomendações de uso e posicionamento de chaves apresentadas para o laço **while** são válidas aqui.

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 (expressão1; expressão2; expressão3)
   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 instrução **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, *expressão3* é avaliada e retorna-se ao **Passo 2**.

A **Figura 1–3** ilustra o funcionamento da instrução **for**.

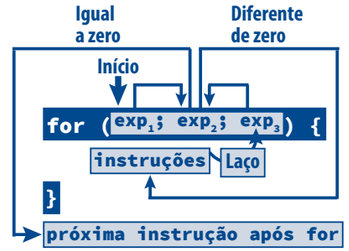

FIGURA 1–3: DIAGRAMA DE FUNCIONAMENTO DO LAÇO FOR

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

```
expressão1;

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

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.

Conforme foi mencionado, 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 instrução **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. **adiante**). Também, 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.

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.

Laços de Repetição 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)
{
      /* Laço externo */
   for (int i = 1; i <= 2; ++i) {
      printf( "\nIteracao do laco for externo no. %d\n", i );

         /* Laço interno (aninhado) */
      for (int 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, ao passo que 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.

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

1.15.2 Desvios Condicionais

Instruções de **desvios condicionais** permitem desviar o fluxo de execução de um programa dependendo do resultado da avaliação de uma expressão (condição).

if-else

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

```
if (expressão)
   instrução1;
else
   instrução2;
```

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 1–4** ilustra o funcionamento da instrução **if-else**.

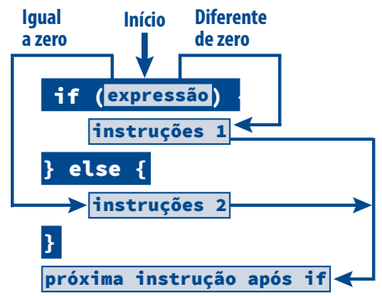

Figura 1–4: Diagrama de Funcionamento da Instrução if-else

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 1–5** ilustra o funcionamento da instrução **if-else** com a parte **else** omitida.

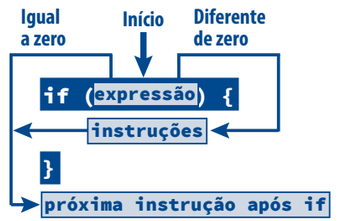

Figura 1–5: Diagrama de Funcionamento da Instrução if sem else

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**. 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**, como, por exemplo:

```
if (!teste) {
   printf("\nVoce nao digitou um numero inteiro\n");
} else if (x % 2 == 0) {
   printf("\nO numero introduzido e' par\n");
} else {
   printf("\nO numero introduzido e' impar\n");
}
```

O uso de chaves em instruções **if-else** aninhadas previne casamentos incorretos entre partes **if** e partes **else**.

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-constanten:
        instruçãon;
    default:
        instruçãod;
}
```

A expressão entre parênteses que segue imediatamente a palavra **switch** deve resultar num valor inteiro; 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 uma ramificação **case** e houver uma parte **default**, a instrução seguindo essa parte será executada.

A **Figura 1–6** ilustra o funcionamento da instrução **switch-case**.

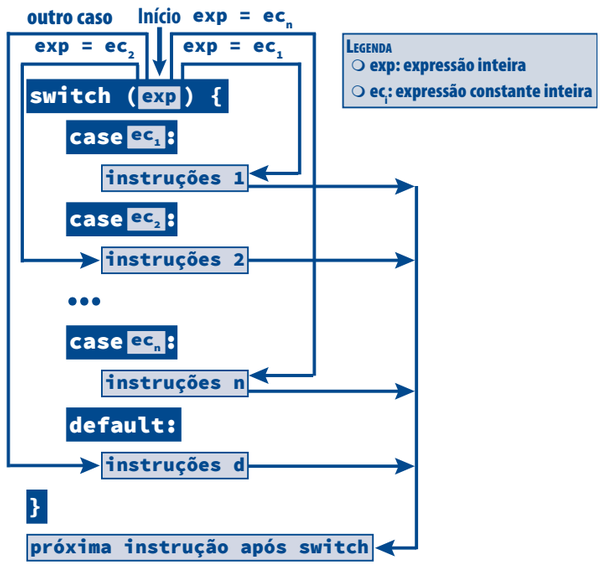

Figura 1–6: Diagrama de Funcionamento da Instrução switch-case

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**.

1.15.3 Desvios Incondicionais

Desvios incondicionais permitem o desvio do fluxo de execução de um programa independentemente da avaliação de qualquer expressão.

break

A instrução **break** foi apresentada na **Seção 1.15.2** 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**).

O uso indiscriminado de desvios incondicionais, como **break**, pode tornar os programas difíceis de ser lidos. Portanto esses desvios devem ser usados com cautela.

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 1.15.1**.
- Laço **do-while**. O fluxo de execução continua com a avaliação da expressão condicional do laço **do-while** e prossegue conforme descrito na **Seção 1.15.1**.
- Laço **for**. O fluxo de execução continua com a avaliação da terceira expressão do laço **for** e prossegue conforme descrito na **Seção 1.15.1**.

Na **Seção 1.15.1**, apresentou-se uma sequência de instruções que poderia ser usada em substituição a uma instrução **for**. Como pode ser inferido do que foi exposto acima, aquela equivalência não é válida quando o corpo do laço **for** contém uma instrução **continue**.

Seja comedido no uso da instrução **continue**, pois seu uso excessivo tende a prejudicar a legibilidade de um programa.

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 */
```

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 2.4**).
- 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.

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 quer 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.

1.16 Outros Operadores

1.16.1 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.

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 e do operador vírgula e sua associatividade é à direita.

1.16.2 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 expressão de um laço **for**. Em outras situações, é melhor evitar o uso do operador vírgula e, ao invés disso, escrever as expressões em instruções separadas.

1.16.3 O Operador sizeof e o Tipo size_t

O operador **sizeof** é um operador unário e de precedência e associatividade iguais às dos demais operadores unários. Esse operador pode receber como operando um tipo de dado, uma constante, uma variável ou uma expressão.

Quando aplicado a um tipo de dado, o operador **sizeof** resulta no número de bytes necessários para alocar uma variável do mesmo tipo. Nesse caso, o operando deve vir entre parênteses. Por exemplo, `sizeof(double)` resulta no número de bytes ocupados por uma variável do tipo **double**.

O resultado do operador **sizeof** é do tipo **size_t**, que é um tipo inteiro sem sinal (v. adiante) definido em vários cabeçalhos da biblioteca padrão de C.

Um tipo inteiro com sinal, como é o caso do tipo primitivo **int** possui um bit de sinal, que indica se um valor desse tipo é positivo ou negativo. Por outro lado, um tipo inteiro sem sinal, como é o caso do tipo derivado **size_t**, não possui tal bit, pois todos seus valores são considerados sempre positivos ou zero. É sempre recomendável não misturar inteiros com e sem sinal (incluindo **size_t**) numa mesma expressão, a não ser que essa expressão seja de atribuição.

Quando o operador **sizeof** é aplicado a uma constante ou variável, o resultado é o número de bytes necessários para armazenar a constante ou variável, respectivamente. Quando aplicado a uma expressão, o operador **sizeof** resulta no número de bytes que seriam necessários para conter o resultado da expressão se ela fosse avaliada; i.e., a expressão em si não é avaliada. Quando o operando do operador **sizeof** é uma expressão, ela não precisa ser colocada entre parênteses; porém, o uso de parênteses com **sizeof** é sempre recomendado para prevenir erros.

1.17 Programas Monoarquivo

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

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

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

Programas monoarquivo são aqueles que consistem de um único arquivo-fonte.

1.17.1 Estrutura de um Programa Simples em C

Um programa simples em C consistindo de um único arquivo fonte possui o formato apresentado esquematicamente na **Figura 1–7**.

```
#include <stdio.h> /* Para usar printf() */
/* Outros cabeçalhos padrão utilizados no programa */
/* Definições de constantes simbólicas utilizadas */
int main(void)
{
    /* Definições de variáveis do programa */
    /* Instruções que executem as ações necessárias */
    /* para funcionamento do programa.              */
    return 0;
}
```

Figura 1–7: Programa Monoarquivo Simples em C

A instrução:

```
return 0;
```

que aparece ao final da maioria dos programas escritos em C indica que o programa foi executado sem anormalidades.

1.17.2 Como Criar um Programa-fonte

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

1.17.3 Criando um Programa Executável

Para obter um programa executável a partir de um programa-fonte usando um ambiente de desenvolvimento (IDE), provavelmente, você precisará apenas de um clique de mouse. Essa é a opção mais rápida e fácil de se obter o resultado desejado.

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

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

Emitindo esse comando, você invocará o compilador e o linker simultaneamente com as opções recomendadas para compilar e ligar o programa-fonte denominado ***arq-fonte*** e o resultado será um arquivo executável denominado ***arq-executável***.

1.18 Endereços e Ponteiros

1.18.1 Endereços

Qualquer variável definida num programa em C possui um endereço que indica o local onde ela encontra-se armazenada em memória. Frequentemente, é necessário utilizar o endereço de uma variável num programa, em vez de seu próprio conteúdo. O endereço de uma variável pode ser determinado por meio do uso do operador de endereço, representado por &. Suponha, por exemplo, a existência da seguinte definição de variável:

```
int  x;
```

então, a expressão:

```
&x
```

resulta no endereço atribuído à variável `x` quando ela é alocada.

1.18.2 Ponteiros

Ponteiro é uma variável capaz de conter um endereço em memória. Um ponteiro que contém um endereço em memória válido é dito apontar para tal endereço. Um ponteiro pode apontar para uma variável de qualquer tipo e ponteiros que apontam para variáveis de tipos diferentes são também considerados de tipos diferentes e incompatíveis.

Para definir um ponteiro é preciso especificar o tipo de variável para a qual ele pode apontar. Assim uma definição de ponteiro em C tem o seguinte formato:

tipo-apontado *variável-do-tipo-ponteiro;

Por exemplo:

```
int *intPtr;
```

define a variável `intPtr` como um ponteiro capaz de apontar para qualquer variável do tipo **int**. Nesse exemplo, diz-se que o tipo da variável `intPtr` é ponteiro para **int** ou, equivalentemente, que seu tipo é **int** *.

O asterisco que acompanha a definição de qualquer ponteiro é denominado **definidor de ponteiro** e a posição exata dele entre o tipo apontado e o nome do ponteiro é irrelevante. Se mais de um ponteiro de um mesmo tipo estiver sendo definido de forma abreviada, deve haver um asterisco para cada ponteiro, como, por exemplo:

```
int *ponteiroParaInteiro1, *ponteiroParaInteiro2;
```

Ponteiros podem ser iniciados da mesma forma que outros tipos de variáveis. Por exemplo, a segunda definição a seguir:

```
int    meuInteiro = 5;
int   *intPtr = &meuInteiro;
```

define a variável `intPtr` como um ponteiro para **int** e inicia seu valor com o endereço da variável `meuInteiro`. No caso de iniciação de um ponteiro com o endereço de uma variável, como no último exemplo, a variável já deve ter sido declarada.

Esquematicamente, as duas variáveis do último exemplo apareceriam em memória como na **Figura 1–8**.

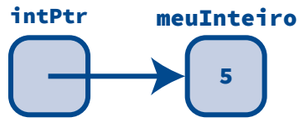

Figura 1–8: Representação Esquemática de um Ponteiro

Ao contrário do que ocorre com os tipos aritméticos, as regras para compatibilidade entre ponteiros são rígidas:

- Dois ponteiros só são compatíveis se eles forem exatamente do mesmo tipo.
- Um ponteiro só é compatível com o endereço de uma variável se a variável for do tipo para o qual o ponteiro aponta.

Infelizmente, apesar de a desobediência a essas regras quase sempre causar graves problemas, alguns compiladores de C apenas emitem mensagens de advertência quando é o caso.

1.18.3 Indireção de Ponteiros

O acesso ao conteúdo do espaço em memória apontado por um ponteiro é efetuado por meio de uma operação de **indireção**. Isto é, a indireção de um ponteiro resulta no valor contido no espaço em memória para onde o ponteiro aponta. Para acessar esse valor, precede-se o ponteiro com o **operador de indireção**, representado por *.

O acesso ao conteúdo de uma variável usando seu nome é denominado acesso direto. O acesso ao conteúdo de uma variável por meio da aplicação do operador de indireção sobre um ponteiro que aponta para a variável é denominado acesso indireto porque ele requer dois passos: **(1)** acesso ao conteúdo do ponteiro para determinar o endereço da variável apontada e **(2)** de posse do endereço dessa variável, acesso ao seu conteúdo. A denominação operador de indireção é derivada do fato de o operador que usa esse nome permitir acesso indireto.

Os operadores de indireção * e de endereço **&** fazem parte do mesmo grupo de precedência em que estão todos os outros operadores unários de C. A precedência desses operadores é a segunda mais alta dentre todos os operadores da linguagem C e a associatividade deles à direita.

Pode-se alterar o conteúdo de uma variável apontada por um ponteiro utilizando o operador de indireção em conjunto com qualquer operador com efeito colateral. Por exemplo, considerando as definições:

```
double  umDouble = 3.6;
double *ptrDouble = &umDouble;
```

a instrução:

```
*ptrDouble = 1.6; /* Altera INDIRETAMENTE o valor de umDouble*/
```

atribuiria o valor `1.6` ao conteúdo da posição de memória apontada por `ptrDouble`. É interessante notar que essa última operação equivale a modificar o valor da variável `umDouble` sem fazer referência direta a ela. Isto é, a última instrução é equivalente a:


```
umDouble = 1.6; /* Altera DIRETAMENTE o conteúdo de umDouble */
```

As duas últimas instruções são funcionalmente equivalentes, mas a segunda é mais eficiente pois o acesso ao conteúdo da variável é feito diretamente.

Na prática, ponteiros são usados quando se tem um espaço em memória associado a uma variável, mas não se tem acesso ao nome dela (v. **Seção 2.1**) ou quando se tem um espaço em memória que não está associado a nenhum nome de variável. Nesse último caso, o espaço em memória é denominado variável anônima (v. **Capítulo 9**).

1.18.4 Ponteiro Nulo

Um **ponteiro nulo** é um ponteiro que não aponta para nenhum endereço válido e, mais importante, esse fato é reconhecido por qualquer sistema hospedeiro. Um ponteiro torna-se nulo quando lhe é atribuído o valor inteiro `0`. Por exemplo:

```
int *ptr;
ptr = 0;  /* Torna p um ponteiro nulo */
```

Qualquer tentativa de acesso ao conteúdo apontado por um ponteiro nulo causa aborto de programa em qualquer hospedeiro no qual ele esteja sendo executado. Em outras palavras, a aplicação do operador de indireção a um ponteiro nulo é reconhecida como irregular por qualquer sistema operacional e causa um erro de execução (i.e., o programa é encerrado abruptamente).

A constante simbólica **NULL**, definida em vários cabeçalhos da biblioteca padrão de C, torna expressões de atribuição e comparação envolvendo ponteiros nulos mais legíveis.

1.19 Exercícios de Revisão

A Linguagem C Padrão (Seção 1.1)

1. (a) O que é padronização de uma linguagem de programação? (b) Por que um padrão de linguagem de programação é desejável? (c) O que é o padrão ISO da linguagem C? (d) Como é popularmente conhecido o padrão corrente da linguagem C?
2. O significa dizer que uma característica de C é dependente de implementação?

Identificadores, Tipos Primitivos e Constantes (Seção 1.2)

3. (a) Quais são as regras para formação de identificadores da linguagem C? (b) O que são palavras-chave? (c) O que são palavras reservadas da linguagem C?
4. Quais dos seguintes nomes não podem ser utilizados como identificadores em C?

   (a) var (c) int (d) $a e) a$
   (g) double (i) VOID (j) void (l) _10
5. Descreva os seguintes tipos primitivos da linguagem C:
   (a) **int**
   (b) **char**
   (c) **double**
6. Quais são os dois formatos de escrita de constantes reais?
7. (a) O que é uma sequência de escape? (b) Em quais situações sequências de escape se fazem necessárias?
8. As sequências de escape `'\n'` e `'\t'` são as mais usadas em escrita na tela. Qual é o efeito de cada uma delas?
9. (a) O que é um string constante? (b) Qual é a diferença entre string constante e caractere constante?

10. (a) Quando um string constante é muito extenso, como ele pode ser separado em partes? (b) Qual é a vantagem que essa facilidade oferece para o programador?

11. (a) A expressão `'A' + 'B'` é legal? (b) Em caso afirmativo, qual é o resultado dessa expressão?

12. Qual é o resultado de `'Z' - 'A'`?

13. (a) Qual é o resultado de `'9' - '0'`? (b) Em geral, qual é o resultado de uma expressão do tipo `'d' - '0'`, em que `d` é um dígito?

Operadores Básicos (Seção 1.3)

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

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

16. (a) O que é efeito colateral de um operador? (b) Quais são os operadores de C que possuem efeito colateral?

17. Quais são os operadores de C que possuem ordem de avaliação de operandos definida?

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

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

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

20. (a) O que é um operador relacional? (b) Quais são os operadores relacionais de C? (c) Quais são os possíveis resultados de aplicação de um operador relacional? (d) Os operadores relacionais de C têm a mesma precedência?

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

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

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

[1] `x > 0 || ++y < 10`

[2] `++y < 10 || x > 0`

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

[1] `y%x && x > 0`

[2] `x > 0 && y%x`

25. Qual é o resultado da expressão `1/3 + 1/3 + 1/3`?

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

Definições de Variáveis (Seção 1.4)

27. (a) O que é uma definição de variável? (b) Que informação é provida para o compilador por uma definição de variável?

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

Operadores de Atribuição (Seção 1.5)

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


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

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

30. Quais são os operadores de atribuição aritmética de C?

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

(a) `d = i = 2.5;`

(b) `i = d = 2.5;`

Conversões de Tipos (Seção 1.6)

32. (a) O que é conversão implícita? (b) Por que às vezes ela é necessária? (c) Em que situações um compilador efetua conversões implícitas? (d) Cite alguns problemas decorrentes de conversões implícitas.

33. (a) Um computador é capaz de realizar a operação `2 + 2.5`? (b) É permitida a escrita dessa expressão num programa em C? (c) O resultado dessa operação é inteira ou real?

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

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

Incremento e Decremento (Seção 1.7)

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

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

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

38. Considere o seguinte trecho de programa:

```
int  i, j = 4;

i = j * j++;
```

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

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

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

Comentários (Seção 1.8)

40. (a) Como comentários podem ser inseridos num programa? (b) Como um compilador lida com comentários encontrados num programa? (c) Por que comentários são necessários num programa? (d) Por que um programa sem comentários compromete seu entendimento?

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

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

A Biblioteca Padrão de C (Seção 1.9)

43. (a) No contexto de programação, o que é uma biblioteca? (b) Como a biblioteca de uma linguagem de programação auxilia o programador? (c) É possível programar utilizando uma linguagem de programação que não possui biblioteca?

44. (a) O que é a biblioteca padrão de C? (b) Por que funções dessa biblioteca não são estritamente consideradas partes integrantes da linguagem C? (c) Por que se diz que a biblioteca padrão de C é um apêndice dessa linguagem?

45. O que é um arquivo de cabeçalho?

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

47. Por que cabeçalhos da biblioteca padrão de C são incluídos usando-se os símbolos `<` e `>`, enquanto na inclusão de cabeçalhos que não pertencem à biblioteca padrão são usadas aspas `"`.

Escrita de Dados na Tela (Seção 1.10)

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

49. (a) O que é um especificador de formato? (b) A que deve corresponder um especificador de formato encontrado num string de formatação numa chamada de **printf()**?

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

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

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

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

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

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

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

(a) `printf("%f", x);`

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

(c) `printf("%3.2f", x);`

(d) `printf("%4.2f", x);`

Leitura de Dados via Teclado 1: Frágil (Seção 1.11)

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

56. (a) O que é prompt e qual é sua importância num programa interativo? (b) Por que a ausência de prompt ou a apresentação de um prompt impreciso pode ser responsável por erros num programa?

57. (a) Descreva o funcionamento da função **getchar()**. (b) Se a função **getchar()** lê apenas caracteres, por que o tipo do valor que ela retorna é **int**, e não **char**? (c) Por que nem sempre a função **getchar()** interrompe a execução do programa e espera que o usuário digite um caractere?

58. Por que leitura de dados via teclado é sempre mais complicada do que escrita de dados na tela?

59. Sabe-se que, via teclado, é possível introduzir apenas caracteres. Então, como é possível um programa receber um número como entrada?

60. Qual deve ser o conteúdo de um string de formatação da função **scanf()**?

61. O que significam os três pontos no protótipo de **scanf()**?

62. Descreva detalhadamente o funcionamento da função **scanf()**.

63. (a) O que significa o valor retornado pela função **scanf()**? (b) Como esse valor deve ser utilizado?

64. Suponha que `x` seja uma variável do tipo **int**. (a) O que há de errado com cada uma das seguintes chamadas de **scanf()** a seguir? (b) Supondo que essas instruções fazem parte de um programa, esses erros serão detectados durante a compilação ou execução do programa?

(i) `scanf("%d", x);`

(ii) `scanf("%lf", &x);`

65. Suponha que a função **scanf()** seja chamada como no seguinte trecho de programa:

```
int x, retorno;

retorno = scanf("%d", &x);
```

O que significa quando o valor atribuído à variável `retorno` é:

(a) `0`?

(b) `1`?

(c) **EOF**?

66. (a) Existe diferença entre os especificadores **%d** e **%i** quando eles são utilizados com **scanf()**? (b) Existe diferença entre esses mesmos especificadores quando eles são utilizados com **printf()**?

67. O que os strings de formatação das funções **printf()** e **scanf()** têm em comum?

68. Suponha que `x` e `y` sejam variáveis do tipo **int**. Quantos valores cada uma das chamadas de **scanf()** a seguir espera ler?

(a) `scanf("%d", &x, &y);`

(b) `scanf("%d %d", &x);`

Constantes Simbólicas (Seção 1.12)

69. (a) O que é uma constante simbólica? (b) Como uma constante simbólica é definida? (c) Que vantagens são obtidas com o uso de constantes simbólicas num programa?

70. Por que não se deve terminar uma definição de constante simbólica com ponto e vírgula?

Enumerações (Seção 1.13)

71. (a) O que é uma enumeração? (b) Para que servem enumerações? (c) Como uma enumeração é definida? (d) Que cuidados o programador deve tomar quando usa enumerações?

72. (a) O que são constantes de enumeração? (b) Como elas são definidas? (c) Quais são as regras de atribuição de valores a constantes de enumerações? (d) Por que, muitas vezes, os valores atribuídos a constantes de uma enumeração não são importantes?

73. Dois tipos de enumerações diferentes podem ter constantes com o mesmo nome num mesmo arquivo de programa?

74. Qual é a vantagem obtida com o uso de enumerações num programa?

75. Por que se diz que o uso correto de enumerações depende apenas do programador, e não do compilador?

76. Qual é o valor atribuído a cada constante na seguinte definição de tipo enumeração?

```
enum {C1 = -1, C2, C3 = 0, C4} umaEnumeracao;
```

Instruções (Seção 1.14)

77. Que tipos de construções são consideradas instruções em C?

78. O que é um bloco de instruções?

79. Por que quando uma expressão constitui uma instrução, ela faz sentido apenas se contiver operadores com efeito colateral?

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

Estruturas de Controle (Seção 1.15)

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

82. Como estruturas de controle são categorizadas?

83. (a) O que são desvios condicionais? (b) Quais são os desvios condicionais da linguagem C?

84. (a) O que são desvios incondicionais? (b) Quais são os desvios incondicionais da linguagem C?

85. (a) O que é um laço de repetição? (b) Quais são os laços de repetição da linguagem C?

86. 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**

87. (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--;
}
```

88. (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);
```

89. (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 */
```

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

```
int i = 10;

while (i++ > 0) {
   ;
}

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

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

```
int i;

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

92. (a) O que é condição de parada de um laço de repetição? (b) Qual é a condição de parada do seguinte laço **while**?


```
int i = 100;

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

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

94. 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);
```

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

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

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

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

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

100. Por que a sequência de instruções equivalente a uma instrução **for** apresentada na **Seção 1.15.1** nem sempre é válida?

Outros Operadores (Seção 1.16)

101. Descreva o funcionamento do operador condicional.

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

103. Qual é o resultado da seguinte expressão?

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

104. Por que não é recomendado usar expressões formadas com o operador condicional como operandos desse mesmo operador?

105. 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.

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

106. Qual é a precedência do operador vírgula?

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

108. Para que serve o operador **sizeof**?

109. (a) Qual é o tipo do valor resultante da aplicação do operador **sizeof**? (b) Esse tipo é primitivo ou derivado?

110. O que há de incomum com o operador **sizeof** em relação a outros operadores de C? Em outras palavras, que característica única (i.e., não encontrada em nenhum outro operador) o operador **sizeof** possui?

111. Se o operador **sizeof** não avalia uma expressão usada como seu operando, como ele pode obter seu resultado?

112. Que cuidado deve ser tomado quando se lida com o valor retornado pelo operador **sizeof**?

113. Por que valores inteiros com sinal não devem ser misturados com valores inteiros sem sinal numa mesma expressão que não seja de atribuição?

114. Por que o resultado da aplicação do operador **sizeof** pode ser obtido pelo compilador; i.e., antes mesmo da execução de um programa?

Programas Monoarquivo (Seção 1.17)

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

116. (a) O que é um sistema de execução com hospedeiro? (b) O que é um sistema livre? (c) O que é um programa de console?

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

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

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

Endereços e Ponteiros (Seção 1.18)

120. Como se obtém o endereço de uma variável?

121. Uma variável de um tipo primitivo sempre contém um valor válido do tipo com o qual ela é definida, mesmo quando ela não é iniciada. No entanto, uma variável que representa um ponteiro nem sempre contém um valor válido. Por quê?

122. (a) Para que serve o operador de indireção? (b) Como deve ser um operando desse operador? (c) Qual é o resultado da aplicação desse operador? (d) Por que o operador de indireção recebe essa denominação?

123. (a) O operador de endereço pode ser aplicado a qualquer tipo de variável? (b) E o operador de indireção?

124. O que é e para que serve um ponteiro nulo?

125. O que há de errado nas seguintes definições de variáveis?

```
int  x1 = -1, x2 = 5;
int *p1 = &x1, p2 = &x2;
```

126. Suponha que `x` seja uma variável. (a) Qual é o significado da expressão `&*&x`? (b) Qual é o significado da expressão `*&*&x`?

1.20 Exercícios de Programação

EP1.1 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.

EP1.2 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
    40             4.4
   ...             ...
    80            26.7
```

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

EP1.3 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.]

EP1.4 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.

EP1.5 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.

EP1.6 **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:

$a = m^2 - n^2$

$b = 2 \cdot m \cdot n$

$c = m^2 + n^2$

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.]

EP1.7 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.