APÊNDICE C

RESPOSTAS PARA OS EXERCÍCIOS DE REVISÃO

Capítulo 1 — Introdução às Linguagens de Programação

1. Consulte a **Seção 1.1**.
2. Porque uma linguagem de máquina é específica para um processador ou uma família de processadores.
3. Palavras mnemônicas e variáveis simbólicas.
4. É uma linguagem que não favorece legibilidade, manutenibilidade e portabilidade. Ela está mais próxima do computador do que do programador.
5. (a) É o contrário de uma linguagem de baixo nível. (b) C, C++ e Java.
6. Linguagens de alto nível favorecem legibilidade, manutenibilidade e portabilidade de programas. O uso de uma linguagem de baixo nível favorece apenas a eficiência de programas.
7. Ambas são linguagens de baixo nível. Cada processador possui uma linguagem de máquina e uma linguagem assembly que são únicas.
8. (a) Variável é um espaço em memória onde dados que serão processados por um programa são armazenados. (b) É uma variável que possui um identificador associado a ela.
9. Linguagens de alto nível favorecem o bom estilo de programação em termos de legibilidade, manutenibilidade e portabilidade.
10. Devido à eficiência do código executável que pode ser produzido por um bom programador de assembly. Os melhores programadores de linguagens de alto nível não conseguem produzir o código mais eficiente possível porque eles dependem do compilador ou interpretador que irá, em última instância, produzir o código executável.
11. Consulte a **Seção 1.2.1**.
12. Não. Nem existe, na prática, mais de um tradutor assembler para um mesmo processador (v. **Seção 1.2.1**).
13. (a) Não. (b) Um compilador ou um interpretador.
14. (a) Consulte a **Seção 1.2.2**. (b) Consulte a **Seção 1.2.3**. (c) Consulte essas duas seções.
15. Consulte a **Seção 1.2.2**.

16. (a) É um programa escrito em assembly ou numa linguagem de alto nível. (b) É um programa-fonte compilado. (c) Programa executável é um programa que já passou pelas fases de compilação e ligação e está pronto para ser executado.

17. (a) Editor de texto ou de programas. (b) Compilador. (c) Linker.

18. Sim. Existem compiladores que produzem código binário mais eficiente para um mesmo programa-fonte do que outros compiladores.

19. Não. Na maioria das vezes, um programa compilado precisa passar pelo posterior processo de ligação.

20. Quase sempre o linker é responsável pela criação de programas executáveis.

21. Consulte a **Seção 1.2.3**.

22. Consulte a **Seção 1.2.3**.

23. Consulte a **Seção 1.2.3**.

24. Consulte a **Seção 1.3**.

25. Sim, porque linguagens de baixo nível não favorecem a legibilidade de qualquer programa.

26. Não. Legibilidade não depende apenas de linguagem; ela também depende do programador.

27. A portabilidade de um programa depende de linguagem de programação, estilo de programação, plataforma e da natureza do programa.

28. Uma forma de programação que favorece as qualidades descritas na **Seção 1.3**.

29. Porque, nesse caso, a eficiência não depende apenas do programador. Ela também depende do compilador utilizado.

30. Veja os últimos dois parágrafos da **Seção 1.2.3**.

31. Quando eles produzem os mesmos resultados usando os mesmos dados de entrada.

32. Porque muitas vezes a escrita de programas eficientes requer muito conhecimento sobre como um programa é compilado, ligado e executado.

33. (a) É um programa que dialoga com o usuário. (b) Editor de texto, navegador de internet, IDE. (c) Controlador de dispositivo e vírus de computador.

34. Consulte a **Seção 1.4**.

35. É um programa que usa apenas console para entrada e saída de dados.

36. Consulte a **Seção 1.5**.

37. Consulte a **Seção 1.5**.

38. (a) Coloração de sintaxe é uma característica de editores de programas que consiste na apresentação de diferentes componentes de programas com cores distintas. (b) Coloração de sintaxe facilita a identificação visual desses componentes.

39. Porque a coloração de sintaxe deixa de ser eficiente.

40. Um carregador (*loader*) prepara um programa para execução.

41. É uma ferramenta de desenvolvimento que auxilia a depuração de programas.

42. Consulte a **Seção 1.6.2**.

43. IDE possui uma interface que integra um conjunto de ferramentas de desenvolvimento, enquanto *toolchain* é um conjunto de ferramentas de desenvolvimento sem integração.

44. (a) É o mesmo que *loader* (v. acima). (b) Sim, pois CodeBlocks permite executar programas sem abandoná-lo.

45. GDB.

46. Consulte a **Seção 1.6.2**.

47. Consulte a **Seção 1.6.2**.

48. `-Wall`.
49. Ele cria programas executáveis.
50. (a) Linker é um programa responsável pela criação de programas executáveis. (b) Compilação cria programas-objeto e ligação cria programas executáveis.
51. (a) É um conjunto de funções para leitura de dados via teclado que facilitam a criação de programas robustos. (b) Essa biblioteca possibilita a criação de programas robustos usando instruções de entrada de dados bastante simples.
52. (a) CodeBlocks é um IDE. (b) É um ótimo ambiente de desenvolvimento, é fácil de usar e é grátis.
53. Consulte a **Seção 1.7.3**.
54. Consulte a **Seção 1.7.4**.
55. Consulte o final da **Seção 1.7.4**.
56. Consulte a **Seção 1.8**.
57. Provavelmente, porque não sabem qual é o tipo de programas que estão construindo ou desconhecem a relação entre Windows e DOS.
58. Consulte a **Seção 1.9**.
59. Consulte o **Prefácio** e a **Seção 1.9**.
60. Consulte a **Seção 1.9**.
61. Consulte a **Seção 1.9**.
62. Consulte a **Seção 1.9**.
63. Consulte a **Seção 1.9**.

Capítulo 2 — Introdução à Construção de Algoritmos

1. Consulte a **Seção 2.1**.
2. (a) Ambos possuem entrada, saída e processamento. (b) Receitas culinárias não são precisas como devem ser algoritmos.
3. São dados que um algoritmo estão capacitados para processar.
4. Consulte a **Seção 2.1**.
5. São aqueles que produzem os mesmos resultados quando recebem os mesmos dados como entrada.
6. Reconhecimento facial, escrita de livro, entendimento de linguagem natural.
7. Consulte a **Seção 2.2**.
8. Depende da experiência do programador que irá converter o algoritmo em programa.
9. Reúso de código é o aproveitamento de trechos de um programa-fonte na criação de outro programa-fonte.
10. Consulte a **Seção 2.3**. (b) Porque linguagem algorítmica não é nem linguagem natural nem linguagem de programação; ela é uma linguagem artificial.
11. Pseudolinguagem facilita a construção e a tradução de algoritmos. Por outro lado, linguagens naturais são inerentemente ambíguas e dependentes de contexto.
12. Consulte a **Seção 2.3**.
13. (a) É código (programa-fonte) escrito em pseudolinguagem. (b) Não.
14. Conteúdo, endereço e nome (identificador).

15. É uma instrução que atribui um valor a uma variável.

16. Significa que a `x` será atribuído seu valor corrente acrescido de `2`.

17. Não, mas o uso de declarações de variáveis num algoritmo facilita sua tradução em programa.

18. Consulte a **Seção 2.5**.

19. Consulte a **Seção 2.5**.

20. Consulte a **Seção 2.5**.

21. Consulte a **Seção 2.5**.

22. Em grupos de precedência, em cada um dos quais os operadores têm a mesma precedência.

23. (a) Para comparar valores numéricos. (b) Consulte a **Seção 2.5.3**.

24. verdadeiro ou falso

25. Operadores lógicos são operadores que combinam constantes, variáveis e expressões lógicas para formar expressões também lógicas.

26. Consulte a **Seção 2.5.4**.

27. É uma tabela que apresenta os resultados da aplicação de um operador lógico em função dos possíveis valores de seus operandos.

28. Basicamente, eles servem para alterar precedência e associatividade de operadores, mas também podem facilitar a leitura de expressões complexas.

29. Na tabela a seguir, suponha que V representa verdadeiro e F representa falso. (Lembre-se que a conjunção e tem precedência maior do que a disjunção ou.)

A	B	C	B e C	A ou B e C
V	V	V	V	V
F	V	V	V	V
V	F	V	F	V
V	V	F	F	V
F	V	F	F	F
F	F	F	F	F
V	F	F	F	V
F	F	V	F	F

30. (a) verdadeiro. (b) verdadeiro. (c) verdadeiro. (d) falso. (e) falso. (f) falso.

31. Para ler dados que serão processados por um algoritmo (ou programa).

32. A instrução leia lê dados num meio de entrada e armazena-os em variáveis.

33. Para exibição de informação para o usuário de um algoritmo.

34. Consulte a **Seção 2.6**.

35. Uma instrução leia deve conter apenas variáveis que indicam os locais nos quais os valores lidos serão armazenados.

36. É uma sequência de caracteres entre aspas.

37. Consulte a **Seção 2.7**.

38. São instruções capazes de alterar o fluxo natural de execução de um algoritmo.

39. Desvios condicionais, desvios incondicionais e laços de repetição.

40. Consulte a **Seção 2.7.1**.

41. Na instrução **enquanto**-**faça**, a expressão condicional é avaliada antes do corpo do laço; na instrução **faça**-**enquanto**, a expressão condicional é avaliada depois do corpo do laço.
42. (a) Bloco é uma sequência de instruções que não apresentam dependências entre si. (b) Endentação é um pequeno espaço em branco horizontal que indica subordinação de uma instrução em relação a outra. (c) Instruções que fazem parte de um mesmo bloco não apresentam endentações entre si.
43. (a) i1 e i5. (b) i3, i4 e i5. (c) i2 e i5. (d) b1 deve ser **F** e b2 deve ser **F**. O valor de b3 pode ser **V** ou **F**.
44. Na instrução 1, a variável L assume o valor **verdadeiro** quando X é igual a Y e assume **falso** em caso contrário. O mesmo ocorre na instrução 2. Portanto essas instruções são funcionalmente equivalentes.
45. Zero.
46. Consulte a **Seção 2.8**.
47. É um nome que sugere o tipo de informação armazenado na variável.
48. (a) Comentários servem para documentar e esclarecer trechos obscuros de algoritmos. (b) Comentários facilitam a leitura de algoritmos.
49. (a) É endentação que usa sempre o mesmo número de espaços e os mesmos formatos. (b) Porque melhora a legibilidade de algoritmos.
50. Consulte a **Seção 2.9**.
51. Consulte a **Seção 2.9.1**.
52. (a) Consulte a **Seção 2.9.3**. (b) Executando-o com casos de entrada qualitativamente diferentes e também manualmente, como um algoritmo.
53. (a) São casos de entrada que produzem resultados que têm significados distintos. (b) Eles testam todas as saídas possíveis de um programa.
54. Exemplos de execução podem ser usados como casos de entrada durante a fase de testes do algoritmo e do programa resultante.
55. Porque erros podem ser introduzidos durante a implementação do programa.
56. Porque um programa pode estar incorreto e, então, será necessário retornar a alguma etapa do processo de desenvolvimento.

Capítulo 3 — Introdução à Linguagem C

1. (a) É a especificação oficial da linguagem. (b) Principalmente, por razões de portabilidade.
2. (a) É o padrão da linguagem C definido por um comitê ISO. (b) C11.
3. Significa que ela não é especificado por nenhum padrão aceito de C e, portanto, não é portável.
4. Dennis Ritch.
5. Consulte a **Seção 3.2.1**.
6. (a) São identificadores que possuem significado especial numa linguagem de programação (b) Não.
7. (a) São identificadores que são ou poderão ser usados pela biblioteca padrão de C. (b) Pode, mas não deve.
8. Na prática não é possível, porque palavras-chave e palavras reservadas usam letras minúsculas.
9. **int**, **double** e **void** são palavras-chave; `$a` e `a$` usam `$`, que não é um caractere permitido em identificadores de C.
10. O conjunto básico de caracteres de C contém letras, dígitos, caracteres gráficos e espaços em branco.
11. (a) É um mapeamento entre caracteres e números inteiros não negativos que os representam. (b) ASCII, EBCDIC e ISO 8859-1.

12. Por meio dos números inteiros que os representam no código de caracteres em uso.

13. (a) `71`. (b) `9`. (c) `57`. (d) `106` (`49 + 57`).

14. Não. `c = 'A'` é portável (e legível) `c = 65` não é portável (nem legível).

15. (a) É um conjunto de valores munido de um conjunto de operações sobre eles (b) É um tipo que faz parte de uma linguagem de programação.

16. (a) Consulte a **Seção 3.4.1**. (b) Consulte a **Seção 3.4.2**. (c) Consulte a **Seção 3.4.3**.

17. Notação convencional e notação científica.

18. (a) É uma forma de representação para alguns caracteres especiais que não possuem representação gráfica. (b) Consulte a **Seção 3.5.3**. (c) Consulte a **Seção 3.5.3**.

19. `'\n'` provoca quebra de linha e `'\t'` causa tabulação horizontal.

20. (a) Um string constante consiste em um ou mais caracteres constantes entre aspas. (b) Um string constante pode conter mais de um caractere; os delimitadores de strings (aspas) e caracteres constantes (apóstrofos) também são diferentes.

21. (a) Dividindo o string em substrings separados por espaços em branco. (b) Melhora a legibilidade.

22. Sim. Existem inúmeros exemplos neste livro.

23. (a) Sim. É a soma do valor inteiro associado a `'A'` com o valor inteiro associado a `'B'`. (b) Não, pois ela depende de implementação. (c) Depende do código de caracteres usado.

24. O resultado de `'Z' - 'A'` depende de implementação (i.e., ele depende do código de caracteres usado, o que não é especificado por nenhum padrão de C).

25. (a) `9` (b) `2` (c) `dígito`.

26. (a) Constante inteira em base decimal. (b) Constante real em notação convencional. (c) Constante real em notação científica. (d) Caractere constante representado como sequencia de escape. (e) String constante.

27. (a) Os operadores unários. (b) O operador de atribuição.

28. (a) É a alteração de valor produzida pelo operador em um de seus operandos. (b) Atribuição, incremento e decremento.

29. Os operadores de conjunção e disjunção, representados, respectivamente, por **&&** e **||**.

30. Não há como alterar essa propriedade.

31. Porque a expressão à direita da atribuição pode produzir dois valores válidos, dependendo da ordem com que os operandos do operador de multiplicação são avaliados.

32. Operadores aritméticos, relacionais e lógicos, nessa ordem, da maior para a menor precedência.

33. (a) Um operador representa uma operação elementar de uma linguagem de programação. (b) É um valor sobre o qual atua um operador. (c) É uma combinação legal de operadores e operandos.

34. (a) `-6`. (b) `-2`. (c) `6`. (d) `-2`. (e) `-6`. (f) `2`. (Esses resultados estão de acordo com os padrões C99 e C11.)

35. (a) É um operador utilizado para efetuar comparações de valores numéricos. (b) Consulte a **Seção 3.7.2**.

36. Alguns usam símbolos diferentes; os tipos de operandos são diferentes; os resultados são diferentes; os operadores relacionais da pseudolinguagem fazem parte de um mesmo grupo de precedência, o que não ocorre com os operadores relacionais de C.

37. Em C, `0` ou `1`. Em linguagem algorítmica, **verdadeiro** ou **falso**.

38. Não (v. **Seção 3.7.2**).

39. (a) `10.5`; (b) `2.5`; (c) `6.75`.

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

41. (a) Negação (**!**), conjunção (**&&**) e disjunção (**||**). (b) `0` ou `1`.

42. (a) Praticamente, nenhuma. (b) Operandos, resultados, símbolos e curto-circuito.

43. (a) É o fato de um operador nem sempre avaliar todos os seus operandos. (b) Conjunção (**&&**) e disjunção (**||**).

44. Não. Na expressão (2) o valor de `y` é sempre incrementado, o que não ocorre sempre no caso da expressão (1) devido ao curto-circuito do operador de disjunção.

45. Se o valor de `x` for `0`, a avaliação da expressão (1) causa o aborto devido a tentativa de cálculo de resto de divisão por `0`. Na expressão (2), o curto-circuito do operador de conjunção impede que a expressão `y%x` seja avaliada quando `x` é `0`.

46. (a) Porque o valor resultante da avaliação de `1/3` é `0`. (b) Zero.

47. (a) `x > 2 && x < 10`; (b) `x%2 == 0 || x%3 == 0`; (c) `x%2 == 0 && x != 6`; (d) `x == 2 || x == 3 || x == 5`;

48. (a) Sim. (b) Zero, porque o resultado de `x = 0` é sempre `0`.

49. O valor da expressão `2/3` é `0`. Portanto a variável `produto` recebe esse valor convertido em **double**.

50. Uma definição de variável especifica o nome e o tipo de uma variável.

51. Uma definição de variável informa o compilador como ele deve interpretar o conteúdo da variável, a quantidade de espaço que deve ser alocada e o identificador associado à variável.

52. Para facilitar rápida identificação visual das categorias às quais os identificadores pertencem.

53. O símbolo `=` representa operador de atribuição; o símbolo `==` representa operador de igualdade.

54. `1`.

55. (a) `i` recebe `2`; `d` recebe `2.0`. (a) `i` recebe `2`; `d` recebe `2.5`.

56. (a) É uma conversão de tipos efetuada automaticamente pelo compilador. (b) Para permitir mistura de tipos numa expressão.

57. Em expressões envolvendo tipos diferentes (incluindo atribuição), em passagem de parâmetros, em retorno de função e quando o tipo de uma variável ou parâmetro é **char**.

58. Consulte a **Seção 3.10.1**.

59. (a) Não. (b) Sim. (c) Real.

60. (a) Para converter `2` em **double**. (b) Não, pois essa conversão ocorreria implicitamente. (c) A conversão explícita melhora a legibilidade da instrução.

61. (a) `2.0`; (b) `2.5`; (c) `2.5`; (d) `2.0`.

62. (a) É uma operação que acrescenta `1` a uma variável. (b) É uma operação que subtrai `1` de uma variável. (c) São os operadores representados por `++` e `--`.

63. Se `x` é uma variável, o resultado de `++x` é `x + 1` e o resultado de `x++` é `x`.

64. (a) `2` (b) `1` ou `2`.

65. (a) `1` (b) `2`.

66. Incremento não pode ter uma expressão como operando.

67. Se a variável `j` for avaliada primeiro, a expressão `j * j++` será igual a `4 * 4`, que resulta em `16`. Se a expressão `j++` for avaliada primeiro, a expressão `j * j++` será igual a `5 * 4`, que resulta em `20`.

68. Não. Essa expressão possui todos os ingredientes para não ser portável (v. **Seção 3.9**).

69. (a) Por meio de delimitadores de comentários. (b) Como se fossem espaços em branco.

70. Para facilitar seu entendimento assim como prover outras informações úteis sobre ele.

71. Um programa sem comentários pode requerer que o leitor faça inferências para tentar entender o que (e como) o programa faz.

72. Quando se está construindo o programa para evitar esquecimento.

73. Comentários encontrados em livros de ensino de programação são didáticos.

74. Porque comentários redundantes desviam a atenção de quem lê o programa.

75. Consulte a **Seção 3.12**.

76. Uma biblioteca é uma coleção de componentes prontos para uso em programas.

77. Estritamente falando, sim. Mas, uma linguagem de programação que não possui biblioteca é severamente limitada do ponto de vista pragmático.

78. Usando os componentes providos por uma biblioteca, o programador não precisa programá-los.

79. (a) É a biblioteca que deve acompanhar qualquer compilador que segue o padrão da linguagem C. (b) Nenhum componente de biblioteca têm significado especial para um compilador de C. Portanto não fazem parte dessa linguagem.

80. Porque, apesar de acompanhar obrigatoriamente qualquer compilador padrão, seus componentes não fazem parte da linguagem em si (v. questão anterior).

81. É um arquivo que contém informações úteis para o compilador sobre os componentes de uma biblioteca ou módulo de biblioteca.

82. (a) Para incluir arquivos de cabeçalhos em programas. (b) O conteúdo do arquivo cujo nome acompanha a diretiva é inserido no arquivo que a contém a partir da linha na qual ela se encontra.

83. (a) Quando se deseja processar entrada ou saída usando a biblioteca padrão. (b) Quando se desejam incluir num programa operações matemáticas sobre números reais que não são contempladas por operadores de C. (c) Quando se desejam processar strings. (d) Para leitura robusta de dados via teclado sem uso da biblioteca padrão.

84. Normalmente, os símbolos `<` e `>` são usados para cabeçalhos da biblioteca padrão, enquanto aspas são usadas para cabeçalhos de outras bibliotecas.

85. *Interação* refere-se à comunicação entre um programa e um usuário. A palavra *iteração* refere-se a repetição e é mais usada no contexto de estruturas de controle.

86. É um programa resistente a qualquer tipo de entrada de dados.

87. Consulte a **Seção 3.14.1**.

88. É uma sequência de caracteres que começa com **%** e especifica como um valor deve ser escrito.

89. (a) É um string que contém especificadores de formato. (b) Quaisquer caracteres.

90. Cada especificador de formato deve corresponder um valor a ser escrito por **printf()**.

91. (a) Usando o especificador de formato **%c**. (b) Usando o especificador de formato **%d**.

92. (a) Um valor do tipo **double** com seis casas decimais. (b) Um valor do tipo **double** com duas casas decimais, largura de campo mínima igual a cinco e alinhado à direita. (c) Um valor do tipo **int** com largura de campo mínima igual a cinco alinhado à direita. (d) Um valor do tipo **int** com largura de campo mínima igual a cinco alinhado à esquerda. (e) Um valor do tipo **int**. (f) Um caractere. (f) Um string.

93. (a) Incluindo `\n` num string de formatação de **printf()**. (b) Incluindo `\t` num string de formatação de **printf()**.

94. Seis.

95. (a) `2.456900`. (b) Um valor que não faz sentido. (c) `2.46`. (d) `2.46`.

96. `4321`. O valor `43` é obtido da primeira chamada de **printf()**; `2` vem da segunda chamada de **printf()** e `1` vem da terceira chamada dessa função.

97. Porque faltou incluir a variável `resultado` como parâmetro de **printf()**.

98. `i = 8, j = 1`

99. `1/3 = 0.000000`

100. Consulte a **Seção 3.14.3**.

101. (a) Usuário comum não sabe o que é *string*. (b) Nem ^*C*.
102. Porque o usuário pode ser induzido a introduzir dados incorretos.
103. A função **printf()** usa um especificador de formato indevido (o correto seria **%f**, e não **%d**).
104. (a) É um identificador que representa um valor constante. (b) Usando **#define**, o nome da constante e seu valor.
105. (a) Porque uma definição de constante simbólica é uma diretiva e diretivas terminam com quebra de linha, e não com ponto e vírgula. (b) Porque esse tipo de erro não é apontado pelo compilador no local onde a constante é definida. (c) Consulte a **Seção 3.15**.
106. Legibilidade, manutenibilidade e portabilidade são favorecidas com o uso de constantes simbólicas.
107. Usando **constante**, o símbolo `=`, o nome da constante e seu valor.
108. Consulte a **Seção 3.16**.
109. Resumidamente, teste procura defeitos num programa e depuração os corrige.
110. Não. Apenas programas executados sob supervisão de um hospedeiro devem incluir uma função **main()**.
111. (a) É um sistema que supervisiona e dá suporte à execução de um programa. (b) É um sistema que não oferece nenhum suporte à execução de um programa.
112. É um programa cuja interface é baseada apenas em texto.
113. Consulte a **Seção 3.17.1**.
114. (a) É o nome da primeira função a ser executada num programa com hospedeiro. (b) Porque o hospedeiro executa essa função logo após o programa ter sido carregado em memória. (c) Sim, dependendo do sistema usado como hospedeiro. Por exemplo, em sistemas da família Microsoft Windows, o nome da primeira função a ser executada é `WinMain()`.
115. Quando uma função **main()** retorna zero, o significado é que o programa que a contém foi executado sem anormalidades.
116. Mensagem de erro indica que um programa não será compilado e mensagem de advertência indica que um programa poderá apresentar problemas quando for executado.
117. (a) É um erro que transgride as regras de uma linguagem de programação. (b) É um erro que causa aborto de programa. (c) É um erro que causa o mau funcionamento de um programa, mas não provoca aborto. (d) Erro de compilação. (e) Erro lógico.
118. Consulte a **Seção 3.18.2**.
119. Porque uma mensagem de advertência pode prever a ocorrência de um erro de execução ou erro lógico.
120. Ocorrerá um erro de execução se o valor introduzido pelo usuário for igual a `0`.

Capítulo 4 — Fluxo de Execução

1. É a sequência e a frequência com que as instruções do programa são executadas.
2. (a) Fluxo natural de execução de um programa é a execução sequencial da primeira à última instrução do programa, sendo cada uma delas executada uma única vez e na ordem em que se encontra no programa. (b) Por meio de estruturas de controle.
3. Esse programa é legal, mas é inútil.
4. (a) Sim. (b) Nenhum.
5. Consulte a **Seção 4.2**.
6. Consulte a **Seção 4.2**.
7. Porque uma expressão sem efeito colateral não altera nenhum dado processado pelo programa.
8. Porque uma instrução (como `x++;`) não pode aparecer fora de uma função.

9. Iniciação é permitida fora de qualquer função, pois não é considerada instrução. Essa é uma das diferenças entre iniciação e atribuição.

10. (a) Ponto e vírgula. (b) Não (v. **Seção 4.2**).

11. (a) Consulte a **Seção 4.5.6**. (b) Consulte as **Seção 4.11.6**.

12. Laços de repetição, desvios condicionais e desvios incondicionais.

13. (a) São instruções que alteram a ordem de execução de outras instruções de um programa de acordo com o resultado da avaliação de uma expressão condicional. (b) **if-else** e **switch-case**.

14. (a) São instruções que permitem o desvio do fluxo de execução de um programa independentemente da avaliação de qualquer condição. (b) **break**, **continue** e **goto**.

15. (a) São instruções que alteram a frequência com que outras instruções são executadas. (b) **while**, **do-while** e **for**.

16. (a) Consulte a **Seção 4.5.1**. (b) Consulte a **Seção 4.5.2**. (c) Consulte a **Seção 4.5.3**.

17. (a) Cinco vezes. (b) `5` e `5`. (c) Sim, pois os resultados das expressões contendo `++` e `--` não são usados.

18. (a) No caso específico em que `x` e `y` assumem os valores iniciais que aparecem nas questões, não há diferença. Mas, se inicialmente o valor de `x` for maior do que ou igual a `y`, haverá diferença. (b) Cinco vezes. (c) `5` e `5`.

19. (a) Cinco vezes. (b) Os valores de `x` e `y` serão, respectivamente, `6` e `4`. (c) Não, pois os valores das expressões `x++` e `y--` são usados.

20. Sim, pois o resultado da expressão `j++` (ou `++j`) não é utilizado.

21. A variável `i` não é iniciada.

22. A condição de parada do laço **while** é atingida apenas quando ocorre overflow na variável `i`.

23. Esse programa soma os termos de uma progressão aritmética com razão `2`, cujo primeiro termo é `2` e o último termo é `100`. Portanto o programa escreve na tela: `Soma = 2550`, conforme era esperado.

24. (a) Essa questão é uma pegadinha, porque aparentemente o valor de `i++` será sempre maior do que zero e, assim, o laço **while** nunca terminará. Acontece, entretanto, que o incremento contínuo da variável `i` terminará acarretando em overflow e essa variável se tornará negativa, encerrando o laço. (b) `Valor final de i: -2147483647`. Esse valor corresponde ao maior valor do tipo **int** com sinal invertido na implementação na qual o programa foi compilado.

25. (a) Sim. O laço encerra quando o valor da expressão `--i` for igual a zero. (b) `Valor final de i: 0`.

26.

```
int i = 0;

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

27. Esse laço nunca termina porque a variável `i` sempre assume `0` como valor.

28. Porque o programador trocou ponto e virgulas por vírgulas, de modo que a condição de para do laço ficou sendo `j = 10, i < j, ++i` e o resultado dessa expressão é `++i`, que nunca será zero. Assim, o laço nunca termina.

29. É uma expressão que, quando resulta num valor diferente de zero, provoca o encerramento do laço.

30. A condição de parada do laço **do-while** desse programa é `valor == 1 && valor == 2`. Mas, não existe nenhum valor que satisfaça essa expressão.

31. `i <= 0`, que é a negação da expressão condicional do laço.

32. (a) Provavelmente, ele pretendia escrever o valor de `i` em cada execução do corpo do laço. (b) Porque o ponto e vírgula na linha inicial do laço **for** encera prematuramente essa instrução. (c) Removendo o ponto e vírgula indevido. (d) Consulte a **Seção 4.5.1**.

33. Esse programa escreve `y` linhas cada uma das quais contém `x` caracteres iguais àquele escolhido pelo usuário.

34. (a) Consulte a **Seção 4.5.3**. (b) **for**.

35. O uso de vírgulas em vez de pontos e vírgulas para separar as expressões do laço **for** causa erro de compilação.

36. (a) 10 vezes. (b) 10 vezes. (c) 11 vezes. (d) 11 vezes. (e) 10 vezes. (f) 11 vezes. (g) Esse laço é infinito e encerra quando ocorre overflow. (h) Idem.

37. (a) 100. (b) 10.

38. O ponto e vírgula na linha contendo **for** encerra prematuramente essa instrução. A correção consiste em remover esse ponto e vírgula.

39. `Botafogo`.

40. Consulte a **Seção 4.5.4**.

41. Consulte a **Seção 4.5.4**.

42. Instruções **break** previnem o efeito cascata de instruções **switch-case**.

43. A expressão `0 < x < 10` resulta sempre em `1`, pois `0 < x` resulta em `0` ou `1` e, em qualquer caso, esse valor é menor do que `10`.

44. (a) O uso do operador de atribuição na expressão condicional da instrução **if**. (b) Escrevendo essa expressão como: `0 = x`. Nesse caso, a expressão continuaria errada, mas o compilador indicaria o erro. (c) Qualquer compilador decente emite uma mensagem de advertência em situações como essa.

45. (a) O identificador `defalut` é considerado um rótulo de instrução. (b) Nada.

46. `Valor de x: 6`.

47. `Valor de x: 5`.

48. (a) `Tchau`. (b) Um número de linhas igual ao valor introduzido, cada uma das quais contendo o caractere `+`. (c) Um número de traços igual ao valor introduzido menos um.

49. Digite o programa, compile-o, execute-o e você conhecerá a resposta.

50. O valor de `x` (ou qualquer outra variável) ou é zero ou diferente de zero. Portanto a terceira instrução **printf()** jamais será executada.

51. O erro está na expressão `numero =! 0`, que é interpretada como negação seguida de atribuição. Portanto à variável `numero` será sempre atribuído `1`, independentemente do valor introduzido pelo usuário.

52. Para que a parte **default** não seja esquecida quando ela for necessária.

53. Esse programa escreve na tela os valores inteiros introduzidos pelo usuário sem repetir nenhum valor.

54. Devido ao uso do operador de atribuição na expressão `debito = 0`.

55. O operador de negação (representado por **!**) tem maior precedência do que o operador de resto de divisão (representado por **%**). Portanto a expressão `!numero%2` resulta sempre em `0` quando a variável `numero` é diferente de `0`.

56. Consulte as **Seções 4.7.1** e **4.7.2**.

57. (a) Consulte a **Seção 4.7.3**. (b) Porque o uso desmesurado de **goto** pode gerar código espaguete.

58. `1`.

59. Será exibido na tela:

```
Final da instrucao for
Final da instrucao while
```

60. `2 4 6 8 10`.

61. Não. O desvio só é permitido para instruções que façam parte da mesma função na qual se encontra a instrução **goto**.

62. Ela pode deixar de ser válida quando o corpo da instrução **for** contém uma instrução **continue**.

63.

(a)
```
0       5       15      30
i = 20
```

(b)
```
1       2       3       4
i = 20
```

(c)
```
1       2       3       4
i = 16
```

(d)
```
1     0     3     2     7     6     13    12    21
i = 10
```

(e)
```
1
i = 1
```

(f)
```
i = 1
```

(g)
```
1     2     3     3     4     5     4     5     6     7
i = 5, j = 4
```

(h)
```
1     3     2     1     0     0     0     0     0     0
i = 5, j = 4
```

64. (a) Se forem traçadas linhas conectando as instruções **goto** às respectivas instruções para as quais são efetuados os desvios provocados por **goto**, essas linhas se cruzarão e estará pronto o prato de código espaguete. (b) Abaixo.

```
#include <stdio.h>

int main(void)
{
   int contador;

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

   putchar('\n');

   return 0;
}
```

65. Instruções `continue` só podem ser usadas em corpos de laços de repetição.

66. Consulte a **Seção 4.8**.

67. Consulte a **Seção 4.8**.

68. `2`.

69. Se você teve dificuldade para responder a questão anterior, sabe a resposta desta questão.

70. `printf("x %se' positivo", x > 0 ? "" : "nao ");`

71. É a menor precedência da linguagem C.

72. Consulte a **Seção 4.9**.
73. (a) `2`. (b) `1`. (c) `1`. (d) `3`.
74. (a) `i` assumirá `-1` e `j` assumirá `0`. (b) `0`. (c) Não. Essa instrução é portável. Usando o operador de soma, ela deixará de sê-lo. (d) Não. Se o operador de conjunção for usado, a expressão `j = i--` não será avaliada, já que a expressão `i = 0` resulta em `0`. (e) Nesse caso especial, a resposta é *sim*.
75. Conjunção, disjunção, vírgula e condicional.
76. Não. Consulte a **Seção 3.9**.
77. (a) Sim. (b) `3`. (c) `2`.
78. Esse programa escreve `d = 2.000000`. A vírgula na expressão `d = 2,5` é interpretada como operador, de modo que à variável `d` é atribuído o valor `2.0`, visto que o operador de atribuição tem maior precedência do que o operador vírgula e levando em consideração a conversão implícita.
79. Porque, nesse caso, a vírgula é considerada um separador de declarações. Portanto não deveria haver um número seguindo-a.
80. Porque eles são obtidos por meio de uma fórmula e, portanto, são previsíveis (desde que se conheçam a semente e a fórmula utilizadas).
81. (a) Um valor entre `1` e `10`. (b) Porque o gerador de números aleatórios usa sempre a mesma semente.
82. Para alimentar o gerador de números aleatórios.
83. (a) Para alimentar o gerador de números aleatórios. (b) A semente que alimenta o gerador de números aleatórios é bem mais difícil de ser replicada.

Capítulo 5 — Subprogramas

1. Consulte a **Seção 5.1**.
2. Sim. Pelo menos a função **main()** deve fazer parte de um programa de console.
3. Consulte a **Seção 5.1**.
4. Por meio do operador de endereço.
5. Uma variável do tipo **int**, por exemplo, sempre armazenará um valor desse tipo, quer ela tenha sido iniciada ou não. Por outro lado, o conteúdo de um ponteiro não iniciado pode não corresponder a um endereço válido.
6. Consulte a **Seção 5.2.3**.
7. (a) Sim. (b) Não.
8. Consulte a **Seção 5.2.4**.
9. A variável `p2` não é ponteiro e, portanto, não deveria ser iniciada com um endereço.
10. (a) Sim. (b) À variável `x` é atribuído `10` por intermédio do ponteiro `p`.
11. (a) `&x`. (b) `x`.
12. Porque `p` é um ponteiro nulo quando lhe é aplicado o operador de indireção.
13. `y = 100`.
14. Porque, na iniciação, o endereço de `x` é atribuído a `p`, enquanto, na atribuição o endereço de `x` é atribuído ao conteúdo apontado por `p`.
15. Porque, na iniciação, um valor do tipo **int** é atribuído a `p`, enquanto, na atribuição um valor do tipo **int** é atribuído ao conteúdo apontado por `p`.
16. Consulte a **Seção 5.3**.

17. Consulte a **Seção 5.3**.
18. Consulte a **Seção 5.3**.
19. Consulte a **Seção 5.4**.
20. (a) Consulte a **Seção 5.5.1**. (b) Por meio de variáveis globais ou com escopo de arquivo.
21. Não. Mas algumas outras linguagens de programação (p. ex., Pascal) permitem isso.
22. (a) O uso de **void** indica que a função não retorna nenhum valor. (b) Nesse caso, o uso de **void** indica que a função não possui nenhum parâmetro.
23. O tipo do parâmetro `y` não foi declarado.
24. Consulte a **Seção 5.4.3**.
25. (a) Sim. (b) Não.
26. (a) Sim. (b) Não.
27. Não. Mas esses tipos devem ser compatíveis.
28. Consulte a **Seção 5.4.3**.
29.

```
int MinhaFuncao(int x)
{
    return x < 0 ? 0 : 1;
}
```

30. O ponteiro `aux` não foi iniciado.
31. Essa função retorna `x` quando `x` é positivo ou zero e retorna `-x` quando `x` é positivo. Portanto ela calcula o valor absoluto de um número inteiro.
32. Quando o valor do parâmetro é menor do que ou igual a zero, essa função não retorna nenhum valor.
33. Seguindo a dica do exercício, se você escrever um programa contendo uma chamada da função `Abs()`, tal como: `Abs(INT_MAX)`, verá que o resultado retornado por ela é 1, o que é um absurdo. Isso acontece devido à ocorrência de overflow na chamada da função **sqrt()**.
34. Quando ele é um parâmetro saída ou de entrada e saída (ou um array, como será visto no **Capítulo 8**).
35. Quando ele é um parâmetro de entrada, mas ocupa muito espaço em memória.
36. O modo de um parâmetro informa se ele é de entrada, saída ou entrada e saída.
37. Consulte a **Seção 5.5.1**.
38. Nunca.
39. (a) É um parâmetro que aparece numa definição de função. (b) É um parâmetro que aparece numa chamada de função. (c) Eles são ligados durante uma chamada de função.
40. (a) Não. (b) Não.
41. Consulte a **Seção 5.5.2**.
42. Se os parâmetros forem compatíveis, haverá conversão do tipo do parâmetro real para o tipo do parâmetro formal. Se os parâmetros não forem compatíveis, poderá ocorrer erro de sintaxe.
43. Consulte a **Seção 5.5.2**.
44. Porque esse é o único tipo de passagem de parâmetros especificado pelo padrão de C.
45. Não.
46. Sim. Nesse caso, o valor retornado é desprezado.
47. Significa que a função retorna um valor, mas ele está sendo desprezado. Mas, o uso de conversão explícita não é estritamente necessário (apenas melhora a legibilidade).

48. (a) `1 2 3 4 5` (b) `1 3 6 10 15` (c) `5 9 13 17 21`

49. Sim. A função `F2()` deve ser definida antes de `F1()`, a não ser que haja uma alusão à função `F2()` antes da definição de `F1()`.

50. Consulte a **Seção 5.6**.

51. Incluindo alusões antes das definições de funções, o programador não precisa se preocupar com a ordem com a qual elas são definidas.

52. Consulte a **Seção 5.6**.

53. Consulte a **Seção 5.6**.

54. Não. Nomes de parâmetros são ignorados em alusões.

55. (a) `void F(int dia, int *mes)` (b) `int *F(int *x, int *y)`

56. (a) `void F(double, char *)` (b) `int *F(void)`

57. O uso de parênteses vazios numa alusão é interpretado como uma alusão sem protótipo.

58. (a) Não. (b) Não. (c) Por uma razão prática: o programador pode criar uma alusão de função copiando e colando o cabeçalho da função.

59. (a) A função `ApresentaMenu()` é chamada antes de ser definida. (b) Colocando-se a definição dessa função ou uma alusão a ela antes da função **main**().

60. Pode-se inferir que essa função possui dois parâmetros: o primeiro deles é um ponteiro para **double** e o segundo deles é do tipo **int**. Pode-se ainda concluir que essa função retorna um valor do tipo **int**.

61. (a) Para uso em alusões de funções e variáveis globais. (b) Neste livro introdutório, essa palavra-chave é redundante. Mas, às vezes, ela é necessária para resolver possíveis ambiguidades em alusões e definições de variáveis globais.

62. Porque o programador não precisa se preocupar com a ordem com que as funções são definidas.

63. Consulte a **Seção 5.9**.

64. Consulte a **Seção 5.9**.

65. Consulte as **Seções 2.9** e **5.9**.

66. (a) Consulte a **Seção 5.8**. (b) Terminais ATM de banco.

67. Porque, tipicamente, programas com interação dirigida por menus possuem várias opções que correspondem a desvios múltiplos e instruções **switch-case** são as mais indicadas para implementar esses desvios.

68. Existem vários exemplos de interação dirigida por menu neste livro. Consulte o **Índice Remissivo**.

69. Consulte a **Seção 5.8.2**.

70. Consulte a **Seção 5.8.3**.

71. É uma variável que é alocada quando o bloco no qual ela é definida é executado e liberada quando encerra a execução desse bloco. (b) No interior de um bloco sem uso de **static**. (c) Escopo de bloco. (d) Seu conteúdo é indefinido.

72. **auto** serve para qualificar variáveis de duração automática. (b) Não. Ela é redundante. (c) Não, porque programas antigos que a usam deixariam de ser compilados.

73. (a) Significa que o espaço alocado para a variável ou parâmetro torna-se livre para uso posterior. (b) Quando encerra a execução do bloco no qual ela é definida. (c) Quando o programa encerra.

74. (a) O conteúdo da variável é indefinido. (b) A variável é implicitamente iniciada com zero.

75. Consulte a **Seção 5.9.3**.

76. (a) Porque uma variável de duração fixa deve ser iniciada antes da execução de qualquer função e uma variável de duração automática só é alocada quando a função que contém sua definição é chamada. (b) Porque

isso imporia uma ordem de alocação de variáveis de duração fixa. Por exemplo, suponha que `x` e `y` sejam variáveis de duração fixa e que fosse permitido iniciar a variável `x` com o valor de `y`. Então, o compilador teria que gerar código para alocar `y` antes de `x`. Mas, o padrão de C especifica apenas que essas variáveis devem ser alocadas ao início da execução do programa que as contém.

77.

```
x = 1    y = 1
x = 1    y = 2
x = 1    y = 3
x = 1    y = 4
x = 1    y = 5
```

78. (a) Não. (b) Sim.

79. Sim. Apesar de a variável `y` ter duração fixa, sua iniciação não usa o valor da variável x. Em outras palavras, a expressão `sizeof(x)` pode ser resolvida em tempo de compilação.

80. Porque a iniciação da variável `k` é ilegal.

81.
```
4        11       22
```

82. (a) A variável `soma` tem escopo de bloco, de modo que ela é iniciada com `0` cada vez que o bloco que contém sua definição é executado. (b) Colocando a definição dessa variável antes do laço **while**.

83. A qualificação da variável `soma` com **static** faz com que ela passe a ter duração fixa e variáveis de duração fixa retêm valores assumidos entre uma chamada de função e outra. Por exemplo, se essa função fosse chamada duas vezes consecutivas como `SomaAteN(5)`, na primeira chamada ela retornaria `15` (valor correto), enquanto na segunda chamada ela retornaria `30` (valor incorreto).

84. (a) Consulte a **Seção 5.10.4**. (b) Variáveis e parâmetros. (c) Rótulos de instruções.

85. (a) Sim. (b) Não.

86. Consulte a **Seção 5.10.4**.

87. (a) Sim, desde que tenham escopos diferentes. (b) Sim, desde que elas sejam definidas em blocos diferentes. (c) Não.

88. (a) Sim. (b) A variável deve ser definida num bloco interno ao corpo da função.

89. Sim. O escopo deve ser de bloco.

90. Nada. O escopo pode ser de bloco, de arquivo ou de programa.

91. Consulte a **Seção 5.10.4**.

92. (a) O escopo de `i` é global; o escopo de `j` começa na linha em que ela é definida e termina ao final do arquivo que contém essa definição; o escopo do parâmetro `k` é todo o bloco que constitui o corpo da função `F()`; o escopo de `m` é o bloco que constitui o corpo da função `F()` a partir da linha que contém a definição dessa variável; o escopo de `n` é o bloco que constitui o corpo da função `F()` a partir da linha que contém a definição dessa variável. (b) O escopo de `F()` é global; o escopo de `G()` é de arquivo. (c) As variáveis `i`, `j` e `n` têm duração fixa; a duração de `m` é automática.

93. Porque, em princípio, qualquer instrução de um programa pode modificar uma variável global, dificultando a depuração e a manutenção do programa.

94.
```
x = 9  y = 2
```

95.

```
i = 0
i = 1
i = 2
```

96. A função `F1()` não será compilada porque o parâmetro `x` e a variável `x` têm escopo de bloco e esse bloco é o mesmo.

97. Uma função não pode ter dois rótulos que usam o mesmo identificador.

98. Esse programa nunca termina porque a condição de parada do laço **for** no corpo da função **main()** nunca é atingida, pois sempre que a função `F()` retorna o valor de `i` é `1`.

Capítulo 6 — Estilo de Programação

1. Consulte a Seção 2.7.6.
2. (a) Não, mas é altamente recomendado. (b) Corpo de estrutura de controle, corpo de função, continuação de instrução ou expressão numa linha subsequente.
3. Porque números reais são representados aproximadamente.
4. O resultado desse programa é apresentado a seguir. Consulte a Seção 7.5 para entender esse resultado.

```
d = 1.000000
d e' diferente de 1.0
```

5. Consulte a Seção 6.2.
6. Consulte a Seção 6.3.
7.

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

(b)
```
for (int i = 0; i <= 100; ++i)
   printf("i = %d", i);
```

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

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

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

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

(g)
```
return valor != OK ? valor : OK;
```

8. Consulte a Seção 6.4.
9. Para facilitar a identificação visual desses identificadores.

10. Consulte a **Seção 6.4**.
11. Consulte a **Seção 6.5**.
12. Quando a constante possui significado próprio que não depende de interpretações subjetivas.
13.

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

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

   if(!valor == 0) {
      continue;
   }

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

14. Consulte a **Seção 6.7**.
15. Consulte a **Seção 6.7**.
16. Consulte a **Seção 6.8**.
17. Consulte a **Seção 6.8**.
18. Consulte a **Seção 6.8**.
19. Consulte a **Seção 6.8**.
20. Quando o programa está sendo escrito, para que não ocorra nenhum esquecimento relacionado àquilo que deve ser comentado.
21. Comentários irrelevantes desviam a atenção de comentários que são realmente importantes.
22. Um comentário que não corresponde exatamente ao código que ele tenta explicar é prejudicial à legibilidade de um programa porque suscita dúvidas no leitor do programa. Por exemplo, o leitor do programa pode ter dificuldade para decidir se o correto é o código ou o comentário que não corresponde ao código.
23. Para separar porções de um programa que não são intimamente relacionadas.
24. Consulte a **Seção 6.9**.
25. No contexto de programação, *interação* diz respeito à comunicação entre um programa e um usuário e *iteração* é execução repetida de instruções.
26. Consulte a **Seção 6.10**.
27. Teclado.
28. Porque reduz a possibilidade de o usuários introduzir dados inconvenientes.
29. Jargões usados na área de computação e o uso abusivo de sons de alerta.
30. É uma informação apresentada pelo programa para o usuário informando-o que o programa será encerrado em seguida.

Capítulo 7 — Reúso de Código e Depuração de Programas

1. Consulte a **Seção 7.1**.
2. Porque programadores experientes possuem um repertório de programas já construídos bem maior do aquele de programadores iniciantes. O repertório mais amplo auxilia programadores experientes a construir novos programas com mais facilidade.
3. Chamadas de funções de biblioteca, copiar trechos de um programa e colar em outro, código default em CodeBlocks.
4. (a) Bibliotecas contêm componentes prontos para serem usados indefinidamente em programas. Assim, o programador não precisa criar tais componentes. (b) Bibliotecas não podem nem devem ser alteradas para satisfazer necessidades específicas.
5. Copiar e colar.
6. Consulte a **Seção 7.3**.
7. Abreviações permitem acelerar o processo de codificação.
8. Consulte a **Seção 7.4.1**.
9. Tentativa de divisão inteira por zero e violação de memória.
10. Testes tentam encontrar algum comportamento anormal de um programa, mas não indicam as causas de uma anormalidade se ela for encontrado. Depuração deve encontrar instruções que causam algum mau funcionamento do programa e corrigi-las. Em tempo: a fase de testes de um programa deve preceder sua fase de depuração.
11. Consulte a **Seção 7.4.3**.
12. Consulte a **Seção 7.4.2**.
13. O erro no trecho de programa 2 é sintático. O erro no trecho de programa 1 é lógico.
14. Consulte a **Seção 7.4.3**.
15. Porque erros de sintaxe são apontados pelo compilador e a correção é relativamente rápida (dependendo da experiência do programador). Erros de execução são mais difíceis de corrigir porque é o próprio programador quem deve encontrar as instruções que os causam.
16. Porque erros de execução deixam *pistas* que são relativamente fáceis de ser seguidas usando-se um bom depurador. Erros de lógica nem sempre deixam rastros.
17. Mensagens de advertência não devem ser negligenciadas por duas razões: (1) uma mensagem de advertência pode antecipar um erro que realmente ocorrerá em tempo de execução; (2) o programador pode adquirir o mau hábito de subestimar essas mensagens.
18. (a) Truncamento consiste no desprezo de alguns algarismos de um número. (b) É um erro decorrente do fato de esses algarismos terem sido desprezados.
19. Consulte a **Seção 6.5**.
20. Número real é um conceito matemático; número de ponto flutuante é uma forma de implementar esse conceito.
21. Seis.
22. (a) Nesse especificador, ***n*** indica o número de casas decimais que serão escritas. (b) Quando se deseja controlar o número de casas decimais que serão escritas por meio de **printf()**.
23. Consulte a **Seção 7.5**.

24. Refere-se a representações de ponto flutuante.
25. Consulte a Seção 7.5.

Capítulo 8 — Arrays

1. Consulte a Seção 8.1.
2. Consulte a Seção 8.1.
3. Consulte a Seção 8.2.
4. Não.
5. Facilidade de manutenção e melhor legibilidade.
6. (a) Consulte a Seção 8.3. (b) Valores entre `0` e `n - 1`, sendo `n` o número de elementos do array. (c) Corrupção de memória.
7. Quando `j` assume `5` como valor, o valor apresentado na tela não faz sentido.
8. (a) Esse programa corrompe memória, pois o índice `5` é inválido para o array `ar[]`. (b) O problema é gravíssimo, pois ocorre corrupção de memória.
9. Uma sequência de Fibonacci:

```
ar[0] =  1
ar[1] =  1
ar[2] =  2
ar[3] =  3
ar[4] =  5
ar[5] =  8
ar[6] = 13
ar[7] = 21
ar[8] = 34
ar[9] = 55
```

10. (a) Esse programa não é portável porque a expressão `ar2[i] = ar1[i++]` não é portável. Como o operador de atribuição não possui ordem de avaliação definida, qualquer um dos seus operandos (`ar2[i]` ou `ar1[i++]`) pode ser avaliado primeiro. Em cada caso, o resultado da atribuição é diferente. (b) A melhor maneira de corrigir esse problema é substituir `i++` na referida expressão por `i` e acrescentar uma instrução na linha seguinte contendo apenas `i++`.
11. Ele lê cinco valores introduzidos via teclado e exibe-os em ordem inversa.
12. (a) Consulte a Seção 8.4. (b) Não.
13. (a) Iniciando apenas o primeiro elemento com zero. (b) Não.
14. Consulte a Seção 8.4.
15. Não é ilegal, mas os valores excedentes não serão usados.
16. Não, porque o compilador calcula o tamanho do array.
17. Consulte a Seção 8.5.
18. (a) **size_t**. (b) Derivado.
19. O operador **sizeof** pode ter um tipo de dado como operando.
20. Para saber o tipo do valor resultante de uma expressão não é necessário conhecer o valor da expressão. Se você ainda não sabia disso, retorne ao Capítulo 3.
21. (a) Por meio da expressão `sizeof(ar)`. (b) Por meio da expressão `sizeof(ar[0])`. (c) Por meio da expressão `sizeof(ar)/sizeof(ar[0])`.

22. Deve-se ter cuidado para não misturar inteiros com sinal e sem sinal numa mesma expressão que não seja de atribuição.
23. Consulte a **Seção 8.5**.
24. Não resolve o problema, porque, após a conversão obtida explicitamente por meio do operador **(int)**, ocorreria uma conversão implícita de atribuição para **size_t**.
25. Porque o parâmetro é interpretado como um ponteiro.
26. `Tamanho = 4  x = 0`
27. Porque o resultado da aplicação do operador **sizeof** independe da avaliação de seu operando; i.e., só é necessário saber o tamanho do operando.
28.
```
i = 10
sizeof(i++) = 4
i = 10
```
29. `Tamanho = 4 bytes`, que é a largura de um ponteiro na implementação usada para testar o programa. Em outra implementação, o resultado pode ser diferente.
30. Consulte a **Seção 8.6**.
31. Consulte a **Seção 8.6**.
32. Consulte a **Seção 8.6**.
33. Pode-se concluir que o tipo do ponteiro `p` é **char** ou um tipo derivado de mesmo tamanho.
34. Não, mas pode-se inferir que o tamanho desse tipo é 4 bytes.
35. Subtração entre ponteiros.
36. (a) Quando `p1` e `p2` são de tipos diferentes. (b) Quando `p1` e `p2` não apontam para elementos de um array.
37. A diferença `p1 - p2` programa que o programa calcula só faria sentido se esses ponteiros apontassem para elementos de um array.
38. (a) Válida. (b) Válida, desde que, nesse instante, `p1` e `p2` estejam apontando para elementos de um array; caso contrário, é inválida. (c) Idem. (d) Válida. (e) Inválida. (f) Inválida.
39. (a) A expressão `*p` resulta no conteúdo da variável do tipo **int** para a qual `p` aponta. (b) A expressão `++p` faz `p` apontar para a próxima variável do **int**. (c) A expressão `++*p` incrementa o conteúdo da variável correntemente apontada por `p`. (d) A expressão `*++p` representa o conteúdo da variável do tipo **int** que segue aquela para a qual `p` aponta correntemente. (e) A expressão `*p++` representa o conteúdo da variável do tipo **int** para a qual `p` aponta; `p` passa a apontar para a variável do tipo **int** seguinte. (f) A expressão `p+++` é ilegal. (g) A expressão `p*++` é ilegal.
40. (a) `&ar[i]` e `ar + i`. (b) `ar[i]` e `*(ar + i)`.
41. (a) É o endereço do primeiro elemento. (b) É o endereço do terceiro elemento. (c) `10`. (d) `12`. (e) `30`.
42. (a) `25`. (b) `15`. (c) `-2`. (d) `3`. (e) `3`.
43. A expressão `++ar` não é válida, pois `ar` é um endereço (constante) e não pode ser modificado.
44.

(a)
```
for (int i = 0; i < sizeof(ar)/sizeof(ar[0]); ++i) {
   printf("%d\t", ar[i]);
}
```

(b)

```
int *p1, *p2 = ar + sizeof(ar)/sizeof(ar[0]);

for (p1 = ar; p2 - p1 > 0; ++p1) {
   printf("%d\t", *p1);
}
```

45. É mais eficiente começar indexação com zero (v. **Seção 8.7**).
46. Consulte a **Seção 8.8**.
47. Sim, indiretamente por intermédio de um ponteiro.
48. Na qualificação de parâmetros que apontam para conteúdos que não devem ser modificados.
49. Constantes simbólicas definidas com **#define** não ocupam espaço em memória, como ocorre com variáveis qualificadas com **const**.
50. A variável `p1` é um ponteiro que aponta para um conteúdo que deve ser mantido constante. A variável `p2` é um ponteiro constante.
51. (a) `p` é um ponteiro para conteúdo constante do tipo **int**. (b) Idem. (c) `p` é ponteiro constante para um conteúdo do tipo **int**. (d) `p` é ponteiro constante para conteúdo constante do tipo **int**. (e) Idem.
52. (a) O parâmetro `x` só pode ser de entrada. (b) O parâmetro `x` pode ter qualquer modo. (c) Entrada.
53. Usar **const** com um parâmetro que não é ponteiro é semelhante a usar **const** com uma variável que não é ponteiro, porque passagem de parâmetro em C se dá apenas por valor. (Se você não entendeu a resposta, releia a **Seção 5.5**.)
54. Consulte a **Seção 8.9**.
55. Como o endereço do array.
56. (a) Sim. (b) Qualificando o parâmetro com **const**.
57. Rigorosamente falando, não. Mas o retorno de uma função pode ser um endereço de array.
58. (a) Porque o espaço ocupado por ele é liberado quando a função retorna. (b) Porque uma variável zumbi pode ter seu valor alterado pelo simples fato de uma função ser chamada, o que torna difícil a percepção do erro por programadores sem muita experiência. (c) Porque o espaço ocupado por uma variável de duração fixa só é liberado quando o programa encerra.
59. (a) i. (b) ii. [Se fosse legal, a função do item (ii) retornaria um zumbi.]
60. (a) O ponteiro `ptr` não é iniciado e a função `F()` não trata o conteúdo apontado por seu parâmetro como constante. (b) Porque a probabilidade de ocorrer violação de memória aumenta, pois haverá um número maior de bytes que poderão ser acessados indevidamente.
61. Não há nenhum problema. Nesse caso, retorna-se um endereço armazenado num parâmetro, e não o endereço desse parâmetro.
62. Não. Ele pode representar um ponteiro para uma única variável.
63. Sim.
64. Consulte a **Seção 8.10**.
65. Para não causar danos à legibilidade do programa que contém tal construção.
66. Porque seus elementos não são arrays.
67. Consulte a **Seção 8.10**.

Capítulo 9 — Caracteres e Strings

1. Consulte a **Seção 9.1**.

2. (a) Não. (b) Sim. Consulte a **Seção 9.1** para entender as razões.
3. (a) É o caractere que termina qualquer string em C. (b) Zero. (c) Para terminar strings em C.
4. Consulte a **Seção 9.1**.
5. Para facilitar a vida do programador. (Ou você prefere a primeira notação?)
6. Sim, mas o array `ar[]` não armazenará um string.
7. Consulte a **Seção 9.2**.
8. (a) Sim. (b) `ar[0]` armazena `'b'`; `ar[1]` armazena `'o'`; `ar[2]` armazena `'l'`; `ar[3]` armazena `'a'`.
9. `oite`.
10. (a) `ar[]` é um array de ponteiros do tipo **char** *. (b) `33`. (**NB**: A questão não solicita o número de bytes alocados apenas para o array. Ela refere-se ao número total de bytes alocados. Assim, deve-se levar em conta o número de bytes alocados para armazenar os strings constantes.)
11. Consulte a **Seção 9.3**.
12. (a) Porque, caso haja tentativa de alteração do string constante, o compilador indicará o erro; caso contrário, o erro se manifestará em tempo de execução e será mais difícil corrigi-lo. (b) `const char *p = "Bola".`
13. (a) `"A"` é um string constante e `'A'` é um caractere constante. (b) **char** *. (c) **char**. (d) Dois. (e) Um.
14. (a) `str[]` é um array que armazena um string que pode ser modificado e `ptr` é um ponteiro para um string constante. (b) Porque `str` é o endereço do array `str[]`.
15. (a) O endereço do string constante `"Bola"`. (b) `'B'`. (c) `'a'`. (d) `'a'`.
16. (a) (i) Após uma tentativa de alteração de string constante na expressão `p[1] = 'a'`, o programa é abortado. (ii) O erro é o mesmo do item (i), mas o compilador indica o erro. (iii) Ocorre o mesmo erro do item (i) acrescido de tentativa de alteração do ponteiro `p` que é definido como **const**. O erro que o compilador indica é devido à tentativa de alteração do ponteiro `p` que deve ser mantido constante. (iv) Esse caso é uma combinação dos casos (ii) e (iii) e o compilador aponta dois erros. (b) Nas situações (ii), (iii) e (iv). (c) Nas situação (i).
17. (a) Incompatibilidade de tipos: o lado esquerdo da atribuição é do tipo **char** e o lado direito é do tipo **char** *. (b) Incompatibilidade de tipos: o lado direito da atribuição é do tipo **char** e o lado esquerdo é do tipo **char** *.
18. Porque na iniciação ao ponteiro `p` atribui-se o endereço do string `"Bola"`. Na atribuição, tenta-se atribuir esse endereço ao conteúdo apontado por `p`.
19. (a) Consulte a **Seção 9.4**. (b) `'l'`.
20. (a) O primeiro programa escreve caracteres correspondentes aos valores inteiros de `1` a `9` no código de caracteres ora utilizado. O segundo programa escreve os dez caracteres (dígitos) que compõem o array. Em seguida ele passa a escrever caracteres correspondentes aos valores indefinidos dos bytes que seguem o array até que um byte de valor nulo seja encontrado. O terceiro programa escreve os dígitos do string, sendo cada dígito escrito numa linha. (b) O segundo programa poderá ser abortado se ele acessar memória que esteja além do limite alocado para o programa.
21. (a) `5`. (b) `4`.
22. `sizeof("Bola") - 1` é equivalente a `strlen("Bola")`.
23. (a) É o endereço do endereço do string `"domingo"`.(b) É o endereço do string `"domingo"`. (c) `'d'`. (d) É o endereço do endereço do string `"quarta"`. (e) É o endereço do string `"quarta"`. (f) `'a'` (que é o caractere de índice `2` do string `"quarta"`). (g) É o endereço do string `"terca"`.
24. Consulte a **Seção 9.5.1**.

25. (a) `Valor de '0': n` [sendo `n` o valor associado ao caractere `'0'` no código de caracteres usado]. (b) `Valor de ` [a sequencia de escape `\0` termina o string de formatação prematuramente]. (c) `Valor de '0': 0`. (d) `Valor de '\0':` [o caractere `'\0'` não possui representação gráfica].

26. **Primeiro erro**: o ponteiro `str` deveria ser incrementado apenas uma vez no laço **while**. **Correção**: usar uma instrução vazia no corpo desse laço. **Segundo erro**: na saída do laço o ponteiro `str` não está apontando para o caractere terminal do string apontado por esse ponteiro. **Correção**: inserir a instrução `--str` entre os dois laços **while**. **Terceiro erro**: a função não retorna um ponteiro para o início do array resultante da concatenação. **Correção**: definir uma variável local do tipo **char** * e iniciá-la com `str`; então, a função deve retornar essa variável local (em vez de `str`).

27. Ele garante que o string recebido como parâmetro não será alterado.

28. O valor retornado por **strlen()** é do tipo **size_t**. Portanto siga as recomendações apresentadas na **Seção 8.5**.

29. (a) (1) O uso de **void** como tipo de retorno não é portável e (2) o array não tem capacidade para armazenar o resultado da chamada de **strcat()**. (b) O erro (2), pois ele causa corrupção de memória.

30. (a) A expressão do laço **while** deveria ser `*++destino = *origem++`. (b) Sim, devido ao uso de **const**.

31. Para permitir que o resultado da cópia possa ser usada em outra operação sobre strings sem que seja necessária uma linha de instrução adicional.

32. Mais ou menos. Se os strings comparados não envolverem caracteres acentuados, a deficiência dessa função pode ser contornada, como foi visto na **Seção 9.10.9**.

33. Principalmente, para verificar se dois strings são exatamente iguais ou não.

34. A função **strcoll()** permite a especificação de uma forma de colação de caracteres. A função **strcmp()** usa sempre um mesmo tipo de colação de caracteres.

35. Quando não se especifica o tipo de colação de caracteres que a função **strcoll()** deve usar.

36. Consulte a **Seção 9.5.6**.

37. Consulte a **Seção 9.5.6**.

38. Consulte a **Seção 9.5.7**.

39. Consulte a **Seção 9.5.8**.

40. O modo mais simples é assim: `p = strchr(str, '\0')`. Mas também se pode obter idêntico resultado com **strrchr()** ou **strlen()**.

41. Assim: `strchr(str, '\0') - str`.

42. Consulte a **Seção 9.5.9**.

43. (a) Porque a função **strtok()** pode alterar seu primeiro parâmetro. (b) Violação de memória com o consequente aborto de programa.

44. O primeiro parâmetro deve apontar para um array com tamanho suficiente para conter o resultado da concatenação.

45. (a) Ele deveria ter sido qualificado com **const**. (b) Só Deus sabe. (Uma instrução vazia seria mais óbvia do que **continue**.) (c) Consulte a **Seção 9.5.3**.

46. Porque a função **strtok()** recebe um string constante como primeiro parâmetro.

47. O segundo parâmetro na primeira chamada de **strcat()** deveria ser um string, mas, em vez disso, é um caractere.

48. (a) `Token: um`. (b) Porque o primeiro parâmetro da chamada de **strtok()** no corpo do laço deveria ser **NULL**.

49. (a) Concatenação de strings. (b) Comprimento de string. (c) Essa função é equivalente a **strcmp()**.

50. `defabcdef`.

51. Essa função retorna `1` se os strings recebidos como parâmetros são iguais ou `0` em caso contrário.

52. A função `F()` funciona assim: os strings são comparados caractere a caractere na expressão condicional do laço **for**. Quando respectivos caracteres são iguais, o corpo do laço é executado. Nesse corpo, verifica-se se o caractere corrente é igual ao caractere terminal. Se esse é o caso, os dois strings são considerados iguais e a função retorna `0` [assim como faz **strcmp**()]. Se os caracteres ora comparados são diferentes, o laço **for** encerra e a função retorna a diferença `str1[i] - str2[i]`. Esse último valor retornado está de acordo com a especificação de **strcmp**(), pois se o caractere `str1[i]` for menor do que `str2[i]`, o resultado dessa diferença será negativo; caso contrário, esse resultado será positivo

53. Os respectivos caracteres dos strings são comparados na expressão condicional do laço **for**, que possui duas condições de parada: (1) os caracteres são diferentes e (2) os caracteres são iguais, mas o primeiro caractere (e o segundo também, obviamente) é igual a `'\0'`. Se o laço **for** encerrar porque a condição de parada (1) foi atingida, o resultado da diferença retornada será positivo se o caractere do primeiro string for maior do que o caractere do segundo string; caso contrário, o valor retornado será negativo. Se o laço **for** encerrar porque a condição de parada (2) foi atingida, a função retorna zero. Em qualquer caso, essa função retorna o valor que **strcmp**() retornaria.

54. Essa função conta o número de ocorrências do caractere representado pelo segundo parâmetro no string representado pelo primeiro parâmetro.

55. (a) Definitivamente, não! (b) Porque muitas implementações de C adotaram essa bizarrice.

56. (a) Consulte a **Seção 9.6**. (b) Idem. (c) Não. O programador pode denominá-los `joao` (sem acento) e `maria`, se preferir.

57. Usando `argv[0]` [substitua *argv* por qualquer outro nome com o qual você tenha batizado o segundo parâmetro de **main**()].

58. Ele deve testar se `argc` é maior do que `2` [substitua *argc* por qualquer outro nome que você tenha atribuído ao primeiro parâmetro de **main**()].

59. Consulte a **Seção 9.7**.

60. (a) **isalnum**(). (b) **isalpha**(). (c) **isdigit**(). (d) **isupper**().

61. Ela retorna `1` quando o string recebido como parâmetro contém apenas letras e dígitos. Caso contrário, ela retorna `0`.

62. **tolower**().

63. Depende da localidade usada no programa que contém essa chamada. Na localidade padrão, o retorno dessa função será zero. Numa localidade que considere o caractere `'ã'` como letra, o retorno dessa função será `1`.

64. Consulte a **Seção 9.8**.

65. Devido ao fato de a ordem de avaliação de parâmetros não ser definida.

66. (a) `F1(ar)` e `F2(ar)`. (b) A chamada `F1(ar)` pode alterar o conteúdo do array.

67. (a) `F1(str)`, `F2(str)` e `F3(&str)`. (b) `F1(str)` e `F3(&str)`. (c) `F3(&str)`.

68. (a) `F2(str2)`. [Mas o compilador GCC não considera erros de compilação quando as funções `F1()` e `F3()` são chamadas. Ele apenas apresenta mensagens de advertência.] (b) Nenhuma. (c) Nenhuma.

69. Sim.

70. Não.

71. (a) Nada. (b) Esse programa é abortado, porque, na primeira chamada da função `EscreveStrSemEspacos()`, há tentativa de modificação de um string constante. (c) Porque a chamada da função **strchr**() no corpo da função `EscreveStrSemEspacos()` retorna o endereço de um string constante e o compilador não tem como saber disso.

72. Consulte a Seção 9.9.1.
73. Quando a função **atoi()** retorna zero, o resultado é ambíguo porque esse valor pode significar o resultado de uma conversão legítima ou ocorrência de erro.
74. Consulte a Seção 9.9.2.
75. Consulte a Seção 9.9.2.
76. Consulte a Seção 9.9.2.
77. Quando o segundo parâmetro de **strtod()** for **NULL** e essa função retornar zero, não será possível saber se o resultado da conversão foi realmente zero ou ocorreu erro de conversão.

Capítulo 10 — Estruturas, Uniões e Enumerações

1. Consulte a Seção 10.1.
2. Arrays são variáveis estruturadas homogêneas, enquanto estruturas são variáveis estruturadas heterogêneas.
3. Consulte a Seção 10.1.
4. É uma construção da linguagem C que permite definir tipos derivados. (b) `*` (ponteiro), `[]` (array), `()` (função) **struct** (estrutura) e **union** (união).
5. Sim.
6. Porque os elementos (campos) possuem interpretações diferentes, apesar de serem do mesmo tipo.
7. Conceitualmente, a resposta lógica seria `46` (`30 + 12 + 4`). Mas, na prática, a resposta não é tão simples assim e pode ser obtida apenas com o uso de **sizeof** (ou conhecimento mais avançado). Assim, se esse operador for usado, o resultado obtido será `48`. Então, de onde vem essa discrepância? O fato é que, na prática, o tamanho de uma estrutura não pode ser determinado como a soma dos tamanhos de seus campos porque pode ser necessária a inclusão de espaços adicionais de preenchimento. Infelizmente, uma discussão sobre esse tópico está além do escopo de um livro de introdução à programação.
8. (a) Consulte as Seções 10.3.2 e 10.8.(b) Não. Em nenhuma definição de tipo é permitida iniciação.
9. Consulte a Seção 10.3.4.
10. Consulte a Seção 10.3.4.
11. Consulte a Seção 10.3.5.
12. (a) e (b):

```
tEst1  est1 = {"Bola", -2};
tEst2  est2 = {&est1, 2.5},
      *pEst2 = &est2;
```

(c)
```
est2.p->ar[3]
```

(d)
```
pEst2->p->ar[3]
```

13. Consulte a Seção 10.4.
14. O operador de indireção.
15. (a) É a precedência mais elevada da linguagem C. (b) À esquerda.
16. (a) Porque o operador ponto tem maior precedência do que o operador de indireção. Portanto o operando esquerdo do operador ponto é `p`, que não é uma estrutura. (b) Primeira correção: `(*p).b = 2.5`; segunda correção (melhor): `p->b = 2.5`.

17. (a) O operador ponto tem maior precedência do que o operador de indireção e como, nesse caso, o resultado da aplicação do operador ponto é um endereço, pode-se aplicar o operador de indireção sobre ele. (b) Com 100% de certeza, o programa será abortado, pois, como o segundo campo da estrutura não foi explicitamente iniciado, o valor que lhe é atribuído (implicitamente) é `0`, que, nesse caso, significa endereço nulo. Assim, quando for aplicado o operador de indireção sobre esse ponteiro, o programa será abortado.
18. Consulte a **Seção 10.5**.
19. Quando o parâmetro é de saída ou de entrada e saída ou quando a estrutura é um parâmetro de entrada, mas ocupa relativamente muito espaço em memória.
20. Porque, tipicamente, na prática, um ponteiro ocupa menos espaço em memória do que uma estrutura.
21. Quando a estrutura deve ser considerada como parâmetro de entrada.
22. Sim. Consulte a **Seção 10.5**.
23. (a) Porque, tipicamente, um ponteiro ocupa menos espaço em memória do que uma estrutura. (b) O programador deve tomar cuidado para não retornar um zumbi.
24. (a) A função do item (ii) não compila. (b) As funções dos itens (i) e (iv) retornam zumbis. (c) As funções dos itens (iii), (v), (vi) e (vii) estão corretamente implementadas. (d) A função do item (iii) é a mais eficiente, pois ela recebe um endereço como primeiro parâmetro e também retorna um endereço. (e) A função do item (vi) é a mais ineficiente, pois ela aloca espaço para uma estrutura e retorna essa estrutura.
25. (a) (i), (ii) e (iv) (b) (iii) Ocorre erro de compilação. (v) O programa é abortado.
26. Consulte a **Seção 10.6**.
27. Quando uniões oferecem oportunidade para economizar memória.
28. Consulte a **Seção 10.6**.
29. (a) Resposta compatível com um livro introdutório: 13 bytes. Resposta sênior: o tamanho da variável `registro` depende de implementação. Mas, o uso de **sizeof** para obter esse tamanho é sempre portável. (b) 8 bytes.
30. Consulte a **Seção 10.7**.
31. Consulte a **Seção 10.7**.
32. Quando ocorre a atribuição ao campo `x` da união, o campo `y` deixa de ser válido e quando ocorre a atribuição ao campo `y` da união, o campo `x` deixa de ser válido. Portanto nenhum resultado exibido faz sentido.
33. Consulte a **Seção 10.8**.
34. Assim: `tEstrutura est = {.b = 5}`.
35. Assim: `tExterna est = {.b.x = 5}`.
36. Assim: `int ar[10] = {[3] = -2}`.
37. Consulte a **Seção 10.9**.
38. Consulte a **Seção 10.9**.
39. Consulte a **Seção 10.9**.
40. Consulte a **Seção 10.9**.
41. Não.
42. Não. Consulte a **Seção 10.9**.
43. Infelizmente, sim.
44. Melhora de legibilidade apenas.
45. Porque compiladores de C não respeitam o conceito de enumeração.
46. `C1` recebe `-1` (óbvio), `C2` recebe `0` (`C1 + 1`), `C3` recebe `0` (óbvio), `C4` recebe `1` (`C3 + 1`).

47. (a) A (aparente) semelhança entre estruturas e enumerações encontra-se no modo como seus tipos são definidos. (b) Estruturas são variáveis estruturadas; enumerações não são variáveis estruturadas.

Capítulo 11 — Processamento de Arquivos

1. Consulte a **Seção 11.1**.
2. Consulte a **Seção 11.1**.
3. Consulte a **Seção 11.1**.
4. Consulte a **Seção 11.1**.
5. Esse cabeçalho contém componentes que lidam com entrada e saída em geral.
6. Consulte a **Seção 11.2**.
7. Consulte a **Seção 11.2**.
8. Consulte a **Seção 11.2**.
9. Consulte a **Seção 11.2**.
10. Consulte a **Seção 11.3**.
11. Consulte a **Seção 11.3**.
12. Por meio de estruturas do tipo **FILE**.
13. Na maioria das vezes, não.
14. (a) Consulte a **Seção 11.2**. (b) No cabeçalho `<stdio.h>`.
15. Consulte a **Seção 11.3.2**.
16. Consulte a **Seção 11.4**.
17. Consulte a **Seção 11.4**.
18. Essa constante simbólica representa o tamanho mínimo que deve ter um array que armazena um nome de arquivo numa dada implementação de C. (Essa definição é equivalente àquela apresentada na **Seção 11.4**.)
19. O arquivo não existe, falha de dispositivo de entrada ou saída, o programa não tem permissão do sistema operacional para acessar o arquivo.
20. (a) Consulte a **Seção 11.4**. (b) Aborto de programa.
21. Consulte a **Seção 11.4**.
22. (a) Não há interpretação de caracteres que representam quebras de linha. (b) Ocorre a referida interpretação. (**NB**: Em sistemas da família Unix não há diferença.)
23. Pode. Mas, normalmente, não deve.
24. Consulte a **Seção 11.4.3**.
25. São modos de abertura de arquivo que permitem leitura e escrita.
26. Consulte a **Seção 11.4.3**.
27. Formato de texto.
28. Modos de abertura para streams binários incluem a letra *b*. (Modos de abertura para streams de texto podem incluir a letra *t*, mas, na prática, raramente ela é usada.)
29. (a) `"r"` é usado com streams de texto; `"rb"` é usado com streams binários. (b) Não há diferença; ambos são usados com streams de texto. (c) A diferença é que `"rt"` é usado com streams de texto, enquanto `"rb"` é usado com streams binários.
30. Consulte a **Tabela 11–3** na **página 561**.

31. Modos de abertura que contêm a letra *w* detonam arquivos que tenham o mesmo nome usado como primeiro parâmetro de **fopen**().
32. (a) Sim. (b) Sim, mas se ele for aberto em modo binário será mais eficiente, pois não haverá interpretação de quebra de linha. Em sistemas da família Unix não faz a menor diferença.
33. Testar se a abertura do arquivo foi bem sucedida.
34. Por meio da constante simbólica **FOPEN_MAX**. Consulte também a Seção 11.15.1.
35. Usando a constante simbólica **FILENAME_MAX**, definida em `<stdio.h>`.
36. A constante simbólica **FILENAME_MAX** deve ser usada para dimensionar arrays que armazenam nomes de arquivos.
37. A constante simbólica **FOPEN_MAX** representa o número máximo de arquivos que a implementação de C ora utilizada garante que podem estar abertos simultaneamente.
38. O problema é que, muito provavelmente, o nome do arquivo introduzido pelo usuário inclui o caractere `'\n'` e esse caractere não faz parte de nenhum nome de arquivo. Esse problema só não ocorre se o usuário digitar um nome de arquivo contendo um número de caracteres igual `FILENAME_MAX - 1`, o que é pouco provável porque esse valor é muito grande. A Seção 11.9.6 mostra como remover o caractere `'\n'` de um string lido com **fgets**().
39. Consulte a Seção 11.5.
40. Um valor que indica se operação que ela tenta realizar foi bem sucedida ou não.
41. Consulte a Seção 11.5.
42. Sim, porque libera o espaço ocupado pela estrutura **FILE** associada ao arquivo.
43. A função **fclose**() deveria receber o ponteiro `p` como parâmetro, e não o string `"teste.bin"`.
44. Consulte a Seção 11.6.
45. Pode ser, mas, hoje em dia, é muito pouco provável.
46. Consulte a Seção 11.6.
47. Porque o valor dessa constante pode indicar que o final do arquivo ora processado foi atingido ou a ocorrência de erro de processamento do mesmo arquivo.
48. Consulte a Seção 11.6.
49. Porque o uso de **EOF** pode ser ambíguo. O valor retornado por **feof**() nunca é ambíguo.
50. A função **ferror**() permite verificar se ocorreu erro numa operação de entrada ou saída.
51. O uso de **EOF** pode ser ambíguo enquanto o valor retornado por **ferror**() não é ambíguo.
52. Ele continuará indicando ocorrência de erro em operações subsequentes de entrada ou saída, mesmo que esse não seja o caso.
53. (a) Por meio de uma chamada de **rewind**(). b) Por meio de uma chamada de **clearerr**().
54. (a) Por meio de uma chamada de **fseek**(), **rewind**() ou **ungetc**(). (b) Por meio de uma chamada de **clearerr**().
55. (a) A função **clearerr**() serve para remover indicativo de erro ou final de arquivo. (b) Porque existem funções que efetuam essa tarefa implicitamente.
56. Consulte a Seção 11.7.
57. Consulte a Seção 11.7.
58. Consulte a Seção 11.7.
59. Consulte as Seções 11.3.2 e 11.7.
60. Para descarregar explicitamente streams de saída (apenas).

61. A função **fflush()** não deve ser usada com parâmetros que representam streams de entrada, como é o caso de **stdin**.

62. Todos os streams de saída correntemente abertos no programa que contém essa instrução serão descarregados.

63. Consulte a **Seção 11.8**.

64. Porque existem funções [p. ex., **scanf()** e **printf()**] que realizam operações de entrada e saída nesses streams sem que eles precisem ser especificados explicitamente.

65. Consulte a **Seção 11.9.1**.

66. Por duas razões. A principal delas é que a função **getchar()** nem sempre retorna um valor associado a um caractere; i.e., ela também pode retornar **EOF**, que, tipicamente não cabe numa variável do tipo **char**. A segunda razão é mais sutil: devido a conversões implícitas de alargamento (v. **Seção 3.10.1**), não se costumam definir varáveis, parâmetros ou tipos de retorno com o tipo **char**. Do ponto de vista prático, essa segunda justificativa não faz diferença num programa, mas pode demonstrar falta de conhecimento do programador.

67. Quando encontra caracteres remanescentes no buffer associado a **stdin**, **getchar()** não interrompe a execução de um programa.

68. Porque o programador tem controle total sobre o que é escrito na tela, mas não tem nenhum controle sobre o que um usuário de seu programa pode digitar.

69. Por intermédio de uma função que lê caracteres e os converte, quando possível, em números, como faz a função **scanf()**, por exemplo.

70. Tipicamente, apenas especificadores de formato e espaços em branco.

71. Significam parâmetros que não têm quantidades ou tipos especificados.

72. Consulte a **Seção 11.9.2**.

73. (a) O número de variáveis que tiveram seus valores modificados pela função **scanf()**. (b) Para testar se algum valor que deveria ter sido lido foi realmente lido.

74. Consulte a **Seção 11.9.3**.

75. Esse programa não testa se o usuário realmente introduziu dois números inteiros.

76. O segundo e o terceiro parâmetros de **scanf()** deveriam ser endereços de variáveis.

77. (a) Essa chamada de **scanf()** espera que o usuário digite exatamente: *Digite um inteiro:* seguido por um número inteiro em base decimal. (b) `x = n`, sendo `n` um valor indeterminado.

78. O fato de ambos usarem especificadores de formato.

79. (a) Sim. (b) Não.

80. Nos itens (a) e (b), no máximo, **scanf()** lerá um número de valores igual ao número de especificadores de formato. No item (b), essa função não atribuirá valor a variáveis que não casarem com respectivos especificadores de formato.

81. (a) Um. (b) Dois.

82. (a) A função **scanf()** não leu o valor que deveria ler. (b) A função **scanf()** leu o valor que deveria ler e o atribuiu à variável `x`. (c) Ocorreu erro de leitura.

83. **EOF**, `0`, `1` ou `2`.

84. A função **scanf()** tenta alterar o conteúdo do string constante apontado por `p`.

85. Porque a segunda chamada de **printf()** interpreta o valor da variável `x` como o endereço de um string.

86. (a) (i) Faltou preceder a variável `x` com **&**; (ii) o especificador de formato deveria ser **%d**. (b) O compilador GCC é capaz de apresentar mensagens de advertência nos dois casos. Mas, nem todo compilador o faz nem tem obrigação de fazê-lo.

87. O especificador de formato **%*n*s**, sendo ***n*** um inteiro positivo, deve ser usado em vez de **%s** e deve-se usar um array com tamanho pelo menos igual a ***n*** para prevenir corrupção de memória.

88. Porque ela não é capaz de ler strings com espaços em branco em seu interior.

89. (a) Esse especificador informa que a função **scanf()** deve ler dígitos e armazená-los num array até que seja encontrado um caractere que não é dígito. (b) Esse especificador é o contrário daquele do item (a).

90. Porque ela causa corrupção de memória e não há remédio possível para isso.

91. Consulte a **Seção 11.9.6**.

92. **stdin**.

93. (a) A função **scanf()** lê os caracteres `'1'`, `'2'` e `'3'`. (b) Os caracteres `'z'` e `'\n'` permanecerão armazenados no buffer. O resto da história encontra-se na **Seção 11.9.6**.

94. (a) Porque o nome do usuário lido provavelmente inclui o caractere `'\n'`. Note que o array que armazena o nome tem `200` elementos. Portanto o problema só não ocorrerá se o nome do usuário tiver mais de `199` caracteres! (b) A **Seção 11.9.6** mostra como remover o caractere `'\n'` de um string lido com **fgets()**.

95. Porque o caractere `'\n'` digitado quando o usuário introduz o caractere `'A'` permanece no buffer associado ao teclado.

96. (a) Nada será lido. (b) O conteúdo do buffer será: '`z`', `'1'`, `'2'`, `'3'` e `'\n'`.

97. `'\n'`.

98. (a) Quando a função **scanf()** é usada com o especificador **%c**. (b) Quando caracteres em branco devem ser saltados no início de uma operação de leitura. (c) Quando esse caractere encerra uma operação de leitura.

99. (a) Espaços em branco (incluindo `'\n'`). (b) Quando **scanf()** lê caracteres.

100. Consulte a **Seção 11.9.3**.

101. (a) O laço **while** seria infinito porque se ocorresse uma leitura indevida, a função **scanf()** continuaria encontrando os mesmos caracteres no buffer de entrada. (b) Porque não cria código espaguete e a codificação é eficiente.

102.

```
int  umInt, nValoresLidos;

while (1) {
   printf("Digite um valor inteiro: ");
   nValoresLidos = scanf("%d", &umInt);

   if (nValoresLidos) {
      break;
   }

   LimpaBuffer();
   printf("Valor incorreto. ");
}
```

103. Para remover caracteres remanescentes após uma operação de leitura.

104. (a) Porque a biblioteca padrão de C não provê nenhuma função que remove caracteres remanescentes no buffer associado a **stdin**. (b) Porque a função **fflush()** deve ser usada apenas com buffers associados a streams de saída.

105. Consulte a **Seção 11.9.5**.

106. Consulte a **Seção 11.10**.

107. Consulte a **Seção 11.10**.

108. (a) Streams de texto ou binários. (b) Streams de texto. (c) Streams binários. (d) Streams de texto.

109. (a) **fgetc()** e **fputc()**. (b) **fgets()** e **fputs()**. (c) **fread()** e **fwrite()**.

110. (a) São partições conceituais (ou lógicas) de um arquivo. (b) É uma partição de um registro.

111. Consulte a **Seção 11.11.1**.

112. Não.

113. Devido ao fato de o operador diferente (representado por `!=`) ter precedência maior do o operador de atribuição, à variável `c` será sempre atribuído `1` ou `0`.

114. Se a função **fgetc()** retornar **EOF**, esse valor será escrito no stream `streamSaida`. Para piorar, a expressão condicional do laço **while** não testa a ocorrência de erro de leitura ou escrita.

115. Eis o programa:

```
rewind(streamA);

while (1) {
   c = fgetc(streamA);

   if (feof(streamA) || ferror(streamA)) {
      break;
   }

   fputc(c, streamB);

   if (ferror(streamB)) {
      break;
   }
}
```

116. (a) Consulte a **Seção 11.9.6**. (b) Consulte a **Seção 11.11.2**.

117. (a) Consulte a **Seção 11.11.3**. (b) Processamento por blocos, sequencial ou via acesso direto.

118. Consulte a **Seção 11.11.3**.

119. **getchar()** só efetua leitura em **stdin**; **fgetc()** permite a especificação de um stream de entrada.

120. A função **fscanf()** deve ser usada em leitura formatada (em streams de texto), enquanto **fread()** deve ser usada em processamento de blocos (em streams binários).

121. Esse programa escreve a última linha do arquivo duas vezes na tela. Uma maneira de corrigir esse programa é substituindo o laço **while** desse programa por:

```
while(1) {
   fgets(linha, sizeof(linha), stream);

   if (feof(stream)) {
      break;
   }

   fputs(linha, stdout);
}
```

122. Esse programa é abortado porque, quando o final do arquivo é atingido, a função **fgets()** retorna **NULL**, de modo que **fputs()** é chamada tendo esse valor como primeiro parâmetro. Como, para um string ser escrito, seus caracteres precisam ser acessados, essa última função aplica o operador de indireção sobre um ponteiro nulo, o que causa o aborto do programa.

123. O tipo da variável `c` deveria ser **int**, e não **char**.

124. (1) O tipo da variável `ch` deveria ser **int**, e não **char**. (2) A chamada de **feof()** deveria ocorrer antes do processamento do caractere, e não depois.

125. Consulte a **Seção 11.12**.

126. Consulte a **Seção 11.12**.

127. **stdin**.

128. Consulte a **Seção 11.12.1**.

129. Consulte a **Seção 11.12.1**.

130. Zero indica que a chamada dessa função foi bem sucedida; um valor diferente de zero indica o contrário.

131. Para streams binários, o valor retornado por **ftell()** representa o número de bytes calculado a partir do início do arquivo. Para streams de texto, o valor retornado por **ftell()** é dependente de implementação.
132. Consulte a **Seção 11.12.1**.
133. (a) Chamar uma função de posicionamento. (b) Chamar uma função de posicionamento ou **fflush()**.
134. Se os dois nomes de arquivo recebidos como argumentos pelo programa forem os mesmos e existir um arquivo com esse nome, seu conteúdo será destruído.
135. Consulte a **Seção 11.13**.
136. Consulte a **Seção 11.13**.
137. A função **rewind()** é recomendada quando se deseja garantir que o processamento de um arquivo começa em seu primeiro byte. Mas, o uso de **fseek()** é preferível.
138. (a) Para garantir que a leitura começa no início do stream. (b) Porque, nesse caso, com certeza, o apontador de posição do arquivo aponta para o primeiro byte desse arquivo.
139. Suponha que stream é um stream que permite acesso direto. Então a chamada de **fseek()**: `fseek(stream, 0, SEEK_SET)` pode ser usada em substituição à chamada de **rewind()**: `rewind(stream)`.
140. Porque a função **fseek()** permite verificar quando ela é bem sucedida, o que não é caso de **rewind()**.
141. Consulte a **Seção 11.14**.
142. Consulte a **Seção 11.14**.
143. Consulte a **Seção 11.14**.
144. Consulte a **Seção 11.14**

Capítulo 12 — Alocação Dinâmica de Memória

1. Consulte as **Seções 8.11.4**, **8.11.6**, **9.10.1**, **9.10.6**, e **10.10.1**.
2. (1) Gerenciamento de uma turma de alunos que pode aumentar ou diminuir de tamanho. (2) Gerenciamento de uma lista de espera. (3) Leitura de um string cujo tamanho máximo não é previamente especificado.
3. Consulte a **Seção 12.1**.
4. Consulte a **Seção 12.1**.
5. (a) Todas as três variáveis que possuem identificadores (i.e., `x`, `y` e `p`) são estáticas, pois o código necessário para suas alocações é gerado quando o programa que as contém é compilado. (b) A única variável dinâmica é a variável anônima alocada por **malloc()**.
6. (a) São. (b) Também são estáticas.
7. Consulte a **Seção 12.2**.
8. (a) Sim. (b) Talvez, sim, mas é melhor assumir que não.
9. Como **malloc()**.
10. Como **free()**.
11. Porque o bloco para o qual esse ponteiro aponta pode ser liberado por **realloc()**.
12. (a) É uma variável que não possui um identificador associado a ela. (b) Porque qualquer variável alocada dinamicamente é anônima e só pode ser acessada indiretamente via ponteiros.
13. Existem sistemas de execução que liberam automaticamente blocos alocados dinamicamente que não podem mais ser usados. Mas, esses sistemas de **coleta de lixo** (*garbage collection*, em inglês) são onerosos. Assim, em nome da eficiência, C e muitas outras linguagens deixam a tarefa de liberação de memória alocada dinâmicamente a cargo do próprio programador. Esse tópico não é discutido em detalhes neste texto, que é um livro introdutório.

14. (1) Chamar **free()** com um ponteiro que aponta para um bloco que já foi liberado. (2) Esquecer de emparelhar cada chamada de função de alocação com uma chamada de **free()**.

15. Se a função **realloc()** retornar **NULL**, o bloco para o qual `p` apontava antes da chamada dessa função não poderá mais ser acessado.

16. A função `CriaArray()` altera os valores dos elementos do array criado dinamicamente, mas não altera o valor do ponteiro para o início desse array representado pelo parâmetro. Portanto o primeiro parâmetro dessa função deveria ser declarado como: `int **array`. Então, as referências ao parâmetro `array` no corpo da função devem ser convenientemente reescritas.

17. Consulte a **Seção 12.3**.

18. Consulte a **Seção 12.3**.

19. Consulte a **Seção 12.3**.

20. Nas funções **malloc()** e **free()**.

21. (a) Porque o operador de indireção não pode ser aplicado sobre um ponteiro do tipo **void** *. (b) Com conversão explícita: `*(int*) p = 5`.

22. Consulte a **Seção 12.4**.

23. Consulte a **Seção 12.4**.

24. Consulte a **Seção 12.4**.

25. (a) É uma variável anônima alocada dinamicamente que foi liberada por meio de **free()**, mas que continua a ser usada. (b) Consulte a **Seção 8.9.4**.

26. (a) Escoamento de memória ocorre quando um programa tem memória alocada para si sem que ela seja efetivamente utilizada. (b) Consulte a **Seção 12.4**.

27. (a) Verificando se o ponteiro ao qual é atribuído o endereço retornado é **NULL**. (b) Porque nem é sempre uma função de alocação dinâmica é bem sucedida.

28. Quando a função **free()** é chamada, o ponteiro `ar` não está mais apontando para o bloco alocado com **malloc()**.