APÊNDICE D

ERROS COMUNS DE PROGRAMAÇÃO EM C

D.1 Introdução

Este apêndice discute erros que são comuns em programação usando a linguagem C. As duas últimas seções apresentam possíveis interpretações para mensagens de erro e advertência frequentemente emitidas pelo compilador GCC.

D.2 Operadores

D.2.1 Uso de Atribuição em vez de Igualdade ou Vice-versa

Algumas vezes, o programador usa distraidamente o operador de igualdade (representado por ==) quando sua intenção é usar o operador de atribuição (representado por =) ou vice-versa. Uma precaução contra esse tipo de erro é, sempre que possível, usar uma constante no lado esquerdo de uma expressão de comparação. Por exemplo, ao analisar uma expressão como a que faz parte da instrução **if** a seguir:

```
int x;
...
if (x = 10)
   ...
```

alguns compiladores podem emitir uma mensagem de advertência, mas isso não impede o programa de ser compilado. Entretanto, se o programador tivesse escrito esse trecho de programa como:

```
int x;
...
if (10 = x)
   ...
```

qualquer compilador indicaria a ocorrência de erro.

D.2.2 Uso Incorreto de Regras de Precedência ou Associatividade

Regras de precedência e associatividade que regem a ordem de aplicação de operadores são facilmente esquecidas, como mostra o seguinte exemplo:

```
while (c = fgetc(stream) != EOF) {
   ...
}
```

Nesse exemplo, a intenção do programador parece óbvia: encerrar o laço **while** quando a função **fgetc()** retornar **EOF**. Mas, provavelmente, ele esqueceu que o operador **!=** tem maior precedência do que o operador **=** e, assim, a expressão que acompanha o laço **while** é interpretada como:

```
c = (fgetc(stream) != EOF)
```

de modo que `c` sempre receberá `0` ou `1`. A correção desse problema é obtida por meio do uso de parênteses:

```
while ((c = fgetc(stream)) != EOF)
```

D.2.3 Uso de && em vez de || ou Vice-versa

Esse problema, na maioria das vezes, é decorrente do uso incorreto das leis de De Morgan (v. **Seção 4.5.4**):

Expressão negada	Expressão Equivalente
`!(A && B)`	`!A || !B`
`!(A || B)`	`!A && !B`

Exemplos:

```
if ( !(x < 0 && y <= 10) )
```

é equivalente a:

```
if ( !(x < 0) || !(y <= 10) ) /* OK */
```

ou:

```
if ( x >= 0 || y > 10 ) /* OK */
```

Mas não é equivalente a:

```
if ( !(x < 0) && !(y <= 10) ) /* INCORRETO! */
```

nem a:

```
if ( x >= 0 && y > 10 ) /* INCORRETO! */
```

Consulte a **Seção 4.5.4** para obter uma completa discussão sobre esse tópico.

D.2.4 Uso de & em vez de &&

Os operadores **&&** e **&** não são intercambiáveis. Mais precisamente, **&&** é um operador lógico, mas **&** *não é operador lógico em C*. Existem várias diferenças entre esses dois operadores, mas elas não serão apresentadas aqui porque o operador **&** (**conjunção sobre bits**) é usado em programação de baixo nível, que não é um tópico discutido neste livro.

D.2.5 Uso de | em vez de ||

Os operadores representados por **||** e **|** não são permutáveis: o símbolo **||** representa um operador lógico, enquanto que o símbolo **|** *não representa operador lógico em C*. O operador representado por **|** (**disjunção sobre bits**) é usado em programação de baixo nível e não foi discutido neste livro.

D.2.6 Os Operadores Lógicos de C Não São Comutativos

Diferentemente do que ocorre em Lógica Matemática, os operadores lógicos **&&** e **||** de C não são comutativos. Isso significa que a ordem com a qual seus operandos são escolhidos é importante. Considere, por exemplo, a seguinte expressão lógica:

```
x > 0 || ++y < 10
```

Nessa expressão, devido ao curto-circuito, a sub-expressão `++y < 10` será avaliada apenas quando `x` não for maior do que `0`. Consequentemente, apenas nessa situação, a variável `y` será incrementada. Por outro lado, na expressão a seguir, obtida da expressão anterior com a inversão dos operandos do operador **||**:

```
++y < 10 || x > 0
```

a variável `y` será sempre incrementada.

Como outro exemplo da importância da escolha da ordem dos operandos de um operador lógico, considere o seguinte fragmento de programa:

```
if (x > 0 && y%x)
   ...
```

Naquilo que diz respeito à avaliação da expressão que acompanha a instrução **if**, o fragmento de programa acima funciona perfeitamente bem. No entanto, se os operandos do operador **&&** forem invertidos:

```
if (y%x && x > 0)
   ...
```

o programa será abortado quando `x` for igual a zero devido à tentativa de divisão inteira por zero.

Na **Seção 12.11**, existem outros exemplos de expressões lógicas nas quais a escolha da ordem dos operandos é essencial.

D.2.7 Suposições sobre Ordem de Avaliação de Operandos

Em C, apenas quatro operadores possuem ordem de avaliação de operandos definida: **&&**, **||**, **?:** e **,** (operador vírgula). Consequentemente, o uso de operadores com efeito colateral sobre uma variável que aparece mais de uma vez numa mesma expressão acarreta, na maioria das vezes, num resultado que não é portável (v. **Seção 3.9**).

Considere o seguinte exemplo:

```
int i = 0, arA[N_ELEMENTOS], arB[N_ELEMENTOS];
...
while (i < N_ELEMENTOS) {
   arA[i] = arB[i++];
}
```

O problema com esse trecho de programa é que o operador de atribuição não possui ordem de avaliação de operandos definida. Logo, apenas se o primeiro operando da atribuição for avaliado primeiro, a provável intenção do programador será satisfeita. Uma solução para esse problema é escrever o laço **while** desse trecho de programa como:

```
while (i < NUM_ELEMENTOS) {
   arA[i] = arB[i];
   ++i;
}
```

É importante notar que a categoria de problemas descrita nesta seção não pode ser resolvida por meio do uso de parênteses, pois esses problemas não são decorrentes de precedência ou associatividade.

D.2.8 Uso de Ponto em vez de Seta ou Vice-versa

O operador **.** (ponto) só pode ser usado quando seu operando esquerdo é uma estrutura e seu operando direto é um campo dessa estrutura. Por outro lado, o operador seta só pode ser usado quando seu operando esquerdo é um ponteiro para estrutura e seu

operando direito é um campo dessa estrutura. Qualquer violação dessas regras causa erro de compilação (v. **Seções 9.3.4** e **D.18.36**).

D.2.9 Uso do Operador de Indireção com o Operador Seta

Quando um parâmetro é um ponteiro para um ponteiro para uma estrutura (v. **Seção 12.11.3**), é necessário usar o operador de indireção em conjunto com o operador seta para acessar um campo dessa estrutura. O problema é que, nesse caso, muitas vezes, o programador esquece que o operador de indireção tem menor precedência que o operador seta, como mostra o seguinte exemplo:

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

typedef struct no {
           int        dado;
           struct no *proximo;
        } tNo, *tLista;

void DestroiListaErrada(tLista *ptrLista)
{
   tLista p;

   if (!*ptrLista) {
      return;
   }

   p = *ptrLista;

   do {
      *ptrLista = *ptrLista->proximo; /* Erro de sintaxe! */

      free(p);

      p = *ptrLista;
   } while (p);
}

int main(void)
{
   return 0;
}
```

A função `DestroiListaErrada()` é semelhante à função `DestroiLista()` apresentada na **Seção 12.11.5**, mas contém um erro de sintaxe na expressão:

```
*ptrLista->proximo
```

O problema com essa expressão é que o operador seta é aplicado antes do operador de indireção. Mas, quando o operador seta é usado, espera-se que seu operando da esquerda seja um ponteiro para uma estrutura, o que não é o caso. Esse problema pode ser corrigido com a aplicação de parênteses:

```
(*ptrLista)->proximo
```

Agora, o operando esquerdo do operador seta é um ponteiro para estrutura, pois, como `ptrLista` é ponteiro para ponteiro para estrutura, `*ptrLista` é um ponteiro para estrutura.

D.3 Estruturas de Controle

D.3.1 Uso Indevido de Ponto e vírgula

Observe o seguinte trecho de programa:

```
int i = 10;
...
while(i > 0);
   --i;
```

A endentação usada pelo programador indica que sua provável intenção era que a instrução:

```
--i;
```

constituísse o corpo do laço **while**. Entretanto, o ponto e vírgula ao final da linha contendo **while** termina prematuramente esse laço.

Outro exemplo:

```
if (x > max);
   max = x;
```

Nesse último exemplo, a instrução:

```
max = x;
```

será sempre executada, independentemente do resultado da avaliação da expressão:

```
x > max
```

Uma prevenção contra esse tipo de erro que termina uma estrutura de controle prematuramente é usar sempre um abre-chaves ao final da linha inicial dessa instrução (v. **Seção 4.10.6**). Obviamente, esse abre-chaves deve ter um fecha-chaves correspondente ao final do corpo da mesma instrução.

As seguintes linhas de programa não requerem ponto e vírgula e sua inclusão pode causar erro de sintaxe ou de lógica:

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

  ```
  while (x < 10 ) {
     ++x;
  }
  ```

  não requer ponto e vírgula para terminar porque o fecha-chaves a termina. Mas, toda instrução **do-while**, como por exemplo:

  ```
  do {
     ++x;
  } while (x < 10 );
  ```

  requer ponto e vírgula para terminar.

D.3.2 Instrução switch-case sem break

Em C, ' possui efeito cascata. Ou seja, uma vez que um caso é selecionado, todas as instruções correspondentes a esse caso e aos casos subsequentes são executadas. Muito raramente, quando se usa uma instrução **switch-case**, esse é o efeito desejado. Isto é,

mais frequentemente, o que o programador deseja é executar exatamente as instruções correspondentes a um dado caso e apenas essas instruções. Esse efeito é obtido com o uso da instrução **break** (v. **Seção 4.6.3**). Portanto, para cada instrução **switch-case**, verifique se não está faltando **break**. O efeito cascata é tão raramente usado que, se você realmente sente necessidade de usá-lo, provavelmente, seu raciocínio está equivocado.

D.3.3 Instrução do-while Confundida com REPEAT-UNTIL

Em muitas linguagens de programação (p. ex., Pascal, Basic, Modula-2), a estrutura de controle equivalente a **do-while** em C repete a execução de seu corpo *até que* uma certa condição seja satisfeita, em vez de *enquanto* uma condição é satisfeita. Nessas linguagens, o programador deve especificar qual é a condição que deve ser satisfeita para que o laço pare. Desse modo, programadores acostumados a programar nessas linguagens tendem a pensar que a estrutura **do-while** de C funciona dessa mesma maneira, mas esse não é o caso. Em C, a única diferença entre **while** e **do-while** é que o corpo da primeira instrução pode não ser executado nenhuma vez, enquanto que o corpo da segunda instrução sempre é executado pelo menos uma vez. Mas, em termos de condição de parada (ou expressão condicional) não há diferença entre **while** e **do-while** (v. **Seção 4.5**).

D.3.4 else que Não Corresponde ao if Desejado

Conforme foi afirmado na **Seção 4.6.2**, uma parte **else** sempre corresponde à parte **if** da estrutura de controle **if-else** mais próxima que ainda não possui uma parte **else** correspondente. Infelizmente, essa regra é tão fácil de apreender quanto de ser esquecida. Por exemplo, considere o seguinte trecho de programa:

```
if (x == 0)
   if (y == 0)
      ++y;
else
   --x;
```

Pelo modo como endentou esse trecho de programa, aparentemente, o programador gostaria que a parte **else** correspondesse à primeira instrução **if**. Entretanto, devido à regra de casamento de partes **if** e partes **else**, essa parte **else** casa com a segunda instrução **if**. O problema pode ser corrigido envolvendo-se a segunda instrução **if** entre chaves, como mostrado a seguir:

```
if (x == 0) {
   if (y == 0)
      ++y;
} else
   --x;
```

Aliás, a melhor recomendação é sempre envolver o corpo de qualquer estrutura de controle com chaves. Portanto, no caso em questão, o melhor seria escrever as estruturas de controle como:

```
if (x == 0) {
   if (y == 0) {
      ++y;
   }
} else {
   --x;
}
```

mesmo sem ser estritamente necessário o uso de chaves envolvendo a instrução `--x`.

D.3.5 Laços de Contagem que Encerram uma Iteração Antes ou Depois

Um erro bastante comum que ocorre com laços de contagem é encerrar um laço **for** uma iteração antes ou depois do esperado. Por exemplo, esse tipo de erro ocorre se a intenção do programador for executar 10 vezes o corpo do laço **for** a seguir:

```
for (i = 0; i <= 10; ++i) {
   ...
}
```

porque ele encerra com uma iteração além do que deveria, pois quando `i` varia entre `0` e `10` (inclusive), ele assume 11 valores. Por outro lado, se a intenção for aquela mencionada, o laço:

```
for (i = 1; i < 10; ++i) {
   ...
}
```

encerra com uma iteração a menos porque o uso do operador representado por <, em vez daquele representado por <=, faz com que a execução do corpo do laço ocorra quando `i` estiver entre `1` e `9` (e não entre `1` e `10`).

D.3.6 Trocar Pontos e Vírgulas por Vírgulas e Vice-versa em Laços for

Usar vírgulas, em vez de pontos e vírgulas, para separar as expressões que fazem parte de uma instrução **for** é comum entre programadores iniciantes na linguagem C. Mas, esse é um tipo de erro sintático que, evidentemente, é capturado pelo compilador.

Agora, trocar vírgulas por pontos e vírgulas e vice-versa, provavelmente resultará num erro lógico. Por exemplo, a instrução **for** do trecho de programa a seguir:

```
int  i, j;

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

é perfeitamente legal e produz um resultado normal. Contudo, se o programador inadvertidamente trocar vírgulas por pontos e vírgulas e vice-versa, o laço **for** resultante:

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

também é legal (do ponto de vista sintático), mas esse laço é infinito (i.e., ele encerrará apenas quando ocorrer overflow na variável `i`).

D.3.7 Alterar uma Variável de Contagem no Corpo de um Laço for

Alterar o valor de uma variável de contagem no corpo do laço **for** do qual ela faz parte pode fazer com que o laço nunca termine. Por exemplo:

```
for (i = 1; i < 10; ++i) {
   ...
   i = 0;
   ...
}
```

D.3.8 continue Não Faz a Execução Continuar na Linha Seguinte

Muitas vezes, o nome da estrutura de controle **continue** instiga o iniciante em programação em C a imaginar que ela deve ser usada para *continuar* a execução de um programa na próxima instrução (o que ocorre naturalmente). Entretanto, o uso dessa

estrutura de controle no corpo de um laço de repetição tem efeito exatamente oposto, conforme foi visto na **Seção 4.7.2**. Além disso, o uso de **continue** fora de um laço de repetição gera erro de compilação (v. **Seção D.18.19**).

D.3.9 Código Morto

Um desvio incondicional (ou instrução **return**) não deve ser seguido por nenhuma instrução que tenha o mesmo nível de subordinação desse desvio (ou, visualmente, nenhuma instrução com a mesma endentação de um desvio incondicional deve segui -lo). Se essa regra não for seguida, as instruções que seguem o desvio incondicional (ou instrução **return**) nunca serão executadas e, portanto, podem ser removidas do programa que as contém.

Às vezes, um bom compilador é capaz de apresentar uma mensagem de advertência relativa ao problema mencionado. Por exemplo, se você compilar o programa:

```
#include <stdio.h>

int main(void)
{
   /* Declarações e instruções que não interessam aqui */

   return 0;

      /* A seguinte instrução nunca será executada */
   printf("\n\t>>> Obrigado por usar este programa.\n");
}
```

o compilador poderá emitir a seguinte mensagem de advertência:

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

e indicar a instrução:

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

como fonte da mensagem de advertência.

Essa mensagem de advertência significa que a chamada de **printf()** no programa acima jamais será executada, sendo, assim, considerada **código morto**. Infelizmente, nem toda versão do compilador GCC é capaz de apresentar essa mensagem de advertência.

D.4 Definições Incorretas de Funções

Esta seção apresenta alguns problemas associados a definições incorretas de funções. Evidentemente, chamadas de funções também podem apresentar problemas. Essa última categoria de problemas será discutida na **Seção D.5**.

D.4.1 Retorno de Zumbis

Uma função não deve jamais retornar o endereço de uma variável local de duração automática ou de um parâmetro, conforme foi discutido em detalhes na **Seção 7.9.4**. Por exemplo:

```
char* UmaFuncao1(const char *str)
{
   char aux[80];
   ...
   return aux; /* Zumbi retornado */
}
```

Para resolver o problema apresentado por essa função, existem duas abordagens:

(1) Usar uma variável de duração fixa (v. **Seção 5.12**). Por exemplo:

```
char* UmaFuncao2(const char *str)
{
   static char aux[80];
   ...
   return aux; /* OK. Não é Zumbi */
}
```

Um possível problema com essa abordagem é que, a cada chamada dessa função o conteúdo do array `aux[]` poderá ser alterado. Portanto, talvez seja necessário fazer uma cópia dele na função que faz a chamada.

(2) Usar alocação dinâmica de memória (v. **Capítulo 12**). Por exemplo:

```
char* UmaFuncao3(const char *str)
{
   char *aux = malloc(80*sizeof(char));
   ...
   return aux; /* OK. Não é Zumbi */
}
```

Essa solução é melhor do que a anterior, mas é necessário que o programador lembre de incluir uma chamada de **free()** no corpo da função que faz a chamada da função `UmaFuncao3()` para liberar o espaço alocado para o array quando ele não for mais necessário.

O efeito zumbi só se manifesta quando a pilha de execução é alterada, o que ocorre sempre que uma função é chamada.

D.4.2 Retorno Imprevisto

Considere a função `F()` definida no programa a seguir.

```
#include <stdio.h>

int F(int x)
{
   if (x > 0) {
      return x;
   }
}

int main(void)
{
   printf( "\nValor retornado por F(10): %d\n",
           F(10) );
   printf( "Valor retornado por F(-10): %d\n",
           F(-10) );

   return 0;
}
```

De acordo com o cabeçalho da função `F()`, ela deve *sempre* retornar um valor do tipo **int**. Mas, conforme pode-se notar, essa função retorna um valor apenas quando o parâmetro é positivo. O que ocorre, então, quando essa função é chamada com um parâmetro negativo ou zero? De acordo com o padrão ISO de C, o resultado é indefinido.

Quando o último programa é executado, ele exibe na tela:

```
Valor retornado por F(10): 10
Valor retornado por F(-10): -1
```

Mas, esse resultado poderia ser diferente, dependendo da implementação de C usada.

Concluindo, assegure-se que uma função cujo tipo de retorno não é **void** sempre retorna um valor para quaisquer que sejam os valores dos parâmetros recebidos pela função.

D.4.3 Negligência de Regras de Escopo

Observe o seguinte fragmento de programa:

```
int x; /* Variável global */

void F()
{
   double x = 2.54;
   ...
   printf("Valor corrente de x: %d");
}
```

O especificador de formato **%d** usado com **printf()** revela que, provavelmente, o programador desejava apresentar na tela o valor da variável global `x`. Entretanto, como, inadvertidamente, foi declarada uma variável local à função com o mesmo nome que a variável global, pelas regras de escopo de C, é a variável local que é considerada válida no corpo da função.

Examine agora o seguinte programa:

```
#include <stdio.h>

int i;

void F()
{
   printf("\nFuncao F() chamada\n", i);

   for (i = 5; i > 0; --i)
      printf("\ti = %d\n", i);
}

int main(void)
{
   for (i = 0; i <= 5; ++i) {
      F();
   }

   return 0;
}
```

O último programa nunca termina, a não ser que o usuário solicite seu encerramento ao sistema operacional (p. ex., digitando `[CTRL]+[C]`). O problema aqui também tem a ver com escopo, mas nesse caso ocorre o oposto do exemplo precedente. Isto é, se fosse definida uma variável denominada `i` no corpo de `F()`, não haveria nenhum problema.

D.4.4 Escoamento de Memória

Qualquer bloco alocado dinamicamente deve ser liberado explicitamente por meio de uma chamada da função **free()**. Se uma função, logo antes de retornar, tem um ponteiro local apontando para um bloco alocado dinamicamente cujo endereço não é retornado, esse espaço jamais poderá ser acessado novamente. Assim, se essa função for chamada muitas vezes, a capacidade do *heap* poderá ser esgotada bem antes do previsto devido ao escoamento de memória causado pela função. Por exemplo:

```
void UmaFuncao(const char *str)
{
   char *p = malloc(strlen(str) + 1);
   ... /* Executa algum processamento */
       /* usando o bloco alocado      */
}
```

Se a função **free()** não for usada antes do retorno dessa função, cada chamada dela causará escoamento de memória.

D.4.5 Uso Incorreto de Variáveis Locais de Duração Fixa

Considere o seguinte programa que demonstra o uso incorreto de uma variável local de duração fixa.

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

#define TAM_ARRAY 50

/****
 * StrEmMaiusculas(): Retorna um string correspondente ao
 *                    string recebido como parâmetro com
 *                    todas as suas letras em maiúsculas
 *
 * Parâmetro: str (entrada) - o string
 *
 * Retorno: Endereço do string correspondente ao string
 *          recebido como parâmetro com todas as suas
 *          letras em maiúsculas. NULL, se o resultado não
 *          couber no array que o armazenaria.
 ****/
char *StrEmMaiusculas(const char *str)
{
   static char maiusculas[TAM_ARRAY];
   char       *p;

      /* Verifica se o resultado caberá no array */
   if (strlen(str) >= TAM_ARRAY) {
      return NULL; /* Não vai caber no array */
   }

      /* Copia no array cada caractere do */
      /* string convertido por toupper()  */
   for (p = maiusculas; *p = toupper(*str); ++p, ++str) {
      ; /* Vazio */
   }
   return maiusculas;
}

int main(void)
{
   char *s1 = "Botafogo",
        *s2 = "Flamengo";

   printf("\n\"%s\" em maiusculas: \"%s\""
          "\n\"%s\" em maiusculas: \"%s\"\n",
          s1, StrEmMaiusculas(s1), s2, StrEmMaiusculas(s2));

   return 0;
}
```

Quando esse programa é compilado e executado, obtém-se o seguinte resultado no meio de saída padrão:

```
"Botafogo" em maiusculas: "BOTAFOGO"
"Flamengo" em maiusculas: "BOTAFOGO"
```

O último programa foi compilado com GCC 4.6.1 e executado no Windows. Se outro compilador fosse utilizado, os strings escritos em letras maiúsculas poderiam ser `"FLAMENGO"` e `"FLAMENGO"` (v. **Seção D.5.1**).

A primeira linha escrita pelo programa sob discussão está correta, mas a segunda linha, evidentemente, está errada. Isso ocorre porque a função **printf()** precisa, ao mesmo tempo, usar os dois strings retornados pela função `StrEmMaiusculas()`. Acontece, porém, que o string retornado por essa função é armazenado numa variável local de duração fixa, de modo que, quando é feita a segunda chamada dessa função, o conteúdo do primeiro string é sobrescrito.

Há duas soluções para o problema descrito acima. A mais simples delas consiste em substituir a chamada única de **printf()** no programa por duas chamadas dessa função, como mostrado abaixo:

```
printf( "\n\"%s\" em maiusculas: \"%s\"",
        s1, StrEmMaiusculas(s1) );
printf( "\n\"%s\" em maiusculas: \"%s\"\n",
        s2, StrEmMaiusculas(s2) );
```

Essa solução funciona porque, quando a chamada da função `StrEmMaiusculas()` sobrescrever o string retornado na primeira chamada, esse string não mais é necessário.

A segunda solução é um pouco mais complicada e requer alterar o código da função `StrEmMaiusculas()` para que ela use alocação dinâmica de memória em vez de uma variável local de duração fixa. Essa solução é adequada se for realmente necessário processar mais de um string retornado por essa função ao mesmo tempo. Uma nova implementação da função `StrEmMaiusculas()` usando alocação dinâmica de memória poderia ser:

```
char *StrEmMaiusculas(const char *str)
{
   char *maiusculas, *p;

      /* Tenta alocar um bloco e checa a alocação */
   if ( !(maiusculas = malloc(strlen(str) +1)) )
      return NULL; /* Não houve alocação */

      /* Copia no array cada caractere do */
      /* string convertido por toupper()  */
   for (p = maiusculas; *p = toupper(*str); ++p, ++str) {
      ; /* Vazio */
   }

   return maiusculas;
}
```

Exercício: Baseado no resultado apresentado pelo programa exibido no início desta seção, o que pode ser concluído com relação à ordem com que o programa avalia parâmetros numa chamada de função?

D.4.6 Tipo de Retorno Omitido

Não existe função com tipo de retorno indefinido, mas, dependendo do padrão seguido pelo compilador utilizado, o programador pode omiti-lo. Isto é, se o programador não

declarar explicitamente o tipo de retorno de uma função, tal compilador assumirá que esse tipo é **int** (e não **void** como alguns programadores podem imaginar). Portanto, essa função deve possuir pelo menos uma instrução **return** (consulte também a **Seção D.4.2**).

O melhor conselho que se pode apresentar com relação a eventuais problemas decorrentes de omissão de tipo de retorno é muito simples: nunca omita o tipo de retorno de uma função. Aliás, os padrões mais recentes da linguagem C (C99 e C11) não permitem mais que o tipo de retorno de uma função seja omitido (v. **Seção 5.4.1**).

D.4.7 Declaração de Parâmetros de Modo Abreviado

Parâmetros são semelhantes a variáveis em muitos aspectos, mas não é permitido declarar parâmetros de modo abreviado, como é permitido para variáveis. Por exemplo se uma função `F()` tem dois parâmetros `x` e `y` do tipo **int**, eles devem ser declarados no cabeçalho da função sem abreviação como:

```
void F(int x, int y)
```

e não como:

```
void F(int x, y)
```

Esse último cabeçalho contém um erro de sintaxe que qualquer compilador detecta (v. **Seção D.18.38**).

D.4.8 Chamada de Função antes de sua Definição

Chamar uma função antes de sua definição pode resultar em erro de compilação ou simplesmente numa mensagem de advertência. Nesse último caso, o compilador (p. ex., GCC) assume que a chamada é correta e, se esse não for o caso, o erro só será detectado na fase de ligação do programa.

A solução para um programa com duas ou três funções é simplesmente alterar a ordem de definições das funções. Para um programa contendo mais do que três funções, a solução mais prática é colocar alusões (v. **Seção 5.6**) de todas as funções, exceto **main()**, antes de suas definições.

D.4.9 Função FazTudo()

Um erro comum de estilo na escrita de uma função é o uso de instruções de entrada e saída quando a função não se destina a entrada ou saída de dados. Por exemplo:

```
void Fatorial(void)
{
   int n, i, fat = 1;

      /* Errado: 'n' deveria ser parâmetro de entrada */
   printf("\nDigite um valor inteiro positivo: ");
   n = LeInteiro();

   if (n < 0) {
      printf("Valor incorreto\n"); /* Errado */
   } else {
      for (i = 1; i <= n; ++i) {
         fat = fat*i;
      }

      printf("Fatorial: %d\n", fat); /* Errado */
   }
}
```

O nome atribuído à função acima sugere que ela deveria calcular o fatorial de um número, mas, de fato, ela efetua três tarefas distintas:

(1) Lê dados no meio de entrada padrão
(2) Calcula fatorial
(3) Escreve dados no meio de saída padrão

Uma definição correta de função que se propõe a calcular o fatorial de um número inteiro seria:

```
int Fatorial2(int n)
{
   int i, fat = 1;

   if (n < 0) {
      return -1; /* Erro de domínio */
   }

   for (i = 1; i <= n; ++i) {
      fat = fat*i;
   }

   return fat;
}
```

D.4.10 Nem Sempre a Biblioteca Padrão Respeita const

Considere o seguinte programa:

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

int main(void)
{
   const char *str = "Bola";
   char       *p = strchr(str, 'l');

   if (p) {
      *p = 't';
      printf("\np = %s\n", p);
   }

   return 0;
}
```

Quando esse programa é executado, ele é abortado e talvez um programador inexperiente tenha dificuldade para determinar a causa do término prematuro desse programa, já que o compilador não oferece nenhuma pista por meio de mensagem de advertência. Mas, examinado-se com atenção esse diminuto programa, a causa do erro pode ser facilmente apontada como sendo a instrução:

```
*p = 't';
```

Não precisa ter muita experiência para constatar que a instrução culpada pelo aborto do programa só poderia ser essa, já que ela é a única instrução que altera algum conteúdo de memória nesse programa. Agora, num programa de maior dimensão, encontrar a origem de um erro dessa natureza pode não ser tão fácil. Contudo, em qualquer caso, a tarefa de depuração seria facilitada se o compilador emitisse uma mensagem de advertência alertando o programador para um possível erro no programa. Então, por que razão o compilador não emite tal mensagem? A resposta trivial a essa questão é que, simplesmente, não há razão para o compilador agir assim. Isto é, o compilador não tem

bola de cristal que o permita saber que o endereço retornado por **strchr()** aponta para um string qualificado com **const**, o que pode causar dano a um programa.

A resposta mais complexa à questão anterior reside no fato de a função **strchr()** não respeitar completamente como constante o string que ela recebe como primeiro parâmetro, uma vez que ele é qualificado com **const** (v. **Seção 8.5.8**). Quer dizer, essa função recebe o endereço de um string que deve ser considerado constante e, de fato, a citada função não altera o string recebido como parâmetro. Porém, por outro lado, ela retorna um endereço que permite alterar o conteúdo do mesmo string.

Provavelmente, a função **strchr()** é implementada de modo equivalente à função `EncontraPrimeiroChar1()` apresentada a seguir:

```
char *EncontraPrimeiroChar1(const char *str, int c)
{
   while (1) {
      if (*str == c) {
           /* A conversão explícita a seguir evita que   */
            /* o compilador emita mensagem de advertência */
         return (char *) str;
      }

      if (!(*str++)) {
         break;
      }
   }

   return NULL;
}
```

A prática de qualificar com **const** conteúdos cujos endereços são retornados por funções é comum em C++, mas, infelizmente, o mesmo não ocorre (apesar de ser aceita) em C. Se a biblioteca padrão de C adotasse essa prática, a função **strchr()** seria provavelmente implementada como a função `EncontraPrimeiroChar2()` a seguir:

```
const char *EncontraPrimeiroChar2(const char *str, int c)
{
   while (1) {
      if (*str == c) {
         return str;
      }

      if (!(*str++)) {
         break;
      }
   }

   return NULL;
}
```

Note que o conteúdo apontado pelo endereço retornado por essa função é qualificado com **const** (v. **Seção 7.8**). Assim, se a função **strchr()** tivesse sido implementada dessa maneira, o compilador emitiria uma mensagem de advertência apontando como suspeita a instrução que causa o aborto do programa apresentado no início dessa seção (v. **Seção D.19.24**).

Existem outras funções da biblioteca padrão de C que podem causar erros se usadas inadvertidamente; dentre as quais, notabilizam-se as seguintes:

- **strrchr()** (v. **Seção 8.5.8**)
- **strstr()** (v. **Seção 8.5.7**)

- **strtod()** (v. **Seção 8.9.2**)

Diferentemente do que ocorre com as demais funções discutidas nesta seção, o que causa a violação de **const** no caso de **strtod()** é seu segundo parâmetro, e não o valor retornado por essa função.

D.4.11 Implementações Incorretas da Biblioteca Padrão

Algumas implementações de funções da biblioteca padrão de C não seguem as especificações do padrão ISO. E isso ocorre até mesmo com implementações bem notáveis, como algumas versões da biblioteca GNU glib que acompanha o compilador GCC. Algumas funções da biblioteca que acompanha o compilador Borland C++ 5.5 também podem ter sido implementadas em desacordo como o padrão ISO.

Quando utilizar uma função de alguma biblioteca cujo comportamento não coincide com o que é especificado pelo padrão ISO de C, consulte FAQs, fóruns de discussão e outros recursos disponíveis na internet para verificar se essa função foi realmente implementada de acordo com o referido padrão. A propósito, a documentação da biblioteca glib, usada pelo compilador GCC, reconhece e classifica como *broken* a implementação de uma dada função que não está em conformidade com o padrão ISO de C.

D.5 Chamadas Incorretas de Funções

D.5.1 Suposições sobre Ordem de Avaliação de Parâmetros

Como foi visto na **Seção 8.8**, a linguagem C não especifica em que ordem os parâmetros passados para uma função são avaliados. Portanto, não se deve fazer nenhuma suposição com relação a essa ordem, pois ela é dependente de implementação. Na prática, isso significa que, se o resultado esperado numa chamada de função depende da ordem na qual os parâmetros são avaliados, partes do programa precisam ser alteradas para eliminar essa dependência.

Cuidado especial deve ser tomado quando chamadas de funções contêm parâmetros que são alterados por operadores com efeitos colaterais. Por exemplo, o programa a seguir demonstra o efeito da ordem de avaliação de parâmetros numa chamada de função:

```
#include <stdio.h>

void ExibeInts(int x, int y)
{
   printf("\n\tPrimeiro valor: %d", x);
   printf("\n\tSegundo valor: %d", y);
}

int main(void)
{
   int  x = 10;

   ExibeInts( x, ++x );
   putchar('\n');

   return 0;
}
```

Quando esse programa é compilado com avaliação de parâmetros da direita para a esquerda, sua execução produz como resultado:

```
        Primeiro valor: 11
        Segundo valor: 11
```

Quando a avaliação de parâmetros é efetuada da esquerda para a direita, o resultado do programa é:

```
Primeiro valor: 10
Segundo valor: 11
```

Outra situação que requer atenção especial para que não ocorram problemas semelhantes ao do programa acima é quando o valor de um parâmetro é resultante de outra chamada de função como mostram os exemplos das **Seções 8.8** e **D.4.5**.

O que há de comum em todos esses casos é o fato de expressões estarem sendo usadas como parâmetros em chamadas de funções. Portanto, a solução para todos esses problemas é armazenar os resultados dessas expressões em variáveis auxiliares e usar essas variáveis na correspondente chamada de função. Por exemplo, no último programa, a solução seria substituir as duas primeiras linhas da função **main()** por:

```
int  x = 10, aux = ++x;

ExibeInts( x, aux );
```

Fazendo essas alterações, mesmo que a ordem de avaliação de parâmetros seja dependente de implementação, só há um resultado possível.

D.5.2 Parâmetros que Não Casam

Quando o compilador desconhece o protótipo de uma função, é possível chamá-la usando-se números e tipos de parâmetros incorretos sem que o compilador perceba. Então, uma violação das regras de casamento (v. **Seção 5.5**) numa chamada de tal função só poderia ser detectada pelo *linker*, o que tornaria a correção do erro mais difícil (v. **Seção 3.18.6**).

Funções com parâmetros variantes [p. ex., **scanf()** e **printf()**] também não permitem ao compilador checar se seus parâmetros são usados corretamente. Assim, essa categoria de funções requer cuidados especiais por parte do programador para garantir que os parâmetros passados para uma função realmente correspondem aos esperados. Algumas vezes, o compilador GCC é capaz de emitir mensagens de advertência quando o tipo de um parâmetro não corresponde ao respectivo especificador de formato em chamadas de **printf()** e **scanf()**, mas nem todo compilador faz isso.

D.5.3 Omissão de Teste de Condição de Exceção

Toda função que retorna um valor indicando ocorrência de uma condição de exceção (v. **Seção 11.5**) deve ter seu valor de retorno testado antes de ser usado. Funções notáveis nessa categoria são as funções de alocação dinâmica de memória (v. **Seção 12.2**), **strtok()** (v. **Seção 8.5.9**) e quase todas as funções do módulo stdio (v. **Seção 11.6**). Todas essas funções retornam valores que indicam quando não conseguem atingir seus objetivos. Portanto, por exemplo, em vez de usar:

```
char *p = malloc(strlen(str) + 1);

strcpy(p, str);
```

use:

```
char *p;

if ((p = malloc(strlen(str) + 1)) != NULL)
   strcpy(p, str);
```

E em vez de usar:

```
FILE *stream = fopen("UmArquivo.txt", "r");

fgets(array, sizeof(array), stream);
```

use:

```
FILE *stream;

if ((stream = fopen("UmArquivo.txt", "r")) != NULL)
   fgets(array, sizeof(array), stream);
```

Em resumo, nunca negligencie o valor retornado por uma função que possa indicar a ocorrência de uma condição de exceção.

D.5.4 Chamadas de Funções sem Parâmetros

Em algumas linguagens de programação (p. ex., Pascal, Basic), uma função sem parâmetros é chamada usando-se apenas seu nome. Em C, não é ilegal usar isoladamente o nome de uma função definida com ou sem parâmetros, mas um nome de função usado desse modo nunca representa uma chamada de função. Por exemplo:

```
void UmaFuncaoSemParametros(void)
{
   ...
}

...

UmaFuncaoSemParametros;
```

Essa última instrução é perfeitamente legal em C, mas desprovida de significado prático. Quer dizer, em C, a instrução:

```
UmaFuncaoSemParametros;
```

significa que se está avaliando o endereço da função `UmaFuncaoSemParametros()` e desprezando o resultado. Isso ocorre porque, em C, o nome de uma função representa seu endereço. Esse tópico não é discutido em nenhum capítulo deste livro, pois está além de seu escopo.

Concluindo, em C, uma função não pode ser chamada sem o uso de um par de parênteses.

D.5.5 Alusões de Funções sem Protótipos

Toda função deve ser aludida usando-se seu protótipo de modo a permitir que o compilador seja capaz de checar a validade de qualquer chamada. Por exemplo:

```
extern void Troca(int *, int *);
```

é melhor do que:

```
extern void Troca();
```

Também, para evitar uma interpretação indesejada, utilize **void** para indicar ausência de parâmetros em alusões (v. **Seção 5.6**).

D.6 Entrada e Saída

D.6.1 Uso Incorreto de scanf()

Um erro bastante comum entre iniciantes em programação em C é passar uma variável como parâmetro para **scanf()**, em vez de passar o endereço da própria variável. Como o padrão de C não requer que compiladores chequem a esperada correspondência entre

especificadores de formato e respectivos endereços de variáveis em chamadas da função **scanf()**, esse erro manifesta-se apenas durante a execução de um programa. Por exemplo:

```
int x;
...
scanf("%d", x); /* Deveria ser scanf("%d", &x); */
```

Apesar de parecer um erro ingênuo, suas consequências podem ser desastrosas. Isto é, esse erro pode acarretar em aborto do programa ou evoluir para um erro lógico. Bons compiladores de C (p. ex., GCC) apontam, por meio de mensagens de advertência, esse tipo de irregularidade.

Agora, observe o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int  i;
   char c;

   for (i = 0; i < 5; ++i) {
      printf(stderr, "Digite um inteiro: ");
      scanf("%d", &c);
      printf ("Valor de i: %d\n", i);
   }

   return 0;
}
```

Quando esse programa é compilado com GCC e executado no Windows, pode-se obter o seguinte:

```
Digite um inteiro: 1
Valor de i: 0
Digite um inteiro: 2
Valor de i: 0
Digite um inteiro: 3
Valor de i: 0
Digite um inteiro: 4
Valor de i: 0
Digite um inteiro: 5
Valor de i: 0
Digite um inteiro: 6
Valor de i: 0
Digite um inteiro: 7
Valor de i: 0
Digite um inteiro: [CTRL]+[C]
```

Como pode ser observado, quando o programa anterior é executado, ele incorre em repetição infinita. Isso ocorre devido ao uso de um parâmetro na função **scanf()** que não casa com o respectivo especificador de formato. Mais especificamente, a chamada de **scanf()** contém o especificador de formato **%d** que faz com que essa função espere ler caracteres que possam ser convertidos num valor do tipo **int**. Finalmente, esse valor deve ser armazenado numa variável do tipo **int**. Acontece, porém, que o segundo parâmetro dessa chamada de **scanf()** é o endereço de uma variável do tipo **char** e uma variável desse tipo não é capaz de conter todos os bytes de um valor do tipo **int** (v. **Seção 3.4**). Portanto, os bytes que sobrarem serão armazenados no espaço em memória que segue essa variável. Como, nesse caso específico, o compilador utilizado

decidiu armazenar a variável `i` que controla o laço **for** exatamente nessa posição, os bytes que sobram são armazenados no espaço reservado à variável `i`.

Exercício: Execute o último programa e, quando instado pelo programa, introduza um valor inteiro bem grande. Se o valor introduzido for suficientemente grande, você verá que o programa encerra imediatamente. Então, explique por que, no exemplo de execução apresentado, o programa entrou em repetição sem fim.

Outro erro comum (e grave) relacionado ao uso de **scanf()** é não checar o valor retornado por essa função para se certificar se algum valor foi realmente lido e armazenado (v. **Seção 10.9**).

Outros erros decorrentes de uso inadequado de **scanf()** são apresentado na **Seção 10.9**.

D.6.2 Uso Incorreto de printf()

O uso incorreto da função **printf()** para apresentação de um string pode conduzir a um sério e famoso problema de segurança, popularmente conhecido como **bug de printf**. Embora esteja além do escopo deste livro discutir esse problema em detalhes, pode-se resumi-lo informando que ele é decorrente do fato de essa função ser utilizada para escrita de strings como:

```
printf(str); /* Perigoso! */
```

em vez de:

```
printf("%s", str); /* Seguro */
```

Essa última instrução representa a única maneira segura de escrever um string usando **printf()**.

D.6.3 Especificadores de Formato Incorretos

Erros bastante frequentes relacionados a especificadores de formato das famílias de funções printf e scanf decorrem da confusão entre os especificadores usados para os tipos reais, como mostra a tabela a seguir.

	ESPECIFICADOR DE FORMATO PARA O TIPO...	
FAMÍLIA	**float**	**double**
printf	—	**%f**
scanf	**%f**	**%lf**

O tipo **float**, que aparece na tabela acima, não é utilizado no corpo principal deste livro. Ele é um tipo primitivo de ponto flutuante de menor largura do que o tipo **double**.

Outro erro bastante frequente é assumir que os especificadores de formato **%i** e **%d** usados com **scanf()** em leitura de valores inteiros são equivalentes. Esse erro, provavelmente, é derivado do fato de os especificadores **%i** e **%d** usados com **printf()** serem, de fato, equivalentes. Mas, esse não é o caso com **scanf()**, como mostra o exemplo a seguir:

```
#include <stdio.h>

int main(void)
{
   int valor, teste;

   printf("\n*** Usando '%%i' ***\n");
   printf("\nDigite um inteiro: ");
   teste = scanf("%i", &valor);
```

```
   if (teste) {
      printf("\nValor lido com '%%i': %d\n", valor);
   } else {
      printf("\nNenhum valor lido\n");
   }

   printf("\n\n*** Usando '%%d' ***\n");
   printf("\nDigite um inteiro: ");
   teste = scanf("%d", &valor);

   if (teste) {
      printf("\nValor lido com '%%d': %d\n", valor);
   } else {
      printf("\nNenhum valor lido\n");
   }

   return 0;
}
```

O exemplo de execução a seguir mostra que os especificadores **%i** e **%d** usados com **scanf()** *não são equivalentes*:

```
*** Usando '%i' ***

Digite um inteiro: 0xab

Valor lido com '%i': 171

*** Usando '%d' ***

Digite um inteiro: 0xab

Valor lido com '%d': 0
```

Observe nesse exemplo de execução que o usuário digitou o mesmo valor nos dois casos, mas eles foram lidos de modos diferentes. No caso de uso do especificador **%i**, o valor foi interpretado como um inteiro na base hexadecimal escrito de acordo com a sintaxe usada por C para constantes inteiras hexadecimais. Em C, uma constante inteira escrita na base hexadecimal deve começar com `0x` ou `0X` e utilizar os dígitos de `0` a `9` e as letras de `A` a `F` (ou `a` a `f`). Esse tópico não havia sido discutido neste livro porque está além do seu escopo.

Em resumo, o especificador **%i** permite a leitura de valores inteiros nas bases decimal, octal e hexadecimal, enquanto que o especificador **%d** permite apenas leitura de valores inteiros na base decimal. Como em programação de alto nível, raramente, é necessário ler valores inteiros em outra base que não seja a base decimal, evite o uso do especificador **%i** com **scanf()**.

No caso de **printf()**, os especificadores **%i** e **%d** são totalmente equivalentes, mas, para evitar confusão, é melhor também evitar o uso do especificador **%i** (mesmo porque ele é redundante).

Usos de especificadores de formato incorretos em funções das famílias scanf e printf não apenas são capazes de causar erros lógicos, como também podem acarretar erros de execução. Esse último caso ocorre principalmente quando o especificador **%s** é usado para escrita de expressões que não representam strings, como mostra o seguinte exemplo:

```
#include <stdio.h>

int main(void)
{
   int   x = 10;
   char *p = "Bola";

   printf("\nValor de x: %d\n", p);
   printf("String: %s\n", x);

   return 0;
}
```

Quando esse programa é executado, ele pode ser abortado devido ao uso do especificador **%s**, que deveria ser usado para escrita de strings, com a variável `x`, que é do tipo **int**.

D.6.4 Caracteres Remanescentes no Buffer Associado à Entrada Padrão

Diversos problemas que se manifestam num programa que efetua leitura em **stdin** (usualmente, associado ao teclado) são causados por caracteres remanescentes no buffer associado a esse meio de entrada. A **Seção 10.9** discute esses problemas e mostra como eles podem ser resolvidos. (Consulte também a **Seção D.14.2**.)

D.6.5 Não Existe Uso Correto para gets()

Certamente, **gets()** é (ou melhor, deixou de ser) a função mais condenada da biblioteca padrão de C. O problema com essa função é que ela não permite limitar o número de caracteres lidos no meio de entrada padrão, como foi discutido na **Seção 10.9.6**. Essa função foi abolida pelo padrão C11, mas, talvez, ainda faça parte da biblioteca que acompanha seu compilador.

O uso de **fgets()** é recomendado em substituição a **gets()**, mas lembre que, diferentemente de **gets()**, a função **fgets()** não descarta o caractere `'\n'` (novamente, v. **Seção 10.9.6**).

D.6.6 Informações que o Usuário Não Lê

Quando um programa produz uma extensa saída de dados na tela, um usuário experiente em console pode redirecioná-la para um arquivo, de forma que ele não será capaz de ler prompts apresentados pelo programa na tela. O programador pode prever esse comportamento do usuário e enviar prompts para **stderr** em vez de **stdout**. Por exemplo, em tal situação, em vez de escrever:

```
printf("Digite o valor de x: ");
```

o programador deve escrever:

```
fprintf(stderr, "Digite o valor de x: ");
```

Outra situação na qual o usuário pode não ser capaz de ler aquilo que o programa escreve na saída padrão é quando o programa é abortado sem que haja tempo de descarregar o buffer associado a **stdout**. Nesse caso, novamente, o programador deve usar **stderr** em vez de **stdout**, conforme foi exposto.

D.6.7 Modo de Abertura Incorreto

A escolha de um modo de abertura incorreto é uma das principais causas de erro em processamento de arquivos. Em particular, conforme foi mencionado na **Seção 10.4.3**,

modos de abertura usados para atualização constituem uma fonte de confusão entre programadores iniciantes. Portanto, se o resultado de um determinado processamento de arquivo é inesperado, certifique-se que está usando o modo de abertura mais adequado para a situação em foco.

Enfim, lembre-se que arquivos de texto só precisam ser abertos em modo de texto se a interpretação de quebra de linha (`'\n'`) se fizer necessária (v. **Seção 11.5**).

D.6.8 Uso Incorreto de EOF

A constante simbólica **EOF** é retornada por diversas funções de entrada ou saída declaradas em `<stdio.h>` para indicar condições de exceção, notadamente quando há tentativa de leitura além do final de um arquivo. Conforme foi afirmado na **Seção 11.2.1**, essa constante deve ser usada com **fgetc()** apenas para arquivos de texto. Mas, mesmo quando essa regra é seguida, muitos programadores não sabem como usar essa constante simbólica corretamente para determinar quando o final de um arquivo foi atingido. Por exemplo, suponha que o seguinte trecho de programa tenha sido escrito com o objetivo de ler cada caractere de um stream de texto e apresentá-lo na tela:

```
char  c;
FILE *stream;
...
while ((c = fgetc(stream)) != EOF)
   putchar(c);
```

O problema com esse trecho de programa é que ele não funciona se o arquivo lido contiver um byte com valor igual ao da constante **EOF**[1]. A solução para o problema apresentado pelo trecho de programa anterior é simples: basta definir a variável `c` como **int**, em vez de **char**.

Além disso, quando a função **fgetc()** retorna **EOF**, a interpretação desse valor é ambígua (v. **Seção 10.6**). Isto é, esse valor pode indicar que houve uma tentativa de leitura além do final do arquivo ou que ocorreu algum tipo de erro de leitura. Portanto, é mais aconselhável usar as funções **feof()** e **ferror()** para testar se houve tentativa de leitura além do final de um arquivo ou se ocorreu algum outro tipo de erro, como foi discutido na **Seção 10.6**.

D.6.9 Uso Incorreto de feof()

A função **feof()** retorna um valor diferente de zero após uma tentativa de leitura além do final do stream que ela recebe como parâmetro, mas algumas vezes é usada incorretamente. Um exemplo de uso incorreto da função **feof()** é o trecho de programa a seguir:

```
while ( !feof(streamEntrada) ) {
    fputc(fgetc(streamEntrada), streamSaida);
}
```

O problema com esse laço **while** é que a função **fputc()** escreverá indevidamente o valor retornado por **fgetc()** (i.e., o valor de **EOF**) quando essa última função tentar ler além do final do arquivo. Maiores detalhes sobre esse problema e sua solução são discutidos na **Seção 11.2.1**.

[1] Na realidade, a explicação precisa não é tão simples assim, mas a conclusão está correta. Apresentar todos os detalhes que levam a essa conclusão está além do escopo deste livro.

D.6.10 Não Testar Ocorrência de Erro com ferror()

Em processamento de arquivos, a melhor filosofia a adotar é a famigerada lei de Murphy: *o que pode dar errado dará*. E, em processamento de arquivos, muita coisa pode dar errada (p. ex., falha eletro-mecânica, alteração de permissão de acesso, arquivo corrompido, etc.). Portanto, após cada operação de leitura ou escrita em arquivo, use a função **ferror()** para testar se ocorreu algum erro. Se esse for o caso, o resultado da última operação deixa de ser confiável e o melhor a fazer é abortar graciosamente o programa e tentar determinar qual foi a causa do erro.

Como exemplo, considere o programa a seguir, que contém um erro de lógica que o impede de funcionar corretamente. Tudo o que esse programa pretende é ler o primeiro caractere do arquivo `Tudor.txt` (v. **Seção 11.6**) e exibi-lo na tela. No entanto, não é isso que acontece.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int   c;

      /* Abre o arquivo somente para leitura */
   stream = fopen("Tudor.txt", "r");

   if (!stream) {
      printf("\nNao foi possivel abrir arquivo\n");
      return 1;
   }
      /* Gera uma condição de erro    */
      /* tentando escrever no arquivo */
   fputc('A', stream);

   printf("\n\t*** Primeira tentativa de leitura ***\n");
   c = fgetc(stream);

      /* Verifica se o final de arquivo foi atingido */
   if (feof(stream)) {
         printf("\n\t>>> Final do arquivo atingido\n");
   } else if (ferror(stream)) { /* Ocorreu erro */
      printf("\n\t>>> Erro na leitura do arquivo\n");
   } else { /* Não ocorreu nenhuma irregularidade */
      printf( "\n\t>>> Primeiro caractere do arquivo: "
              "%c\n", c );
   }

      /* Verifica se há alguma sinalização */
      /* de erro ou de final de arquivo    */
   if(ferror(stream) || feof(stream)) {
         /* Zera indicadores de erro */
         /* e de final de arquivo    */
      clearerr(stream);
   } else { /* O caractere já foi lido e */
            /* não há mais nada a fazer  */
      fclose(stream);
      return 0;
   }
```

```
   printf("\n\t*** Segunda tentativa de leitura ***\n");

   c = fgetc(stream);

      /* Verifica se o final de arquivo foi atingido */
   if (feof(stream)) { /* Final de arquivo foi atingido */
         printf("\n\t>>> Final do arquivo atingido\n");
   } else if (ferror(stream)) { /* Ocorreu erro */
      printf("\n\t>>> Erro na leitura do arquivo\n");
   } else { /* Não ocorreu nenhuma irregularidade */
      printf( "\n\t>>> Primeiro caractere do arquivo: "
              "%c\n", c );
   }

   fclose(stream);

   return 0;
}
```

Quando compilado com GCC, esse programa produz como resultado:

```
*** Primeira tentativa de leitura ***

>>> Erro na leitura do arquivo

*** Segunda tentativa de leitura ***

>>> Primeiro caractere do arquivo: e
```

Em condições normais, o resultado apresentado pelo programa está incorreto porque não deve ter havido nenhum erro de leitura, como ele informa, nem o primeiro caractere do arquivo é `'e'`. Ou seja, o caractere correto é `'H'`, mas esse caractere foi lido na primeira tentativa de leitura que o programa informou, incorretamente, que não ocorreu.

O problema com o programa acima é que, quando é efetuada a leitura do primeiro caractere do arquivo, já havia um indicativo de erro provocado pela tentativa de escrita no arquivo, cujo modo de abertura não permite essa operação. Então, esse indicativo de erro permanece ativo até a chamada de **clearerr()**, que só ocorre após a leitura do primeiro caractere. Portanto, quando o programa testa se ocorreu erro de leitura, ele se depara com esse indicativo de erro e informa, incorretamente, que o erro ocorreu durante a leitura do caractere. Concluindo, a moral da história é que se deve verificar a ocorrência de erro e adotar as devidas providências após cada operação de leitura ou escrita. No caso específico do último programa, essa regra não foi seguida após a tentativa de escrita no arquivo.

Quando o último programa é compilado com Borland C++ 5.5, ele produz como resultado:

```
*** Primeira tentativa de leitura ***
>>> Erro na leitura do arquivo
*** Segunda tentativa de leitura ***
>>> Primeiro caractere do arquivo: H
```

Aparentemente, o último resultado apresentado está mais correto do que o resultado anterior porque, pelo menos, o programa escreve o caractere desejado. Além disso, se você refletir bem sobre esse resultado, concluirá que a mensagem que informa que ocorreu erro na primeira tentativa de leitura também está relativamente correta. Caso contrário, como o programa escreveria o primeiro caractere do arquivo na segunda tentativa? Se você raciocinou desse modo, está correto. Contudo, acontece que, nesse

caso, o erro no resultado do programa é duplo: tanto o programa quanto a função **fgetc()** da biblioteca usada com o compilador Borland estão equivocados. Quer dizer, o programa continua errado, mas o erro cometido por essa função compensou o erro do programa, de modo que o resultado parece correto.

O erro apresentado pelo programa acima quando compilado pelo compilador Borland é decorrente do fato de a função **fgetc()** que faz parte da biblioteca que acompanha esse compilador não ter sido implementada como preconiza o padrão ISO da linguagem C. Ou seja, essa função não lê um caractere (como deveria) num arquivo quando há um indicativo de erro para esse arquivo.

O último resultado apresentado pelo programa em discussão mostra que funções da biblioteca padrão de C podem ser implementadas em desacordo com o padrão da linguagem (v. discussão na **Seção D.4.11**).

D.6.11 Passagem de Leitura para Escrita e Vice-versa

Quando um arquivo é aberto em algum modo de atualização (v. **Seção 10.4.3**), deve-se atentar para o fato de não ser possível passar imediatamente de uma operação de leitura para escrita e vice-versa. Quer dizer:

- Entre uma operação de escrita e leitura (nessa ordem), deve haver uma chamada de **fflush()** (v. **Seção 10.7**), **fseek()** ou **rewind()** (v. **Seção 11.3.1**).
- Entre uma operação de leitura e escrita (nessa ordem), deve haver uma chamada de **fseek()** ou **rewind()** (v. **Seção 11.3.1**). Mas, essas funções não precisam ser chamadas se houver tentativa de leitura além do final do arquivo.

Para ilustrar o que foi exposto, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

      /* Cria um novo arquivo de texto */
      /* em modo de atualização        */
   if (!(stream = fopen("Teste", "w+"))) {
      printf("Arquivo nao foi aberto\n");
      return 1;
   }
      /* Escreve um string no arquivo */
   fprintf(stream, "AEIOU\n");

      /* Checa ocorrência de erro */
   if (ferror(stream)) {
      printf("\nErro 1\n");
      return 1;
   }
      /* Volta ao início do arquivo */
   rewind(stream);

      /* Lê um caractere e descarta-o */
   (void) fgetc(stream);

      /* Verifica se ocorreu erro de leitura */
   if (ferror(stream)) {
      printf("\nErro 2\n");
      return 1;
   }
```

```
   /* Tenta escrever um caractere no arquivo */
   fputc('X', stream);

   /* Verifica se ocorreu erro de escrita */
   if (ferror(stream)) {
      printf("\nErro 3\n");
      return 1;
   }

   fclose(stream); /* Fecha o arquivo */

   return 0;
}
```

No programa acima, a chamada de **fputc**():

```
fputc('X', stream);
```

não consegue escrever o caractere `'X'` no arquivo porque a operação antecedente no arquivo foi de leitura [i.e., a chamada de **fgetc**()] e, entre essa operação e a chamada de **fputc**() não ocorreu nenhuma chamada de função de posicionamento. Portanto, para corrigir o problema basta inserir logo antes da chamada de **fputc**(), a seguinte chamada de **fseek**():

```
fseek(stream, 0, SEEK_CUR);
```

Deve-se notar que essa chamada de **fseek**() não tenta mover o indicador de posição do arquivo, já que o deslocamento (segundo parâmetro) nessa chamada é zero. Ou seja, a referida chamada serve apenas para permitir a passagem de leitura para escrita.

D.6.12 Uso de fseek() com Arquivos de Texto

Como foi afirmado na **Seção 11.3.1**, as únicas chamadas portáveis da função **fseek**() para streams de texto são:

- `fseek(stream, 0, SEEK_CUR);`
- `fseek(stream, 0, SEEK_END);`
- `fseek(stream, 0, SEEK_SET);`
- `fseek(stream, ftell(stream), SEEK_SET);`

Qualquer outro formato de chamada de **fseek**() com arquivo de texto está fadado à ausência de portabilidade. Para constatar essa afirmação, considere o seguinte programa.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int   i;

   if(!(stream = fopen("Teste", "w+"))) {
      printf("Arquivo nao foi criado\n");
      return 1;
   }

   for (i = '0'; i <= '9'; ++i) {
      fprintf(stream, "%c\n", i);
   }

   if (fseek(stream, 3, SEEK_SET)) {
      printf("Indicador de posicao nao foi movido\n");
      return 1;
   }
```

```
    putchar(fgetc(stream));
    fclose(stream);
    return 0;
}
```

O último programa escreve, num stream de texto, os dígitos de `'0'` a `'9'`, sendo que cada dígito é escrito numa linha. Então, o programa move o indicador de posição para o byte de índice `3` e escreve na tela o caractere lido nessa posição. Quando esse programa é executado em sistemas da família Windows/DOS, ele escreve `1` na tela, enquanto que quando ele executado num sistema da família Unix, ele escreve quebra de linha. Portanto, esse programa não é portável.

D.6.13 fclose() Também Pode Falhar

Conforme foi visto na **Seção 10.5**, não testar o valor retornado por **fclose()** só é admissível quando se tenta fechar um arquivo logo antes do encerramento de um programa, já que o programa será, de qualquer modo, encerrado e não há mais nada que possa ser feito com relação a um eventual erro de fechamento. Mas, se o programa deve prosseguir com sua execução após o fechamento do arquivo, deve-se testar se essa operação foi bem sucedida e, se não foi o caso, o programa deve adotar as devidas precauções.

É importante notar ainda que fechar um arquivo mais de uma vez, sem nenhuma outra abertura intercalada dele pode acarretar num erro grave. Quer dizer, esse erro é equivalente a chamar a função **free()** repetidas vezes para liberar um mesmo bloco alocado dinamicamente, que, conforme foi visto na **Seção 12.2.3**, pode trazer graves consequências. Isso ocorre porque a estrutura do tipo **FILE** que contém informações sobre o arquivo é alocada dinamicamente e, quando a função **fclose()** é chamada, ela, provavelmente, chama **free()** para liberar o espaço ocupado por essa estrutura.

D.6.14 Função que Recebe Stream (Arquivo Aberto)

Uma função que recebe um parâmetro do tipo **FILE** *, que representa um stream aberto, deve proceder do seguinte modo:

- Se a função deve processar o arquivo sequencialmente a partir de seu início, antes de processá-lo, ela precisa chamar **rewind()** ou **fseek()** para deslocar o indicador de posição do arquivo para seu início.
- Se a função deve processar o arquivo sequencialmente a partir do ponto em que se encontra do indicador de posição do arquivo ou por meio de acesso direto, ela não deve chamar **rewind()** [ou efetuar chamada equivalente de **fseek()**]. Mas, para garantir que não ocorre falha devido à passagem de leitura para escrita ou vice-versa, é recomendado efetuar uma chamada de **fseek()** antes de qualquer operação de leitura ou escrita, mesmo que o deslocamento nessa chamada seja zero (v. **Seção D.6.11**).
- Em nenhuma circunstância, a função deve fechar o arquivo, pois a função que passa o stream como parâmetro real espera ainda tê-lo aberto após a chamada.

Por outro lado, uma função que passa um stream aberto para outra função deve contar com o fato de que ele continuará aberto ao final da execução da função chamada. Entretanto, a não ser que seja explícito na especificação da função chamada, a função que efetua a referida chamada não deve esperar que o indicador de posição do arquivo permaneça inalterado.

Uma função que contrarie as regras expostas acima poderá incorporar erros lógicos ao programa que a usa.

D.6.15 Função que Abre Arquivo

Uma função que abre um arquivo deve fechá-lo após seu processamento. Além disso, a chamada de **fclose()** utilizada no fechamento do arquivo deve ser testada (v. **Seção 10.5**).

Por outro lado, uma função que chama outra que abre arquivo não deve esperar que esse arquivo esteja aberto quando a função chamada retornar.

D.6.16 rewind() Pode Falhar, mas Não Informa

Conforme foi discutido na **Seção 11.4**, um programa que utiliza a chamada:

```
fseek(stream, 0, SEEK_SET)
```

é mais robusto do que aquele que chama **rewind()**:

```
rewind(stream)
```

Mas, trocar **rewind()** por **fseek()** só faz sentido se o valor retornado por esta última função for testado para verificar se o movimento do indicador de posição de arquivo foi realmente bem sucedida.

D.6.17 Função ungetc() Usada Repetidamente

O padrão de C não garante a inserção de dois ou mais caracteres num stream com o uso repetido da função **ungetc()** sem chamadas intercaladas de operações de leitura no mesmo stream, como mostram os dois trechos de programa a seguir:

Trecho de Programa 1

```
ungetc('A', stdin);  /* Inserção garantida      */
ungetc('B', stdin);  /* Pode não haver inserção */
```

Trecho de Programa 2

```
ungetc('A', stdin);  /* Inserção garantida */
getchar();
ungetc('B', stdin);  /* Inserção garantida */
```

D.6.18 Uso de fseek() e SEEK_END com Streams Binários

Um stream binário pode ser terminado com número não especificado de bytes nulos. Portanto, de acordo com o padrão ISO C11, uma chamada de **fseek()** tendo **SEEK_END** como terceiro parâmetro pode não fazer sentido, como mostra o seguinte exemplo:

```
int TamanhoDeArquivo(FILE *stream)
{
      /* Tenta mover o indicador de posição */
      /* para o final do arquivo           */
   if (fseek(stream, 0, SEEK_END)) {
      return -1; /* O movimento não foi possível */
   }

      /* Retorna o número de bytes do */
      /* início até o final do arquivo  */
   return ftell(stream);
}
```

Devido à observação apresentada no início desta seção, na função acima, a chamada de **fseek()**:

```
fseek(stream, 0, SEEK_END)
```

pode não fazer sentido quando o stream recebido pela função for binário. Assim, a função `TamanhoDeArquivo()` pode não funcionar em certas situações. Contudo, em sistemas operacionais que seguem o padrão Posix, essa função tem garantia de perfeito funcionamento.

D.6.19 Uso de ftell() com Streams de Texto

Em streams de texto, o resultado de uma chamada de **ftell()** deve ser usado apenas como deslocamento numa chamada de **fseek()**. Como exemplo de mau uso de **ftell()** considere novamente a função `TamanhoDeArquivo()`, apresentada como exemplo na **Seção D.6.18**, quando essa função recebe um stream de texto como parâmetro.

D.6.20 Arquivos de Texto

Conforme foi visto na **Seção 10.2**, um sistema operacional pode impor restrições a arquivos de texto. Portanto, para máxima portabilidade, um programa não deve criar um arquivo de texto:

- Vazio (i.e., sem nenhum caractere)
- Contendo espaços em branco antes de uma quebra de linha
- Contendo uma linha parcial (i.e., que omite `'\n'`) ao final de um arquivo
- Caracteres que não tenham representação gráfica, com exceção de `'\t'` e `'\n'`

D.7 Uso Incorreto de Diretivas de Pré-processamento

Existem diversos erros que podem estar associados ao uso indevido de diretivas de pré-processamento, mas aqui serão abordados apenas alguns problemas relativos à diretiva **#define**.

D.7.1 Definições Incorretas de Constantes Simbólicas

O erro mais frequentemente cometido numa definição de constante simbólica é o uso de ponto e vírgula para terminá-la (v. **Seção 3.15**). Algumas vezes, esse tipo de erro não causa danos a um programa, como por exemplo:

```
#include <stdio.h>

#define PI 3.14;

int main(void)
{
   double r = 2.54, c;

   c = 2*r*PI;

   printf("\nCircunferencia: %f\n", c);

   return 0;
}
```

Quando a constante `PI` é substituída pelo pré-processador, a instrução que a usa no programa acima resulta em:

```
c = 2*r*3.14;;
```

Portanto, o segundo ponto e vírgula é considerado instrução vazia e não tem nenhum efeito negativo sobre o programa, de maneira que ele funciona perfeitamente bem.

Agora, considere a mesma definição incorreta de constante simbólica do programa anterior utilizada num programa ligeiramente diferente:

```
#include <stdio.h>

#define PI 3.14;

int main(void)
{
   double r = 2.54, a;

   a = PI*r*r;

   printf("\nArea: %f\n", a);

   return 0;
}
```

Quando o pré-processador substitui a constante simbólica na instrução que a utiliza, o resultado obtido é:

```
a = 3.14;*r*r;
```

Ou seja, nesse caso, o ponto e vírgula adicional usado na definição da constante simbólica dividiu a referida instrução em duas. A primeira delas:

```
a = 3.14;
```

é perfeitamente legal. Mas a segunda instrução:

```
*r*r;
```

não faz nenhum sentido e causa erro de compilação. Agora, a mensagem de erro apresentada nesse último caso é uma fonte de tormento para programadores inexperientes por duas razões:

(1) A instrução indicada pelo compilador como origem da mensagem de erro é:

```
a = PI*r*r;
```

que aparenta estar absolutamente correta. E ela realmente estaria correta, se não fosse o fato de a constante simbólica que ela utiliza ter sido definida incorretamente. Mas, por que o compilador aponta essa instrução como incorreta e não a definição de constante simbólica que é a fonte real do erro? A resposta a essa questão pode lhe intrigar, mas o fato é que, quando o compilador traduz seu programa, ele sequer toma conhecimento de qualquer constante simbólica. A tarefa de substituição de constantes simbólicas fica a cargo do pré-processador, que cria um arquivo temporário com todas as constantes simbólicas já devidamente substituídas. Portanto, nesse caso, apesar de o compilador indicar a causa do erro numa instrução do arquivo-fonte original, não foi essa a instrução que ele encontrou no referido arquivo temporário. Ou seja, nesse arquivo a linha correspondente que ele encontrou foi:

```
a = 3.14;*r*r;
```

(2) A mensagem de erro apresentada pelo compilador não parece fazer sentido. Nesse caso específico, a mensagem apresentada pelo compilador GCC é:

```
error: invalid type argument of unary '*' (have 'double')
```

Traduzindo, o que a essa mensagem de erro quer dizer é que o operador de indireção, representado por `*`, foi usado indevidamente sobre um operando do tipo **double**. Entretanto, apesar de parecer esdrúxula, essa mensagem de erro está absolutamente correta e refere-se à segunda instrução resultante da mencionada divisão da instrução original:

```
*r*r;
```

Nessa instrução, o primeiro asterisco é interpretado como operador de indireção e, como seu operando `r` é do tipo **double**, a expressão resultante é obviamente inválida, pois o operador de indireção pode ser aplicado apenas sobre endereços e ponteiros.

Considerando o arrazoado acima, pode-se concluir que, quando um compilador aponta uma instrução como origem de erro e essa instrução inclui uma constante simbólica, deve-se verificar se a verdadeira origem do erro é a definição da constante.

D.7.2 Definições de Tipos Usando #define

Alguns programadores inexperientes podem confundir o uso de **typedef** com a diretiva **#define**, pois, em algumas situações, seus usos são, de fato, equivalentes. Por exemplo, a definição de tipo:

```
typedef int tInteiro;
```

poderia ser substituída pela seguinte diretiva:

```
#define tInteiro int
```

Entretanto, a diretiva **#define** é inadequada para substituir declarações de tipos mais complexas. Por exemplo, suponha que você deseja definir um tipo que represente um ponteiro para o tipo **char**. Então, você definiria corretamente esse tipo como:

```
typedef char *tPonteiroParaChar;
```

No entanto, a tentativa de definição desse tipo por meio da diretiva **#define**:

```
#define tPonteiroParaChar char *
```

é inapropriada.

Para entender melhor o problema, suponha que você deseja declarar dois ponteiros do tipo `tPonteiroParaChar`. Obviamente, com a primeira definição de tipo não haveria nenhum problema, mas se a segunda definição fosse usada, o pré-processador substituiria:

```
tPonteiroParaChar p1, p2;
```

por:

```
char *p1, p2;
```

ou seja, a primeira variável seria reconhecida como ponteiro para **char**, mas a segunda variável seria considerada do tipo **char** (e não do tipo **char** *).

Muito provavelmente, o resultado obtido com essa última definição de tipos não era aquele esperado pelo programador. Para evitar esse tipo de problema, use **typedef** para definir tipos e nunca use **#define** com esse propósito.

D.8 Os Lancinantes Erros Causados por Ponteiros

Erros causados pelo uso indevido de ponteiros são tipicamente creditados a:

- Tentativa de acesso a um conteúdo que não faz parte do espaço de memória reservado para o programa. Nesse caso, ou um ponteiro não foi iniciado (v. **Seção D.8.2**) ou ele passou a apontar para um endereço inválido após uma determinada operação (v. **Seção D.9.1**). Esse tipo de erro eventualmente manifesta-se por meio de erro de aborto de programa.
- Acesso a um bloco de memória válido, mas que não está exclusivamente associado ao ponteiro que efetua o acesso. Um ponteiro com essa característica

é considerado órfão e o conteúdo para onde ele aponta é o infame zumbi (v. **Seção D.8.1**).

- Indireção de ponteiro não iniciado (v. **Seção D.8.2**). Nesse caso, os sintomas são diversos e o programa pode comportar-se de uma maneira aparentemente aleatória. Isto é, ora ele funciona corretamente, ora ele não funciona corretamente, ora ele é abortado, etc. Em suma, descobrir a causa de erro num programa que apresenta esse tipo de comportamento é relativamente fácil, mas nem sempre é fácil determinar exatamente qual é a instrução que causa tal comportamento.
- Indireção de ponteiro nulo. Existem duas origens possíveis para essa causa de erro: (1) o ponteiro foi iniciado com **NULL** e o programador esqueceu de atribuir-lhe um endereço válido ou (2) o ponteiro recebeu **NULL** como valor retornado por uma função e o programador esqueceu de checar o valor retornado. Em qualquer caso, o programa será inexoravelmente abortado (v. **Seção D.8.3**).

As seções a seguir abordarão as causas de erros enumeradas acima e apontarão possíveis precauções que o programador pode adotar para prevenir-se contra essa categoria de erros.

D.8.1 Ponteiros Órfãos

Um **ponteiro órfão** é aquele que aponta para um bloco que, em princípio, é válido, mas que está livre para alocação a qualquer instante. Em outras palavras, um ponteiro órfão contém o endereço de um bloco que já foi liberado e, assim, ele aponta para um zumbi prestes a reencarnar (v. **Seção D.4.1**).

Os problemas de um programa contendo um ponteiro órfão começam quando ao ponteiro é aplicado o operador de indireção, como mostra o trecho de programa a seguir:

```
int *p1, *p1;

p1 = p2 = malloc(sizeof(int));
...
free(p1); /* Tanto p1 quanto p2 estão órfãos */
...
*p2 = 5; /* Problema à vista */
```

Nesse último fragmento de programa, deve parecer óbvio para qualquer programador de C que, após a chamada de **free()**, `p1` torna-se um ponteiro órfão. Mas, talvez, passe desapercebido o fato de `p2` também tornar-se um ponteiro órfão e é possível que o bloco para o qual `p2` apontava antes da chamada de **free()** tenha sido alocado novamente quando o operador de indireção é aplicado sobre esse ponteiro.

Uma medida capaz de prevenir (em parte) o aludido problema é, sempre que o bloco apontado por um ponteiro for liberado, atribuir **NULL** a esse ponteiro. Por exemplo:

```
free(p); /* Libera o bloco previamente alocado */
p = NULL; /* Invalida o ponteiro */
```

Se você acha tedioso escrever duas instruções (em vez de uma) cada vez que um ponteiro é liberado, use a seguinte definição de macro depois das diretivas **#include** de seu programa:

```
#define LIBERA(x) do { free(x); x = NULL; } while(0)
```

Macros com parâmetros não constituem um tópico deste livro e discutir como elas são definidas requer uma digressão que está além do escopo pretendido. Mas, usar (i.e., chamar) uma macro com parâmetros é tão fácil quanto chamar uma função, apesar de requer muita cautela por parte do programador porque numa chamada de macro não há verificação de ligação de parâmetros, como ocorre com chamadas de funções (v. **Seção 5.5**). Apenas, siga a recomendação e livre-se de parte da maldição que ronda o uso de ponteiros.

A macro `LIBERA()` chama a função **free()** e, em seguida, torna nulo o ponteiro usado como parâmetro. Desse modo, o objetivo dela é transformar um possível erro lógico num erro de execução, que é mais fácil de ser resolvido (v. **Seção 6.4**). Então, em vez de chamar `free(p)` para liberar o bloco apontado por `p` e que não é mais necessário use:

```
LIBERA(p);
```

Infelizmente, para o caso apresentado no exemplo mais recente, o uso dessa macro seria eficaz apenas se houvesse tentativa de uso do ponteiro `p1`, mas esse não seria o caso com o ponteiro `p2`. Por exemplo, a última instrução do trecho de programa a seguir:

```
int p1 = malloc(sizeof(int));
int p2 = p1;
...
LIBERA(p1);
*p1 = 10;
```

causaria o aborto imediato do programa, visto que a macro `LIBERA()`[2] atribui **NULL** a `p1`. Entretanto, se a instrução a seguir substituísse a última instrução do trecho de programa acima:

```
*p2 = 10;
```

ela seria considerada perfeitamente legal, mas poderia causar um erro lógico mais adiante no programa.

É importante salientar ainda que, quando se têm dois ponteiros apontando para um mesmo bloco alocado dinamicamente, como foi o caso do último exemplo, não se deve chamar a função **free()** [nem a macro `LIBERA()`] para cada ponteiro. Isto é, essa função (ou macro) deve ser chamada apenas para um deles. Aliás, é importante relembrar aqui que o ponteiro passado como parâmetro para a função **free()** [ou para a macro `LIBERA()`] deve estar apontando para o *início de um bloco alocado dinamicamente*.

Deve-se notar ainda que ponteiros órfãos e zumbis também ocorrem em situações que não envolvem retorno de funções nem alocação dinâmica de memória, como mostra o exemplo a seguir:

```
int *p = NULL;

...

{
   int x = 10; /* x é uma variável local a este  */
               /* bloco e tem duração automática */
   p = &x;
   ... /* Até aqui, tudo bem */
}
   /* A partir daqui, p está apontando para um zumbi */
...
printf("%d", *p); /* Só Deus sabe o que será escrito */
```

[2] A justificativa para o uso de um par de parênteses seguindo um nome de macro é evitar que macros com parâmetros sejam confundidas com constantes simbólicas, que são macros sem parâmetros.

No último trecho de programa, o problema surge a partir do fecha-chaves, pois, nesse ponto, a memória alocada para `x` é liberada, o que faz com que o ponteiro `p` passe a apontar para um zumbi.

D.8.2 Indireção de Ponteiro Não Iniciado

Ponteiro não iniciado é aquele usado antes de lhe ser atribuído um endereço válido. Ponteiros não iniciados podem causar os mesmos problemas de ponteiros órfãos (v. **Seção D.8.1**) e ainda podem causar aborto de programa. Por definição, um ponteiro órfão nunca causa aborto, porque ele sempre aponta para um endereço que faz parte do espaço de endereçamento do programa que o contém. Ponteiros órfãos podem ser responsabilizados apenas por erros lógicos.

Ponteiros não iniciados são relativamente mais fáceis de ser detectados, principalmente quando causam aborto. A indireção de um ponteiro não iniciado pode ocorrer explicitamente como no seguinte fragmento de programa:

```
int *p; /* Nenhuma iniciação de p */

... /* Trecho no qual a p não é atribuído valor */

*p = 0; /* Para onde p aponta? */
```

Além disso, a indireção de um ponteiro não iniciado pode ocorrer por meio de uma chamada de função como (v. **Seção 8.5.4**):

```
char *p; /* Nenhuma iniciação de p */

... /* Trecho no qual a p não é atribuído valor */

strcpy(p, "Este programa sera' abortado ou pior!");
```

O melhor que pode acontecer ao programa contendo a chamada de **strcpy()** acima é ele ser abortado, pois assim o programador terá alguma noção do que ocorreu de errado. Se o programa não for abortado, talvez o ponteiro `p` esteja apontando para algum endereço no espaço de memória reservado para a execução do programa. Nesse caso, o ponteiro alterará o conteúdo de um bloco que o programador não terá ideia de qual seja.

Existem vários sintomas que um programa pode apresentar em decorrência do uso de um ponteiro não iniciado:

- O programa pode ser abortado enquanto executa uma função que se tem certeza que não contém *bugs* (p. ex., uma função de biblioteca). Nesse caso, um ponteiro não iniciado pode ter tentado corromper o código da função.
- Uma função é chamada, mas nunca inicia sua execução ou nunca retorna. Nesse caso, pode ser que um ponteiro não iniciado tenha corrompido a pilha de execução.
- O programa ora funciona corretamente ora não funciona. Aqui, talvez, um ponteiro não iniciado esteja corrompendo aleatoriamente alguma variável do programa, dependendo do valor indefinido recebido pelo ponteiro.

Para evitar erros decorrentes de ponteiros não iniciados, o programador deve, antes de usar qualquer ponteiro, perguntar a si mesmo: *Para qual variável (bloco) esse ponteiro está apontando?* Por exemplo, antes de chamar a função **strcpy()** como no trecho de programa a seguir:

```
strcpy(str, "bola");
```

pergunte-se: *Para onde o ponteiro str está apontando?*

Se você não souber responder a essa pergunta, provavelmente, o ponteiro não deve estar apontando para um endereço válido.

D.8.3 Indireção de Ponteiro Nulo

Aplicar o operador de indireção a um ponteiro nulo causa o aborto do programa que executa tal operação e todo programador de C deve conhecer esse fato. O que alguns programadores não percebem é que algumas funções da biblioteca padrão aplicam esse operador a um ponteiro recebido como parâmetro sem antes testá-lo para saber se é nulo. Ou seja, é responsabilidade do programador certificar-se que não está passando um ponteiro nulo para uma função que não espera recebê-lo. Por exemplo, programa a seguir mostra o que ocorre quando a função **strcpy()** é chamada tendo como primeiro parâmetro um ponteiro nulo:

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

int main(void)
{
   char *p = NULL, ar[] = "Bola";

   ... /* Um longo trecho de programa que impede o    */
       /* programador de enxergar o desastre iminente */

   strcpy(p, ar);

   return 0;
}
```

Esse programa é abortado porque a função **strcpy()** aplica o operador de indireção em ambos os ponteiros recebidos como parâmetros sem antes testar se algum deles é nulo (v. **Seção 8.5.4**). Apesar de o programa ser abortado, essa situação ainda é melhor do aquela que ocorreria se o ponteiro não tivesse sido iniciado com **NULL**.

Uma situação comum na qual ocorre indireção de ponteiro nulo é aquela em que o valor atribuído ao ponteiro é recebido como retorno de uma função de biblioteca. Como exemplos mais comuns de funções que podem retornar **NULL** têm-se: **malloc()**, **calloc()**, **realloc()**, **fopen()**, **fgets()** e algumas funções declaradas em `<string.h>`, tais como **strtok()** e **strchr()**. A prevenção para esse tipo de erro é sempre, sem exceção, testar o valor retornado por uma função que pode retornar **NULL** antes que esse valor seja usado.

D.8.4 Ponteiro Incrementado Apontará para Outro Endereço

Poucas afirmações são tão óbvias mesmo para um programador iniciante em C do que o título desta seção. Infelizmente, até programadores mais experientes esquecem com relativa frequência essa evidente assertiva. Observe, como exemplo, o seguinte programa que contém uma função que tenta imitar a função **strcpy()** da biblioteca padrão (v. **Seção 8.5.4**):

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

char *CopiaString(char *destino, const char *origem)
{
   while (*destino++ = *origem++)
      ; /* Intencionalmente vazio */
```

```
      /* A cópia do string foi perfeita,    */
      /* mas o retorno é um desastre porque */
      /* o ponteiro 'destino' não está      */
      /* apontando para o string copiado no */
      /* array (primeiro parâmetro).        */
   return destino;
}

int main(void)
{
   char *p, str[30];

      /* Usar a função CopiaString() */
      /* como a seguir não apresenta */
      /* nenhum problema:            */
   CopiaString(str, "bola");

   printf("\nString copiado: \"%s\"", str);

      /* Usar a função CopiaString() */
      /* como a seguir representa    */
      /* um problema:                */
   p = CopiaString(str, "Problema");

   printf("\nString copiado: \"%s\"", p);

      /* O pior ocorre agora: o resultado da    */
      /* concatenação caberia confortavelmente  */
      /* no array str se p apontasse para o     */
      /* inicio do array, mas não esse é o caso */
   strcat(p, " com Tiranossauro Rex");

   printf("\nString concatenado: \"%s\"\n", p);

   return 0;
}
```

Quando executado, esse programa pode exibir o seguinte (ou algo parecido) na tela:

```
String copiado: "bola"
String copiado: "   Ò-+w¶©-wP$>"
String concatenado: "F+w"
```

A segunda e a terceira linhas escritas na tela podem ser diferentes em outras execuções do programa. O importante é notar que elas simplesmente não fazem nenhum sentido. Além disso, esse programa poderá ser abortado.

O problema que ocorre no programa acima é que a função **`CopiaString()`** incrementa o ponteiro recebido como primeiro parâmetro e, ao final, retorna esse ponteiro que, nesse instante, está apontando para o próximo byte além do final do string copiado. Uma correção para esse problema consiste em guardar o endereço inicial do array que recebe a cópia, como mostrado a seguir:

```
char *CopiaString2(char *destino, const char *origem)
{
   char *inicio = destino;

   while (*destino++ = *origem++)
      ; /* Intencionalmente vazio */

   return inicio;
}
```

D.8.5 void * é Tipo de Ponteiro Genérico, mas void ** não o É

A razão pela qual se deu preferência ao uso de uma macro em vez de uma função na **Seção D.8.1** foi que o uso de função seria muito mais complicado. Isto é, uma função parcialmente equivalente à macro `LIBERA` apresentada naquela seção seria definida como:

```
void Libera(void **p)
{
   free(*p);
   *p = NULL;
}
```

A definição da função `Libera()` é relativamente simples, mas seu uso (i.e., chamadas) não seria tão simples. Quer dizer, apesar de um parâmetro de o tipo **void*** ser considerado ponteiro genérico e, portanto não requerer conversão explícita (v. **Seção 12.3**), um parâmetro do tipo **void**** , como aquele da função `Libera()`, não é considerado ponteiro genérico. Assim, chamadas da função `Libera()` requerem conversões explícitas relativamente complicadas e propensas a erros, como mostrado a seguir:

```
int *ar;

ar = malloc(100*sizeof(int));

/* ... */ // Processa o array

   /* Conversão explícita é necessária aqui */
Libera((void **)&ar);
```

D.9 Arrays e Strings

D.9.1 Desrespeito aos Limites de Arrays

Uma alteração no espaço antes do índice inferior ou após o índice superior de um array pode corromper variáveis alocadas na pilha ou no heap, dependendo do fato de o array ter sido alocado estática ou dinamicamente, respectivamente. De qualquer modo, o resultado é o mesmo: corrupção de memória.

Desrespeitar os limites de um array ocorre com frequência em laços de repetição, como mostra exemplo apresentado na **Seção 7.3**.

É importante notar que apenas consultar um valor que está além das fronteiras de um array não é tão problemático, apesar de não fazer sentido. O problema maior ocorre quando se altera tal valor, como mostra o seguinte trecho de programa:

```
int *p, ar[5] = {1, 2, 3, 4, 5};

p = ar + 10; /* p aponta para além dos limites do array */

   /* A instrução a seguir não faz sentido, */
   /* mas não há maiores consequências para */
   /* o funcionamento do programa.          */
printf("Conteudo apontado por p = %d\n", *p);

   /* A instrução a seguir é catastrófica */
*p = 0;
```

Strings também são arrays e o mesmo cuidado deve ser tomado para que as fronteiras de um string não sejam ultrapassadas. Considere, por exemplo, a função

`RemoveBrancosFinais()` definida no programa a seguir e cujo objetivo é remover os espaços em branco finais de um string.

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

char *RemoveBrancosFinais(char *str)
{
      /* Faz p está apontar para o  */
      /* último caractere do string */
   char *p = strchr(str, '\0') - 1;
      /* Substitui cada caractere considerado */
      /* espaço em branco por '\0'            */
   while(isspace(*p))
      *p-- = '\0';

   return str;
}

int main(void)
{
   char s1[] = "bola      ";
   char s2[] = "          ";
   char s3[] = "";

   printf("String \"%s\" sem espacos finais: ", s1);
   printf("\"%s\"\n", RemoveBrancosFinais(s1) );

   printf("String \"%s\" sem espacos finais: ", s2);
   printf("\"%s\"\n", RemoveBrancosFinais(s2) );

      /* O programa será certamente    */
      /* abortado antes de chegar aqui */

   printf("String \"%s\" sem espacos finais: ", s3);
   printf("\"%s\"\n", RemoveBrancosFinais(s3) );

   return 0;
}
```

A função `RemoveBrancosFinais()` funciona bem quando o string recebido como parâmetro contém caracteres seguidos de espaços em branco [p. ex., o string `s1` definido na função **main()**], mas pode causar o aborto do programa se o string recebido contiver apenas espaços em branco [p. ex., o string `s2` definido em **main()**] ou for vazio [p. ex., o string `s3` definido em **main()**]. O problema nesses dois últimos casos é que a função `RemoveBrancosFinais()` poderá alterar bytes localizados antes do início do string recebido como parâmetro.

Exercício: Reescreva a função `RemoveBrancosFinais()` de tal modo que, se ela receber um string contendo apenas espaços em branco ou for vazio, ela retorne um string vazio.

Quando uma variável definida logo antes ou depois de um array assume valores inesperados, deve-se suspeitar de corrupção de memória causada por acesso indevido além dos limites do array. Por exemplo, considerando as definições de variáveis a seguir:

```
int     i;
double  ar[10];
double  d;
```

Se alguma das variáveis `i` e `d` definidas acima eventualmente assumir um valor estranho, acessos ao array `ar[]` que alteram seus elementos são possíveis suspeitos.

D.9.2 Strings Constantes Devem Ser Considerados Constantes

O programador deveria respeitar strings constantes assim como respeita (compulsoriamente) constantes numéricas. Infelizmente, muitas vezes, erros decorrentes do desrespeito a essa regra ocorrem por mero desconhecimento por parte do programador.

Não custa repetir o que foi afirmado na **Seção 8.3**: muitas implementações de C armazenam strings constantes num espaço em memória reservado apenas para leitura. Portanto, tentar alterar o conteúdo de um string constante pode causar aborto de programa.

Para permitir que o compilador detecte um provável erro antes que ele ocorra durante a execução do programa, acostume-se a declarar ponteiros para strings constantes usando **const**, como mostra o fragmento de programa a seguir:

```
const char *s = "Bola";
...
*s = 'C'; /* O compilador aponta o erro */
```

Nesse exemplo, o compilador apresenta uma mensagem de erro relacionada com a instrução que tenta alterar o conteúdo do string constante devido ao uso de **const** na definição da variável `s` (v. **Seção D.18.17**). Sem o uso de **const**, talvez o programa contendo o fragmento acima fosse abortado e o programador nem imaginaria por qual razão.

D.9.3 Comparação Incorreta de Strings

Considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   char *s1 = "bola";
   char s2[] = "bola";

   if (s1 == s2) {
      printf("\nOs strings \"%s\" e \"%s\" sao iguais\n",
             s1, s2);
   } else {
      printf("\nOs strings \"%s\" e \"%s\" sao diferentes\n",
             s1, s2);
   }

   return 0;
}
```

Quando esse programa é executado, ele produz como resultado:

```
Os strings "bola" e "bola" sao diferentes
```

O surpreendente resultado apresentado pelo programa acima é derivado da instrução **if**, que, em vez de comparar strings, na realidade, verifica se dois ponteiros apontam para o mesmo endereço. Portanto, se o objetivo for determinar se dois strings são iguais ou diferentes, deve-se usar a função **strcmp()** (v. **Seção 8.5.6**).

Se a comparação tiver como objetivo ordenar strings, a melhor escolha é a função **strcoll()** (v. **Seção 8.5.6**). Nesse caso, deve-se, antes de comparar os strings, efetuar a chamada `setlocale(LC_COLLATE, "")`, de modo que seja utilizada a localidade ora em vigor no sistema operacional usado (v. **Seção 8.5.6**).

D.9.4 Strings Constantes sem Acessibilidade

Quando se atribui o endereço de um string constante a um ponteiro, esse ponteiro se torna o único meio de acesso ao string. Se, subsequentemente, o mesmo ponteiro for associado a outro endereço, perde-se esse meio de acesso apesar de o string ainda encontrar-se armazenado em memória. Assim, tem-se um caso de escoamento de memória, pois o string continuará ocupando espaço em memória inutilmente, já que ele jamais poderá ser acessado novamente. Por exemplo:

```
char *p = "pitomba"; /* p torna-se o único meio de */
                     /* acesso ao string "pitomba" */
...
p = "umbu"; /* p deixa de apontar para "pitomba" */
            /* e passa a apontar para "umbu"     */
```

Nesse trecho de programa, após a segunda atribuição feita ao ponteiro `p`, o string `"pitomba"` continua armazenado em memória, mas não poderá mais ser acessado, gerando, assim, um desperdício de memória.

D.9.5 Uso de sizeof em vez de strlen()

O operador **sizeof** não deve ser usado em substituição a **strlen()** para calcular o comprimento de um string, exceto em circunstâncias especiais (v. final desta seção). Por exemplo:

```
char  *p = "bolada";
char   s1[80] = "bolada";
char   s2[] = "bolada";
int    t;

  /* Resulta no número de bytes... */
t = sizeof(p);  /* ...num ponteiro */
t = sizeof(s1); /* ...no array s1  */
t = sizeof(s2); /* ...no array s2  */
```

Em nenhuma dessas instruções, o resultado produzido é aquele esperado (i.e., `6`). Portanto, não use **sizeof** para calcular o número de caracteres visíveis num string.

Na realidade, existem situações nas quais é possível determinar o tamanho de um string usando o operador **sizeof**. Por exemplo:

```
char s[] = "bola";

printf("Tamanho do string: %d", sizeof(s) - 1);
```

Mas, esse é um caso especial e, em prol de um bom estilo de programação, é melhor ser esquecido.

D.9.6 Funções que Não Limitam o Número de Caracteres Escritos

Muitas funções declaradas no cabeçalho `<string.h>` [p. ex., **strcpy()**] não são capazes de limitar o número de caracteres escritos num array. Mas existem funções equivalentes declaradas nesse mesmo cabeçalho [p. ex., **strncpy()**] que possuem essa capacidade. Para evitar possível corrupção de memória, o programador deve dar preferência a funções de processamento de strings que limitam o número de caracteres escritos no array que armazena o resultado da operação. Por exemplo, considere o seguinte programa:

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

#define TAM_ARRAY  12

int main(void)
{
   int   x = 5;
   char  s[TAM_ARRAY];
   int   y = 5;
   strcpy(s, "um string que corrompe memoria");

   printf( "\nString apos chamada de strcpy():"
           "\n\t \"%s\"\n", s );

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

   return 0;
}
```

Quando o programa é executado, o resultado exibido na tela pode ser o seguinte (ou algo parecido):

```
String apos chamada de strcpy():
         "um string que corrompe memoria"

x = 1836020338, y = 1868767333
```

Note que, em virtude de corrupção de memória, os valores das variáveis `x` e `y` (que deveriam ser `5`) são apresentados como valores que não fazem sentido.

Agora, substituindo a chamada da função **strcpy()** no programa acima pela chamada de **strncpy**():

```
strncpy(s, "um string que corrompe memoria", TAM_ARRAY);
```

o resultado do programa passará a ser o seguinte:

```
String apos chamada de strcpy():
         "um string qu‡"

x = 5, y = 5
```

Certamente, o resultado da execução do programa após a alteração ainda não satisfaz o programador, mas, pelo menos, não houve corrupção de memória, como ocorreu com o programa antes da alteração. Se você não percebeu, o problema com o último resultado apresentado é que aparece um caractere bizarro após o último caractere copiado para o array. A origem desse estranho resultado é o fato de a função **printf()** tentar escrever um string armazenado no array `s[]`, mas, de fato, esse array não armazena um string (v. **Seção D.9.7**).

Algumas funções usadas em formatação em memória declaradas em `<stdio.h>` [p. ex., **sprintf()** vista na **Seção 10.10.1**] também apresentam a mesma possibilidade de corromper memória. Novamente, essas funções possuem equivalentes que limitam o número de caracteres escritos [p. ex., **snprintf**()].

D.9.7 Funções que Nem Sempre Produzem Strings

Algumas funções declaradas no cabeçalho `<string.h>` [p. ex., **strncpy**()] nem sempre incluem o caractere terminal de string nos resultados de suas operações, como mostra o último exemplo apresentado na **Seção D.9.6**. Portanto, para garantir que o resultado de

uma dessas operações seja realmente um string, pode ser necessário inserir um caractere `'\0'` ao final de um array resultante da operação, como mostra o programa a seguir:

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

#define TAM_ARRAY  12

int main(void)
{
   int   x = 5;
   char  s[TAM_ARRAY];
   int   y = 5;

   strncpy(s, "um string que corrompe memoria", TAM_ARRAY);

   s[TAM_ARRAY - 1] = '\0';

   printf( "\nString apos chamada de strcpy():"
           "\n\t \"%s\"\n", s );

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

   return 0;
}
```

Quando esse programa é executado ele escreve na tela:

```
String apos chamada de strcpy():
        "um string q"

x = 5, y = 5
```

Observe que o programa apresentado acima foi obtido a partir do programa discutido no final da **Seção D.9.6** com o acréscimo da instrução:

```
str[TAMANHO - 1] = '\0';
```

logo após a chamada de **strncpy**().

D.9.8 Strings Constantes versus Caracteres Constantes

A **Seção 8.4** discute alguns enganos frequentes cometidos por iniciantes em C quando lidam com caracteres isolados e strings. Alguns exemplos apresentados naquela seção são:

```
char *p;
...
*p = "A"; /* ERRADO: *p deveria receber um caractere e */
          /* não um ponteiro para um string constante  */

p = 'A'; /* ERRADO: p deveria receber um ponteiro */
         /* para char e não um valor do tipo char */
```

Confusões entre iniciações e atribuições envolvendo strings também são comuns entre iniciantes em C:

```
char *p = "String"; /* OK: Quem está recebendo */
                    /* um valor é p e não *p   */
...
*p = "String"; /* ERRADO: Quem está recebendo */
               /* um valor é *p e não p       */
```

Consulte a **Seção 8.4** para obter maiores detalhes sobre os exemplos apresentados acima.

D.9.9 Alocação de Espaço Insuficiente para Conter um String

Conforme foi enfatizado na **Seção 8.5**, um array usado para armazenar o resultado de uma operação com strings deve ter capacidade suficiente para conter todos os caracteres (inclusive `'\0'`) resultantes dessa operação. Infelizmente, muitas vezes, o programador esquece essa regra. Considere o seguinte fragmento de programa como exemplo:

```
char *p;
...
strcpy(p, "Bola");
```

Esse é o pior erro dentre todos dessa categoria, pois o local indefinido para onde o ponteiro `p` aponta em memória não pode armazenar um único caractere sequer. Mesmo assim, esse tipo de erro é muito comum entre iniciantes em programação.

Outro exemplo:

```
char *p = malloc(strlen("Bola"));
...
strcpy(p, "Bola");
```

Esse erro, aparentemente, não é tão mal quanto o anterior, porque, aqui, o programador esqueceu apenas de armazenar espaço para o caractere terminal de string (`'\0'`).

Qualquer um dos exemplos apresentados nesta seção pode causar aborto de programa, devido a corrupção de memória ou, pior, originar um erro lógico.

D.10 Alocação Dinâmica de Memória

D.10.1 Zumbis Também Assombram o Heap

Se uma função aloca memória dinamicamente e retorna um ponteiro para o espaço alocado, ela não deve chamar **free()** para liberar esse espaço antes de retornar. Ou seja, a responsabilidade de liberar o espaço alocado passa a ser da função que chama aquela que aloca espaço. Por exemplo:

```
char *Funcao1(const char *str)
{
   char *p = malloc(strlen(str) + 1);
   ... /* Executa algum processamento */
       /* usando o bloco alocado      */

   free(p); /* ERRADO: gestação de um zumbi */

   return p; /* Retorno do bebê zumbi */
}

char *Funcao2(const char *str)
{
   char *pAux = Funcao1("Bode");

   ...

   free(pAux); /* CORRETO: quem chama uma função    */
               /* que aloca espaço dinamicamente    */
               /* tem a responsabilidade de liberar */
               /* o espaço alocado                  */
}
```

D.10.2 Uso Incorreto de free()

Existem dois erros comuns decorrentes do uso incorreto da função **free()**:

(1) **Liberação múltipla de um bloco alocado dinamicamente**. Nesse caso, o ponteiro usado como parâmetro de **free()** já foi usado numa chamada anterior dessa função sem que lhe fosse atribuído outro endereço de bloco alocado dinamicamente entre as duas chamadas de **free()**. Por exemplo:

```
int *p = malloc(n*sizeof(int));

...

free(p);

... /* Neste trecho, p não recebe */
    /* o endereço de outro bloco  */

free(p); /* Liberação dupla */
```

No instante em que é feita a segunda chamada de **free()**, talvez, o bloco para onde ele apontava antes da primeira chamada, tenha sido realocado. A solução para esse problema é o uso da macro `LIBERA`, descrita na **Seção D.8.1**, em substituição a **free()**. Desse modo, a segunda chamada de **free()** não terá efeito porque o ponteiro `p` será nulo. Mas, o uso dessa macro será ineficaz se houver mais de um ponteiro apontando para um mesmo bloco alocado dinamicamente e a função **free()** for chamada com ambos os ponteiros, como, por exemplo:

```
int *p1 = malloc(sizeof(int)),
    *p2 = p1;
...
free(p1);
free(p2);
```

O problema com esse último trecho de programa é que tanto `p1` quanto `p2` apontam para o mesmo bloco alocado dinamicamente. Assim, a primeira chamada de **free()** está perfeitamente correta, mas a segunda chamada não está porque o bloco apontado pelo ponteiro `p2` já foi liberado na primeira chamada.

(2) **Liberação de bloco inválido**. A passagem de um ponteiro inválido para **free()** pode corromper o heap ou fazer o programa apresentar um comportamento errático. Nesse caso, um ponteiro inválido usado como parâmetro de **free()** é aquele que não aponta para início de um bloco alocado dinamicamente com **malloc()**, **calloc()** ou **realloc()**. Por exemplo:

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

int main(void)
{
   int *p1, *p2;

   p1 = malloc(sizeof(int));

   p2 = p1 + 116; /* 116 veio da cartola */

   free(p2);

   return 0;
}
```

Quando esse programa é executado, ele é abortado, porque o ponteiro `p2` usado como parâmetro na chamada de **free()** não aponta para um bloco válido.

D.10.3 realloc(): As Aparências Enganam

Conforme foi visto na **Seção 12.2.4**, o programador não deve jamais supor que **realloc**() retornará o mesmo endereço que lhe é passado como primeiro parâmetro numa chamada dessa função. Portanto, é sempre recomendado atribuir o endereço retornado por **realloc**() a um ponteiro diferente daquele usado como primeiro parâmetro. E essa recomendação é válida mesmo quando o tamanho desejado para um novo bloco [segundo parâmetro de **realloc**()] é menor do que ou igual ao tamanho do bloco apontado pelo primeiro parâmetro dessa função. Não seguir essa recomendação pode fazer com que o bloco apontado pelo primeiro parâmetro seja perdido.

Agora, às vezes, a recomendação acima parece não ser seguida, apesar de o raciocínio empregado pelo programador estar absolutamente correto. Por exemplo, considere o esboço de função apresentado a seguir que tenta redimensionar um array para que ele seja capaz de armazenar mais um elemento:

```
tElemento *InsereElemento(tElemento ar[], int *n, tElemento e)
{
   ar = realloc(ar, (*n + 1)*sizeof(tElemento));

   if (!ar) {
      return NULL;
   }

   /* A função prossegue usando 'ar' */

   return ar;
}
```

A função `InsereElemento()` parece não seguir as recomendações de uso de **realloc**(), porque ela usa o mesmo ponteiro que recebe o endereço retornado por **realloc**() como parâmetro dessa função. Mas, nesse caso, as aparências enganam o leitor desatento, porque o endereço que é alterado é aquele armazenado no parâmetro formal `ar`, e não aquele armazenado no parâmetro real que foi usado na chamada da função `InsereElemento()`. Além disso, como essa função retorna quando **realloc**() falha, ela não precisa mais do endereço do antigo bloco.

D.10.4 Lista Encadeada Não É Array

Os elementos (nós) de uma lista encadeada não são necessariamente adjacentes (e é exatamente por isso que cada nó armazena o endereço do próximo nó da lista). Portanto, eles não devem ser acessados por meio de índices ou incrementos de ponteiros como ocorre com arrays. Logo, se `p` for um ponteiro para um nó de uma lista encadeada, não caia na tentação de substituir uma instrução como:

```
p = p->proximo; /* Modo correto de acessar o próximo */
                /* elemento de uma lista encadeada   */
```

por:

```
p++; /* Acesso ERRADO ao próximo elemento da lista */
```

ou:

```
++p; /* Acesso ERRADO ao próximo elemento da lista */
```

ou:

```
p[2]; /* Modo ERRADO de acessar um elemento da lista */
```

D.11 Operações Inteiras

D.11.1 Detectando Overflow

Overflow ocorre quando o valor absoluto do resultado de uma operação é tão grande que não pode ser representado no tipo utilizado (i.e., o resultado não pode ser acomodado numa variável desse tipo).

A **Seção 4.12.7** discute uma maneira de detecção de overflow de números inteiros durante a avaliação de uma expressão. Mas, a técnica apresentada naquela seção só se aplica quando nenhum operando da expressão é negativo.

A maneira mais simples e portável de determinar se uma operação inteira resultará em overflow é testar se, considerando os operandos como inteiros sem sinal (**unsigned**), o resultado da operação é um valor negativo. Por exemplo, se `x` e `y` são duas variáveis do tipo **int**, a instrução **if** a seguir mostra como esse teste de overflow pode ser realizado numa operação de adição:

```
if ((int) ((unsigned)x + (unsigned)y) < 0)
   printf("Ocorrera' overflow na soma x + y");
```

O tipo **unsigned int** (ou, abreviadamente, **unsigned**) é um tipo inteiro primitivo que difere do tipo **int** pelo fato de não possuir bit de sinal (v. **Seção 7.5**). Isto é, todos os bits usados no armazenamento de uma variável do tipo **unsigned** são significativos.

O programa a seguir mostra como usar na prática a técnica apresentada:

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

void ExibeSoma(int x, int y)
{
   if ((int) ((unsigned)x + (unsigned)y) < 0) {
      printf("\nHavera' overflow na soma: %d + %d\n", x, y);
   } else {
      printf("\n%d + %d: %d\n", x, y, x + y);
   }
}

int main(void)
{
   ExibeSoma(INT_MAX, 1);
   ExibeSoma(INT_MAX, -1);

   return 0;
}
```

Quando executado, o último programa apresenta como resultado:

```
Havera' overflow na soma: 2147483647 + 1

2147483647 + -1: 2147483646
```

Além de adição, outras operações aritméticas também podem causar overflow. Até mesmo a operação que determina o valor absoluto de um número inteiro pode causar overflow, se o valor utilizado como operando for igual ao menor valor que seu tipo pode armazenar. Por exemplo, se função `AbsErrado()` a seguir:

```
int AbsErrado(int x)
{
   return x < 0 ? -x : x;
}
```

for chamada como:

```
AbsErrado(INT_MIN)
```

ela retornará o seguinte valor:

```
-2147483648
```

que representa o valor da constante **INT_MIN**, definida em `<limits.h>`, o que, obviamente, está incorreto (pois um valor absoluto não pode ser negativo).

O que causa overflow, nesse caso, é o fato de haver mais números inteiros negativos do que números inteiros positivos em representações binárias tipicamente utilizadas. Portanto, quando o menor número inteiro é convertido em número positivo, o resultado não pode ser contido numa variável do tipo inteiro em questão.

Uma forma de corrigir o erro apresentado pela função `AbsErrado()` seria redefini-la como:

```
int Abs(int x)
{
   if (x == INT_MIN) {
      printf("\nOcorrera' Overflow: x = %d\n", x);
      return x;
   }
   return x < 0 ? -x : x;
}
```

Essa função retorna os mesmos valores que a função `AbsErrado()`, mas, pelo menos, informa a ocorrência de overflow.

D.11.2 Cuidado com size_t!

O tipo **size_t** é o tipo do resultado da aplicação do operador **sizeof** e é utilizado por várias funções da biblioteca padrão de C. O problema com o uso desse tipo é que ele é um tipo sem sinal (v. **Seção D.11.1**) e o tipo inteiro majoritariamente usado neste livro é **int**, que é um tipo com sinal.

Como foi discutido na **Seção 7.5**, misturar tipos inteiros com sinal e sem sinal numa expressão que não seja de atribuição pode levar a resultados indigestos. Além disso, frequentemente, o programador esquece que um valor do tipo **size_t** nunca pode ser negativo, como mostra o exemplo a seguir:

```
void ExibeStrInvertido(const char *s)
{
   size_t tam;

      /* O laço a seguir nunca encerrará */
   for (tam = strlen(s) - 1; tam >= 0 ; --tam) {
      putchar(s[tam]);
   }
}
```

A função `ExibeStrInvertido()` tenta exibir o string recebido como parâmetro de trás para a frente e o raciocínio usado pelo programador parece correto. Acontece que, como o tipo da variável `tam` usada pela função é **size_t** e uma variável desse tipo nunca assume um valor negativo, o laço **for** usado pela função nunca encerrará. Para corrigir o erro apresentado pela referida função, basta substituir o tipo da variável `tam` por **int**.

D.11.3 Divisão Inteira por Zero

Conforme foi afirmado na **Seção 3.7.1**, divisão inteira ou resto de divisão inteira por zero é uma operação ilegal que causa aborto de qualquer programa que tenta realizar tal operação, como mostra o seguinte exemplo:

```
#include <stdio.h>

int main(void)
{
   int x = 5, y = 0, resto;

      /* A seguinte instrução causa o aborto do programa */
   resto = x%y;
      /* A seguinte instrução não será executada */
   printf("\nResto da divisao: %d\n", resto);

   return 0;
}
```

Para evitar que um programa seja eventualmente abortado, antes de qualquer operação de divisão ou resto de divisão, deve-se testar se o divisor da operação é zero, como mostra o programa a seguir:

```
#include <stdio.h>

int main(void)
{
   int x = 5, y = 0, resto;

   if (!y) {
      printf("\nNao e' possivel calcular resto "
             "de divisao por zero\n");
   } else {
      resto = x%y;
      printf("\nResto da divisao: %d\n", resto);
   }
   return 0;
}
```

D.12 Operações Reais

D.12.1 Arredondamentos

Uma dada operação real pode não resultar no valor esperado devido a arredondamento. Como exemplo de erro de arredondamento, considere o seguinte programa[3]:

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

int main(void)
{
   double x;

      /* O valor atribuído a x certamente será zero */
   x = pow(10, 40) +
       700         -
       pow(10, 40) +
       pow(10, 45) +
       500         -
       pow(10, 45);
```

[3] Esse programa é adaptado de um exemplo proposto em Dolenc, A. *et alii* (v. **Bibliografia**).

```
   printf("\nPrimeira expressao:\n\tValor de x = %f\n", x);

      /* Alterando-se a ordem das parcelas pode-se */
      /* obter o resultado correto. Por exemplo:   */

   x = pow(10, 40) -
       pow(10, 40) +
       pow(10, 45) -
       pow(10, 45) +
       700         +
       500;

   printf("\nSegunda expressao:\n\tValor de x = %f\n", x);

   return 0;
}
```

Quando executado, esse programa produz o seguinte resultado na tela:

```
Primeira expressao:
        Valor de x = 0.000000

Segunda expressao:
        Valor de x = 1200.00000
```

Analisando-se o resultado do último programa, nota-se que apesar de o erro absoluto ser grande (`1200`), na realidade, o erro relativo é bem pequeno. Isto é, a razão entre o erro absoluto e o maior valor envolvido nessas expressões é da ordem de 10^{-42}. Mesmo assim, nota-se que o simples arranjo de operandos numa expressão aritmética real é capaz de produzir resultados bem discrepantes.

D.12.2 Comparações

Devido ao modo aproximado com que são armazenados, números reais não devem ser comparados usando-se operadores relacionais. Em vez disso, deve-se testar se a diferença entre os dois números que se desejam comparar é suficientemente pequena para que esses números sejam considerados iguais, como foi visto na **Seção 5.14.6**. Contudo, em seu livro *The Art of Computer Programming Volume II* (v. **Bibliografia**), Donald Knuth recomenda que, para testar se dois valores reais são tão próximos que se pode considerar que eles são iguais, deve-se levar em consideração a diferença relativa entre os valores comparados, e não a diferença absoluta, como foi feito na implementação da função `ComparaDoubles()`, definida na **Seção 5.14.6**. Isso significa que, nessa função, a expressão:

```
fabs(d1 - d2) <= DELTA
```

deveria ser substituída por:

```
fabs(d1 - d2) <= DELTA*Max(fabs(d1), fabs(d2)
```

onde `Max()` é uma função que retorna o maior dentre dois valores do tipo **double** recebidos como parâmetros. Portanto, se um programa requer mais precisão do que aqueles apresentados neste livro, a função apresentada da **Seção 5.14.6** deve ser corrigida como se mostra a seguir.

```
double Max(double d1, double d2)
{
   return d1 > d2 ? d1 : d2;
}
```

```
int ComparaDoubles2(double d1, double d2)
{
   /* Verifica se o valor relativo da diferença entre  */
   /* os números é menor do que ou igual à precisão    */
   /* determinada pela constante DELTA. Se for o caso, */
   /* eles são considerados iguais. Caso contrário,    */
   /* verifica qual deles é menor ou maior.            */

   if (fabs(d1 - d2) <= DELTA*Max(fabs(d1), fabs(d2))) {
      return 0; /* d1 e d2 são considerados iguais */
   } else if (d1 < d2) {
      return -1; /* d1 é menor do que d2 */
   } else {
      return 1; /* d1 é maior do que d2 */
   }
}
```

D.12.3 Overflow e Underflow

O conceito de overflow para números reais é o mesmo que foi apresentado para números inteiros (v. **Seção D.11.1**), mas detecção de overflow para números reais é mais fácil. Quer dizer, ocorrência de overflow pode ser detectada após a execução de uma operação real, por exemplo, por meio de uma comparação com a constante **HUGE_VAL**, definida no cabeçalho `<math.h>`, como mostra o seguinte programa:

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

int main(void)
{
   double umDouble = 2*DBL_MAX;

   if (umDouble >= HUGE_VAL) {
      printf ("\nOcorreu overflow\n");
   }

   printf ("Valor de umDouble: %f\n", umDouble);

   return 0;
}
```

A constante simbólica **DBL_MAX**, que aparece no programa acima, é definida no cabeçalho `<float.h>` e representa o maior valor possível para o tipo **double**. Nesse programa, foi propositalmente atribuído à variável `umDouble` um valor que não pode ser contido numa variável do tipo **double**. Como resultado, essa variável recebeu o valor **HUGE_VAL** indicando a ocorrência de overflow.

Underflow ocorre quando o valor absoluto do resultado de uma operação real é tão pequeno que a implementação do tipo real utilizado não consegue representar tal valor e, portanto, ele é considerado zero. Esse problema não ocorre em operações inteiras.

Detecção de underflow é bem mais complicada do que detecção de overflow e discutir esse tópico está além do escopo deste livro.

D.12.4 Confundir Divisão Real com Divisão Inteira

Um erro frequente consiste em atribuir o resultado de uma divisão inteira a uma variável real com a expectativa de que essa divisão seja considerada real. Esse tipo de erro é óbvio quando uma atribuição ou iniciação é curta, como em:

```
double x = 2/3;
```

Nesse último exemplo, tão logo o valor de `x` seja exibido, descobre-se o erro. Mas, se a expressão for longa ou envolver variáveis inteiras, o erro poderá ser mais difícil de detectar. Por exemplo:

```
int x = 2, y = 3;
...
double z = 2.54 + 1.73 + x/y + 3.14 + 1.6;
```

Nessa última expressão, é mais difícil detectar o erro porque a expressão `x/y`, que não contribui em nada para o resultado (pois resulta em zero), aparece camuflada em meio aos demais termos da expressão.

D.12.5 Divisão Real por Zero

Ao contrário do que ocorre com tentativa de divisão inteira por zero, uma divisão real por zero não causa aborto de programa. Entretanto, o resultado dessa última operação não produz um número real, como mostra o seguinte exemplo:

```
#include <stdio.h>

int main(void)
{
   double x = 5, y = 0, divisao;

   divisao = x/y; /* Programa NÃO será abortado, mas o */
                  /* resultado não será um número real */

   printf("\nQuociente: %f\n", divisao);

   return 0;
}
```

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

```
Quociente: inf
```

A saída do programa acima indica que o resultado da divisão de um valor do tipo **double** por zero não é um valor do tipo **double**. (O valor `inf` escrito na tela representa o conceito de infinito em C.)

Concluindo, para evitar que um programa apresente um resultado inconveniente devido a uma divisão real por zero, antes de qualquer operação de divisão, deve-se testar se o divisor da operação é zero, como mostra o exemplo apresentado na **Seção D.11.3**.

D.12.6 Não Há Representação Fidedigna de Números Reais

Conforme foi mostrado na **Seção 6.5**, há uma infinidade (literalmente) de números reais que não podem ser representados em computador. Nem mesmo números tão simples em base decimal quanto `0.1` possuem representação binária exata.

Ignorar esse conhecimento pode acarretar sérios desastres (literalmente, de novo). Por exemplo, durante a Guerra do Golfo em 1991, um bug no sistema de mísseis Patriot foi responsável (indiretamente) pela morte de 28 soldados americanos. E esse bug foi causado por um erro de truncamento do funesto número `0.1`.

A **Seção 6.5** lida com problemas de representação de ponto flutuante num nível conveniente para um livro introdutório de programação. Se você estiver desenvolvendo um programa (p. ex., um sistema de mísseis) que requer mais precisão no tratamento de números de ponto flutuante procure um texto especializado sobre o assunto[4].

[4] Um bom começo é o excelente artigo: *What Every Computer Scientist Should Know About Floating-Point Arithmetic* de David Goldberg (v. **Bibliografia**).

D.13 Comentários

D.13.1 Comentários Não Terminados

Quando o programador esquece de fechar um comentário apropriadamente, trechos de seu programa podem ser *engolidos* pelo comentário. Por exemplo:

```
a = b; /* Este comentário não termina onde deveria e...
c = d; /* a instrução c = d está dentro do comentário   */
```

Erros decorrentes de uso incorreto de comentários são comuns, mas muito fáceis de encontrar se for utilizado um editor de programas razoável; i.e., um editor que empregue uma convenção gráfica que diferencie comentários de outras construções de C. Por exemplo, o editor do ambiente CodeBlocks apresenta comentários com coloração destacada, o que facilita a identificação visual deles.

D.13.2 Comentários Mal Posicionados

Considere o seguinte fragmento de programa:

```
int a, b, *p;
...
a = b/*p; /* Divide b por *p */
```

O problema com essa última instrução é que a combinação de caracteres `/*` entre `b` e `p` é interpretada como abertura de comentário. Nesse caso, ocorre um erro de compilação, mas poderia ter sido convertido num erro lógico se o programador tivesse colocado ponto e vírgula ao final da última linha:

```
int a, b, *p;
...
a = b/*p; /* Divide b por *p */;
```

Novamente, como foi dito na **Seção D.13.1**, usando-se um bom editor de programas torna-se relativamente fácil identificar esse tipo de erro visualmente. Por exemplo, usando o editor de programas do ambiente CodeBlocks, a última linha do primeiro trecho de programa acima apareceria com coloração destacada, o que facilitaria a identificação do erro.

A correção do erro acima é trivial: basta colocar `*p` entre parênteses ou usar um espaço em branco entre `/` e `*` antes de `p`.

D.13.3 Comentários Aninhados

Não se devem aninhar comentários de um mesmo tipo, pois nem todo compilador aceita esse tipo de aninho. Levando isso em consideração, é recomendado usar apenas um tipo de comentário em cada programa, pois, se for necessário remover temporariamente um trecho de programa contendo comentários, utiliza-se o outro tipo de comentário disponível (v. **Seção 6.4.3**).

D.14 Erros de Portabilidade

Um erro de portabilidade pode não impedir um programa de ser compilado normalmente numa implementação de C específica, mas não há garantia de que ele será compilado noutra implementação. Existem três erros de portabilidade tão comuns que merecem destaque especial neste apêndice. Esses erros serão descritos nas subseções a seguir.

D.14.1 void main()

Conforme foi discutido na **Seção 8.6**, a função **main()** pode ter dois protótipos diferentes:

```
int main(void)
```

```
int main(int argc, char *argv[])
```

Em nenhum desses protótipos da função **main()**, o padrão ISO de C especifica que o tipo de retorno dessa função seja diferente de **int**. Portanto, apesar de muitas implementações permitirem o uso de **void** como tipo de retorno de **main()**, essa bizarrice é uma extensão da linguagem C e, assim, não é portável.

D.14.2 fflush(stdin)

A função **fflush()** serve para descarregar apenas buffers associados a streams de saída (v. **Seção 10.7**). A biblioteca padrão de C não provê nenhuma função para descarregar buffers associados a streams de entrada (como é o caso de **stdin**), mas é fácil implementar uma função para descarga do buffer associado a um stream de entrada, como é o caso da função `LimpaBuffer()`, apresentada na **Seção 10.9.4**.

D.14.3 conio.h

O cabeçalho `conio.h` é uma extensão bastante comum entre implementações de C (v. **Seção 10.14**). Nesse cabeçalho, são declaradas funções que realizam operações de entrada e saída que não são contempladas pela biblioteca padrão de C. Provavelmente, as três funções declaradas nesse cabeçalho mais utilizadas são `getche()`, `getch()` e `clrscr()`, que foram descritas na **Seção 10.14**.

Nem as funções declaradas em `conio.h` nem mesmo esse cabeçalho são portáveis. Isto é, apesar de ser comum, esse cabeçalho pode não existir numa dada implementação de C.

D.14.4 system() É Portável, mas seu Parâmetro Não o É

A função **system()** faz parte do módulo stdlib da biblioteca padrão de C e, portanto, é portável. Contudo, o parâmetro que essa função recebe representa um comando do sistema operacional usado como hospedeiro para o programa que usa a referida função. Assim, chamadas de **system()** raramente apresentam portabilidade. Por exemplo, a chamada:

```
system("cls");
```

funciona em sistemas operacionais da família Windows/DOS e tem como efeito apagar o conteúdo da tela. Mas, essa mesma chamada de **system()** não tem nenhum efeito se o programa em questão for executado num sistema da família Unix.

A abordagem que se propõe aqui para facilitar a portabilidade é confinar chamadas de **system()** em funções definidas pelo programador ou ter seus parâmetros definidos como constantes simbólicas. Por exemplo, uma função para limpeza de tela poderia ser definida como no programa a seguir:

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

```
int LimpaTela(void)
{
   return system("cls");
}

int main(void)
{
   (void) LimpaTela(); /* Despreza o retorno da função */

   return 0;
}
```

No entanto, a alternativa a seguir é mais eficiente e também promove a portabilidade:

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

#define COMANDO_LIMPA_TELA  "cls"

int main(void)
{
   (void) system(COMANDO_LIMPA_TELA);

   return 0;
}
```

D.15 Escopo

D.15.1 Ocultação de Variáveis e Parâmetros

Uma variável com escopo de bloco ou parâmetro pode ser ocultado por uma variável definida num bloco aninhado (v. **Seção 5.13.5**). Por exemplo:

```
int F(int x)
{
   {
      double x; /* Oculta o parâmetro x */
      ...
   }
}
```

De modo semelhante, variáveis com escopo de arquivo podem ser ocultadas por variáveis com escopo de bloco e variáveis globais podem ser ocultadas por variáveis com escopo de arquivo ou bloco (v. **Seção 5.13**).

D.15.2 Colisões de Identificadores

Nunca utilize o mesmo identificador para rotular simultaneamente uma variável e uma função que fazem parte do mesmo escopo, pois o não cumprimento dessa regra pode acarretar erros de ligação difíceis de ser resolvidos.

De um modo geral, erros causados por colisões de identificadores não são fáceis de detectar. Portanto, evite-os seguindo recomendações de estilo apresentadas ao longo deste livro.

D.15.3 Variáveis Definidas em Laços de Repetição

Conforme foi discutido na **Seção 5.13**, variáveis definidas em blocos têm escopo de bloco, mas, muitas vezes, o programador parece esquecer esse fato quando lida com laços de repetição, como mostra o exemplo abaixo:

```
int SomaAteN(int n)
{
   while (--n) {
      int soma = 0;

      soma = soma + n;
   }
   return soma;
}
```

Se você tentar compilar um programa contendo a função `SomaAteN()`, verá que o compilador apresenta a mensagem de erro (v. **Seção D.18.7**):

```
'soma' undeclared (first use in this function)
```

Essa mensagem de erro informa que a variável `soma` não foi declarada na função `SomaAteN()`, o que, claramente, não é o caso. Ou seja, o que ocorre na realidade é que essa variável está sendo utilizada num local onde ela não é válida.

Outra situação na qual o programador frequentemente esquece as regras de escopo é quando uma variável é definida na primeira expressão de um laço **for**, como no seguinte exemplo:

```
int SomaAteN2(int n)
{
   for (int i =1, soma = 0; i <= n; ++i) {
      soma = soma + i;
   }

   return soma;
}
```

Nesse caso, o escopo da variável `soma` é restrito ao laço **for**. Quando um programa contendo a função `SomaAteN2()` é compilado, ele apresenta a mesma mensagem de erro discutida antes.

D.16 Iniciação

D.16.1 Iniciação Não É o Mesmo que Atribuição

Existem vários exemplos neste livro que mostram que, apesar das aparências, iniciação e atribuição são operações distintas em C. Portanto, nem toda operação válida em atribuição também é válida em iniciação e vice-versa. A **Tabela D-1** resume as principais diferenças entre iniciação e atribuição.

Operação	Iniciação	Atribuição
Armazenar numa variável de duração fixa o resultado de uma expressão que envolve variáveis	Ilegal (mas há uma exceção apresentada na **Seção 5.12.3**)	Permitido
Armazenar um valor numa variável definida com **const**	Recomendável	Ilegal
Armazenar um valor num parâmetro	Ilegal	Permitido
Armazenar um string num array usando a notação de string constante	Permitido	Impossível

Tabela D–1: Iniciação versus Atribuição

Operação	Iniciação	Atribuição
Armazenar simultaneamente mais de um valor em elementos de um array	Permitido	Impossível
O símbolo = representa um operador?	Não	Sim
A operação resulta num valor?	Não	Obviamente
Armazenamento de valor numa variável pode ocorrer implicitamente?	Sim	Não faz sentido
Pode ocorrer fora de uma função?	Sim	Nunca
Armazenar simultaneamente mais de um valor em campos de uma estrutura	Permitido	Impossível, a não ser em atribuição entre estruturas

Tabela D–1: Iniciação versus Atribuição

D.16.2 Iniciação de Variáveis em Laços de Repetição

Se uma variável de duração automática for definida no corpo de um laço de repetição, a iniciação ocorrerá sempre que o corpo do laço for executado, mas, às vezes, o programador esquece esse fato. Considere, como exemplo, o seguinte programa que foi construído para calcular a soma compreendida entre `1` e o número inteiro positivo introduzido pelo usuário:

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

int SomaAteN(int n)
{
   if (n <= 0) {
      return 0;
   }

   while (1) {
      int soma = 0;

      soma = soma + n--;

      if (!n) {
         return soma;
      }
   }
}

int main(void)
{
   int n;

   printf("\nDigite um inteiro positivo: ");
   if ((n = LeInteiro()) <= 0) {
      return 1;
   }

   printf( "\nSoma dos inteiros entre 1 e %d: %d\n",
           n, SomaAteN(n) );

   return 0;
}
```

Esse programa é compilado sem problemas, mas, quando executado, ele informa que a soma que ele calcula é sempre `1`, independentemente do valor inteiro positivo introduzido pelo usuário.

O erro lógico encontrado no programa acima reside na função `SomaAteN()`. Mais precisamente, a variável `soma` definida no corpo do laço **while** é iniciada com zero a cada passagem pelo corpo desse laço. Assim, na última passagem pelo corpo do laço (quando `n` vale `1`), o valor corrente da variável `soma` (que é zero) é somado ao valor de `n` e o resultado é retornado. Assim, a função `SomaAteN()` sempre retorna `1`. A solução para esse erro consiste em definir e iniciar a variável `soma` antes do laço **while**.

É tentador imaginar que o uso de **static** na definição da variável `soma` resolve o problema dessa função. Mas esse não é o caso, pois se a duração dessa variável se tornar fixa, ela reterá o último valor que lhe for atribuído. Assim, se essa função for chamada várias vezes num programa tendo valores positivos como parâmetros, o único resultado correto que ela retornará será o primeiro.

D.17 Outros Problemas Comuns

D.17.1 Uso de Variáveis Não Iniciadas

Um erro bastante frequente é o uso do conteúdo de uma variável antes de ela assumir um valor conhecido; i.e., antes de ela ser iniciada. Uma variável também é considerada não iniciada quando a ela é atribuído o valor de uma variável não iniciada ou de uma expressão contendo uma variável não iniciada. Por exemplo, o programa a seguir mostra que atribuir a uma variável um valor proveniente de variáveis não iniciadas é equivalente a não iniciar a variável:

```
#include <stdio.h>

int main(void)
{
   int x, y;          /* x e y não são iniciadas */
   int soma = x + y;  /* Logo, soma também não é iniciada */

   printf("\nDigite dois numeros para somar: ");
   scanf("%d %d", &x, &y);

   printf("\nA soma dos numeros e' igual a: %d\n", soma);

   return 0;
}
```

Quando esse programa é executado, o resultado provavelmente não fará sentido:

```
Digite dois numeros para somar: 5 7

A soma dos numeros e' igual a: 30
```

Lembre-se que variáveis de duração fixa são iniciadas implicitamente com zero e que variáveis de duração automática não são iniciadas implicitamente (v. **Seção 5.12.3**). Assim, quando uma variável assume um valor aparentemente aleatório, ela deve ser checada cuidadosamente para assegurar que foi devidamente iniciada.

D.17.2 Conversões Inadequadas de Tipos

Conforme foi discutido nas **Seções 3.10** e **5.5**, há cinco situações nas quais podem ocorrer conversões automáticas em C:

(1) Numa atribuição ou iniciação de variável envolvendo tipos diferentes.
(2) Numa expressão (excluindo atribuição) na qual tipos diferentes são misturados.
(3) Quando há necessidade de conversão de alargamento (v. **Seção 3.10**).
(4) Numa passagem de parâmetro quando o tipo do parâmetro real não coincide com o tipo do parâmetro formal correspondente e a conversão é possível.
(5) Num retorno de função quando o tipo do valor retornado não coincide com o tipo de retorno declarado no cabeçalho da função e a conversão é possível.

Com exceção de conversão de alargamento, em qualquer das demais situações enumeradas podem ocorrer erros em virtude de conversões automáticas. A **Seção 3.10** apresenta alguns exemplos de erros de programação decorrentes de misturas de tipos e conversões automáticas.

Resumindo, o melhor conselho que se pode oferecer aqui é que se devem evitar, sempre que possível, conversões automáticas, tomando-se o cuidado de nunca misturar valores de tipos diferentes. Mas, se for inevitável misturar tipos numa expressão, deve-se avaliar cuidadosamente as consequências das conversões implícitas que poderão ocorrer.

D.17.3 Identificadores Trocados

Erros decorrentes da troca inadvertida de um identificador por outro podem ocorrer quando se utilizam identificadores muito parecidos uns com os outros ou quando os identificadores são desprovidos de representatividade (v. **Seção 4.10.4**). A adoção de um estilo de programação que promova a representatividade de identificadores minimiza a possibilidade de ocorrência desse tipo de erro, como mostra o exemplo seguir:

```
int x, y; /* É fácil confundir x com y */

int idade, matricula; /* É muito difícil confundir */
                      /* essas duas variáveis      */
```

D.17.4 Acesso a Campo Inválido de União

Como foi visto na **Seção 9.6**, os campos de uma união são mutuamente exclusivos. Assim, o erro mais comum que ocorre quando se usam uniões consiste em atribuir um valor a um campo de uma união e, em seguida, acessar outro, como ilustrado no fragmento de programa a seguir:

```
typedef union {
           int    x;
           double y;
        } tUniao;

tUniao u = {5};

   /* Um longo trecho que não altera o valor de  */
   /* 'u' e impede o programador de lembrar qual */
   /* é o campo válido ao seu final              */
...

printf("u.y = %f\n", u.y);
```

Qualquer que seja o valor exibido pelo trecho de programa acima, ele não fará sentido.

A melhor precaução para esse tipo de erro consiste em seguir os seguintes passos:

1. Definir uma variável que indique qual é o campo válido de uma união em qualquer instante. Idealmente, essa variável é de um tipo enumeração, no qual cada constante corresponde exatamente a um campo da união à qual se refere.
2. Cada alteração de campo de uma união deve ser acompanhada pela correspondente alteração da variável indicadora mencionada.
3. Antes de acessar um campo de uma união, deve-se testar, usando a variável indicadora, se esse campo é válido.

O programa a seguir ilustra a abordagem delineada acima:

```
#include<stdio.h>

typedef union {
           int    x;
           double y;
        } tUniao;

typedef enum {CAMPO_INT, CAMPO_DOUBLE} tIndicador;

int main(void)
{
   tUniao     u = {5};
   tIndicador indicador = CAMPO_INT;

   if (indicador == CAMPO_DOUBLE) {
      printf("u.y = %f\n", u.y);
   } else {
      printf("u.x = %d\n", u.x);
   }

   return 0;
}
```

D.17.5 Esgotamento de Memória

Nos dias atuais, a causa mais provável de aborto de programa decorrente de esgotamento da memória disponível para um programa é recursão infinita. Como este livro não lida com recursão, quando ocorre esgotamento de memória (ou, mais precisamente, esgotamento de pilha) em programas propostos neste livro, é provável que o programador tenha tido a intenção de chamar uma função e, inadvertidamente, chamou a própria função que efetua a chamada. Outra possibilidade é a ocorrência de uma cadeia recursiva sem fim envolvendo duas ou mais funções, como mostra o programa a seguir:

```
#include<stdio.h>

extern void F2(void);

void F1(void)
{
   printf("\nChamada de F1()\n");
   F2();
}

void F2(void)
{
   printf("\nChamada de F2()\n");
   F1();
}
```

```
int main(void)
{
   F1();
   return 0;
}
```

Quando esse programa é executado, ele é abortado por esgotamento de pilha (*stack overflow*).

D.17.6 Compilador de C++ Não É Compilador de C

Provavelmente, você já topou ou topará com alguma das seguintes afirmações:

- **C é subconjunto de C++**. Baboseira. Existem construções que são válidas em C que não são válidas em C++ e existem construções que são sintaticamente válidas nas duas linguagens, mas têm semânticas (i.e., interpretações) diferentes.
- **C++ é superconjunto de C**. Idem.
- **C++ é um melhoramento ou evolução de C**. C e C++ representam paradigmas de programação diferentes do mesmo modo que bicicleta e motocicleta representam paradigmas diferentes de transporte, apesar de, nesses dois casos, os paradigmas compartilharem características comuns.

Se você já leu ou ouviu alguma dessas afirmações releve a ignorância de quem as profere, pois nenhuma delas corresponde à realidade. É verdade que C e C++ compartilham várias características, mas elas são linguagens distintas usadas em contextos diferentes.

E quanto à afirmação de que compilador de C++ compila programas escritos em C? Bem, qualquer compilador conhecido de C++ é capaz de compilar um programa escrito em C, desde que ele seja informado que o programa deve ser considerado um programa escrito em C (p. ex., por meio de extensão de arquivo-fonte). Caso contrário, o programa será compilado como se fosse um programa escrito em C++ (v. **Seção 3.18.1**).

D.18 Mensagens de Erro Comuns do Compilador GCC

Esta seção apresenta possíveis interpretações para mensagens de erro emitidas pelo compilador GCC com as quais provavelmente o programador já se deparou ou se deparará algum dia. A maioria dos exemplos apresentados nesta seção não tem nenhum significado prático e foram incluídos com o único propósito de ilustrar mensagens de erro. Além disso, As mensagens de erros apresentadas aqui são relativas à versão 4.6.1 do compilador GCC e podem não ser emitidas por outra implementação de C.

D.18.1 lvalue required as left operand of assignment

O tipo de erro a que se refere essa mensagem ocorre quando o operando esquerdo do operador de atribuição é uma constante. Frequentemente, esse tipo de erro é provocado pela não observância de regras de precedência ou troca indevida do operador de igualdade pelo operador de atribuição, como ilustra o seguinte exemplo:

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

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

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x, &y);

   if (x != 0 && y = x) {
      printf("x = %d, y = %d\n", x, y);
   }

   return 0;
}
```

O erro desse programa ocorre na expressão:

```
x != 0 && y = x
```

Como o operador de atribuição é aquele de menor precedência nessa expressão, quando ele é aplicado, seu operando esquerdo é o resultado da avaliação da subexpressão:

```
x != 0 && y
```

que, obviamente, é um valor constante e, por isso, não deveria ser um operando esquerdo do operador de atribuição. No exemplo acima, provavelmente, a intenção do programador era escrever a referida expressão como:

```
x != 0 && y == x
```

Dependendo da versão do compilador GCC utilizada, a mensagem de erro exibida em substituição à mensagem em discussão poderá ser:

```
invalid lvalue in assignment
```

D.18.2 lvalue required as increment (decrement) operand

Essa mensagem de erro indica que existe uma tentativa de aplicação do operador de incremento (ou decremento) sobre um operando que não pode ser alterado. Por exemplo:

```
int main(void)
{
   int ar[5], i;

   for (i = 0; i < 5; ++i) {
      *ar++ = i; /* Ilegal: ar não pode ser alterado */
   }

   return 0;
}
```

Nesse exemplo, o operador de incremento está sendo aplicado sobre o identificador `ar` usado na definição do array `ar[]`. Mas, conforme foi visto na **Seção 7.7**, o nome de um array representa seu endereço e, portanto, deve ser considerado constante.

Dependendo da versão do compilador GCC utilizada, a mensagem de erro exibida em substituição à mensagem em discussão poderá ser:

```
invalid lvalue in increment
```

D.18.3 expected identifier or '(' before '{' token

Tipicamente, o erro indicado por essa mensagem ocorre quando é inserido um ponto e vírgula após o cabeçalho de uma função, como mostra o exemplo a seguir:

```
int Soma(int x, int y);
{
   int soma;

   soma = x + y;

   return soma;
}
```

Nesse exemplo, a linha:

```
int Soma(int x, int y);
```

é interpretada como uma alusão de função (v. **Seção 5.6**). Assim, o bloco que o segue se encontra num local onde não deveria haver nenhuma instrução.

D.18.4 void value not ignored as it ought to be

Essa mensagem de erro é gerada quando uma chamada de função que não retorna valor é usada como operando numa expressão. Por exemplo, a compilação do programa a seguir resulta na apresentação da mensagem de erro em questão:

```
#include <stdio.h>

void Soma(int x, int y)
{
   int soma;

   soma = x + y;
}

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

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x, &y);

   s = Soma(x, y);

   printf("Soma: %d + %d = %d\n", x, y, s);

   return 0;
}
```

Existem duas situações que originam a mensagem de erro que dá título a esta seção:

(1) O tipo de retorno da função não deveria ser **void**.

(2) O tipo de retorno da função é realmente **void**, mas ela é chamada sem que esse fato seja levado em consideração.

D.18.5 invalid use of void expression

O tipo de erro a que essa mensagem se refere é semelhante àquele da **Seção D.18.4**. A única diferença é que o valor (inexistente) retornado pela função é usado como parâmetro de outra função. Por exemplo, a compilação do programa a seguir gera essa mensagem de erro:

```
#include <stdio.h>

void Soma(int x, int y)
{
   int soma;

   soma = x + y;
}
```

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

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x, &y);

   printf("Soma: %d + %d = %d\n", x, y, Soma(x, y));

   return 0;
}
```

A mensagem de erro usada como título desta seção também é emitida quando um ponteiro para **void** não é convertido num ponteiro de tipo conhecido antes da aplicação do operador de indireção ou ele é usado numa operação de aritmética de ponteiros (v. **Seção 12.3**).

D.18.6 'x' redeclared as different kind of symbol

Essa mensagem de erro é tipicamente decorrente de colisão de identificadores e é acompanhada da mensagem adicional:

```
note: previous definition of 'x' was here
```

Considere, por exemplo, a função `Soma()`, definida a seguir.

```
int Soma(int x, int y)
{
   int x;

   return x + y;
}
```

Nas mensagens em questão, `'x'` é o identificador ao qual a mensagem se refere. Obviamente, mensagens nas quais aparecem outro identificador em lugar de `x` tem a mesma interpretação. Note ainda que, estranhamente, nesse caso, o compilador informa que há dois erros no programa sob discussão, quando, de fato, há apenas um.

Nessa função, o primeiro parâmetro e a variável local à função são declarados com o mesmo nome `x` no mesmo escopo. Esse fato dá origem à mensagem de erro em foco.

D.18.7 'x' undeclared (first use in this function)

Essa mensagem de erro indica que a variável `x` não foi declarada, como ilustra o programa seguir:

```
#include <stdio.h>

int main(void)
{
   x = 0;

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

   return 0;
}
```

Uma mensagem de erro dessa natureza também aparece quando uma variável é definida, mas é usada fora de seu escopo (v. **Seção D.15.3**).

Essas mensagens de erro são acompanhas pela mensagem:

```
note: each undeclared identifier is reported only once for each
function it appears in
```

Essa última mensagem informa que o compilador indica apenas uma ocorrência de cada identificador não declarado numa mesma função, mesmo que ele apareça várias vezes nessa função.

D.18.8 conflicting types for 'F'

Provavelmente, existe um conflito entre uma alusão de função (nesse caso, denominada *F*) e a definição da mesma função.

Considere como exemplo o seguinte programa:

```
#include <stdio.h>

extern int F(int x);

int main(void)
{
   int i, j;

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &i, &j);

   printf("Soma: %d + %d = %d\n", i, j, F(i));

   return 0;
}

int F(int x, int y)
{
   return x + y;
}
```

Nesse exemplo, a alusão da função `F()` contém apenas um parâmetro do tipo **int**, enquanto que, na definição da mesma função, há dois parâmetros.

A mensagem que intitula esta seção é acompanha pela mensagem:

```
note: previous declaration of 'F' was here
```

Por causa dessa mensagem adicional, o compilador informa que há dois erros no programa, mas, de fato, há apenas um erro.

D.18.9 too few arguments to function 'F'

O que origina essa mensagem é o fato de uma função ter sido chamada com um número de parâmetros menor do que o esperado. Por exemplo, o programa a seguir causa a apresentação dessa mensagem de erro:

```
#include <stdio.h>

int F(int x, int y)
{
   return x + y;
}

int main(void)
{
   int i, j;

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &i, &j);

   printf("Soma: %d + %d = %d\n", i, j, F(i));

   return 0;
}
```

A mensagem de erro em questão também precipita a emissão de uma mensagem informando o local onde a função é definida:

```
note: declared here
```

Por causa dessa mensagem adicional, o compilador ainda informa que há dois erros no programa acima, quando, na realidade, existe apenas um erro.

D.18.10 too many arguments to function 'F'

Essa mensagem de erro é semelhante àquela da **Seção D.18.9**, mas aqui ocorre o contrário. Isto é, a função `F()` a que se refere a mensagem é chamada com um número de parâmetros maior do que o esperado. As demais observações apresentadas na **Seção D.18.9** também são válidas no presente caso.

D.18.11 incompatible type for argument n of 'F'

Na mensagem de erro que intitula esta seção, `n` é um valor inteiro positivo e `F` é o nome de uma função.

O que, provavelmente, origina uma mensagem de erro dessa espécie é o fato de o parâmetro formal na chamada da função `F()` ser um ponteiro e o parâmetro real correspondente ser um valor real (p. ex., do tipo **double**) ou vice-versa, como mostra o exemplo a seguir:

```
#include <stdio.h>

void F(int *p)
{
   *p = 0;
}

int main(void)
{
   double i = 2.5;

   F(i);

   return 0;
}
```

Nesse caso, o compilador GCC aponta a existência de dois erros, mas só há um erro no programa acima. Apesar dessa falha, o compilador apresenta uma mensagem adicional mais elucidativa:

```
note: expected 'int *' but argument is of type 'double'
```

D.18.12 invalid operands to binary op

Essa mensagem de erro indica que um dos operandos de um operador binário é inválido e ela é complementada por uma explicação mais detalhada entre parênteses e na mesma linha da mensagem. O tipo de erro apontado por essa mensagem é mais comum quando ponteiros são utilizados numa expressão sem a devida aplicação do operador de indireção, como mostra o exemplo a seguir:

```
#include <stdio.h>

int main(void)
{
   int i = 5, *pi = &i;

   printf("%d", pi * i);

   return 0;
}
```

Nesse programa, o primeiro operando do operador de multiplicação [v. segundo parâmetro de **printf()**] é um ponteiro, o que, obviamente, torna a expressão ilegal, já que não existe operação de multiplicação sobre ponteiros.

D.18.13 initializer element is not constant

O que gera essa mensagem de erro é o fato de uma variável de duração fixa ser iniciada com uma expressão que requer o uso do conteúdo de uma variável. Por exemplo, as linhas de código a seguir dão origem a esse tipo de mensagem em qualquer local de um programa em que se encontrem:

```
int        x = 5;
static int y = x; /* Ilegal */
```

Nesse trecho de programa, tenta-se atribuir à variável `y`, que é de duração fixa, o conteúdo de outra variável, o que é ilegal (v. **Seção 5.12.3**).

D.18.14 expected '=', ',', ';', 'asm' or '__attribute__' before '{' token

Apesar do aspecto tenebroso e pouco elucidativo dessa mensagem de erro, ela é mais provável de ser exibida quando o programador esquece de incluir parênteses no cabeçalho de uma definição de função, como no programa a seguir:

```
#include <stdio.h>

int main
{
   int x = 2;

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

   return 0;
}
```

A mensagem de erro em foco também é emitida quando existe algo acompanhando o cabeçalho de uma função que impeça o compilador de concluir que o cabeçalho está completo.

D.18.15 parameter 'x' is initialized

Uma provável situação na qual o compilador emite a mensagem em apreço é quando o programador esquece o abre-chaves que segue o cabeçalho de uma função, como ocorre na função `F()` a seguir:

```
int F()
   int x = 0;

   return x;
}
```

Existe um estilo obsoleto de declaração de parâmetros de função, que não é discutido neste livro, no qual os parâmetros são declarados logo após o fecha-parênteses no cabeçalho da função. Assim, quando, em vez de abre-chaves, o compilador encontra uma iniciação de variável, ele acredita tratar-se de uma declaração de parâmetro no formato obsoleto (mas considerado legal). Então, ele informa que o erro encontrado deve-se ao fato de um parâmetro estar sendo iniciado, o que não é permitido (v. **Seção 5.4.1**).

D.18.16 declaration for parameter 'x' but no such parameter

A mais provável causa de emissão dessa mensagem é a mesma apontada na **Seção D.18.15**.

D.18.17 assignment of read-only location '*s'

Se você se deparar com essa mensagem de erro, provavelmente, seu programa está tentando alterar um conteúdo em memória que você declarou que deveria ser considerado constante, como no programa a seguir:

```
int main(void)
{
   const char *s = "Bola";

   *s = 'C'; /* O compilador aponta o erro */

   return 0;
}
```

A mensagem de erro decorrente desse programa deve deixá-lo radiante, pois, seguindo recomendação apresentada na **Seção 8.3**, você trocou um erro de execução por um erro de compilação.

D.18.18 missing terminating " character

Existem duas causas prováveis para essa mensagem de erro:

(1) Foram esquecidas as aspas ao final de um string.

(2) As aspas finais de um string foram incluídas, mas são precedidas pelo caractere de escape `'\'`. Nesse caso, as aspas são interpretadas como parte do string e não como um delimitador de string.

O segundo caso mencionado acima é exemplificado no programa a seguir:

```
#include <stdio.h>

int main()
{
  printf("Alo mundo\");

  return 0;
}
```

Nesse programa, o caractere `'\'` que precede a segunda ocorrência de aspas na chamada de **printf()** faz com que elas sejam consideradas parte integrante do string, e não como delimitadoras desse string.

A mensagem de erro em questão certamente virá acompanhada de uma ou mais mensagens de erro ou advertência. Na compilação do último programa, por exemplo, as seguintes mensagens serão emitidas:

```
warning: missing terminating " character
error: expected expression before 'return'
error: expected ';' before '}' token
```

D.18.19 continue statement not within a loop

O que causa essa mensagem de erro é o uso de uma instrução **continue** fora do corpo de um laço de repetição, como ilustra o programa a seguir:

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

```
int main(void)
{
   double x = 2;
   continue; /* Ilegal */
   printf("\nx = %d\n", x);
   return 0;
}
```

Lembre-se que, no contexto de programação em C, a melhor interpretação para a instrução **continue** é *salte* e não *continue* (v. **Seção 4.7.2**).

D.18.20 break statement not within loop or switch

O que provoca essa mensagem de erro é o uso de uma instrução **break** fora do corpo de um laço de repetição ou de uma instrução **switch-case**, como é o caso no programa a seguir:

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

void ImprimeRaiz(double x)
{
   if (x < 0) {
      break; /* Ilegal */
   }
   printf("Raiz quadrada: %f\n", sqrt(x));
}
int main(void)
{
   ImprimeRaiz(2.5);
   return 0;
}
```

D.18.21 unterminated comment

Essa mensagem de erro é provocada pela falta de fechamento de um comentário, como no programa a seguir:

```
#include <stdio.h>

int main(void)
{
      /* Um comentário interminável
   printf("Alo mamae\n");
   return 0;
}
```

D.18.22 expected declaration or statement at end of input

A mais provável causa dessa mensagem de erro é a ausência de fecha-chaves ao final da função **main()**, como ocorre com o seguinte programa:

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

```
int main(void)
{
   int x = 2;
   printf("\nx = %d\n", x);
   return 0;
   /* Deveria haver fecha-chaves aqui */
```

D.18.23 expected ',' or ';' before 'identificador'

O provável erro apontado por essa mensagem é a falta de vírgula ou ponto e vírgula na linha que antecede o identificador indicado na mensagem. Por exemplo, quando o seguinte programa é compilado:

```
#include <stdio.h>
int main(void)
{
   int x = 2, y = 3
   printf("x/y = %d", x/y);
   return 0;
}
```

o compilador apresenta a mensagem de erro:

```
error: expected ',' or ';' before 'printf'
```

D.18.24 expected '{' at end of input

Uma causa provável para a exibição dessa mensagem de erro é a ausência de abre-chaves após o cabeçalho da função **main()**, como ocorre no programa a seguir:

```
#include <stdio.h>
int main(void)
   return 0;
}
```

D.18.25 syntax error before 'identificador'

Numa mensagem de erro desse tipo, o compilador é vago e apenas informa que há um erro antes do identificador especificado na mensagem. Versões antigas do compilador GCC emitem mensagens dessa natureza. Portanto, se você se deparar com uma dessas mensagens, atualize sua versão de GCC. (A versão 4.6.1 do compilador GCC, utilizada para testar a maioria dos exemplos deste livro, não emite mensagens de erro dessa espécie.)

D.18.26 syntax error before 'T' token

Para um compilador de C, token é uma sequência de caracteres que forma uma construção válida na linguagem C. Portanto, como na mensagem de erro discutida na **Seção D.18.25**, o compilador informa apenas que há um erro antes do token especificado na mensagem. Se você topar com uma mensagem dessa natureza, utilize uma versão mais recente do compilador GCC. (A versão 4.6.1 do compilador GCC não emite mensagens de erro dessa natureza.)

D.18.27 expected 'token 1' before 'token 2'

Essa mensagem de erro é vaga e informa apenas que o compilador esperava encontrar o primeiro token mencionado na mensagem antes do segundo token citado na mesma mensagem. Por exemplo, quando o seguinte programa é compilado,

```
#include <stdio.h>

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

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

   return 0;
}
```

a seguinte mensagem (entre outras) é emitida:

```
error: expected ';' before '}' token
```

Note que o erro no último programa é a ausência de aspas necessárias para encerrar o string de formatação de **printf**().

D.18.28 expected expression before 'T' token

Uma mensagem de erro desse tipo é vaga e apenas indica que o compilador esperava encontrar uma expressão antes do token mencionado na mensagem. O programa apresentado como exemplo na **Seção D.18.27**, quando compilado, causa a apresentação de uma mensagem dessa natureza:

```
expected expression before '}' token
```

D.18.29 expected declaration specifiers before 'T' token

A expressão *expected declaration specifiers before 'T' token* significa que o compilador esperava encontrar uma declaração de parâmetro no estilo obsoleto de cabeçalho de C antes de encontrar o token especificado na mensagem. Nesse estilo obsoleto, os parâmetros são declarados logo após o fecha-parênteses no cabeçalho da função. Apesar de obsoleto, esse estilo de cabeçalho ainda é aceito (mas seu uso não é recomendado).

Uma provável situação na qual o compilador emite uma mensagem desse tipo é quando o programador esquece ou troca por fecha-chaves, o abre-chaves que segue o cabeçalho de uma função, como ocorre na função **main**() do seguinte programa:

```
#include <stdio.h>

int main(void)
   int x = 2;

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

   return 0;
}
```

Quando esse programa é compilado, o compilador emite a seguinte mensagem:

```
expected declaration specifiers before '}' token
```

D.18.30 expected declaration specifiers before 'identificador'

A expressão *expected declaration specifiers before 'identificador'* significa que o compilador esperava encontrar uma declaração de parâmetro no estilo obsoleto de cabeçalho de C antes de encontrar o identificador especificado na mensagem de erro (v. **Seção D.18.29**).

Tipicamente, o compilador emite uma mensagem dessa natureza quando o programador esquece ou troca por fecha-chaves, o abre-chaves que segue o cabeçalho de uma função. Por exemplo, quando o programa da **Seção D.18.29** é compilado, a seguinte mensagem é emitida:

```
expected declaration specifiers before 'printf'
```

D.18.31 No such file or directory

Tipicamente, essa mensagem de erro é emitida quando o nome de um arquivo de cabeçalho é digitado incorretamente, como, por exemplo:

```
#include <sting.h>
```

Nesse exemplo, a alvoroçada mensagem de erro emitida pelo compilador é:

```
fatal error: sting.h: No such file or directory
```

que indica que o pré-processador não encontrou o arquivo **`sting.h`**. (Note que a grafia do nome do arquivo de cabeçalho está incorreta, pois falta a letra *r*.)

D.18.32 redefinition of 'x'

Muitas vezes, essa mensagem de erro surge quando se tenta incluir num programa uma instrução fora do corpo de qualquer função, como por exemplo:

```
#include <stdio.h>

int x = 0;

x = 2*(x + 1); /* ILEGAL */

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

   return 0;
}
```

Frequentemente, a mensagem de erro em foco é acompanhada por outras mensagens de erro ou advertência decorrentes do mesmo fato. Por exemplo, na compilação do último programa, são emitidas as seguintes mensagens adicionais:

```
warning: data definition has no type or storage class
warning: type defaults to 'int' in declaration of 'x'
note: previous definition of 'x' was here
error: initializer element is not constant
```

D.18.33 duplicate label 'rotulo'

Essa mensagem de erro é decorrente do uso de dois rótulos que usam o mesmo identificador numa mesma função e é acompanhada pela mensagem adicional:

```
note: previous definition of 'rotulo' was here
```

Considere, por exemplo, a função **`F()`**, definida a seguir. Nessa função, o rótulo **`rotulo`** é definido duas vezes, o que acarreta a mensagem de erro mencionada.

```
int F(void)
{
   int x = 2, y = 1;

rotulo:
   x = x + y;
```

```
   if (x < 10) {
      goto rotulo;
   }

   {
   rotulo: /* ILEGAL: rótulos devem  */
           /* ser únicos numa função */
      y = y + x;
   }

   return y;
}
```

D.18.34 invalid initializer

Provavelmente, a principal causa da mensagem de erro em questão é a ausência de chaves em torno do iniciador de um array, uma estrutura ou uma união, como mostra o programa a seguir:

```
#include <stdio.h>

int main(void){
   int ar[10] = 1; /* Erro de Sintaxe */

   printf("%d", ar[0]);

   return 0;
}
```

Nesse programa, a iniciação do array `ar[]` deveria ser assim:

```
int ar[10] = {1};
```

D.18.35 invalid type argument of '->'

A mensagem de erro em apreço é tipicamente ocasionada quando o operando esquerdo do operador seta não é um ponteiro para estrutura, como mostra o programa a seguir:

```
#include <stdio.h>

typedef struct {
           double x, y;
        } tPonto;

int main(void)
{
   tPonto umPonto = {2.5, -2.5};

   printf("Ponto: (%f, %f)\n", umPonto->x, umPonto->y);

   return 0;
}
```

No exemplo acima, a variável `umPonto` usada como operando esquerdo do operador seta é uma estrutura, e não um ponteiro para estrutura.

D.18.36 request for member 'x' in something not a structure or union

A mensagem de erro em questão é emitida quando o operando esquerdo do operador ponto não é uma estrutura ou o operando direito não é um campo da estrutura representada pelo operando esquerdo, como mostra o programa a seguir:

```
#include <stdio.h>

typedef struct {
           double x, y;
        } tPonto;

int main(void)
{
   tPonto umPonto = {2.5, -2.5}, *ptrPonto = &umPonto;

   printf("Ponto: (%f, %f)\n", ptrPonto.x, ptrPonto.y);

   return 0;
}
```

Nesse exemplo, a variável `ptrPonto` usada como operando esquerdo do operador ponto é um ponteiro para estrutura, e não uma estrutura.

D.18.37 wrong type argument to increment (decrement)

A variável usada como operando do operador de incremento (ou decremento) é provavelmente uma variável estruturada, como uma estrutura ou união. O programa a seguir ilustra uma situação na qual a mensagem de erro em foco é emitida:

```
typedef struct {
           double x, y;
        } tPonto;

int main(void)
{
   tPonto ponto;

   ++ponto; /* ILEGAL */

   return 0;
}
```

D.18.38 unknown type name 'y'

Essa mensagem de erro informa que o compilador não encontrou um identificador associado a um tipo de dado conforme ele esperava encontrar. A mais provável causa desse tipo de mensagem de erro é o uso de abreviação, que é permitida em definições de variáveis, mas não é o caso com declarações de parâmetros, como mostra o seguinte exemplo:

```
void Soma(int x, y)
{
   return x + y;
}
```

Nesse exemplo, o compilador esperava encontrar um nome de tipo que seria usado para definir o tipo do segundo parâmetro da função `Soma()`, mas, em vez disso, encontrou o nome de um parâmetro.

D.18.39 invalid lvalue in assignment

O tipo de erro a que se refere essa mensagem é o mesmo apresentado na **Seção D.18.1**.

D.18.40 invalid lvalue in increment

O tipo de erro a que se refere essa mensagem é o mesmo apresentado na **Seção D.18.2**.

D.19 Advertências Comuns do Compilador GCC

Esta seção apresenta possíveis interpretações para mensagens de advertência frequentemente emitidas pelo compilador GCC versão 4.6.1. A maioria dos exemplos apresentados nesta seção é desprovida de significado prático. Quer dizer, esses exemplos foram incluídos com o único propósito de ilustrar mensagens de advertência.

As mensagens de advertências apresentadas aqui são classificadas de acordo com a gravidade da situação que elas antecipam:

Crítica	A instrução que originou a mensagem certamente resultará em erro de execução (aborto) ou erro lógico quando o programa for executado.
Média	A instrução que originou a mensagem pode prejudicar a eficiência do programa, mas não há maiores consequências.
Irrelevante	A mensagem de advertência pode ser ignorada, pois a instrução que a origina não causa nenhum dano.

Quando uma mensagem de advertência for classificada em mais de uma dessas categorias, a gravidade da situação depende do contexto no qual a mensagem é emitida. Além disso, a despeito da classificação apresentada, lembre-se *sempre* de dar a devida atenção a qualquer mensagem de advertência emitida por um compilador. Enfim, tenha sempre em mente que:

Um programa robusto, eficiente e que incorpora boas práticas de estilo de programação não origina nenhuma mensagem de advertência.

Além disso:

Um programador que leva a sério sua profissão não admite que seu programa seja maculado por qualquer mensagem de advertência emitida por um compilador considerado de boa qualidade.

D.19.1 format '%?' expects type 'T *', but argument n has type 'T'

Gravidade: Crítica

O título desta seção especifica o formato de uma categoria de advertências. Nesse título, **%?** representa um especificador de formato de **scanf**(), `T` representa um tipo e `n` representa o número de ordem de um parâmetro.

O que origina uma mensagem de advertência dessa natureza é a ausência do operador de endereço **&** no enésimo parâmetro numa chamada de função da família *scanf* [p. ex., **scanf**(), **fscanf**()]. Por exemplo:

```
#include <stdio.h>

int main(void)
{
   int x;

   printf("Digite um valor inteiro: ");
   scanf("%d", x);  /* Operador & está ausente */

   printf("\nVoce digitou: %d\n", x);

   return 0;
}
```

Nesse exemplo, a mensagem de advertência emitida pelo compilador é:

```
warning: format '%d' expects argument of type 'int *', but
         argument 2 has type 'int'
```

Essa mensagem indica precisamente que o segundo parâmetro de **scanf()** deveria ser do tipo **int** *, mas, de fato, é **int**. Para corrigir o erro apontado pelo compilador, deve-se preceder a variável `x` com o operador de endereço (representado por **&**). Se o programa não for assim corrigido, apresentará um erro lógico ou, quem sabe, será abortado. Esse último caso ocorrerá se o valor de `x`, interpretado como um endereço, estiver fora do espaço de endereçamento do programa.

D.19.2 format '%?' expects a matching 'T *' argument

Gravidade: Crítica/Média

O título desta seção especifica o formato de uma categoria de advertências. Nele, **%?** representa um especificador de formato de **scanf()** (ou qualquer outro membro da família scanf) e `T` representa um tipo.

Uma mensagem de advertência dessa categoria indica que um especificador de formato de **scanf()** deveria ter um endereço de variável associado a ele, mas esse endereço está ausente, como mostra o exemplo a seguir:

```
#include <stdio.h>

int main(void)
{
   int x, y;

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x);

   printf("\nValores lidos: x = %d, y = %d\n", x, y);

   return 0;
}
```

Nesse programa, a chamada de **scanf()** inclui dois especificadores de formato, mas há apenas um endereço de variável. Portanto, nesse caso, a correção do problema indicado pela mensagem de advertência seria chamar a referida função como:

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

Nesse mesmo programa, a gravidade do erro apontado pelo compilador é crítica, pois o programa deixa claro que precisa realmente ler dois valores. Agora, se o valor da variável `y` não tivesse importância, a gravidade do erro seria média, pois haveria apenas desperdício de memória.

D.19.3 too many arguments for format

Gravidade: Crítica/Média

A origem dessa mensagem é o fato de o número de especificadores de formato num string de formatação de uma função da família *scanf* [p. ex., **scanf()**, **fscanf()**] ser menor do que o número de respectivos endereços de variáveis. Ou seja, o problema apontado pela mensagem em questão é o contrário daquele discutido na **Seção D.19.2**.

Considere como exemplo o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int x, y;

   printf("Digite dois valores inteiros: ");
   scanf("%d", &x, &y);

   printf("\nValores lidos: x = %d, y = %d\n", x, y);

   return 0;
}
```

Nesse programa, a chamada de **scanf()** tem apenas um especificador de formato, mas há tentativa de leitura de dois valores (pois há dois endereços de variáveis). Portanto, nesse caso, a correção do problema indicado pela mensagem de advertência seria chamar a função **scanf()** como:

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

Nesse mesmo programa, a gravidade do erro apontado pelo compilador é crítica, pois o programa deixa claro que pretende ler dois valores. Se o valor da variável `y` não tivesse importância, haveria apenas desperdício de memória e a gravidade do erro seria considerada média.

D.19.4 spurious trailing '%' in format

Gravidade: Crítica

O que acarreta essa mensagem de advertência é uma chamada de função da família *scanf* [p. ex., **scanf()**] contendo o caractere `'%'` sem acompanhamento de outros caracteres que complementem um especificador de formato. Por exemplo:

```
#include <stdio.h>

int main()
{
   int x;

   printf("Digite um valor inteiro: ");
   scanf("%", &x); /* Especificador de formato incompleto */

   printf("Voce digitou: %d\n", x);

   return 0;
}
```

Nesse programa, o especificador de formato de **scanf()** está incompleto (deveria ser **%d**).

Além da mensagem em questão, o resultado da compilação do último programa também apresenta a mensagem de advertência:

```
too many arguments for format
```

pela mesma razão apresentada na **Seção D.19.3**.

D.19.5 unknown escape sequence: '\0dd'

Gravidade: Média

O que dá origem a essa mensagem é o fato de um string conter um caractere de escape `\` sem que esse caractere em conjunto com o caractere que o segue constituam uma sequência de escape conhecida. A sequência de escape, denotada por `'\0dd'`, que aparece ao

final da mensagem de advertência em questão, representa o valor inteiro em base octal associado ao caractere que segue o caractere \. Esses dois caracteres constituem, inadvertidamente, a sequência de escape desconhecida mencionada na referida mensagem.

Considere como exemplo o seguinte programa:

```
#include <stdio.h>

int main()
{
  printf("Alo mundo\ ");

  return 0;
}
```

Quando esse programa é compilado, aparece a mensagem:

```
warning: unknown escape sequence: '\040'
```

Nessa mensagem, `40` é o valor inteiro em base octal correspondente ao espaço em branco que segue o caractere \. Esses dois caracteres constituem uma sequência de escape desconhecida. Provavelmente, a intenção do programador seria escrever a sequência de escape `'\n'`, usada como quebra de linha.

D.19.6 variable 'x' set but not used

Gravidade: Média

Uma variável à qual é atribuído um valor e que não é usada origina essa mensagem de advertência, como mostra o seguinte exemplo:

```
#include <stdio.h>

int main(void)
{
   int x, y;

   x = 2;
   y = 3;

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

   return 0;
}
```

Nesse programa, a mensagem de advertência em apreço é oriunda do fato de à variável `x` ter sido atribuído um valor, mas esse valor não ser usado.

É interessante notar que, se o programa acima for ligeiramente alterado, resultando no programa funcionalmente equivalente:

```
#include <stdio.h>

int main(void)
{
   int x = 2, y = 3;

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

   return 0;
}
```

a mensagem de advertência apresentada pelo compilador será:

```
warning: unused variable 'x'
```

o que mostra, mais uma vez, que atribuição e iniciação são construções bem parecidas, mas, realmente, diferentes (v. **Seção D.16.1**).

D.19.7 'x' is used uninitialized in this function

Gravidade: Crítica

A mensagem de advertência em foco indica que a variável **x** é usada sem que lhe tenha sido atribuído nenhum valor. Por exemplo, a compilação do programa a seguir:

```
#include <stdio.h>
int main()
{
   int x, y;

   printf("Digite um valor inteiro: ");
   scanf("%d", &y);

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

   return 0;
}
```

apresenta essa mensagem de advertência.

Quando esse programa é executado, ele pode apresentar como resultado:

```
Digite um valor inteiro: 5
x = 2147344384, y = 5
```

o que demonstra a gravidade da situação, pois o valor de **x** exibido não faz nenhum sentido. (O valor de **x** em outra execução do programa provavelmente será provavelmente diferente desse valor.)

D.19.8 unused variable (parameter) 'x'

Gravidade: Crítica/Média

Esta mensagem de advertência:

```
unused variable 'x'
```

indica que a variável referenciada por **x** foi definida, mas não foi usada pelo programa. Existem duas causas possíveis para esse problema:

(1) A referida variável não é necessária, mas, no instante em que a declarou, o programador imaginou que ela seria útil. Essa situação acomete tanto programadores iniciantes quanto aqueles mais experientes e, nesse caso, não causa grave dano a um programa. Se você remover a declaração da variável que não é usada pelo programa, o compilador não emitirá mais a referida mensagem de advertência. Nesse caso, a gravidade é média, pois haverá apenas desperdício de memória.

(2) A variável é realmente necessária, mas não foi usada como deveria, porque, provavelmente, o programador usou outra variável em seu lugar. A gravidade, nesse caso, é severa, pois o programa não funcionará satisfatoriamente.

Exemplo da primeira situação:

```
#include <stdio.h>
int main(void)
{
   int x, y, soma;

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x, &y);

   printf("Soma: %d + %d = %d\n", x, y, x + y);

   return 0;
}
```

No programa acima, a variável **soma** pode ser removida sem alterar a funcionalidade do programa.

Exemplo da segunda situação:

```
#include <stdio.h>

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

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x, &y);

   x = x + y;

   printf("Soma: %d + %d = %d\n", x, y, x);

   return 0;
}
```

No último programa, a variável **soma** deveria ter sido usada para armazenar o resultado da soma que o programa calcula, mas, em vez disso, o programador usou a variável **x**. Nesse caso, o programador deve usar a variável **soma** no lugar da primeira ocorrência de **x** na expressão:

```
x = x + y;
```

ou reestruturar o programa, de modo que a variável **soma** não seja mais necessária.

Quando um parâmetro é declarado no cabeçalho de uma função e não é usado em seu corpo, o compilador emite uma mensagem semelhante àquela discutida acima, mas usa *parameter*, em vez de *variable*, no teor da mensagem. Por exemplo, quando o seguinte programa é compilado:

```
#include <stdio.h>

int Soma(int x, int y, int z)
{
   return x + y;
}

int main(void)
{
   printf("Soma: %d\n", Soma(1, 2, 3));

   return 0;
}
```

o compilador emite a seguinte mensagem:

```
warning: unused parameter 'z'
```

visto que o parâmetro **z** foi declarado no cabeçalho da função acima mas não foi utilizado no corpo dessa função.

A análise da situação indicada por essa última mensagem de advertência é a mesma apresentada no início desta seção.

D.19.9 suggest parentheses around assignment used as truth value

Gravidade: Crítica/Irrelevante

O compilador encontrou uma expressão condicional cujo valor é resultante da avaliação de uma atribuição. Essa mensagem de advertência pode identificar um grave problema, como, por exemplo:

```
if (x = 0) {
   ...
}
```

Essa instrução **if** é sempre incorreta, pois o valor da expressão:

```
x = 0
```

é sempre zero. No entanto, o laço **while** a seguir:

```
while (*ar++ = *s++)
   ; /* Instrução vazia */
```

está provavelmente correto, pois o fato de o programador ter indicado explicitamente que o corpo do laço é vazio sugere que ele está consciente do que escreveu (v. **Seção 8.5.4**).

Na aludida mensagem de advertência, o compilador sugere que o programador utilize um par de parênteses adicional (i.e., redundante) em torno da expressão de atribuição. Para que serve esse par de parênteses? Bem, esses parênteses não alteram em nada a tradução do programa (i.e., o código gerado será o mesmo com eles ou sem eles), de modo que eles apenas fazem o compilador interpretar como consciente a intenção do programador e, assim, ele não mais emitirá a referida mensagem de advertência. Portanto, se você está ciente que a expressão está correta, não custa nada agradar o compilador.

D.19.10 implicit declaration of function 'F'

Gravidade: Irrelevante

A mensagem de advertência em questão é gerada quando o compilador encontra uma chamada da função `F()` sem antes ter encontrado sua definição ou uma alusão a ela. Se essa função fizer parte de seu programa, basta incluir uma alusão correspondente (v. **Seção 5.6**) antes das definições das funções ou rearranjar a ordem em que se encontram essas definições. Entretanto, só é razoável seguir essa última recomendação se o programa contiver um número muito pequeno de funções. Por exemplo, quando o programa a seguir é compilado:

```
#include <stdio.h>

int main(void)
{
   int x, y;

   printf("Digite dois valores inteiros: ");
   scanf("%d %d", &x, &y);

   x = x + y;

   printf("Soma: %d + %d = %d\n", x, y, Soma(x, y));

   return 0;
}

int Soma(int x, int y)
{
   return x + y;
}
```

o compilador apresenta a seguinte mensagem de advertência:

```
warning: implicit declaration of function 'Soma'
```

Para que o compilador não emita mais essa mensagem, o programador pode colocar uma alusão da função `Soma()` antes da função **main()** ou mover a própria definição da função `Soma()` para essa mesma posição.

Se a função à qual a mensagem de advertência em foco se refere faz parte de uma biblioteca, provavelmente o programador esqueceu de incluir o cabeçalho que contém a alusão da função. Por exemplo, quando é compilado, o programa a seguir:

```
int main(void)
{
   printf("Alo mundo");
   return 0;
}
```

origina a seguinte mensagem de advertência:

```
implicit declaration of function 'printf'
```

O que causa essa mensagem de advertência é a ausência da diretiva:

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

O possível erro apontado pela mensagem em questão pode ser considerado irrelevante, porque ele é incapaz de causar danos a um programa. Na pior hipótese, o programador terá pela frente um erro de ligação, quando a referida função for chamada indevidamente (v. **Seção D.4.8**).

D.19.11 control reaches end of non-void function

Gravidade: Crítica

A mensagem de advertência em apreço é oriunda da definição de uma função cujo tipo de retorno não é **void**, mas que não retorna nenhum valor. Ou seja, essa função deveria retornar um valor, mas não contém nenhuma instrução **return** que realize isso. A função `Soma()` definida a seguir, por exemplo, causa a apresentação da referida mensagem de advertência.

```
int Soma(int x, int y)
{
   int soma;
   soma = x + y;
}
```

O comportamento de um programa contendo uma função como `Soma()`, definida acima, apresentará um comportamento indeterminado. Para corrigir o erro dessa função, deve-se acrescentar, antes do final do seu corpo, a instrução:

```
return soma;
```

A mensagem de advertência em discussão era muito comum na função **main()** de muitos programas, pois essa função deve sempre retornar um valor do tipo **int** que, frequentemente, é esquecido. No entanto, de acordo com os padrões C99 e C11, na ausência de uma instrução de retorno numa função **main()**, o compilador deve acrescentar a instrução:

```
return 0;
```

ao corpo dessa função.

D.19.12 return type defaults to 'int'

Gravidade: Irrelevante

A mensagem de advertência que intitula esta seção é decorrente do fato de o programador ter omitido o tipo de retorno numa definição de função. Por exemplo:

```
Soma(int x, int y) /* Ilegal em C99 e C11 */
{
   int soma;

   soma = x + y;

   return soma;
}
```

Como o programador omitiu o tipo de retorno na definição da função `Soma()` acima, o compilador considerou-o como sendo **int** (v. **Seção 5.4.1**).

A partir do padrão ISO C99, omitir o tipo de retorno de uma função passou a ser considerado ilegal. Infelizmente, poucos compiladores adotaram essa norma. Mas, mesmo que seu compilador negligencie esse preceito, não proceda assim.

D.19.13 passing argument n of 'F' makes pointer from integer without a cast

Gravidade: Crítica

A mensagem de advertência em questão chama a atenção do programador para um casamento de parâmetros que certamente resultará num grave erro de execução. Por exemplo:

```
#include <stdio.h>

void Troca(int *p1, int *p2)
{
   int aux = *p1;

   *p1 = *p2;
   *p2 = aux;
}

int main(void)
{
   int i = 2, j = 5;

   printf( "\n\t>>> ANTES da troca <<<\n"
           "\n\t    i = %d, j = %d\n", i, j );

   Troca(&i, j);

   printf( "\n\t>>> DEPOIS da troca <<<\n"
           "\n\t    i = %d, j = %d\n", i, j );

   return 0;
}
```

O problema com esse programa é que, na chamada da função `Troca()`, o segundo parâmetro, que deveria ser do tipo **int** * (i.e., ponteiro para **int**), é do tipo **int**. Quando esse programa é executado, ele é abortado, o que demonstra a gravidade da situação.

D.19.14 passing argument n of 'F' from incompatible pointer type

Gravidade: Crítica

Essa mensagem de advertência indica um erro grave: o enésimo parâmetro formal da função `F()` é de um tipo de ponteiro incompatível com o tipo de ponteiro do respectivo parâmetro real usado na chamada. Por exemplo:

```
#include <stdio.h>

void Troca(int *p1, int *p2)
{
   int aux = *p1;

   *p1 = *p2;
   *p2 = aux;
}

int main(void)
{
   double i = 2.5, j = 5.2;

   printf( "\n\t>>> ANTES da troca <<<\n"
           "\n\t    i = %f, j = %f\n", i, j );

   Troca(&i, &j);

   printf( "\n\t>>> DEPOIS da troca <<<\n"
           "\n\t    i = %f, j = %f\n", i, j );

   return 0;
}
```

Nesse programa, os parâmetros formais da função `Troca()` são do tipo **int** * e os parâmetros reais na chamada da função são do tipo **double** *. Portanto, nesse caso, o casamento entre parâmetros formais e reais não é adequado.

Quando o programa em questão é executado, ele apresenta o seguinte resultado:

```
>>> ANTES da troca <<<

    i = 2.500000, j = 5.200000

>>> DEPOIS da troca <<<

    i = 2.500002, j = 5.199997
```

Note que a principal diferença entre a mensagem de advertência discutida nesta seção e aquela apresentada na **Seção D.19.13** é que, aqui, a incompatibilidade entre parâmetros envolve ponteiros de tipos diferentes, enquanto que, na **Seção D.19.13**, a incompatibilidade envolve parâmetros formais que são ponteiros e parâmetros reais que não são ponteiros ou vice-versa.

D.19.15 passing argument n of 'F' makes integer from pointer without a cast

Gravidade: Crítica

Essa mensagem de advertência informa que haverá uma conversão de um endereço num valor de tipo inteiro em virtude do casamento entre o enésimo parâmetro formal e o respectivo parâmetro real. Por exemplo:

```
#include <stdio.h>

int F(int x)
{
   return x;
}
```

```
int main(void)
{
   int i = 2, j;
   j = F(&i);
   printf("\ni = %d, j = %d\n", i, j);
   return 0;
}
```

Nesse programa, o endereço da variável `i` é convertido em **int** durante a chamada da função `F()`. Essa conversão é crítica porque o padrão de C não prevê o que poderá acontecer quando o programa for executado.

D.19.16 return makes pointer from integer without a cast

Gravidade: Crítica

Essa mensagem indica que uma função que deveria retornar um endereço, na realidade, retornará um valor inteiro convertido em endereço. Por exemplo:

```
#include <stdio.h>

int *F(int x)
{
   return x;
}

int main(void)
{
   int i = 2, j;
   j = *F(i);
   printf("\ni = %d, j = %d\n", i, j);
   return 0;
}
```

Quando executado, esse programa poderá ser abortado. Portanto, a situação é crítica.

D.19.17 return makes integer from pointer without a cast

Gravidade: Crítica

Essa mensagem indica que uma função que deveria retornar um valor inteiro, retornará um endereço convertido em inteiro. Por exemplo:

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

int F(void)
{
   int *p = malloc(sizeof(int));

   *p = 5;

   return p;
}
```

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

   i = F();

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

   return 0;
}
```

A situação relacionada com essa mensagem de advertência é crítica porque o padrão de C não prevê o que poderá acontecer quando o programa for executado.

D.19.18 return from incompatible pointer type

Gravidade: Crítica

Quando o tipo de retorno de uma função é um tipo de ponteiro e a função retorna um endereço de outro tipo, o compilador emite essa mensagem de advertência. A situação relacionada com essa mensagem de advertência é crítica porque o padrão de C não prevê o que poderá acontecer quando o programa for executado. Considere o seguinte programa como exemplo dessa situação:

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

int *F(void)
{
   double *p = malloc(sizeof(double));

   *p = 2.54;

   return p;
}

int main(void)
{
   int *x;

   x = F();

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

   return 0;
}
```

Quando executado, o programa poderá apresentar como resultado:

```
x = -2061584302
```

Esse resultado sem sentido apresentado pelo programa poderia ser outro (igualmente sem sentido), o que demonstra a gravidade da situação.

D.19.19 function returns address of local variable

Gravidade: Crítica

A mensagem de advertência em questão anuncia o nascimento de um bebê zumbi que, oportunamente, sairá do seu sarcófago, que é a pilha de execução, para assombrar o programador.

Brincadeiras à parte, a referida mensagem de advertência informa que a função a que ela se refere retornará o endereço de um parâmetro ou de uma variável de duração automática (v. **Seções 7.9.4** e **D.4.1**).

A seguir, o último exemplo de zumbi deste livro:

```
#include <stdio.h>

int *F(int x)
{
   return &x; /* x é um zumbi */
}

int main(void)
{
   int i = 5, j;

   j = *F(i);

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

   return 0;
}
```

A função `F()` do programa acima, retorna o endereço do parâmetro `x`, que, conforme foi visto na **Seção 7.9.4**, será liberado ao retorno da função. Entretanto, nesse caso, o efeito fantasmagórico não aparece porque o programa é curto demais para permitir que o zumbi se manifeste.

D.19.20 assignment (initialization) makes integer from pointer without a cast

Gravidade: Crítica

Essa mensagem de advertência indica que há uma conversão de endereço em número inteiro em decorrência de uma atribuição (ou iniciação), como ilustra o programa a seguir:

```
#include <stdio.h>

void Troca(int *p1, int *p2)
{
   int aux = p1; /* Errado: deveria ser 'aux = *p1' */

   *p1 = *p2;
   *p2 = aux;
}

int main(void)
{
   int i = 2, j = 3;

   printf( "\n\t>>> ANTES da troca <<<\n"
           "\n\t    i = %d, j = %d\n", i, j );

   Troca(&i, &j);

   printf( "\n\t>>> DEPOIS da troca <<<\n"
           "\n\t    i = %d, j = %d\n", i, j );

   return 0;
}
```

Nesse exemplo, a variável `aux` da função `Troca()` é do tipo **int**, mas é iniciada com o parâmetro `p1`, que é do tipo **int** *.

Quando o programa acima é executado ele apresenta como resultado:

```
>>> ANTES da troca <<<

    i = 2, j = 3

>>> DEPOIS da troca <<<

    i = 3, j = 2293580
```

O valor desconcertante da variável **j** comprova a gravidade da situação.

D.19.21 assignment (initialization) from incompatible pointer type

Gravidade: Crítica

A mensagem de advertência supracitada é emitida quando o compilador detecta uma atribuição ou iniciação entre ponteiros de tipos diferentes. Por exemplo, quando o programa a seguir é compilado,

```
#include <stdio.h>

int main(void)
{
   int x = 2;
   double *p = &x;

   printf("\nx = %d, *p = %f\n", x, *p);

   return 0;
}
```

o compilador apresenta a mensagem:

```
warning: initialization from incompatible pointer type
```

Quando executado, o último programa pode apresentar como resultado:

```
x = 2, *p = 0.000000
```

O resultado apresentado pelo programa demonstra a gravidade da situação.

D.19.22 assignment (initialization) discards 'const' qualifier from pointer target type

Gravidade: Crítica

Essa mensagem indica que o valor de uma variável qualificada com **const** pode ser alterado em virtude de uma atribuição de seu endereço a um ponteiro, como ocorre, por exemplo, no seguinte programa:

```
#include <stdio.h>

int main(void)
{
   const int varConstante = 0;
   int      *ptr;

   ptr = &varConstante;

   *ptr = 1;

   printf("\nvarConstante = %d\n", varConstante);

   return 0;
}
```

Na compilação desse programa, a mensagem de advertência apresentada quando o programa é compilado:

```
warning: assignment discards 'const' qualifier from pointer
target type
```

é decorrente da atribuição:

```
ptr = &varConstante;
```

Se a intenção do programador for realmente alterar o valor da variável para a qual o ponteiro `ptr` aponta, para evitar que essa mensagem de erro seja emitida, o programador deve usar conversão explicita na atribuição como em:

```
ptr = (int *) &varConstante;
```

Caso o programador desejasse resguardar a variável `varConstante` de possíveis alterações provocadas pelo ponteiro `ptr`, o conteúdo apontado por esse ponteiro deveria ser qualificado com **const**, como na seguinte definição:

```
const int *ptr;
```

D.19.23 passing argument n of 'F' discards 'const' qualifier from pointer target type

Gravidade: Crítica

Essa mensagem indica que o endereço de uma variável qualificada com **const** foi passado como parâmetro real para uma função e o parâmetro formal com o qual esse endereço casa não é devidamente qualificado com **const**. Por exemplo:

```
#include <stdio.h>

int F(int *p)
{
   *p = 0;
   return 0;
}

int main(void)
{
   const int varConstante = 0;
   printf("\nF() = %d\n", F(&varConstante));
   return 0;
}
```

Quando esse programa é compilado, compilador emite a seguinte mensagem adicional:

```
note: expected 'int *' but argument is of type 'const int *'
```

A mensagem é classificada como crítica porque se a intenção do programador não fosse resguardar o conteúdo da variável `varConstante`, ele não deveria tê-la definido com **const**.

É importante ressaltar que não é o fato de a função `F()` alterar o conteúdo apontado por seu parâmetro que ocasiona a mensagem de advertência sob discussão. Isto é, a razão para tal mensagem é a capacidade que a função tem para efetuar essa alteração, quer ela realmente ocorra ou não. Por exemplo, se função `F()` fosse substituída por:

```
int F(int *p)
{
   printf("\n*p = %d\n", *p);
   return 0;
}
```

o compilador continuaria a emitir a mesma mensagem de advertência. Para sanar o problema, o parâmetro `p` dessa última função `F()` deveria ser qualifica com **const**, já que a função não altera o conteúdo para o qual esse parâmetro aponta.

D.19.24 return discards 'const' qualifier from pointer target type

Gravidade: Crítica

Suponha que a função `EncontraPrimeiroChar1()` apresentada na **Seção D.4.10** tivesse sido definida como:

```
char *EncontraPrimeiroChar3(const char *str, int c)
{
   while (1) {
      if (*str == c) {
         return str;
      }

      if (!(*str++)) {
         break;
      }
   }

   return NULL;
}
```

Então, a mensagem de advertência que intitula esta seção alertaria o programador para o fato de a função retornar um endereço que permite alterar o conteúdo de um string que deve ser mantido constante.

A correção da função `EncontraPrimeiroChar3()` que evita a mensagem de advertência em foco consiste em qualificar o retorno dessa função com **const**, conforme foi mostrado na **Seção D.4.10**.

D.19.25 excess elements in array initializer

Gravidade: Crítica/Irrelevante

Essa mensagem indica que há mais elementos numa iniciação de array do que o tamanho do array, como ocorre, por exemplo, no seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int i, ar[3] = {-1, -2, 8, 4};

   for (i = 0; i < 4; ++i) {
      printf("ar[%d] = %d\n", i, ar[i]);
   }

   return 0;
}
```

Quando há mais valores numa iniciação do que o número de elementos do array ora sendo iniciado, os valores em excesso são meramente descartados. Portanto, a gravidade dessa situação é crítica se o array realmente precisa armazenar todos os elementos presentes na iniciação, mas seu tamanho foi subdimensionado. Por outro lado, se o tamanho do array estiver correto, os elementos excedentes não causarão nenhum dano ao programa, de forma que, nesse caso, o problema indicado pela mensagem é irrelevante.

D.19.26 dereferencing 'void *' pointer

Gravidade: Irrelevante

Essa advertência é decorrente da mensagem de erro discutida na **Seção D.18.5**; i.e., ela é emitida quando há tentativa de aplicação do operador de indireção sobre um ponteiro genérico que não foi convertido explicitamente para um tipo de ponteiro conhecido.

Essa mensagem de advertência é considerada irrelevante porque ela acompanha uma mensagem de erro, o que significa que o programa que a provoca não será compilado.

D.19.27 extra tokens at end of #include directive

Gravidade: Irrelevante

O uso de ponto e vírgula ao final de uma diretiva **#include** resulta nessa mensagem de advertência, mas o erro apontado por ela não causa nenhum dano a um programa.

D.19.28 left-hand operand of comma expression has no effect

Gravidade: Crítica

Conforme foi visto na **Seção 4.9**, o resultado do operador vírgula é o operando da direita; i.e., o operando esquerdo não contribui para o resultado. Portanto, quando não ocorre alteração do valor de uma variável no operando esquerdo desse operador, esse operando pode ser removido sem modificar o resultado da expressão. É com relação a esse fato que a mensagem de advertência em questão alerta o programador. O programa a seguir ilustra esse arrazoado:

```
#include <stdio.h>

int main(void)
{
   int i, j;

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

   return 0;
}
```

Nesse programa, o operando do primeiro operador vírgula no laço **for** é a variável `i`. Provavelmente, a intenção do programador seria iniciar a variável `i`, assim como fez com a variável `j`. Portanto, como nenhuma variável é alterada no operando constituído pela variável `i`, o compilador emite a mensagem de advertência em discussão.

Quando o programa acima é executado, o resultado é imprevisível, o que mostra que a gravidade da situação indicada pela referida mensagem é crítica.

D.19.29 overflow in implicit constant conversion

Gravidade: Crítica

A mensagem de advertência em apreço indica que uma conversão de atribuição resulta num valor grande demais para caber na variável que recebe esse valor. O programa a seguir ilustra essa situação:

```
#include <stdio.h>

int main(void)
{
   int i;

   i = 1.85E100;

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

   return 0;
}
```

Nesse exemplo, o valor `1.85E100` convertido em inteiro é grande demais para caber na variável `i`, que é do tipo **int**.

D.19.30 'return' with a value, in function returning void

Gravidade: Crítica/Irrelevante

O que origina a mensagem de erro que intitula esta seção é o fato de uma função cujo tipo de retorno é **void** retornar um valor. A função `F()` a seguir causa a emissão da mensagem de advertência em questão:

```
void F(void)
{
   return 0;
}
```

A gravidade do erro indicado pela mensagem de advertência mencionada é considerada crítica se o tipo de retorno da função não deveria ser **void**, mas foi inadvertidamente definido assim. Caso contrário, a gravidade é considerada irrelevante. Em qualquer caso, uma tentativa de usar o valor retornado pela função `F()` causa erro de compilação (v. **Seção D.18.5**).

D.19.31 statement with no effect

Gravidade: Crítica/Média

Uma instrução que não modifica o valor de qualquer variável ou parâmetro é inútil e pode ser removida de um programa sem alterar sua funcionalidade. Existem duas exceções para essa afirmação: (1) quando a instrução consiste de uma chamada de função e (2) quando ela é uma instrução **return**.

Como exemplo, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int x = 0, y = 0;
      /* A seguinte instrução não tem nenhum efeito */
   2*(x + 1);

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

   return 0;
}
```

Nesse programa, a instrução:

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

não altera o resultado apresentado pelo programa e, portanto, pode ser removida dele.

A mensagem de advertência em questão talvez indique que o programador tenha esquecido de atribuir o valor resultante da expressão a uma variável. Por exemplo, no programa acima, é possível que a intenção do programador fosse atribuir a expressão em foco à variável `y`, como:

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

Nesse caso, a mensagem de advertência aponta uma situação crítica. Caso contrário, se a expressão em discussão foi incluída por mera negligência, a gravidade do problema indicado pela mensagem é considerada média, pois torna o programa menos eficiente, já que haverá instruções em linguagem de máquina que não contribuem em nada para a funcionalidade do programa.

D.19.32 data definition has no type or storage class

Gravidade: Irrelevante

Tipicamente, essa mensagem de advertência está associada à mensagem de erro apresentada quando se tenta incluir num programa uma instrução fora do corpo de qualquer função, como foi discutido na **Seção D.18.32**.

Essa mensagem de advertência é considerada irrelevante porque, de qualquer modo, o programa contendo o erro mencionado não será compilado.

D.19.33 "/*" within comment

Gravidade: Crítica/Irrelevante

Essa mensagem de advertência pode indicar uma situação crítica quando parte de um programa é indevidamente considerada como comentário, como mostra o programa a seguir:

```
#include <stdio.h>

int main(void)
{
   int a = 5, b = 2, *p = &b;

   a = a/*p; /* Atualiza o valor de a */;

   printf("\na = %d, b = %d, *p = %d\n", a, b, *p);

   return 0;
}
```

Observe no programa acima que, se não existisse um ponto e vírgula ao final da linha que contém o comentário, ocorreria um erro sintaxe, em vez de emissão de uma mensagem de advertência.

Em situações diferentes daquela ilustrada nesse exemplo, a mensagem de advertência em questão pode ser considerada irrelevante.

D.19.34 wrong type argument to increment (decrement)

Gravidade: Crítica

Essa mensagem de advertência informa que o operando do operador de incremento (ou decremento) é aceitável do ponto de vista sintático, mas, talvez, a expressão resultante produza um valor inesperado. Considere, por exemplo, o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int   ar[] = {1, 2, 3};
   void *p = ar;

   p++;

   printf("\n*p = %d\n", *(int *)p);

   return 0;
}
```

Nesse programa, sobre o ponteiro `p` não deveria ser efetuada nenhuma operação aritmética, já que ele é um ponteiro genérico que não foi convertido para um tipo de ponteiro conhecido (v. **Seção 12.3**). Quando esse programa é executado, ele produz como resultado:

```
*p = 33554432
```

que não faz absolutamente nenhum sentido.

De acordo com o padrão ISO de C, o resultado de uma operação aritmética sobre um ponteiro genérico sem a devida conversão é indefinido e depende do compilador utilizado. No caso do compilador GCC, nesse contexto, um ponteiro genérico é interpretado como um ponteiro para **char**. Assim, o incremento de `p` no programa em discussão faz esse ponteiro apontar para o segundo byte do primeiro elemento do tipo **int** armazenado no array `ar[]`, o que explica o estranho resultado apresentado pelo último programa.

D.19.35 operation on 'x' may be undefined

Gravidade: Crítica

Essa mensagem de advertência indica que o resultado de uma operação sobre uma variável ou um parâmetro, cujo identificador é `x`, não é definido pelo padrão de C. Ou, em outras palavras, o resultado dessa operação não é portável. Por exemplo, o programa seguir causa a emissão dessa mensagem de advertência:

```
#include <stdio.h>

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

   y = x + ++x;

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

   return 0;
}
```

O problema com esse programa é que, na expressão:

```
x + ++x
```

a ordem de avaliação dos operandos `x` e `++x` não é definida (v. **Seção 3.9**).

Quando esse último programa é compilado com GCC, ele produz como resultado:

```
y = 2
```

Entretanto, se o mesmo programa tivesse sido compilado com outro compilador, o resultado poderia ser:

```
y = 1
```