Leitura e Validação de Dados II

Todo programa interativo deve ser tolerante a falhas dos usuários. Isto é, ele deve ser capaz de aceitar a introdução de entradas que não são apropriadas e responder adequadamente indicando a causa do erro do usuário. Esta seção apresenta o projeto e a implementação de uma abordagem para leitura e validação de dados de entrada mais sofisticada do que aquela apresentada na **Seção 3.7**. Conforme poder-se-á constatar, essa tarefa não é tão trivial quanto aparenta ser.

Na corrente versão, será dado enfoque apenas à leitura e validação de valores de entrada propriamente ditas. Em outras palavras, o foco desta seção será a função `LeValor()` introduzida na **Seção 3.7**. Será demonstrado também como apresentar mensagens de erro mais elaboradas que indiquem precisamente a causa de uma entrada de dados inválida. Algumas partes do programa que ilustra o método da **Seção 3.7** sofreram pequenas modificações para se adaptarem à nova abordagem e serão brevemente discutidas aqui. Se você ainda não leu a **Seção 3.7**, é recomendável que a leia antes de prosseguir.

O objetivo principal aqui é apresentar alternativas que superem as limitações da versão da função `LeValor()` apresentada na **Seção 3.7**. Como recordação, as limitações apresentadas pela primeira versão desta função são:

- A função consegue distinguir "o" (*ó*) de zero, mas pode considerar uma entrada que não corresponde à intenção do usuário. Por exemplo:

  ```
  Introduza um numero inteiro positivo: 2o
  O valor lido foi: 2
  ```

- A função não consegue indicar corretamente uma condição de *overflow*. Por exemplo:

  ```
  Introduza um numero inteiro positivo: 222222222222222222222222
  O valor -1 nao e' valido
  ```

- A função causa o aborto do programa quando é introduzido um valor negativo que cause *underflow*. Por exemplo, a interação a seguir:

  ```
  Introduza um numero inteiro positivo: -222222222222222222222222
  ```

  causa o encerramento prematuro do programa apresentado na versão anterior.

- A função não indica precisamente a causa de um erro. Por exemplo:

  ```
  Introduza um numero inteiro positivo: alo2
  O valor introduzido nao e' valido
  ```

A nova versão da função `LeValor()` a ser apresentada aqui é capaz de superar todas as limitações apresentadas acima. Por exemplo, caso o usuário introduza "`2o`" (leia-se *ó* e não zero), a função de leitura e validação é capaz de identificar o erro e apontá-lo corretamente, conforme ilustrado a seguir:

```
Introduza um numero de ponto flutuante: 2o
Entrada ilegal. Foi encontrado um erro na posicao indicada
abaixo:
    2o
     ^
```

Na implementação do método de leitura e validação de dados desenvolvida aqui, é feita a suposição de que o valor a ser lido é do tipo **double**, mas esta suposição não constitui uma limitação do método. Isto é, ela foi considerada apenas para demonstrar o uso prático do método. Você não deverá ter dificuldades em modificar esta suposição de modo a adequá-la às suas necessidades.

Descrição Geral do Método

O algoritmo a ser seguido pela função `LeValor()`, que faz a leitura e validação de dados, é o seguinte:

1. Leia o conteúdo do buffer de entrada padrão como um *string*.
2. Tente converter o *string* lido para o tipo de dados esperado pelo programa.
3. Usando o resultado da tentativa de conversão para o tipo desejado faça o seguinte:

   3.1 Se o *string* lido *não* puder ser convertido para o tipo adequado, apresente uma mensagem de erro indicando precisamente por que o *string* não pode ser convertido e retorne 0. Neste caso, o argumento da função que recebe o valor lido e validado é *indefinido*.

   3.2 Se a conversão do *string* para o tipo desejado for bem sucedida, faça o seguinte:

      3.2.1 Se este valor for *válido* conforme a expectativa do programa, retorne 1. O argumento da função que recebe o valor lido e validado conterá o valor convertido.

      3.2.2 Se este valor *não* for válido conforme a expectativa do programa, retorne 0. O argumento da função que recebe o valor lido e validado conterá o valor convertido.

Funções, Variáveis e Tipos da Biblioteca Padrão Utilizados

Aqui serão apresentados as funções, variáveis globais e tipos da biblioteca padrão de C utilizados na implementação da presente abordagem. As apresentações seguem o seguinte modelo:

- **#include <*arquivo*>**: Representa o arquivo que você deve incluir para estar apto a utilizar o respectivo objeto.
- **Protótipo**: No caso de funções, apresenta o protótipo. No caso de variável global, indica como a mesma pode ser aludida. No caso de macros, utiliza-se uma analogia com protótipo de função (já que o termo *protótipo* não se aplica exatamente a macro).
- **Descrição**: Contém uma descrição sucinta do objeto.
- **Uso**: Descreve o papel desempenhado pelo objeto na implementação da abordagem apresentada.
- **Informações Adicionais**: Sugestão sobre referências que podem ser consultadas para obter uma descrição mais detalhada do objeto.

Os objetos são apresentadas na ordem em que aparecem pela primeira vez na função `LeValor()`.

Tipo size_t

- `#include <stddef.h>`
- **Descrição**: Este tipo é utilizado por argumentos e valores de retorno de várias funções da biblioteca padrão de C. Ele corresponde simplesmente a um inteiro sem sinal e, muitas vezes, pode ser substituído por **int** ou **unsigned int**, mas por uma questão de portabilidade, é melhor utilizá-lo.
- **Uso**: Utilizado como tipo da variável `nCaracteresLidos`, que representa o comprimento [calculado pela função **strlen()**] do *string* lido.
- **Informações Adicionais**: Desnecessárias.

Função fflush()

- `#include <stdio.h>`
- **Protótipo**: `int fflush(FILE *stream);`

- **Descrição**: Descarrega**)** *buffers* associados com *streams* de saída. Retorna 0 se bem sucedida ou **EOF** (constante definida em `<stdio.h>`) se detecta algum erro.
- **Uso**: Utilizada para garantir que uma saída de dados (um prompt, mais precisamente) é apresentada imediatamente ao usuário. O uso desta função nestas circunstâncias pode ser desnecessário em algumas implementações de C (por exemplo, Borland C++), mas, para garantir portabilidade, é melhor utilizá-la.
- **Informações Adicionais**: **Capítulo 12**.

Função fgets()

- `#include <stdio.h>`
- **Protótipo**: `char *fgets(char *str, int n, FILE *stream);`
- **Descrição**: Função para leitura de *strings* num *stream* especificado. O conteúdo lido é armazenado no array apontado por `str` e a leitura encerra quando `n-1` caracteres são lidos ou quando a função encontra o caractere `'\n'`. O n-ésimo caractere é o caractere terminal de *string* `'\0'` anexado ao final do array apontado por `str`. Se for bem sucedida, a função retorna o *string* `str`; caso contrário (i.e., quando ela encontra o final do *stream* antes de ler `n-1` caracteres ou encontrar `'\n'`), ela retorna **NULL**.
- **Uso**: Esta função é utilizada para ler o conteúdo do *buffer* de entrada padrão e armazená-lo como um *string* (passo 1 do algoritmo).
- **Informações Adicionais**: **Seção 12.8.2**.

Função strlen()

- `#include <string.h>`
- **Protótipo**: `size_t strlen(const char *s);`
- **Descrição**: Esta função**)** retorna o comprimento do *string* que recebe como argumento.
- **Uso**: Utilizada para calcular o número de caracteres lidos no buffer de entrada padrão (passo 2 do algoritmo).
- **Informações Adicionais**: **Seção 8.4.3**.

Variável errno

- `#include <errno.h>`
- **Protótipo**: `extern int errno;`
- **Descrição**: Variável global que armazena um número que representa uma condição de erro. Esta variável é alterada por várias funções da biblioteca padrão quando encontram uma condição de erro.
- **Uso**: Utilizada para verificar se houve algum erro na tentativa de conversão do *string* para o tipo de dados especificado pelo programa (passo 3 do algoritmo).
- **Informações Adicionais**: **Volume II**.

Função vfprintf()

- `#include <stdio.h>`
- **Protótipo**:
```
int vfprintf( FILE *stream, const char *formato,
              va_list argumentos );
```
- **Descrição**: Esta função**)** funciona exatamente como a função **printf()**, exceto pelos fatos de (1) permitir a especificação de um stream de saída (argumento `stream`) e (2) ter um argumento do tipo **va_list** (`argumentos`) ao invés de argumentos variáveis, como ocorre com **printf()**. Quando é bem sucedida, a função retorna o número de bytes escritos; caso contrário, ela retorna **EOF**.
- **Uso**: Utilizada pela função `ImprimeMensagemDeErro()` para imprimir mensagens de erro no stream padrão de erros **stderr** (passo 3.1 do algoritmo).
- **Informações Adicionais**: **Capítulo 12**.

Função strtod()

- `#include <stdlib.h>`
- **Protótipo**: `double strtod(const char *s, char **ptrFinal);`
- **Descrição**: Converte**)** um *string* para um valor do tipo **double**. A função encerra o processamento do *string* logo que encontra um caractere que não pode ser interpretado como parte de um número de ponto flutuante. Neste caso, o argumento `ptrFinal` apontará para o caractere no *string* de entrada que causou o enceramento da função.

- **Uso**: Utilizada para tentar converter o *string* lido num valor do tipo **double** (passo 2 do algoritmo).
- **Informações Adicionais**: **Volume II**.

Função strerror()

- `#include <string.h>`
- **Protótipo**: `char *strerror(int numeroDoErro);`
- **Descrição**: Retorna**)** um *string* contendo a mensagem de erro associada ao argumento `numeroDoErro`.
- **Uso**: Utilizada imprimir uma mensagem de erro do sistema quando ocorre algum erro na tentativa de conversão do *string* para o tipo de dados especificado pelo programa e o programa não é capaz de identificar o tipo de erro encontrado (passo 3.1 do algoritmo).
- **Informações Adicionais**: **Volume II**.

Tipo va_list e Macros va_start e va_end

- `#include <stdarg.h>`
- **Protótipos (macros)**:

```
void va_start(va_list argumentos, ultimoArgumentoFixo);
tipo va_arg(va_list argumentos, tipo);
void va_end(va_list argumentos);
```

- **Descrição**: Esse tipo e essas macros são utilizados em definições de funções com argumentos variáveis.
- **Uso**: Utilizados para implementar a função `ImprimeMensagemDeErro()` (v. adiante).
- **Informações Adicionais**: **Seção 3.4**.

Leitura e Validação de Valores

A função `LeValor()` apresentada a seguir lê um *string* no meio de entrada padrão, tenta convertê-lo para um valor do tipo **double** e verifica se o valor convertido satisfaz os critérios especificados utilizando a estratégia delineada no algoritmo apresentado na

Seção 10.9.1. (As linhas de interesse da função foram numeradas para facilitar a discussão que segue a apresentação da função.)

```
 1. unsigned LeValor(double *valor, unsigned (*pfVerifica)(double))
    {
 2.    size_t  nCaracteresLidos;
 3.    char    strEntrada[TAMANHO_DO_ARRAY];
 4.    char   *ptrFinal;

 5.    while (1) {
 6.       printf("Introduza um numero de ponto flutuante: ");
 7.       fflush(stdout);

 8.       if (!fgets(strEntrada, sizeof(strEntrada), stdin))
             return 0;

 9.       nCaracteresLidos = strlen(strEntrada);

10.       if (strEntrada[nCaracteresLidos - 1] == '\n') {
11.          strEntrada[--nCaracteresLidos] = '\0';

12.          errno = 0;

13.          *valor = strtod(strEntrada, &ptrFinal);

14.          if (!errno && nCaracteresLidos && !*ptrFinal)
                break;

15.          IndicaErro(strEntrada, ptrFinal);

16.       } else {
17.          LimpaBuffer();
18.          ImprimeMensagemDeErro("A entrada foi muito grande. "
                                   "O numero maximo de caracteres"
                                   " e' %d.\n", TAMANHO_DO_ARRAY);
          }
       } /* while */

19.    if (!pfVerifica)
20.       return 1;

21.    if (pfVerifica(*valor))
22.       return 1;
       else {
23.       ImprimeMensagemDeErro("O valor %lf nao satisfaz criterios "
                                "do programa.\n", *valor);
24.       return 0;
       }
    }
```

Comentários sobre a função LeValor()

1. A função `LeValor()` retorna 1 quando é bem sucedida (i.e., quando lê e valida o valor **double** conforme esperado) e retorna 0 quando fracassa em sua tarefa. Ela possui dois argumentos:

- `valor` - argumento de saída representado por um ponteiro para **double** que apontará para o valor lido em caso de sucesso. Caso a função não seja bem sucedida, o conteúdo apontado por `valor` é indefinido
- `pfVerifica` - argumento de entrada representado por um ponteiro para uma função de verificação que recebe um valor **double** como entrada e retorna 1 se o valor lido satisfaz os critérios de validade e 0, em caso contrário.

2. A variável local `nCaracteresLidos` armazena o número de caracteres lidos na entrada padrão. Esta variável é do tipo **size_t** porque este é o tipo do valor retornado pela função de biblioteca **strlen()** que calcula o comprimento do *string* lido no meio de saída.

3. A variável local `strEntrada` é um array de caracteres que conterá o *string* lido. O número de elementos deste array é dado pela constante simbólica `TAMANHO_DO_ARRAY`, definida no início do arquivo que contém a definição da função `LeValor()`.

4. A variável local `ptrFinal` é um ponteiro que apontará para o caractere do *string* que causar o final da tentativa de conversão do *string* para **double**.

5. Este é um laço de repetição infinita (pois a condição sempre resulta em 1) cuja saída será feita no interior do corpo do laço (v. adiante).

6. Apresenta um prompt para o usuário solicitando uma entrada de dados.

7. A chamada `fflush(stdout)` garante**)** que o usuário lê o prompt apresentado pela função **printf()** imediatamente. O uso desta função pode ser desnecessário em algumas implementações, mas, para garantir portabilidade, é melhor utilizá-la.

8. A chamada `fgets(strEntrada, sizeof(strEntrada), stdin)` lê o conteúdo do buffer da entrada padrão **stdin** e armazena-o no array `strEntrada`. O número máximo de caracteres que esta função tenta ler é dado por `sizeof(strEntrada) - 1`. O valor retornado por esta chamada de **fgets()** é testado pela instrução **if**. Esta função retorna **NULL** apenas quando o buffer de entrada está vazio. Neste caso, não há mais nada a ser feito e a função `LeValor()` retorna 0.

9. Esta instrução utiliza a função **strlen()** para calcular o número de caracteres lidos no meio de entrada e armazenados no array `strEntrada`.

10. Quando a função **fgets()** encontra o caractere `'\n'` antes de retornar, ela inclui este caractere como penúltimo caractere do *string* lido. Este **if** testa exatamente se o caractere `'\n'` foi lido pela função **fgets()**.

11. Neste ponto, o caractere `'\n'` foi encontrado no *string* lido. Este caractere precisa ser removido antes de tentar fazer a conversão do *string*, caso contrário a função de conversão indicará uma condição de erro (porque o caractere `'\n'` não faz parte de nenhum valor **double**). Na realidade, esta instrução realiza duas tarefas:

- Remove o caractere `'\n'` atribuindo o caractere terminal de *string* `'\0'` ao elemento do array que contém `'\n'`
- Subtrai 1 do número de caracteres lidos, visto que este valor foi reduzido com a exclusão de `'\n'`.

12. É necessário iniciar a variável **errno** com zero, pois assim pode-se garantir que, se o valor desta variável for diferente de zero após a chamada de **strtod()**, a causa deste novo valor terá sido um erro na tentativa de conversão feita por esta função.

13. Esta instrução tenta converter o *string* armazenado no array `strEntrada` num valor do tipo **double** que será armazenado no conteúdo apontado pela variável `valor`, caso a conversão seja bem sucedida. O parâmetro `ptrFinal` utilizado na chamada da função de conversão **strtod()** é um ponteiro que irá apontar para o primeiro caractere no *string* armazenado em `strEntrada` que causar a interrupção da tentativa de conversão.

14. Quando satisfeita, a condição `!errno && nCaracteresLidos && !*ptrFinal` indica que nenhum erro foi detectado na conversão feita pela função **strtod()**. Neste caso, a execução da instrução **break** causa a saída do laço **while**. A condição que indica a inexistência de erro na conversão é uma conjunção de três componentes:

- `!errno`: esta condição é satisfeita quando a variável **errno** é igual a zero, que é o valor que ela tinha antes da chamada da função **strtod()**; portanto, esta função não alterou esta variável e nenhum erro ocorreu.
- `nCaracteresLidos`: esta condição é satisfeita quando a variável `nCaracteresLidos` é diferente de zero; ou seja, havia caracteres a ser convertidos.
- `!*ptrFinal`: esta condição é satisfeita quando a variável `ptrFinal` aponta para o caractere `'\0'` que indica o final do *string*; ou seja, quando satisfeita, esta condição indica que todo o *string* foi utilizado na conversão e, portanto, não houve nenhum caractere no *string* que causasse o final prematuro da tentativa de conversão.

15. Se a execução do laço **while** chegou a este ponto é porque ocorreu algum erro na tentativa de conversão. A chamada da função `IndicaErro()` (v. adiante) mostra precisamente ao usuário a causa do erro encontrado.

16. A cláusula **else**, associada ao **if** da linha 9, corresponde ao fato de não ter sido encontrado o caractere `'\n'` no *string* de entrada. A chamada da função **fgets()** na linha 7 permite que o usuário digite no máximo `sizeof(strEntrada) - 1` caracteres, incluindo o caractere `'\n'`, que corresponde a `[ENTER]` que ele digita para enviar sua entrada de dados para o programa. Portanto, o fato de o caractere `'\n'` não ter sido encontrado no *string* significa que o usuário digitou caracteres além do número permitido.

17. Pelo menos o caractere `'\n'` encontra-se no buffer de entrada, já que ele não foi encontrado no *string* de entrada. É necessário limpar o buffer de entrada antes da próxima tentativa de leitura de dados.

18. A chamada da função `ImprimeMensagemDeErro()` (v. adiante) informa ao usuário que o número de caracteres digitados foi além do número permitido.

19. Neste ponto do programa, sabe-se que a entrada do usuário foi realmente do tipo especificado (**double**). Resta determinar se este valor satisfaz outros critérios especificados na chamada da função `LeValor()`. Aqui, testa-se se foi passado um valor nulo como valor do argumento `pfVerifica`.

20. O fato de o ponteiro para função `pfVerifica` ser nulo indica que não há função de verificação para testar o valor lido; i.e., qualquer valor válido até então será definitivamente válido. Portanto, neste caso, retorna-se 1 indicando que o valor foi validado.

21. A chamada de função `pfVerifica(*valor)` utiliza o ponteiro para função `pfVerifica` passado como parâmetro para chamar a verdadeira função de validação que é definida em algum local do programa.

22. O valor lido foi convertido para o tipo **double** e, além disso, ele satisfaz as condições de validação especificadas. Esta instrução simplesmente, retorna 1 indicando o sucesso absoluto.

23. O valor lido e convertido para **double** *não* satisfaz as condições de validação especificadas. Esta instrução apresenta uma mensagem de erro correspondente para o usuário (v. **Seção 10.9.10**).

24. Esta instrução simplesmente retorna 0 indicando que o valor não foi validado.

Apresentação de Mensagens de Erro

A função `ImprimeMensagemDeErro()` apresenta mensagens de erro no *stream* padrão de erros **stderr**. Esta função foi desenvolvida mais por conveniência do que por estrita necessidade.

```
static unsigned ImprimeMensagemDeErro(const char *formato, ...)
{
   va_list argumentos;
   int     resultado;

   va_start (argumentos, formato);
   resultado = vfprintf(stderr, formato, argumentos);
   va_end (argumentos);

   return resultado;
}
```

A função `ImprimeMensagemDeErro()` tem os seguintes argumentos de entrada:

- `formato`: este argumento deve ser um *string* de formatação do mesmo tipo usado pela função **printf()**.
- `...`: representa aquilo que será impresso usando `formato`.

É desnecessário utilizar qualquer chamada da função **fflush()** na função `ImprimeMensagemDeErro()` porque esta função imprime no stream padrão de erros **stderr** que é um *stream* sem *buffering* (v. **Seção 12.3**).

Observe que esta função foi definida como **static** porque ela interessa apenas ao módulo que implementa a função `LeValor()`. Em outras palavras, ela é apenas uma função que auxilia a implementação da função `LeValor()`e, portanto, não deve fazer sentido para outras partes de programas que utilizam a função `LeValor()`.

A função `ImprimeMensagemDeErro()` retorna exatamente aquilo que é retornado pela função **vfprintf()**. Para entender melhor a implementação desta função refira-se à **Seção 3.4**, que trata de funções com listas de argumentos variáveis.

Indicação Precisa de Erros

A função `IndicaErro()` indica precisamente a posição onde ocorre um erro na entrada de dados do usuário. (As linhas de interesse da função foram numeradas para facilitar a discussão que segue a apresentação da função.)

```
 1. static void IndicaErro(const char *str, const char *ptrErro)
```

```
   {
2.    static const char *const mensagem =
             "Entrada ilegal. Foi encontrado um erro na posicao "
             "indicada abaixo:\n";

3.    if (errno != 0) {
4.       ImprimeMensagemDeErro("%s\n", strerror(errno));
5.       ImprimeMensagemDeErro(mensagem);
6.    } else
7.       ImprimeMensagemDeErro(mensagem);

8.    ImprimeMensagemDeErro("    %s\n", str);
9.    ImprimeMensagemDeErro("    %*c\n", (int) (ptrErro - str) + 1,
                            '^');
   }
```

Comentários sobre a função IndicaErro()

1. Esta é função é definida como **static** pela mesmas razões expostas para a função `ImprimeMensagemDeErro()` e tem os seguintes argumentos de entrada:

- `str`: este argumento é um *string* que representa a entrada de dados do usuário.
- `ptrErro`: este argumento é um ponteiro para o primeiro caractere que causa um erro de conversão do *string*.

2. A variável local `mensagem` armazena o conteúdo inicial da mensagem de erro que será apresentada ao usuário. Ela foi definida como **static** para que ela não seja iniciada a cada chamada da função. O conteúdo desta variável não deve ser modificado (justificativa para o primeiro qualificador **const** na definição da variável) e ela aponta sempre para este mesmo *string* (justificativa para o segundo qualificador **const** na definição da variável).

3. Verifica se houve erros detectados na conversão e indicados na variável global **errno**.

4. Um erro foi assinalado pela variável **errno**. A chamada de função `strerror(errno)` retorna o *string* fornecido pelo sistema corresponde ao erro. A chamada da função `ImprimeMensagemDeErro()` imprime a mensagem de erro correspondente (v. **Seção 10.9.10**).

5. Esta instrução imprime a primeira parte da mensagem indicativa de erro que será apresentada.

6. A cláusula **else** indica que não houve erro indicado pela variável **errno**, mas ocorreu algum erro indicado de outro modo (caso contrário esta função não teria sido chamada).

7. Imprime a primeira parte da mensagem indicativa de erro, como descrito no comentário sobre a linha 5.

8. Esta instrução simplesmente imprime o *string* lido na entrada de dados.

9. Esta chamada da função `ImprimeMensagemDeErro()` solicita que seja impresso, na linha seguinte, o caractere `'^'` na posição correspondente ao caractere do *string* de entrada onde ocorreu o erro. Esta posição é dada pela diferença entre o endereço do caractere onde ocorreu o erro, representado por `ptrErro`, e o endereço inicial do *string*, representado por `str`, mais um (i.e., esta posição é dada por: `ptrErro - str + 1`). O asterisco no *string* de formatação na chamada da função informa que devem ser saltados espaços em número correspondente ao respectivo argumento [dado por `(int) (ptrErro - str) + 1`]. Depois que este número de espaços é saltado, o caractere `'^'` é impresso exatamente onde é esperado.

Função LimpaBuffer()

A função `LimpaBuffer()` apresentada a seguir lê e descarta todos os caracteres que porventura tenham sido deixados no buffer de entrada em alguma tentativa de leitura de dados.

```
void LimpaBuffer(void)
{
    int valorLido = getchar(); /* valorLido deve ser int! */

    while ((valorLido != '\n') && (valorLido != EOF))
        valorLido = getchar();
}
```

A função `LimpaBuffer()` complementa a implementação da abordagem apresentada aqui. A finalidade e implementação desta função são as mesmas apresentadas na **Seção 8.8.4**.

Chamada da Função de Leitura

Antes de chamar a função `LeValor()` é necessário definir uma ou mais funções de validação de dados. Uma tal função de validação deve retornar um valor diferente de zero se o valor é válido e zero em caso contrário, como:

```
static unsigned VerificaValor(double valor)
{
      return ((valor >= 10.0) && (valor < 20.0));
}
```

Esta função considera que o valor recebido como argumento é válido se for maior do que ou igual a 10 e menor do que 20.

A função **main()** apresentada a seguir utiliza a função `LeValor()` para ler valores de acordo com a validação especificada pela função `VerificaValor()`.

```
int main(void)
{
   double   valorLido; /* Armazena os valores válidos lidos */
   unsigned teste; /* Armazena valor retornado por LeValor() */
   int      nValoresLidos = 0; /* Valores válidos lidos */

   printf("\nIntroduza valores de ponto-flutuante "
          "entre 10 (inclusive) e 20.\n"
          " \nSerao lidos tres valores validos.\n\n");

   do { /* Lê três valores válidos */
      teste = LeValor(&valorLido, VerificaValor);
      if (teste) { /* Valor lido foi validado */
         ++nValoresLidos; /* Mais um valor válido foi lido */
         printf("\nO valor lido foi: %lf\n", valorLido);
      }
   } while(nValoresLidos < 3);

   return 0;
}
```

Comentários sobre a função main()

A função **main()** apresentada aqui funciona basicamente de modo semelhante à função correspondente apresentada na versão da **Seção 3.7**. Devido á simplicidade desta função, os comentários necessários para seu completo entendimento são incluídos em seu corpo.

Apresentar o valor lido ao usuário não é tão crítico na função **main()** apresentada aqui quanto era na versão anterior apresentada na **Seção 3.7**. Aquela versão podia ler valores que não correspondiam à intenção do usuário, mas a nova versão apresentada aqui não possui esta limitação. Portanto, a apresentação do valor lido aqui foi feita apenas a título de ilustração. Num caso real, você substituiria a última chamada de **printf()** por uma chamada de função que processasse o valor lido.

Exemplo de Interação

A seguir, é apresentado um exemplo de sessão de interação de um programa que utiliza as funções implementadas aqui (caracteres em negrito representam entrada de dados do usuário).

```
Introduza valores de ponto-flutuante entre 10 (inclusive) e 20.
Serao lidos tres valores validos.


Introduza um numero de ponto flutuante: 2o.0
Entrada ilegal. Foi encontrado um erro na posicao indicada abaixo:
    2o.0
     ^
Introduza um numero de ponto flutuante: 2e19

O valor lido foi: 2000000000000000000.000000

Introduza um numero de ponto flutuante: 5e455
Result too large

Entrada ilegal. Foi encontrado um erro na posicao indicada abaixo:
    5e455
        ^
Introduza um numero de ponto flutuante: 5e-10
O valor lido foi: 0.000000

Introduza um numero de ponto flutuante: 5e-455
Result too large

Entrada ilegal. Foi encontrado um erro na posicao indicada abaixo:
    5e-455
         ^
Introduza um numero de ponto flutuante: -5
O valor -5.000000 nao satisfaz os criterios do programa.

Introduza um numero de ponto flutuante: 19.99999999999
O valor lido foi: 20.000000
```

Limitações da Implementação

A seguir são enumeradas limitações conhecidas da implementação de leitura e validação de dados apresentada aqui.

A Função LeValor() Funciona para um Tipo de Dados Apenas

Claramente, a função `LeValor()` funciona adequadamente apenas para leitura de valores do tipo **double**. A seguir são propostas alternativas para superar esta limitação:

Transformar a função em macro. Esta proposta é inconveniente para uma função tão complexa quanto a função `LeValor()`, além de padecer de todos os outros males que acometem as macros (v. **Seção 5.3**).

Utilizar um molde (*template*) de função. Esta é uma boa solução, mas moldes existem apenas em C++.

Escrever uma função para cada tipo de dados possível. Em C, esta é a melhor solução. Apesar de parecer ser a mais trabalhosa, em última instância, você vai ler apenas números inteiros e de ponto-flutuante. Portanto, você irá precisar definir apenas duas ou três funções uma ou duas funções para leitura de valores inteiros e uma função para leitura de valores de ponto-flutuante. Consulte o **Volume II** para conhecer outras funções de conversão que poderiam ser utilizadas.

A Função LeValor() Pode Imprimir Mensagens de Erro Ilegíveis

A função `IndicaErro()` imprime a mensagem de erro do sistema (em inglês e sem muita clareza) quando **errno** é diferente de zero. Isto ocorre, por exemplo, quando a tentativa de conversão gera uma condição de *overflow*. Evidentemente, estas mensagens não são convenientes para um usuário comum. Uma solução simplista seria suprimir estas mensagens substituindo o seguinte trecho da função `IndicaErro()`:

```
if (errno != 0) {
   ImprimeMensagemDeErro("%s\n", strerror(errno));
   ImprimeMensagemDeErro(mensagem);
} else
   ImprimeMensagemDeErro(mensagem);
```

por simplesmente:

```
ImprimeMensagemDeErro(mensagem);
```

Mas, se esta solução for adotada, perde-se parte da capacidade de apontar erros precisamente. Este problema precisa ser analisado mais profundamente, mas esta análise está além do escopo deste livro.

A Solicitação de Dados ao Usuário É Imprecisa

O prompt apresentado pela função `LeValor()` na linha 6 é fixo e não corresponde precisamente à especificação de valores válidos esperados pelo programa.

A solução para este problema é simples. Primeiro, acrescente um novo argumento que represente um *string* de *prompt* na função `LeValor()`. Assim, o protótipo da função `LeValor()` tornar-se-á:

```
unsigned LeValor( double *valor, unsigned (*pfVerifica)(double),
                  const char *prompt );
```

Feita esta mudança, altera-se a instrução 6 da função `LeValor()` para:

```
printf(prompt);
```

Agora, basta você chamar a função `LeValor()` utilizando o *prompt* desejado. Por exemplo:

```
teste = LeValor(&valorLido, VerificaValor2,
                "Introduza um numero real par: ");
```

Quando um Valor Não É Validado, a Mensagem Apresentada É Imprecisa

Este problema é semelhante ao anterior e sua solução também é semelhante àquela proposta de solução. Primeiro, acrescente um argumento que represente um *string* de erro de validação no cabeçalho da função `LeValor()`. Assim, o protótipo da função `LeValor()` tornar-se-á:

```
unsigned LeValor(double *valor, unsigned (*pfVerifica)(double),
                 const char *prompt, const char *msgErro);
```

Depois, altera-se a instrução na linha 23 da função `LeValor()` para:

```
ImprimeMensagemDeErro("Valor introduzido: %lf. %s\n",
                      *valor, msgErro);
```

Agora, a função `LeValor()` pode ser chamada com o prompt e a mensagem de erro desejados. Por exemplo:

```
teste = LeValor(&valorLido, VerificaValor2,
                "Introduza um numero real par: ",
                "Este valor nao e' par.");
```