CAPÍTULO 10
CONSTRUÇÕES COMPLEXAS

10.1 Introdução

Os criadores de C agiram de forma muito parcimoniosa na escolha dos elementos de composição da linguagem. Por exemplo, enquanto algumas linguagens utilizam `BEGIN` e `END` como delimitadores de blocos, C utiliza simplesmente { e } com o mesmo objetivo. Outro exemplo: em algumas linguagens, um ponteiro é declarado utilizando `POINTER TO` como construtor de tipo; em C, o construtor de tipo correspondente é simplesmente *****. Assim, construções envolvendo ponteiros podem tornar-se muito complicadas.

Uma conseqüência dessa economia de símbolos é que C permite a escrita de programas inteiros criptográficos. Existe até mesmo uma competição bastante popular entre membros da comunidade de programadores de C, denominada ***International Obfuscated C Code Contest***[1], dedicada aos amantes da escrita de programas em C quase indecifráveis. Um exemplo de programa apresentado por um dos contendores é:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

O objetivo deste capítulo não é introduzi-lo na ***arte*** de escrita de programas ininteligíveis. Programas como o que acaba de ser apresentado

[1] Maiores informações sobre esta competição são encontradas no site ***www.ioccc.org***.

são curiosos, e examiná-los pode até ser divertido, mas não satisfazem os critérios de legibilidade, manutenibilidade e portabilidade que se deve procurar alcançar. Portanto, o objetivo deste capítulo será ensiná-lo a interpretar e evitar o quanto possível algumas construções da linguagem C que, devido à sua natureza, são consideradas complexas.

Este capítulo discute o conceito de ponteiro para função. Este importante tópico de programação em C permite a construção de programas que seriam difíceis de criar sem ele. A última seção apresenta um uso prático de ponteiros para funções. Ela apresenta uma versão bem mais sofisticada de leitura e validação de dados apresentada na **Seção 3.7**. Esta seção é longa e pode ser omitida dependendo da ênfase do curso ou de disponibilidade de tempo.

10.2 Operadores e Declaradores

O conceito de operadores foi apresentado logo no **Capítulo 1**, dada a sua importância fundamental para o restante do material. O conceito de declaradores ainda não foi formalmente apresentado, apesar de já ter sido vastamente utilizado. Um **declarador** serve simplesmente para construir tipos. Esta seção apresenta o conceito de operador de memória e discute declaradores que freqüentemente são confundidos com operadores.

10.2.1 Operadores de Memória

Operadores de memória são operadores utilizados em referências a posições e conteúdos de memória. Todos estes operadores já foram exaustivamente discutidos anteriormente e serão incluídos aqui apenas por uma questão de complemento e para facilidade de referência.

Os operadores de memória de C, juntamente com seus significados, são apresentados na **Tabela 32**.

OPERADOR	NOME	USO	INTERPRETAÇÃO
&	Endereço	&x	O endereço da variável x
*	Indireção	*p	Conteúdo armazenado na posição apontada por p
[]	Elemento de array	a[i]	Conteúdo do elemento do array a de índice i
()	Chamada de função	F()	Chamada da função F()
.	Campo de estrutura	e.c	Conteúdo do campo c da estrutura e
->	Campo de estrutura	p->c	Conteúdo do campo c da estrutura apontada por p.

Tabela 32: Operadores de memória

Os operadores **[]**, **()**, **->** e **.** (ponto) fazem parte de um mesmo grupo de precedência. Este grupo tem a maior precedência dentre todos os operadores da linguagem C. A associatividade destes operadores é da esquerda para a direita. Os operadores unários **&** e ***** fazem parte do grupo de precedência que reúne todos os operadores unários. Este grupo de precedência segue imediatamente o grupo dos operadores **[]**, **()**, **->** e **.** em ordem de precedência e a associatividade dos operadores deste grupo é da direita para a esquerda.

10.2.2 Declaradores

Declaradores (ou **construtores de tipos**) são utilizados para declarar ou definir tipos, variáveis ou funções. Alguns declaradores usados em C compartilham símbolos utilizados como operadores[2]. Isto é, os conjuntos de símbolos **[]**, **()** e *****, utilizados como operadores, conforme visto anteriormente, são também utilizados como declaradores, conforme mostra a **Tabela 33**.

[2] Os demais declaradores de C usam as palavras-chave **struct**, **union** e **enum**.

SÍMBOLO	OPERADOR DE...	DECLARADOR DE...
*	indireção	ponteiro
[]	acesso a elemento de array	array
()	chamada de função	função

Tabela 33: Operadores e declaradores correspondentes

Fazer distinção num programa entre declaradores e operadores correspondentes é fácil, basta verificar o tipo de construção na qual um desses símbolos aparece. Isto é, se a construção trata-se de uma declaração ou definição, o símbolo está sendo usado como declarador; caso contrário, se o símbolo é utilizado numa expressão, ele representa um operador.

Além de compartilharem símbolos, esses declaradores e operadores são semelhantes sob outros aspectos. Ou seja, eles também possuem precedência e associatividade. A **Tabela 34** apresenta precedências e associatividades dos declaradores mostrados na **Tabela 33**.

GRUPO DE DECLARADORES	PRECEDÊNCIA	ASSOCIATIVIDADE
[] e **()**	Mais alta	À esquerda
*	Mais baixa	À direita

Tabela 34: Precedência e associatividade de declaradores

Para interpretar declarações que envolvem o uso de mais um desses declaradores, é preciso estar ciente das propriedades de precedência e associatividade apresentadas na **Tabela 34**.

10.3 Arrays de Ponteiros

Existem situações em programação nas quais se faz necessário utilizar um array de ponteiros. Considere, por exemplo, a seguinte declaração:

```
char  *arrayDePonteiros[5];
```

A variável `arrayDePonteiros` é um array de ponteiros para caracteres (e não um ponteiro para um array de caracteres), pois **[]** tem maior precedência do que *****, conforme visto na **Seção 10.2**.

Arrays de ponteiros são utilizados com freqüência como arrays de ***strings*** constantes. Considere o seguinte exemplo, no qual a função `ImprimeMes()` deve receber como entrada um inteiro não-negativo (entre 1 e 12), representando a ordem de um mês, e imprimir o nome do mês correspondente.

```
void  ImprimeMes(unsigned  int  m)
{
    static char *mes[13] = {"", "janeiro", "fevereiro", "marco",
"abril", "maio", "junho", "julho", "agosto", "setembro", "outubro",
"novembro", "dezembro"};
   if ( !m || (m > 12) )
      printf("Valor ilegal para numero de mes.\n");
   else
      printf("%s\n", mes[m]);
}
```

A função `ImprimeMes()` tem funcionamento bastante simples. A variável `mes` é um array de ponteiros para o tipo **char** e, em conformidade com sua iniciação, cada ponteiro aponta para o início de um ***string***.

A única coisa que pode parecer estranha à primeira vista com relação a este exemplo é que a variável `mes` foi declarada como sendo um array de 13 ponteiros, em vez de 12, como parece ser mais sensato. A razão para o acréscimo do ***string*** "" como endereço do elemento inicial do array é que ele torna o código mais eficiente à custa de apenas um byte de memória, que é o espaço ocupado por este ***string***, mais o espaço necessário para armazenar um ponteiro. Em outras palavras, a razão pela qual um ponteiro extra com um valor inútil foi utilizado é que não se torna necessário subtrair 1 do índice `m` quando o nome do mês é impresso na segunda chamada de **printf()**. Mais precisamente, se o array fosse declarado com 12 ponteiros, em vez de 13, a segunda chamada de **printf()** deveria ser escrita como:

```
printf("%s\n", mes[m - 1]);
```

Assim, este exemplo ilustra uma prática comum em programação em C, que consiste em descartar o elemento inicial de um array quando os índices começam naturalmente em 1, como a ordem dos meses do ano. A única desvantagem de proceder assim é que deve-se alocar espaço para um elemento do array que nunca é utilizado. Por outro lado, operações aritméticas adicionais tornam a execução do programa mais lenta e aumentam o tamanho do código gerado pelo compilador.

10.4 Ponteiros para Ponteiros

Ponteiro para ponteiro é uma construção utilizada com freqüência em programação em C[3]. Um ponteiro que aponta para um ponteiro pode ser declarada precedendo-se o nome da variável por dois asteriscos. Por exemplo:

```
double  **p;
```

define `p` como sendo um ponteiro para um objeto do tipo **double**. Para acessar o valor apontado por `p`, é necessária uma indireção dupla. Por exemplo:

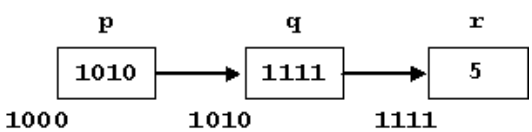
```
double d = **p;
```

atribui um valor do tipo **double** a `d`.

Agora, considere as seguintes definições:

```
int   r = 5;
int  *q = &r;
int **p = &q;
```

Estas definições resultam numa alocação de memória que poderia ser representada como no esquema a seguir:



[3] Os sistemas operacionais Mac OS e Palm OS, por exemplo, utilizam ponteiros para ponteiros (denominados ***handles***) num esquema de gerenciamento de memória que tenta reduzir os efeitos de fragmentação de ***heap*** (v. **Seção 11.3**).

No esquema acima, os números escritos abaixo dos retângulos representam endereços (arbitrários) das variáveis (escritas acima dos retângulos), enquanto os números dentro dos retângulos representam os valores das variáveis.

Note, ainda, que, no último exemplo, existem três maneiras equivalentes de se atribuir o valor `10` à variável `r`:

```
r = 10;   /* Acesso direto */
*q = 10;  /* Acesso indireto */
**p = 10; /* Acesso duplamente indireto */
```

Ponteiros para ponteiros são usados com freqüência como parâmetros formais quando se deseja modificar o valor do próprio ponteiro (e não o valor do ***conteúdo*** apontado); i.e., quando o ponteiro representa um parâmetro de saída ou de entrada/saída. Por exemplo, considere os protótipos das funções `F1()`, `F2()` e `F3()` a seguir:

```
void F1(char *p)
```

• A função `F1()` poderá alterar o conteúdo apontado pelo ponteiro recebido como argumento, mas não poderá alterar o próprio ponteiro.

```
void F2(char **p)
```

• A função `F2()` poderá alterar tanto o conteúdo do ponteiro cujo endereço é recebido como argumento quanto o conteúdo apontado por este ponteiro.

```
void F3(const char **p)
```

• A função `F3()` poderá alterar o conteúdo do ponteiro cujo endereço é recebido como argumento, mas, devido ao uso de **const**, não poderá modificar o conteúdo apontado por este ponteiro.

Para tornar este último exemplo mais palpável, suponha que se tenham as seguintes definições:

```
char ar[20] = "bola";
char *str = ar;
```

Então, as funções `F1()`, `F2()` e `F3()` poderiam ser chamadas conforme mostrado a seguir:

```
F1(str); /* O conteúdo do string para onde str aponta pode ser */
         /* alterado por F1(), mas o ponteiro str não pode */

F2(&str); /* O conteúdo do string para onde str aponta pode ser */
        /* alterado por F2(), assim como o ponteiro str */

F3(&str); /* O conteúdo do string para onde str aponta não pode */
        /* ser alterado por F3(), mas o ponteiro str pode */
```

Exemplos práticos de uso de ponteiros para ponteiros como argumentos de função são apresentados no **Capítulo 11**.

10.5 Ponteiros para Funções

Ponteiros para funções constituem uma ferramenta bastante poderosa em programação em C. Antes de introduzir este conceito, entretanto, é necessário que se examine mais profundamente o modo como o compilador de C interpreta uma chamada de função.

Não apenas variáveis são armazenadas na memória do computador durante a execução de um programa. O próprio código executável do programa também é armazenado em memória. Conseqüentemente, do mesmo modo que cada variável possui um endereço em memória, cada instrução em linguagem de máquina de um programa possui um endereço. Quando uma função é definida, o compilador de C trata o nome da função como sendo o endereço da função em memória. Ou seja, o nome da função é tratado como um ponteiro para a primeira instrução da função em linguagem de máquina. Uma chamada da função no programa causa a transferência do fluxo de execução do programa para este endereço.

Assim como ocorre com variáveis, o endereço de uma função não pode ser modificado num programa. Entretanto, pode-se ter um ponteiro que aponte para várias funções em diferentes pontos de um programa, do mesmo modo que se pode ter um ponteiro capaz de apontar para várias variáveis em instantes diferentes.

10.5.1 Definições de Ponteiros para Funções

Para definir um ponteiro para função, deve-se preceder o nome da variável por asterisco, como na declaração de qualquer ponteiro, mas, aqui, deve-se também colocar o asterisco e o ponteiro entre parênteses. A declaração termina com um par de parênteses contendo os tipos dos argumentos das funções para onde o ponteiro pode apontar[4]. Portanto, uma definição de ponteiro para função assume a seguinte forma:

tipo (**nome-do-ponteiro*)(*tipos-dos-argumentos*);

onde ***tipo*** denota o tipo de retorno das funções para as quais o ponteiro pode apontar e ***tipos-dos-argumentos*** representa a lista dos tipos dos argumentos destas funções.

Como exemplo de definição de um ponteiro para função, considere:

```
int  (*pf)(float);
```

que define `pf` como sendo um ponteiro para funções cujo tipo de retorno é **int** e que têm apenas um argumento do tipo **float**. Isto significa dizer, por exemplo, que `pf` pode apontar para uma função `F1()` com um argumento do tipo **float** cujo tipo de retorno é **int**, mas não pode apontar para uma função `F2()` sem argumento nem para uma função `F3()` com um argumento do tipo **float**, mas cujo tipo de retorno é **float**.

[4] Esta declaração de tipo é semelhante àquela que aparece numa alusão de função e, como ocorrem alusões, o programador pode optar por não declarar os tipos dos parâmetros, obtendo, assim, um ponteiro que pode apontar para qualquer função que tenha o tipo de retorno especificado (i.e., funções com quaisquer tipos de argumentos). Esta prática, no entanto, não é recomendável e não será apresentada aqui.

Note que, se os parênteses em torno de `*pf` na definição apresentada no exemplo anterior forem omitidos, o compilador tratará a declaração como se fosse uma alusão a uma suposta função `pf()` cujo argumento é do tipo **float** e cujo tipo de retorno é um ponteiro para o tipo **int**.

10.5.2 Atribuição de Valores a um Ponteiro para Função

Como o compilador trata nomes de funções como endereços, atribuir um valor a um ponteiro para função requer simplesmente atribuir ao ponteiro um nome de função compatível com o ponteiro. Por exemplo:

```
extern  int  F1(float); /* Alusão da função F1() */
int  (*pf)(float); /* Ponteiro para função que recebe um float */
                   /* como argumento e retorna um valor int  */

pf = F1; /* pf passa a apontar para a função F1() */
```

Outras possibilidades de atribuição a `pf` consideradas incorretas são exemplificadas a seguir:

```
pf = F1(2.5);
```

• Esta instrução não é ilegal, mas é uma conversão para ponteiro dependente de implementação, pois `F1(2.5)` é uma chamada de função que retorna **int** e, portanto, está tentando-se atribuir um valor do tipo **int** a um ponteiro.

```
pf = &F1(2.5);
```

• A expressão `&F1(2.5)` é ilegal, pois está tentando-se obter o endereço do valor retornado por `F1()`.

```
pf = &F1;
```

• Estritamente falando, a expressão `&F1` é incorreta, pois está tentando-se obter o endereço de um endereço, mas compiladores ISO não irão considerar isto um erro. Eles simplesmente ignorarão o operador **&**.

Outros pontos importantes que o programador deve considerar quando atribui um valor a um ponteiro para função são:

(1) Os tipos de retorno na declaração da função e na declaração do ponteiro para função devem ser os mesmos.

(2) Os tipos dos argumentos na declaração da função e na declaração do ponteiro para função devem ser os mesmos.

Por exemplo, se foi declarado um ponteiro para função com argumento **float** e que retorna um valor **int**, deve-se atribuir a este ponteiro o endereço de uma função cujos tipos de argumentos e de retorno sejam respectivamente iguais a estes tipos. Por exemplo, dadas as seguintes declarações:

```
extern  float  F2(float);
extern  int  F3(void);
int  (*pf)(float);
```

os resultados das seguintes atribuições seriam indefinidos:

```
pf = F2;
```

• `pf` e `F2` são incompatíveis, pois seus tipos de retorno são diferentes.

```
pf = F3;
```

• `pf` e `F3` são incompatíveis, pois seus tipos de argumentos são diferentes.

Apesar das incompatibilidades apontadas nas atribuições acima, elas são legais; i.e., elas compilam num compilador padrão de C. Em tais casos, o padrão ISO de C apenas requer que o compilador emita uma mensagem de advertência (v. **Seção 3.2.5**).

Conversões entre ponteiros para funções que têm tipos de parâmetros diferentes efetuadas com o uso de conversão explícita evitam que o compilador emita mensagens de advertência. Mas, se uma tal conversão for usada para chamar a função apontada, o comportamento é indefinido. Considere os seguintes exemplos:

```
extern int F1(void);
int (*fptr1) (void) = F1; /* OK */
extern int F2(int);
int (*fptr2) (int) = F2; /* OK */

fptr1 = (int (*) (void))F2; /* OK */

(*fptr1)(); /* Resultado indefinido, pois F2() */
            /* deve receber um argumento int   */
```

Um ponteiro para função não deve ser convertido num ponteiro para um tipo de dados, ou vice-versa, pois o resultado é indefinido. Por exemplo:

```
extern int F1(void);
int *p = F1; /* Compila, mas o resultado é indefinido */
```

10.5.3 Chamada de Função Mediante Ponteiro

Chamar uma função por meio de um ponteiro ao qual se atribuiu o endereço dela é similar a chamá-la utilizando o próprio nome da função. Por exemplo:

```
extern  int  F1(char c);
int          (*pf)(char);
int          resultado;
char         a;

pf = F1;

resultado = pf(a);  /* Chama a função F1() por meio do ponteiro */
                    /* pf, passando a como  argumento */
```

Uma outra notação que pode ser utilizada para chamar uma função utilizando um ponteiro para função consiste em preceder o ponteiro por * e envolver ambos, o ponteiro e o asterisco, com parênteses. Utilizando esta sintaxe, a chamada de função do último exemplo poderia ser escrita como[5]:

```
resultado = (*pf)(a);
```

[5] Compiladores de C muito antigos aceitam apenas este formato de chamada. Além disso, alguns programadores de C preferem-no por considerá-lo mais legível, pois chama a atenção para o fato de que `pf` é um ponteiro para função e não um nome de função.

Observe que, se, no último exemplo, o par de parênteses em torno de `*pf` fosse omitido, o lado direito da atribuição seria interpretado como:

```
*(pf(a))
```

pois uma chamada de função tem precedência maior do que a do operador de indireção. Esta última expressão é, obviamente, ilegal, tendo em vista a declaração de `pf`.

10.5.4 Retornando Ponteiros para Funções

Uma função pode retornar um ponteiro para função. No exemplo a seguir, a função `EscolheFuncao()` é definida com o argumento `condicao` do tipo **unsigned** e retorna um ponteiro para uma função que recebe um argumento do tipo **float** e cujo tipo de retorno é **int**.

```
extern  int  F1(float);
extern  int  F2(float);

int  (*EscolheFuncao(unsigned condicao))(float)
{
   if (condicao)
       return  F1;

   return  F2;
}
```

A função `EscolheFuncao()` do exemplo acima, retorna um ponteiro para a função `F1()` se o argumento `condicao` é diferente de zero ou um ponteiro para a função `F2()` em caso contrário (lembre-se de que `F1` e `F2` são endereços de funções e, portanto, são compatíveis com ponteiros para funções). A função `EscolheFuncao()` poderia ser chamada conforme ilustrado no fragmento de programa a seguir:

```
int  (*pf)(float);
...
pf = EscolheFuncao(2);
```

O uso de definições de tipo facilita a definição de funções que retornam ponteiros para funções e favorece a legibilidade. Por exemplo,

usando uma definição de tipo, a função `EscolheFuncao()` poderia ser redefinida como:

```
typedef int (*tFuncPtr) (float);

tFuncPtr EscolheFuncao2(unsigned condicao)
{
   if (condicao)
       return  F1;

   return  F2;
}
```

Claramente, este conjunto de declarações é mais inteligível do que a definição anterior da função `EscolheFuncao()`.

10.5.5 Ponteiros para Funções como Parâmetros de Funções

O uso mais comum de ponteiros para funções é como parâmetros de funções. Uma situação típica é o uso de ponteiros para funções como argumentos de funções que implementam algoritmos de integração numérica. Neste caso, o argumento que é um ponteiro para função representa exatamente a função a ser numericamente integrada. Um exemplo prático de uso de ponteiros para funções como argumentos de funções será apresentado na **Seção 10.9**.

Como parâmetro formal na definição de uma função, um ponteiro para função deve ser declarado exatamente do mesmo modo apresentado na **Seção 10.5.1**. Por exemplo:

```
void FuncaoComPonteiroParaFuncao(float f, int (*pf)(float))
{
       /* Corpo da função FuncaoComPonteiroParaFuncao() */
}
```

Na chamada de uma função que tem um ponteiro para função como um de seus argumentos, deve-se utilizar como parâmetro real correspondente a este argumento apenas o nome de uma função compatível com ele. Por exemplo, suponha que se tenham as seguintes definições de funções no mesmo programa da função `FuncaoComPonteiroParaFuncao()` do exemplo acima:

```
float   x;

int  F1(float umFloat)
{
 /* Corpo da função F1() */
}

int  F2(float umFloat)
{
 /* Corpo da função F2() */
}

void  F3(float umFloat)
{
 /* Corpo da função F3() */
}
```

Então, as seguintes chamadas seriam perfeitamente legais:

```
FuncaoComPonteiroParaFuncao(x, F1);
FuncaoComPonteiroParaFuncao(x, F2);
```

Entretanto, a chamada:

```
FuncaoComPonteiroParaFuncao(x, F3);
```

produziria um resultado indefinido, pois a função `F3()` não é compatível com o segundo argumento da função `FuncaoComPonteiroParaFuncao()`.

10.6 Interpretando Construções Complexas

Algumas vezes, declarações em C tornam-se tão complexas que fica difícil entender o que está sendo declarado. Considere, por exemplo, a seguinte declaração:

```
char  *(*(*x)())[5];
```

Ela define a variável `x` como sendo um ponteiro para uma função que retorna um ponteiro para um array com 5 ponteiros para o tipo **char**.

Uma forma de evitar declarações complexas como a desse exemplo é criar definições de tipos intermediárias, como mostrado a seguir:

```
typedef  char  *tAPC[5]; /* Array com 5 ponteiros para chars */
typedef  tAPC  *tFPAPC(); /* Função retornando um ponteiro para
*/
                /* um array com 5 ponteiros para chars */
tFPAPC  *x; /* Equivalente à declaração: char  *(*(*x)())[5]; */
```

Na maioria das vezes, declarações difíceis de ser decifradas são compostas pelos declaradores de ponteiro *****, array **[]** e função **()**. Portanto, antes da apresentação de um procedimento para entendimento e composição de declarações complexas envolvendo estes operadores, é conveniente recordar as regras de precedência e associatividade destes declaradores:

- Os declaradores de array **[]** e de função **()** têm a mesma precedência, e esta precedência é maior do que a precedência do operador de ponteiro *****.
- Os declaradores **[]** e **()** são associados da esquerda para a direita, enquanto o declarador * é associado da direita para a esquerda.

A melhor estratégia para decifrar declarações complexas é começar com o nome da variável sendo declarada e, então, examinar cada parte da declaração a partir do declarador de maior precedência que esteja mais próximo da variável. Esta regra é alterada pela presença de parênteses que modifiquem a precedência dos declaradores. Considere, por exemplo, a seguinte declaração:

```
char  *x[8];
```

Aplicando o procedimento descrito acima, esta declaração poderia ser decifrada por meio dos seguintes passos:

1. `x[8]` é um array
2. `*x[8]` é um array de ponteiros
3. `char *x[8]` é um array de ponteiros para o tipo **char**

Parênteses podem ser utilizados para modificar a ordem de precedência de declaradores numa declaração. Por exemplo, considere a declaração:

```
int  (*x[8])();
```

que poderia ser decifrada por meio dos seguintes passos:

1. `x[8]` é um array
2. `(*x[8])` é um array de ponteiros
3. `(*x[8])()` é um array de ponteiros para funções
4. `int (*x[8])()` é um array de ponteiros para funções que retornam **int**

Se esta última declaração tivesse sido escrita sem parênteses:

```
int  *x[8]();
```

ela seria interpretada como:

1. `x[8]` é um array
2. `x[8]()` é um array de funções
3. `*x[8]()` é um array de funções que retornam ponteiros
4. `int *x[8]()` é um array de funções que retornam ponteiros para **int**

Portanto, esta última declaração seria inválida, uma vez que a linguagem C não permite arrays de funções.

10.7 Compondo Construções Complexas

Construções complexas são compostas utilizando a mesma abordagem descrita na **Seção 10.6** para decifrá-las. Suponha, por exemplo, que se deseje um ponteiro para um array de ponteiros para funções que retornam ponteiros para arrays de elementos do tipo **char**. Uma declaração com este significado poderia ser composta utilizando os seguintes passos:

1. `(*x)` é um ponteiro
2. `(*x)[]` é um ponteiro para um array
3. `(*(*x)[])` é um ponteiro para um array de ponteiros
4. `(*(*x)[])()` é um ponteiro para um array de ponteiros para funções

5. `(*(*(*x)[])())` é um ponteiro para um array de ponteiros para funções que retornam ponteiros

6. `(*(*(*x)[])())[]` é um ponteiro para um array de ponteiros para funções que retornam ponteiros para arrays

7. `char (*(*(*x)[])())[]` é um ponteiro para um array de ponteiros para funções que retornam ponteiros para arrays de elementos do tipo **char**

Note que, no último exemplo, devido à precedência mais baixa do declarador ***** em relação aos declaradores **[]** e **()**, um par de parênteses é acrescentado sempre que um declarador ***** é utilizado.

Declarações e instruções complexas são sempre condenáveis[6]. Isto é, em termos de legibilidade, a melhor estratégia é mesmo evitar o uso de declarações ou instruções complexas, como aquela apresentada aqui, por meio do uso de definições de tipos intermediários, conforme foi descrito na **Seção 10.6**.

10.8 Algoritmos de Classificação I

Classificar (ou **ordenar**) uma lista de valores em ordem alfabética ou numérica é uma operação bastante comum em programação. Embora o processo manual de classificar valores seja relativamente fácil, muitos algoritmos de classificação para implementação via computador não são triviais. Existem vários algoritmos de classificação, e aquele que será apresentado aqui é um dos mais ineficientes em termos de uso de recursos computacionais. Por outro lado, este algoritmo, denominado **método da bolha** (***bubble sort***), é talvez o mais fácil de ser entendido[7].

10.8.1 O Método da Bolha

A idéia que norteia o algoritmo de ordenação ***bubble sort*** é a de comparar elementos adjacentes do array que representa a lista a ser ordenada, a partir

[6] A não ser que sua intenção seja participar do *International Obfuscated C Code Contest*.

[7] A biblioteca de C oferece uma função mais eficiente, denominada **qsort()**, que implementa o método de classificação conhecido como ***quick sort***. Esta função será apresentada na **Seção 11.9**.

dos primeiros dois elementos, e trocá-los quando o elemento de índice menor é maior do que o elemento de índice maior (ou o contrário, quando se deseja classificação em ordem decrescente, ao invés de em ordem crescente). Após comparar os dois primeiros elementos, compara-se o segundo elemento com o terceiro, então o terceiro com o quarto e assim por diante até que o final do array seja atingido. Uma **passagem** pelo array consiste em uma seqüência de comparações entre elementos adjacentes, do início ao final do array. Numa dada passagem, caso haja pelo menos uma troca entre elementos, uma nova passagem se faz necessária. O algoritmo ***bubble sort*** requer que sejam feitas passagens pelo array até que o mesmo esteja completamente ordenado; i.e., até que se consiga uma passagem pelo array sem que haja troca alguma de elementos.

A função `BubbleSort()` a seguir implementa este algoritmo para um array de elementos do tipo **int**.

```
#define  FALSE  0
#define  TRUE   1

void BubbleSort(int lista[], unsigned  int  tamanhoDaLista)
{
   unsigned  i, ordenada = FALSE;
   int       aux;

      /* Enquanto a lista não estiver ordenada, */
      /* continua executando passagens por ela  */
   while (!ordenada){
      ordenada = TRUE; /* Supõe que a lista está ordenada */
      for (i = 0; i < tamanhoDaLista - 1; i++){
            if (lista[i] > lista[i+1]){ /* Compara elementos
adjacentes */
          ordenada = FALSE; /* Pelo menos um elemento fora de
ordem */

               /* Troca elementos adjacentes */
          aux = lista[i];
          lista[i] = lista[i+1];
          lista[i+1] = aux;
        }  /* final do if */
      } /* final do for */
   } /* final do while */
} /* final da funcao BubbleSort */
```

A função **main()** a seguir chama a função `BubbleSort()` para ordenar um array de elementos inteiros:

```
int main(void)
{
   int  arrayDeInts[] = {12, 55, 21, 1, 6, 8, 17, 220, 5, 83};
   int  i, tamanhoAr;

   tamanhoAr = sizeof(arrayDeInts)/sizeof(arrayDeInts[0]);

   for (i=0; i < tamanhoAr; i++) { /* Imprime array original */
      printf(" %d ", arrayDeInts[i]);
   }

   BubbleSort(arrayDeInts, tamanhoAr);

   printf("\n\n\n");

   for (i=0; i < tamanhoAr; i++) { /* Imprime array ordenado */
      printf(" %d ", arrayDeInts[i]);
   }

   return 0;
}
```

Para melhorar o entendimento do algoritmo seguido pela função `BubbleSort()`, é útil incluir, imediatamente antes do laço **for** na função `BubbleSort()`, um outro laço **for** que imprime os elementos do array sendo ordenado, a cada passagem do algoritmo.

Exercício: Acrescente um laço **for** na função `BubbleSort()` que imprima os elementos do array sendo ordenado a cada passagem pelo array. Implemente esta mudança juntamente com o programa do último exemplo e examine cuidadosamente as posições dos elementos do array a cada passagem do algoritmo.

10.8.2 Generalizando a Função BubbleSort() I

Freqüentemente, ponteiros para funções são utilizados como um mecanismo para executar várias operações similares sem a necessidade de duplicação de código. Suponha que você deseje classificar um array

de elementos do tipo **int** em ambas as ordens, crescente e decrescente. Obviamente, uma forma de se fazer isto seria escrever duas funções de classificação: uma para classificar em ordem crescente (como a função `BubbleSort()` apresentada acima) e outra para classificar em ordem decrescente.

Se você observar atentamente a função `BubbleSort()` apresentada anteriormente, verificará que o que determina a ordem crescente obtida pela função é apenas a comparação:

```
lista[i] > lista[i+1]
```

Portanto, uma função para classificar em ordem decrescente teria como única diferença da função `BubbleSort()` a substituição da comparação:

```
lista[i] > lista[i+1]
```

por:

```
lista[i] < lista[i+1]
```

Seria, portanto, um grande desperdício escrever duas funções que diferem apenas por um operador relacional. Uma outra forma de resolver o problema seria substituir a comparação `lista[i] > lista[i+1]` na função `BubbleSort()` pela chamada de uma função, denominada `Compara()`, como:

```
Compara(lista[i], lista[i+1])
```

Quando a ordem de classificação fosse crescente, esta função deveria retornar 1 quando `lista[i] > lista[i+1]` e 0 em caso contrário. Se a ordem de classificação fosse decrescente, esta função deveria retornar 1 quando `lista[i] < lista[i+1]` e 0 em caso contrário. Portanto, na realidade, seriam necessárias duas funções:

```
int   ComparaCrescente(int   a, int  b)
{
    return a > b;
}
```

e

```
int   ComparaDecrescente(int   a, int  b)
{
    return a < b;
}
```

Utilizando ponteiros para funções, o problema poderá ser finalmente resolvido. Mais especificamente, utiliza-se como argumento adicional um ponteiro para função que possa apontar ora para `ComparaCrescente()` ora para `ComparaDecrescente()`, de acordo com a classificação desejada. Assim, um novo argumento, denominado `Compara`, será acrescentado à lista de parâmetros de `BubbleSort()` para indicar se a classificação será em ordem crescente ou decrescente. A função `BubbleSort()` generalizada desta maneira pode então ser reescrita como apresentado a seguir:

```
void BubbleSort2( int lista[], unsigned  int  tamanhoDaLista,
                  int  (*Compara)(int, int) )
{
   unsigned  i, ordenada = FALSE;
   int       aux;

      /* Enquanto a lista não estiver ordenada, */
      /* continua executando passagens por ela  */
   while (!ordenada){
     ordenada = TRUE; /* Supõe que a lista está ordenada */

      for (i = 0; i < tamanhoDaLista - 1; i++){
     if (Compara(lista[i], lista[i+1])){ /* Compara vizinhos */
        ordenada = FALSE;  /* Um elemento está fora de ordem */

               /* Troca elementos adjacentes */
            aux = lista[i];
            lista[i] = lista[i+1];
            lista[i+1] = aux;
         }  /* if */
      } /* for */
   } /* while */
} /* BubbleSort */
```

A função **main()** a seguir chama a função `BubbleSort2()` com

um array de elementos inteiros para classificá-lo nas ordens crescente e decrescente:

```
int main(void)
{
   int  arrayDeInts[] = {12, 55, 21, 1, 6, 8, 17, 220, 5, 83};
   int  i, tamanhoAr;

 tamanhoAr = sizeof(arrayDeInts)/sizeof(arrayDeInts[0]);

   printf("Ordem original:\n");
   for (i=0; i < tamanhoAr; i++) {
     printf(" %d ", arrayDeInts[i]);
   }

  BubbleSort2(arrayDeInts, tamanhoAr, ComparaCrescente);

   printf("\n\nOrdem crescente:\n");
   for (i=0; i < tamanhoAr; i++) {
     printf(" %d ", arrayDeInts[i]);
   }
  BubbleSort2(arrayDeInts, tamanhoAr, ComparaDecrescente);

   printf("\n\nOrdem decrescente:\n");
   for (i=0; i < tamanhoAr; i++) {
     printf(" %d ", arrayDeInts[i]);
   }

   return 0;
}
```

Este exemplo de generalização de `BubbleSort()` utilizando ponteiros para funções certamente não apresenta a melhor solução para este caso em particular. Sua apresentação aqui deve-se simplesmente à utilidade do exemplo, que é relativamente simples, como um dispositivo didático.

10.9 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[8].

Na corrente versão, será dado enfoque apenas à leitura e validação de valores de entrada propriamente dita. 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: 222222222222
22222222222222
O valor -1 nao e' valido
```

- A função causa o aborto do programa quando é introduzido um valor negativo que cause ***underflow***[9]. Por exemplo, a interação a seguir:

[8] Esta seção foi inspirada no artigo *Getting Interactive Input in C* escrito, por Thomas Wolf (v. **Bibliografia**).

[9] Supunha-se, na versão apresentada na **Seção 3.7**, que o programa deveria ler apenas valores positivos.

```
Introduza um numero inteiro positivo: -22222222222
22222222222222
```

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.

10.9.1 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.

10.9.2 Funções, Variáveis e Tipos da Biblioteca Padrão Utilizados

Aqui serão apresentadas as funções, variáveis globais e tipos da biblioteca padrão de C utilizadas 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 detectar 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 pelo fato de (1) permitir a especificação de um stream de saída (argumento `stream`) e de (2) ter um argumento do tipo **va_list** (`argumentos`) em vez 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 encerramento 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**.

10.9.3 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 (v. **Seção 10.9.10**).

7. A chamada `fflush(stdout)` garante que o usuário leia 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 ao `[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.

10.9.4 Apresentação de Mensagens de Erro

A função `ImprimeMensagemDeErro()` apresenta mensagens de erro no *stream* padrão de erros **stderr** (v. **Seção 12.6**). 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()** (v. **Volume II**). Para entender melhor a implementação desta função, refira-se à **Seção 3.4**, que trata de funções com listas de argumentos variáveis.

10.9.5 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 função é definida como **static** pelas 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 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.

10.9.6 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**.

10.9.8 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 for 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 à 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.

10.9.9 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: 20000000000000000000.000000

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
```

10.9.10 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, precisará 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.");
```

10.10 Exercícios de Revisão

1. (a) O que é um declarador de tipo? (b) Quais são os declaradores de tipos disponíveis em C?

2. (a) O que são operadores de memória? (a) Quais são os operadores de memória de C?

3. (a) Como um array unidimensional de ponteiros pode ser utilizado para representar uma coleção de ***strings***? (b) Como um ***string*** neste array pode ser acessado?

4. Considere os seguintes protótipos de função:

```
void F1(char *p)
void F2(char *p)
void F3(char **p)
void F4(const char **p)
```

Considere ainda o array `ar` definido como:

```
char ar[20] = "bola";
```

(a) Quais funções poderiam ser chamadas usando o array `ar` e como ele seria utilizado nas possíveis chamadas?

(b) Em quais destas chamadas o conteúdo do array `ar` pode ser alterado ***sem problemas***?

(c) Em quais destas chamadas o valor de `ar` pode ser alterado?

5. Considere os mesmos protótipos de função do exercício anterior e o ponteiro `str` definido como:

```
char *str = "bola";
```

(a) Quais funções poderiam ser chamadas usando o ponteiro `str` e como ele seria utilizado nas possíveis chamadas?

(b) Em quais destas chamadas o conteúdo apontado por `str` pode ser alterado ***sem problemas***?

(c) Em quais destas chamadas o valor de `str` pode ser alterado?

6. Qual é a relação existente entre o nome de uma função e um ponteiro?

7. Escreva uma declaração apropriada para cada uma das seguintes situações:

(a) Um ponteiro para uma função que recebe dois argumentos do tipo **int** e retorna um valor do tipo **long**.

(b) Um ponteiro para uma função que recebe dois ponteiros para o tipo **int** como argumentos e retorna um ponteiro para um valor do tipo **long**.

(c) Uma função que recebe um ponteiro para função como parâmetro e retorna um ponteiro para um valor do tipo **int**. O parâmetro deve ser compatível com funções que possuem um parâmetro do tipo **int** e retornam um valor do tipo **long**.

8. Suponha que se tenham os seguintes protótipos:

```
void   UmaFuncao(float f, double (*pf)(double));
float  F1(double d);
float  F2(float f);
double F3(float f);
double F4(double d);
```

Quais das chamadas da função `UmaFuncao()` a seguir são ilegais e por quê?

(a) `UmaFuncao(2.4, F1);`

(b) `UmaFuncao(3.5f, F2);`

(c) `UmaFuncao(2.2, F3(1.5));`

(d) `UmaFuncao(1.8, &F4);`

(e) `UmaFuncao(2.8, &F4(2.1));`

(f) `UmaFuncao(2.4, F4);`

9. Suponha que se tenham os seguintes protótipos:

```
void   OutraFuncao(double (*pf)());
float  F1(double d);
float  F2(float f);
double F3(float f);
double F4(double d);
```

Quais das chamadas da função `OutraFuncao()` a seguir são ilegais e por quê?

(a) `OutraFuncao(F1);`

(b) `OutraFuncao(F2);`

(c) `OutraFuncao(&F3);`

(d) `OutraFuncao(F4);`

10. Dadas as seguintes iniciações:

```
static   int  a[2][3] = { {-3, 14,5} { 1, -10, 8} };
static  int  *b[] = { a[0], a[1] };
int          *p = b[1];
```

quais são os resultados das avaliações das seguintes expressões:

(a) `*b[1]`

(b) `*(++p)`

(c) `*(*(a+1) + 1)`

(d) `*(--p-2)`

11. Interprete as seguintes declarações e diga quais são legais e quais são ilegais:

(a) `int **p`

(b) `int (*pa)[]`

(c) `int *ap[]`

(d) `int (*pf)()`

(e) `int af[]()`

(f) `int fa()[]`

(g) `int ff()()`

(h) `int (**ppa)[]`

(i) `int (**ppf)()`

(j) `int *(*pap)[]`

(k) `int (*ppa)[][]`

(l) `int (*paf)[]()`

(m) `int *(*pfp)()`

(n) `int (*pfa)()[]`

(o) `int (*pff)()()`

(p) `int **app[]`

(q) `int (*apa[])[]`

(r) `int (*apf[])()`

(s) `int *aap[][]`

(t) `int aaf[][]()`

(u) `int *afp[]()`

(v) `int afa[]()[]`

(w) `int (*fpa())[]`

(x) `int (*fpf())()`

(y) `int *fap()[]`

(z) `int faf()[]()`

12. Utilize definições intermediárias de tipos para tornar a seguinte declaração mais legível:

```
char (*(*(*x)[])())[];
```

13. Encontre uma maneira mais simples de generalizar a função `BubbleSort()` apresentada na **Seção 10.8.1** sem utilizar ponteiros para função.

10.11 Exercícios de Programação

EP10.1) A função `ImprimeMes()` apresentada na **Seção 10.3** poderia, em vez de imprimir o nome do mês, retorná-lo de modo que a função que a chama possa fazer aquilo que desejar. Reimplemente esta função de forma que ela retorne o nome do mês. Escreva também uma função **main()** para testar a nova implementação de `ImprimeMes()`.

EP10.2) Modifique a função `BubbleSort()` apresentada neste capítulo de modo que, em vez de rearranjar os elementos de um array, ela armazene a ordem correta de classificação num novo array denominado `listaOrdenada[]`. Escreva um programa para testar esta nova versão de `BubbleSort()`.

EP10.3)

(a) Escreva um módulo em C, denominado `Alunos`, que exporte o seguinte:

(i) O tipo `tAluno`, definido como uma estrutura capaz de conter as seguintes informações sobre um aluno: ***nome***, ***matrícula***, ***três notas*** (variando de 0.0 a 10.0) e a ***média*** dessas notas.

(ii) As macros `MAX_NOME` e `CARACTERES_EM_MATRICULA`, que representam, respectivamente, o número máximo de caracteres permitidos no nome de um aluno e o número (exato) de caracteres permitidos numa matrícula.

(iii) Uma função que lê, no meio de entrada padrão, os dados a serem armazenados numa variável do tipo `tAluno`,

com exceção da média que será calculada. Esta função deve utilizar funções do módulo ***Interface*** (v. adiante) para leitura e validação dos dados introduzidos. A condição de validação da matrícula de um aluno é que ela contenha apenas dígitos na quantidade exata especificada pela constante `CARACTERES_EM_MATRICULA`. A condição de validação do nome de um aluno é que ele contenha apenas letras, com o número máximo de caracteres representado pela constante `MAX_NOME`. [**Sugestão**: utilize as funções **isdigit()** e **isalpha()** do módulo ctype descritas no **Volume II**.]

(iv) Uma função que recebe um registro de aluno (i.e., uma estrutura do tipo `tAluno`) como parâmetro e calcula a média do aluno.

(v) Uma função que recebe um arranjo de estruturas do tipo `tAluno` representando uma turma e calcula a média da turma.

(vi) Uma função que recebe um arranjo de estruturas do tipo `tAluno` representando uma turma e ordena este arranjo em ordem alfabética crescente de nomes de alunos. [**Sugestão**: utilize a função **qsort()** apresentada na **Seção 11.9.1**.]

(vii) Uma função que recebe um arranjo de estruturas do tipo `tAluno` representando uma turma e ordena este arranjo em ordem crescente de médias. [**Sugestão**: utilize a função **qsort()** apresentada na **Seção 11.9.1**.]

(viii) Uma função que recebe um arranjo de estruturas do tipo `tAluno` representando uma turma e um ***string*** representando uma matrícula e retorna um ponteiro para o elemento do arranjo (estrutura) cuja matrícula é igual à matrícula fornecida como parâmetro. Caso a matrícula não seja encontrada no arranjo, esta função deve retornar **NULL**.

(b) Escreva um módulo em C, denominado ***Leitura***, que exporte o seguinte:

(i) Uma função para leitura de notas de alunos. (**Sugestão**:

Utilize a função `LeValor()` apresentada na **Seção 10.9**.)

(ii) Uma função para leitura de nomes e matrículas de alunos. (**Sugestão**: Estude a **Seção 8.8** e aprenda como utilizar a função `LeString()` para ler e validar ***strings***.)

Este módulo deve conter ainda outras funções auxiliares de implementação (por exemplo, uma função para limpeza de buffer), mas estas funções auxiliares não devem ser exportadas pelo módulo.

(c) Escreva um módulo em C, denominado ***Interface***, que exporte o seguinte:

(i) Uma função que faça a apresentação do programa.

(ii) Uma função que faça a despedida do programa.

(iii) Uma função que limpe a tela. (**Sugestão**: Não existe função desta natureza na biblioteca padrão de C, mas usualmente várias implementações possuem uma função denominada `clrscr()` num módulo denominado `conio`.)

(iv) Uma função que apresente na tela um menu contendo as seguintes opções:

Acrescentar um aluno na turma.
Remover um aluno da turma.
Modificar um registro de aluno.
Enumerar todos os alunos da turma em ordem alfabética ou de média crescente.
Imprimir a média da turma.
Sair do programa.

(v) Uma função que leia e valide opções de menu. Esta função deve retornar 0 se a opção for inválida ou o caractere correspondente se a opção for válida. Utilize para validação desta opção uma função local (**static**) que receba como entrada um caractere representando uma opção e um ***string***

representando a concatenação de todas as opções válidas (por exemplo, no menu apresentado acima, este ***string*** seria "ARMEIS"). Esta função local deve retornar 1 se o caractere representa uma opção válida e zero em caso contrário. Além disso, esta função não deve fazer distinção entre letras maiúsculas e minúsculas.

(d) Escreva um módulo em C, denominado `Operacoes`, que implemente as operações oferecidas ao usuário no menu descrito no item (c).

(e) Escreva um programa que mantenha um conjunto de registros de alunos (turma) num arranjo de estruturas do tipo `tAluno` contendo um número máximo de registros dado pela constante `NUMERO_MAX_ALUNOS`, definida pelo programa. Este programa deve utilizar os módulos descritos nos itens (a), (b), (c) e (d) para oferecer ao usuário as opções de operações apresentadas no menu descrito no item (c), executar cada operação escolhida pelo usuário e apresentar resultados ou mensagens de erro, conforme for o caso.