Capítulo 8

Arrays

OBJETIVOS

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia relacionada à linguagem C:
  - ❒ Variável estruturada
  - ❒ Array
  - ❒ Definição de array
  - ❒ Iniciação de array
  - ❒ Índice
  - ❒ Zumbi
  - ❒ Fator de escala
  - ❒ Operador sizeof
  - ❒ Array unidimensional
  - ❒ Array multidimensional
  - ❒ Acesso sequencial de array
- Descrever como deve ser escrita a iniciação de um array unidimensional
- Discernir a diferença em termos de iniciação entre arrays de duração fixa e arrays de duração automática
- Descrever e usar as palavras-chave **sizeof** e **const** da linguagem C:
- Determinar o número de bytes ocupados por um array e por um elemento de array
- Diferenciar constantes definidas usando **const** e **#define**
- Classificar variáveis estruturadas em homogêneas ou heterogênea
- Descrever precisamente as operações aritméticas permitidas sobre ponteiros
- Expressar o endereço e o valor de um elemento de array usando aritmética de ponteiros
- Justificar o fato de indexação de arrays em C começar em zero
- Descrever como um parâmetro formal que representa um array deve ser declarado e como ele deve casar com um parâmetro real correspondente
- Saber que uma função em C nunca recebe ou retorna um array em si
- Interpretar declaração de variável ou parâmetro que usa **const**
- Identificar e evitar erro de programação causado por zumbi
- Descrever o que é linha e coluna de um array bidimensional

8.1 Introdução

UMA VARIÁVEL ESTRUTURADA (ou **agregada**) é aquela que contém componentes que podem ser acessados individualmente. Uma variável estruturada pode ser **homogênea**, quando seus componentes são todos de um mesmo tipo, ou **heterogênea**, quando seus componentes podem ser de tipos diferentes. O tipo de uma variável estruturada é considerado um **tipo estruturado** (ou **agregado**).

Array é uma variável estruturada e homogênea que consiste numa coleção de variáveis do mesmo tipo armazenadas contiguamente em memória. Cada variável que compõe um array é denominada **elemento** e pode ser acessada usando o nome do array e um **índice**. Em C, índices são valores inteiros não negativos e o elemento inicial de um array sempre tem índice igual a zero.

Arrays constituem o foco principal deste capítulo, que aborda ainda outros tópicos intimamente relacionados ao uso de arrays, como:

- O operador **sizeof** (Seção 8.5)
- Aritmética de ponteiros (Seção 8.6) e a relação entre ponteiros e arrays (Seção 8.7)
- A palavra-chave **const** (Seção 8.8)

8.2 Definições de Arrays

A definição de um array, em sua forma mais simples, consiste em:

***tipo nome-do-array*[*tamanho-do-array*];**

Nesse esquema de definição, *tipo* é o tipo de cada elemento do array *e tamanho-do-array* consiste numa expressão inteira, constante e positiva que especifica o número de elementos do array. Por exemplo, a seguinte linha de um programa:

```
double notas[50];
```

define a variável `notas[]` como um array de 50 elementos, cada um dos quais é uma variável do tipo **double**[1].

8.3 Acesso a Elementos de um Array

Os elementos de um array são **acessados** por meio de índices que indicam a posição de cada elemento em relação ao elemento inicial do array. Mais precisamente, o elemento inicial possui índice `0`, o segundo tem índice `1` e assim por diante. Como a indexação dos elementos começa com `0`, o último elemento de um array possui índice igual ao número de elementos do array menos um.

Esquematicamente, a notação usada para acesso aos elementos de um array é:

***nome-do-array*[*índice*]**

Para todos os efeitos, um elemento de um array pode ser considerado uma variável do tipo usado na definição do array, de modo que qualquer operação permitida sobre uma variável do tipo de um elemento de um array também é permitida para o próprio elemento. Por exemplo:

```
double  notas[50];

notas[0] = 5.0;  /* Atribui 5.0 ao elemento inicial do array */
      ...
notas[49] = 7.5; /* Atribui 7.5 ao último elemento do array */
notas[50] = 9.0; /* Essa referência é problemática */
```

[1] A razão pela qual este e muitos outros textos sobre C usam colchetes quando fazem referência a um array (como em `notas[]`, por exemplo) é que o nome de um array considerado isoladamente representa seu endereço (v. Seção 8.7).

Nesse último exemplo, a referência ao elemento `notas[50]` é problemática, pois acessa uma porção de memória que não faz parte do array.

O padrão ISO não requer que um compilador de C faça verificação de acesso além dos limites de um array. Portanto o programador pode acidentalmente acessar porções de memória que não foram alocadas para o array (como na última linha do exemplo acima) e isso pode trazer consequências imprevisíveis. Algumas vezes, a porção de memória acessada pode pertencer a outras variáveis; outras vezes, áreas especiais de memória podem ser indevidamente acessadas, o que poderá causar aborto do programa. Frequentemente, esse tipo de erro é causado porque o programador excede em um o valor do índice usado na expressão condicional de um laço **for** que acessa sequencialmente os elementos de um array, como mostra o seguinte exemplo:

```
#include <stdio.h>

int main(void)
{
   int  ar[10], i;

   for  (i = 0; i <= 10; i++) {
      ar[i] = 0;
   }

   return 0;
}
```

Nesse exemplo, o array `ar[]` foi definido com capacidade para conter `10` elementos e, portanto, ele deve ser acessado apenas com índices variando entre `0` e `9`. Entretanto, o laço **for** do exemplo possui um erro que faz com que ao elemento de índice `10` (inválido, portanto) seja atribuído o valor `0`. Como não existe o elemento `ar[10]`, o compilador colocará zero numa porção de memória que não pertence ao array `ar[]`, mas que, provavelmente, pertence à variável `i`. Isso é provável porque a definição da variável `i` foi feita logo em seguida à declaração de `ar[]`, como mostra a Figura 8–1.

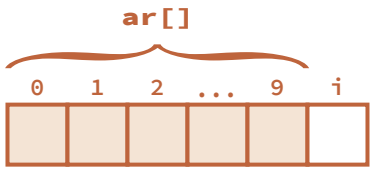

Figura 8–1: Possível Alocação de Array e Variável de Contagem

O erro incorporado no último programa poderá causar um laço de repetição infinito, uma vez que à variável `i` é atribuído zero sempre que o elemento `ar[10]` é acessado, de modo que ela nunca assume um valor que encerre o laço.

Como exemplo correto de acesso sequencial aos elementos de um array considere o programa a seguir que exibe um array ora do primeiro ao último elemento, ora do último ao primeiro elemento.

```
#include <stdio.h>

int main(void)
{
   int ar[5] = {1, -1, 2, 0, 4},
       i;

      /* Exibe o array do primeiro ao último elemento */
   for (i = 0; i < 5; ++i) {
      printf("ar[%d] = %2d\n", i, ar[i]);
   }

   printf("\n\n"); /* Salta uma linha */
```

```
      /* Exibe o array do último ao primeiro elemento */
   for (i = 4; i >= 0; --i) {
      printf("ar[%d] = %2d\n", i, ar[i]);
   }

   return 0;
}
```

Quando executado o último programa produz o seguinte resultado na tela:

```
ar[0] =  1
ar[1] = -1
ar[2] =  2
ar[3] =  0
ar[4] =  4

ar[4] =  4
ar[3] =  0
ar[2] =  2
ar[1] = -1
ar[0] =  1
```

Uma situação na qual números mágicos (v. **Seção 6.5**) aparecem com frequência é na definição e acesso a elementos de arrays. Aqui, o problema não é apenas de legibilidade, mas também de manutenibilidade. Suponha, por exemplo, que você tenha definido um array como:

```
double notas[20]; /* 20 é número mágico */
```

e seu programa utiliza o valor constante `20` em vários outros locais. Se você ou algum outro programador desejar alterar seu programa com o objetivo de aumentar o tamanho do array para, diga-se, `25` terá que decidir quais dos demais valores iguais a `20` distribuídos no programa representam o tamanho do array. O fragmento de programa a seguir ilustra o que foi exposto:

```
double notas[20]; /* 20 é número mágico */
...
for (i = 0; i < 20; ++i) { /* 20 é um número mágico */
   ...
}
```

Um bom estilo de programação recomenda que esse trecho de programa seja substituído por:

```
#define NUMERO_DE_ELEMENTOS 20
...
double notas[NUMERO_DE_ELEMENTOS];
...
for (i = 0; i < NUMERO_DE_ELEMENTOS; ++i) {
   ...
}
```

No laço **for** acima, a constante `20` foi substituída por `NUMERO_DE_ELEMENTOS`, mas o valor `0` não foi substituído, pois ele não pode ser considerado um número mágico, visto que ele sempre representa o primeiro índice de um array e, portanto, tem significado próprio (v. **Seção 6.5**).

8.4 Iniciações de Arrays

Como ocorre com outros tipos de variáveis, arrays de duração fixa e arrays de duração automática diferem em termos de iniciação.

8.4.1 Arrays de Duração Fixa

Como padrão, arrays com duração fixa têm todos os seus elementos iniciados automaticamente com zero, mas podem-se iniciar todos ou alguns elementos explicitamente com outros valores. A iniciação de elementos de um array é feita por meio do uso de expressões constantes, separadas por vírgulas e entre chaves, seguindo a definição do array. Uma construção dessa natureza é denominada um **iniciador**. Considere, por exemplo:

```
static  int  meuArray1[5];
static  int  meuArray2[5] = {1, 2, 3.14, 4, 5};
```

Nesse exemplo, todos os elementos de `meuArray1` serão iniciados com `0`, enquanto `meuArray2[0]` recebe o valor `1`, `meuArray2[1]` recebe `2`, `meuArray2[2]` recebe `3` (aqui ocorre uma conversão de tipo), `meuArray2[3]` recebe `4` e `meuArray2[4]` recebe `5`.

Não é ilegal incluir numa iniciação um número de valores maior do que o permitido pelo tamanho do array, mas, nesse caso apenas o número de valores igual ao tamanho do array é usado na iniciação. Por exemplo, na iniciação do array `ar1[]` a seguir:

```
int ar1[3] = {1, 2, 3, 4, 5};
```

serão usados apenas os valores `1`, `2` e `3`, visto que o array deve ter apenas três elementos.

Não é necessário iniciar todos os elementos de um array e, se houver um número de valores de iniciação menor do que o número de elementos do array, os elementos remanescentes serão iniciados implicitamente com zero. Nesse último caso, a iniciação implícita dos elementos remanescentes é independente da duração do array. Por exemplo:

```
int ar2[5] = {2, -1};
```

Nesse exemplo, na iniciação do array `ar2[]`, são atribuídos `2` e `-1` ao primeiro e ao segundo elementos, respectivamente e zero aos demais elementos.

Quando todos os elementos de um array são iniciados, o tamanho do array pode ser omitido, pois, nesse caso, o compilador deduz o tamanho do array baseado no número de valores de iniciação. Por exemplo:

```
static  int  meuArray2[] = {1, 2, 3.14, 4, 5};
```

é o mesmo que:

```
static  int  meuArray2[5] = {1, 2, 3.14, 4, 5};
```

Essa última característica é válida também para arrays de duração automática e, na prática, é mais usada com strings (v. **Capítulo 9**).

8.4.2 Arrays de Duração Automática

Arrays de duração automática (i.e., aqueles definidos dentro de funções sem o uso de **static**) também podem ser iniciados explicitamente. As regras para iniciação explícita de elementos de um array de duração automática são similares àquelas usadas para arrays de duração fixa. Isso inclui a iniciação com `0` dos elementos não iniciados, desde que haja a iniciação explícita de pelo menos um elemento. Entretanto, arrays de duração automática não são iniciados implicitamente. Ou seja, se não houver nenhuma iniciação explícita, o valor de cada elemento será indefinido; i.e., eles receberão o conteúdo indeterminado encontrado nas posições de memória alocadas para o array.

Para fixar melhor a diferença entre arrays de duração fixa e arrays de duração automática em termos de iniciação implícita considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   static int arFixo[5];
   int        arAuto[5];
   int        i;

   printf("\n\n*** Array de Duracao Fixa ***\n");

   for (i = 0; i < 5; ++i) {
      printf("\n  %d\t    %d", i, arFixo[i]);
   }

   printf("\n\n*** Array de Duracao Automatica ***\n");

   for (i = 0; i < 5; ++i) {
      printf("\n  %d\t    %d", i, arAuto[i]);
   }
   return 0;
}
```

Quando esse programa foi executado, ele produziu como resultado:

```
*** Array de Duracao Fixa ***

  0         0
  1         0
  2         0
  3         0
  4         0

*** Array de Duracao Automatica ***

  0         2293592
  1         4200662
  2         4200568
  3         28
  4         0
```

No programa acima, foram definidos dois arrays: **`arFixo[]`** com duração fixa e **`arAuto[]`** com duração automática e nenhum deles é iniciado explicitamente. Observe o resultado apresentado pelo programa e note que os elementos do array de duração fixa foram iniciados implicitamente com zero, enquanto os elementos do array de duração automática não foram iniciados e, por isso, os valores dos elementos desse array não fazem sentido. Se você compilar e executar o programa acima, obterá os mesmos valores para os elementos do array de duração fixa, mas os valores apresentados para os elementos do array de duração automática provavelmente serão diferentes.

Embora não seja comum, a iniciação de elementos de um array de duração automática também pode incluir expressões, como mostra o seguinte exemplo:

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

int main(void)
{
   double x = 2.54, y = 1.6,
          ar[4] = { sqrt(2), 2*x, x + y, sqrt(5) };

   for (int i = 0; i < 4; ++i) {
      printf("\nar[%d] = %3.2f", i, ar[i]);
   }
```

```
   return 0;
}
```

A iniciação do array `ar[]` nesse programa seria inválida se ele tivesse duração fixa (i.e., se sua definição fosse precedida por **static**).

8.5 O Operador sizeof e o Tipo size_t

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

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

```
#include <stdio.h>

int main(void)
{
   size_t tamanhoTipoDouble;

   tamanhoTipoDouble = sizeof(double);

   printf( "\n>>> Bytes ocupados por uma variavel do"
           "\n>>> tipo double: %d\n", tamanhoTipoDouble );

   return 0;
}
```

O resultado do operador **sizeof** é do tipo **size_t**, que é um tipo inteiro sem sinal (v. adiante) definido em vários cabeçalhos da biblioteca padrão de C. Por isso, no último programa, a variável `tamanhoDoTipoDouble` é definida com esse tipo.

Um tipo inteiro com sinal, como é o caso do tipo primitivo **int** possui um **bit de sinal**, que indica se um valor desse tipo é positivo ou negativo. Por outro lado, um tipo inteiro sem sinal, como é o caso do tipo derivado **size_t**, não possui tal bit, pois todos seus valores são considerados sempre positivos ou zero.

Quando o operador **sizeof** é aplicado a uma constante ou variável, o resultado é o número de bytes necessários para armazenar a constante ou variável, respectivamente. Quando aplicado a uma expressão, o operador **sizeof** resulta no número de bytes que seriam necessários para conter o resultado da expressão *se ela fosse avaliada*; i.e., a expressão em si *não é avaliada*. Considere, como exemplo, o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int    i = 0;
   size_t tamanhoDaExpressao;

   tamanhoDaExpressao = sizeof(++i);

   printf( "\n>>> Bytes ocupados pela expressao ++i: %d", tamanhoDaExpressao );

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

   return 0;
}
```

Quando executado, esse programa produz o seguinte resultado:

```
>>> Bytes ocupados pela expresao ++i: 4
>>> Valor de i: 0
```

De acordo com o resultado apresentado pelo último programa, a expressão `++i` não foi avaliada pelo operador **sizeof** e, por isso, a variável `i` não foi incrementada.

Quando o operando do operador **sizeof** é uma expressão, ela não precisa ser colocada entre parênteses como foi feito no último programa; porém, o uso de parênteses com **sizeof** é sempre recomendado para prevenir erros.

O tamanho, em bytes, de um array pode ser determinado aplicando-se o operador **sizeof** ao nome do array. Entretanto, nesse caso, não se deve utilizar nenhum índice; caso contrário, o tamanho resultante será o de um único elemento do array (ao invés do tamanho de todo o array). O programa a seguir demonstra esses fatos.

```
#include <stdio.h>

int main(void)
{
   int    ar[] = {-1, 2, -2, 7, -5};
   size_t tamanhoArray, tamanhoElemento, nElementos;

   tamanhoArray = sizeof(ar);

   tamanhoElemento = sizeof(ar[0]);

   nElementos = tamanhoArray/tamanhoElemento;

   printf("\n>>> Bytes ocupados pelo array: %d\n", tamanhoArray);

   printf( "\n>>> Bytes ocupados por um elemento do array: %d\n", tamanhoElemento );

   printf("\n>>> Numero de elementos do array: %d\n", nElementos);

   return 0;
}
```

Esse último programa produz como resultado:

```
>>> Bytes ocupados pelo array: 20
>>> Bytes ocupados por um elemento do array: 4
>>> Numero de elementos do array: 5
```

A expressão:

```
sizeof(ar[0])
```

resulta no tamanho do elemento de qualquer array `ar[]` porque qualquer array tem pelo menos um elemento, que é o elemento `ar[0]`. Portanto pode-se concluir que a expressão:

```
sizeof(ar)/sizeof(ar[0])
```

resulta sempre no número de elementos de um array `ar[]`, desde que ela seja avaliada dentro do escopo no qual o array é definido (v. Seção 8.9).

É importante observar que uma aplicação do operador **sizeof** é avaliada em tempo de compilação e não durante a execução do programa que o contém. Por exemplo, o compilador é capaz de avaliar a última expressão que calcula o número de elementos de um array, pois, para tal, ele não precisa conhecer o conteúdo do array, o que só ocorre durante a execução do programa.

Conforme foi afirmado, o resultado do operador **sizeof** é do tipo **size_t**, que é um tipo inteiro sem sinal. Existem outros tipos inteiros sem sinal em C que este livro evitou usar, porque eles são fontes de erros difíceis

de detectar num programa. Mas, agora, com a inexorável necessidade de uso do tipo **size_t**, torna-se inevitável uma séria discussão sobre esses tipos.

Cada valor de um tipo inteiro com sinal utiliza um bit que serve exatamente para indicar qual é o sinal (positivo ou negativo) do valor. Num tipo inteiro sem sinal, todos os valores são considerados positivos e não existe bit de sinal. Portanto, mesmo que dois tipos inteiros, um com sinal e o outro sem sinal, tenham a mesma largura (v. **Seção 3.4.1**), eles são considerados distintos, de modo que, em nível de máquina, não existe operação possível entre um valor com sinal e outro sem sinal. Assim, para uma linguagem de alto nível permitir mistura de um operando com sinal e outro sem sinal numa expressão, um deles deve ser convertido no tipo do outro antes de a operação ser realizada. No caso específico de mistura de operandos com e sem sinal que têm a mesma largura, o valor do operando com sinal é convertido num valor sem sinal. Quando o valor convertido é positivo, isso não acarreta problema. Mas, quando o número convertido é negativo, o resultado da conversão é um valor positivo enorme e bem diferente do valor original que foi convertido. Isso ocorre porque o bit de sinal, que é `1` quando o número é negativo, será interpretado como um bit significativo.

Para apreciar o tipo de erro que pode ocorrer quando se misturam valores com sinal e valores sem sinal, considere o seguinte programa:

```
#include<stdio.h>

int main(void)
{
   int    ar[] = {1, 2, 3, 4, 5, 6, 7}, i;
   size_t nElementos;

      /* Calcula o número de elementos do array */
   nElementos = sizeof(ar)/sizeof(ar[0]);

      /* Na avaliação da expressão i <= nElementos - 2, i, que vale -1, é    */
      /* convertido em size_t, que é um tipo sem sinal. Portanto o valor de  */
      /* i convertido torna-se um valor muito grande sem sinal e o corpo do  */
      /* laço não é executado nunca. A solução é definir nElementos como int */
   for(i = -1; i <= nElementos - 2; i++) {
      printf("%d\t", ar[i + 1]);
   }

   printf("\n>>> Valor de i = %u\n", (size_t)i);

   return 0;
}
```

Esse programa é bastante simples e o que ele tenta realizar é apenas apresentar na tela os valores dos elementos do array `ar[]` e, apesar de ele tentar atingir seu objetivo de uma forma ineficiente e não convencional, o raciocínio empregado no laço **for** utilizado pelo programa é absolutamente correto do ponto de vista funcional. Mas, se você compilar e executar esse programa, não obterá a escrita na tela de nenhum elemento do array. Por que isso ocorre?

Poucos programadores inexperientes de C saberão responder essa pergunta, mas, se você entendeu a discussão referente a mistura de tipos com e sem sinal, estará apto a acompanhar o que ocorre durante a execução da instrução **for** do último programa:

1. A variável `i` recebe o valor `-1` na entrada do laço **for**.
2. A expressão `i <= nElementos - 2` é avaliada e é aqui que ocorre o problema. Como a variável `nElementos` é de um tipo sem sinal (**size_t**), o operando direito do operador representado por `<=` será desse tipo e o operando esquerdo (`i`) precisará ser convertido em um valor sem sinal. Como `i` vale `-1`, esse valor convertido num número sem sinal é enorme e, certamente, será maior do que `nElementos - 2`. Portanto o corpo do laço **for** não será executado nenhuma vez.

Para determinar qual é o valor verdadeiro de `i` usado na avaliação da expressão `i <= nElementos - 2`, foi inserida a seguinte chamada de **printf()** entre as instruções **for** e **return** do programa sob discussão:

```
printf("Valor de i = %u", (size_t)i);
```

Nessa chamada de **printf()**, o especificador de formato **%u** é usado para a escrita de um número inteiro sem sinal.

A solução para o problema apresentado pelo último programa é evitar que ocorra a conversão do valor de `i` num número sem sinal. Isso é obtido definido-se o tipo da variável `nElementos` como **int**. Essa singela alteração faz com que o programa funcione, pois, nesse caso, ocorrerá conversão de atribuição de um número inteiro sem sinal para um número inteiro com sinal, e não o contrário (v. Seção 3.10.1).

Com base na discussão apresentada acima, o conselho a ser seguido é:

Recomendação	*Jamais misture operando inteiro com sinal com operando inteiro sem sinal numa mesma expressão, a não ser numa expressão de atribuição na qual o operando esquerdo seja uma variável do tipo int.*

8.6 Aritmética de Ponteiros

A linguagem C permite que as operações aritméticas apresentadas na Tabela 8–1 sejam executadas com ponteiros (supondo que `p`, `p1` e `p2` sejam ponteiros de quaisquer tipos).

Operação	Exemplo
Soma de um valor inteiro a um ponteiro	`p + 2`
Subtração de um valor inteiro de um ponteiro	`p - 3`
Incremento de ponteiro	`++p` ou `p++`
Decremento de ponteiro	`--p` ou `p--`
Subtração entre dois ponteiros do mesmo tipo	`p1 - p2`

Tabela 8–1: Operações Aritméticas sobre Ponteiros

Operações aritméticas sobre ponteiros, entretanto, devem ser interpretadas de modo diferente das operações aritméticas usuais. Por exemplo, se `p` é um ponteiro definido como:

```
int  *p;
```

a expressão:

```
p + 3
```

deve ser interpretada como o endereço do espaço em memória que está três variáveis do tipo **int** adiante do endereço da variável para o qual `p` aponta. Isto é, como `p` é um endereço, `p + 3` também será um endereço. Mas, em vez de adicionar `3` ao valor do endereço armazenado em `p`, o compilador adiciona `3` multiplicado pelo tamanho (i.e., número de bytes) da variável para a qual `p` aponta. Nesse contexto, o tamanho da variável apontada pelo ponteiro é denominado **fator de escala**. Com exceção da última operação apresentada na Tabela 8–1, as demais operações sobre ponteiros envolvem a aplicação de um fator de escala, que corresponde à largura do tipo de variável para o qual o ponteiro aponta.

Suponha, por exemplo, que o endereço correntemente armazenado no ponteiro `p` definido acima seja `e` e que variáveis do tipo **int** sejam armazenadas em 4 bytes. Então, `p + 3` significa, após a aplicação do fator de escala, o endereço `e + 3*4`, que é igual ao endereço `e + 12`. A Figura 8–2 ilustra esse argumento.

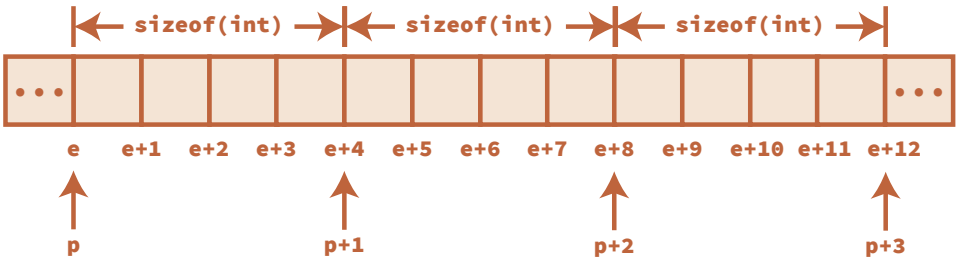

FIGURA 8–2: SOMA DE UM INTEIRO A UM PONTEIRO

Se, no exemplo anterior, o ponteiro `p` tivesse sido definido como `char *p`, então `p + 3` significaria `e + 3` [porque `sizeof(char)` é sempre `1`]. Concluindo, `p + 3` sempre significa o endereço da terceira variável do tipo apontado pelo ponteiro `p` após aquela correntemente apontada por ele.

Subtrair um inteiro de um ponteiro tem uma interpretação semelhante. Por exemplo, `p - 3` representa o endereço da terceira variável do tipo apontado pelo ponteiro `p` que precede a variável correntemente apontada por ele, como mostra a **Figura 8–3**, que considera as mesmas suposições da **Figura 8–2**.

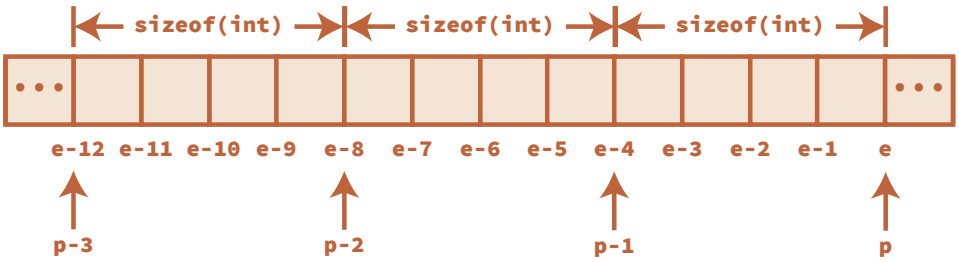

FIGURA 8–3: SUBTRAÇÃO DE UM INTEIRO DE UM PONTEIRO

Operações de incremento e decremento de ponteiros também são bastante comuns em programação em C. Nesses casos, os operadores de incremento e decremento são utilizados para fazer um ponteiro apontar para a variável posterior e anterior, respectivamente, à posição atual do ponteiro. Em qualquer caso, o fator de escala mencionado é aplicado à operação.

A subtração de dois ponteiros é legal, desde que os ponteiros sejam do mesmo tipo, mas só faz sentido quando eles apontam para elementos de um mesmo array. Essa operação resulta num número inteiro cujo valor absoluto representa o número de elementos do array que se encontram entre os dois ponteiros. O seguinte programa demonstra o que foi exposto:

```
#include <stdio.h>

int main(void)
{
   int ar[] = {-1, 2, -2, 7, -5}, nElementos;

   nElementos = &ar[3] - &ar[0];

   printf( "\n>>> Numero de elementos entre &ar[3] e "
           "&ar[0]: %d\n", nElementos );

   return 0;
}
```

O resultado desse último programa é:

```
>>> Numero de elementos entre &ar[3] e &ar[0]: 3
```

A **Figura 8–4** ilustra o resultado apresentado pelo programa anterior:

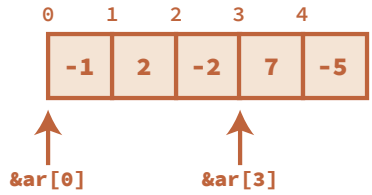

FIGURA 8–4: SUBTRAÇÃO DE DOIS PONTEIROS

O programa em seguida, apresenta alguns exemplos legais e ilegais de aritmética de ponteiros:

```
#include <stdio.h>

int main(void)
{
   double  ar[] = {0.0, 2.5, 3.2, 7.5, 1.6},
          *p1, *p2;
   int     j;
   char   *p3;

   p1 = &ar[0];  /* Legal */

   p2 = p1 + 4;  /* Legal */
   printf("\n*p2 = %f", *p2);

   j = p2 - p1;  /* Legal - resultado: j recebe 4  */
   printf("\np2 - p1 = %d", j);

   j = p1 - p2;  /* Legal - resultado: j recebe -4 */
   printf("\np1 - p2 = %d", j);

   p1 = p2 - 2;  /* Legal - ponteiros são compatíveis */
   printf("\n*p1 = %f", *p1);

   p3 = p1 - 1;  /* Legal, mas os ponteiros */
                 /* não são compatíveis     */
   printf("\n*p3 = %f", *p3);
   printf("\n*p3 = %d", *p3);
// j = p1 - p3;  /* ILEGAL: os ponteiros  não são compatíveis */

   return 0;
}
```

Quando executado, esse programa exibe como resultado:

```
*p2 = 1.600000
p2 - p1 = 4
p1 - p2 = -4
*p1 = 3.200000
*p3 = 3.199999
*p3 = 0
```

A seguir, serão apresentados comentários sobre as instruções de interesse do programa anterior:

```
p1 = &ar[0];  /* Legal */
```

Essa atribuição faz com que **p1** aponte para o primeiro elemento do array **ar[]**.

```
p2 = p1 + 4;  /* Legal */
```

Essa instrução faz com que **p2** aponte para a quarta variável do tipo **double** adiante do endereço apontado por **p1**. Como, correntemente, **p1** aponta para o primeiro elemento do array **ar[]**, a atribuição fará com que **p2** aponte para o elemento **ar[4]**.

```
printf("\n*p2 = %f", *p2);
```

Essa chamada de **printf()** escreve o valor apontado por `p2`; ou seja, o valor do elemento `ar[4]`.

```
j = p2 - p1;  /* Legal */
```

Como, na segunda instrução do programa em questão, `p2` recebeu o valor `p1 + 4`, o resultado de `p2 - p1` é `4`.

```
j = p1 - p2;  /* Legal */
```

Se você entendeu a explicação anterior, o resultado é obvio: `j` recebe `-4`.

```
p1 = p2 - 2;  /* Legal */
```

A última atribuição feita a `p2` fez com que ele apontasse para o elemento `ar[4]` (v. acima). Portanto, nessa atribuição, `p1` receberá o endereço da segunda variável do tipo **double** que antecede `p2`; ou seja, após essa atribuição, `p1` apontará para o elemento `ar[2]`.

```
printf("\n*p1 = %f", *p1);
```

Essa chamada de **printf()** exibe na tela o valor apontado por `p1`; ou seja o valor do elemento `ar[2]`.

```
p3 = p1 - 1;  /* Legal, mas os ponteiros não são compatíveis */
```

Infelizmente, como foi discutido na **Seção 5.2**, apesar de o compilador reconhecer que os ponteiros são incompatíveis e emitir uma mensagem de advertência, de acordo com o padrão ISO, ele considera tal operação legal. A partir daí, qualquer referência ao ponteiro `p3` raramente faz algum sentido.

```
printf("\n*p3 = %f", *p3);
printf("\n*p3 = %d", *p3);
```

Quando o programa em questão foi testado, essas duas chamadas de **printf()** produziram a seguinte escrita na tela:

```
*p3 = 3.199999
*p3 = 0
```

Conforme foi antecipado, como o valor de `p3` é resultado de uma operação entre dois ponteiros incompatíveis, os resultados obtidos com operações de indireção desse ponteiro são imprevisíveis e, provavelmente, não fazem sentido.

```
// j = p1 - p3; /* ILEGAL - os ponteiros não são compatíveis */
```

A operação `p1 - p3` é considerada ilegal porque subtração entre ponteiros só é permitida se os ponteiros forem exatamente do mesmo tipo. Essa instrução foi comentada para permitir que o programa fosse compilado.

8.7 Relações entre Ponteiros e Arrays

Esta seção mostra como ponteiros podem ser usados para acessar elementos de um array. Para começar, suponha a existência das seguintes definições:

```
int   ar[4] = {-1, 12, 9, 2};
int  *p;
```

Então, a atribuição a seguir faz com que `p` aponte para o início do array `ar[]` (i.e., para o elemento `ar[0]`):

```
p = &ar[0];
```

A situação neste instante é ilustrada na Figura 8–5.

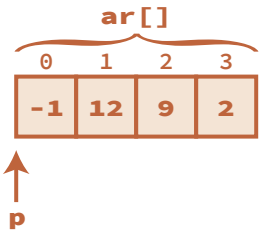

FIGURA 8–5: RAZÃO PELA QUAL INDEXAÇÃO DE ARRAYS COMEÇA EM ZERO 1

Portanto a indireção do ponteiro `p` (i.e., `*p`) resulta no valor de `ar[0]` (i.e., `-1`). Além disso, utilizando-se aritmética de ponteiros pode-se ter acesso aos outros elementos do array. Isto é, `p + 1` refere-se ao endereço de `ar[1]` e `*(p + 1)` resulta em `ar[1]`; `p + 2` refere-se ao endereço de `ar[2]` e `*(p + 2)` resulta em `ar[2]`; e assim por diante, como mostra a Figura 8–6.

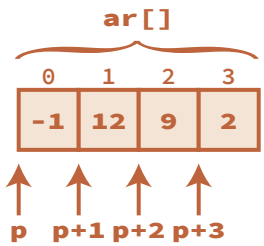

FIGURA 8–6: RAZÃO PELA QUAL INDEXAÇÃO DE ARRAYS COMEÇA EM ZERO 2

Pode-se, portanto, inferir que, em geral, se `p` aponta para o início do array `ar[]`, então a seguinte relação é válida para qualquer inteiro `i`:

`*(p + i)` *é o mesmo que* `ar[i]`

Agora, em C, o nome de um array considerado isoladamente é interpretado como o endereço do array (i.e., o endereço do elemento de índice `0`). Ou seja,

`ar` *é o mesmo que* `&ar[0]`

Combinando-se as duas relações anteriores, obtém-se a seguinte equivalência:

`*(ar + i)` *é o mesmo que* `ar[i]`

Em consequência dessa última relação, quando um compilador de C encontra uma referência com índice a um elemento de um array (p. ex., `ar[2]`), ele adiciona o índice ao endereço do array (p. ex., `ar + 2`) e aplica o operador de indireção a esse endereço [p. ex., `*(ar + 2)`], obtendo, assim, o valor do elemento do array.

Devido às relações apresentadas, ponteiros e nomes de arrays podem ser utilizados de modo equivalente tanto com o operador de indireção, representado por `*`, quanto com colchetes, que também representam um operador, como será visto na Seção 10.4. É importante lembrar, entretanto, que conteúdos de ponteiros podem ser modificados, enquanto o valor atribuído a um nome de array não pode ser modificado porque ele representa o endereço do array (variável) e o endereço de uma variável não pode ser modificado. Em termos práticos, isso significa, por exemplo, que o nome de um array (sem índice) não pode sofrer o efeito de nenhum operador com efeito colateral.

A seguir, estão apresentados alguns exemplos de instruções legais e ilegais envolvendo arrays e ponteiros:

```
double  ar[5], *p;
p = ar;
```

Essa instrução é legal e é equivalente a `p = &ar[0]`.

```
ar = p;  /* ILEGAL */
```

Essa instrução é ilegal, pois `ar` é o endereço do array `ar[]` e não se pode alterar o endereço de um array (ou qualquer outra variável).

```
ar++;  /* ILEGAL */
```

Essa instrução é ilegal porque se trata de outra tentativa de alterar o endereço `ar` do array `ar[]`.

```
p++;
```

Essa instrução é legal porque ponteiros podem ser incrementados.

```
ar[1] = *(p + 3);
```

Essa instrução é legal, pois os valores `ar[1]` e `*(p + 3)` são ambos do tipo **double**.

Com o conhecimento adquirido nesta seção, pode-se responder uma questão que intriga muitos iniciantes na linguagem C, que é: *Por que a indexação de arrays começa em 0 e não em 1, como seria mais natural?* Para entender a resposta, suponha que um ponteiro `p` aponte para o elemento de índice `1` de um array, como mostra a **Figura 8–7**.

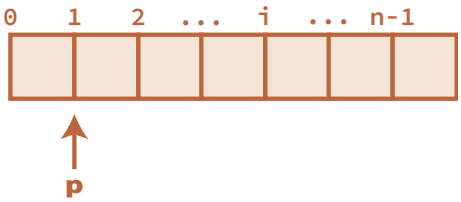

FIGURA 8–7: RAZÃO PELA QUAL INDEXAÇÃO DE ARRAYS COMEÇA EM ZERO 3

Então, utilizando o ponteiro `p`, acessos aos elementos do array seriam obtidos por meio das expressões mostradas na seguinte tabela:

ÍNDICE	EXPRESSÃO
1	*p
2	*(p + 1)
3	*(p + 2)
...	...
i	*(p + i - 1)

Assim, em geral, para acessar o elemento de índice `i` do array seria necessária uma operação aritmética a mais (i.e., subtrair `1` de `p + i`) do que seria o caso se `p` apontasse para o elemento de índice `0`. Ocorreria o mesmo se a indexação de arrays começasse em `1`: a cada acesso a um elemento de um array seria necessário subtrair `1`. Portanto, do ponto de vista de eficiência, a indexação de arrays em C é justificável.

8.8 Uso de const

A palavra-chave **const**, quando aplicada na definição de uma variável, especifica que a variável não pode ter seu conteúdo alterado após ser iniciada. Por exemplo, após a iniciação:

```
const int varConstante = 0;
```

não seria mais permitido à variável **varConstante** ter seu conteúdo modificado. Mas, uma variável definida com o qualificador **const** tem um significado bem distinto daquele de uma constante simbólica (v. **Seção 3.15**). Por exemplo, na definição de constante simbólica a seguir:

```
#define MINHA_CONSTANTE  0
```

MINHA_CONSTANTE difere de **varConstante** basicamente porque **MINHA_CONSTANTE** não tem espaço em memória alocado para si, enquanto **varConstante** terá espaço alocado. Em consequência disso, não se pode, por exemplo, atribuir o endereço de uma constante simbólica a um ponteiro, mas pode-se fazer isso com variáveis constantes, como o seguinte exemplo mostra:

```
int  *ptr1 = &varConstante;    /* Legal  */
int  *ptr2 = &MINHA_CONSTANTE; /* ILEGAL */
```

Não se pode garantir que uma variável constante não seja modificada indiretamente. Por exemplo, considerando a definição de **ptr1** apresentada acima, é permitido que a variável **varConstante** seja modificada por meio de uma instrução como:

```
*ptr1 = 1;
```

A propósito, qualquer compilador decente emite uma mensagem de advertência quando a um ponteiro capaz de alterar o conteúdo de uma variável constante (como **ptr1** no último exemplo) é atribuído o endereço dessa variável.

Numa definição de ponteiro, a palavra-chave **const** pode aparecer precedida pelo símbolo * ou não. Nos dois casos, os significados das definições são diferentes. Por exemplo, na segunda definição a seguir:

```
int          x;
int  *const  ponteiroConstante = &x;
```

a variável **ponteiroConstante** é considerada um ponteiro que deve apontar sempre para a variável **x** (i.e., o valor do ponteiro não deve mudar). Mas, esse ponteiro pode ser utilizado para alterar o conteúdo da variável para a qual ele aponta. Esse uso de **const** é semelhante àquele apresentado antes; afinal, todo ponteiro é uma variável.

Considere, agora, outro tipo de uso de **const** que aparece na segunda definição de variável a seguir:

```
int         x;
int  const *ponteiroParaConstante = &x;
```

Essa última declaração é equivalente a:

```
const int *ponteiroParaConstante = &x;
```

Em ambos os formatos, a variável **ponteiroParaConstante** é considerada como um ponteiro para uma variável que não pode ser modificada (indiretamente) por meio desse ponteiro. Mas, obviamente, pode ser modificada diretamente por meio da própria variável. Nesse caso, o valor do ponteiro em si pode ser modificado, de modo que ele possa apontar para outra variável.

Se você sentir dificuldade em interpretar os dois usos de **const** com ponteiros, utilize **leitura árabe**. Isto é, leia a definição da variável da direita para a esquerda partindo da própria variável. Quando encontrar um asterisco, traduza-o como *ponteiro* e quando encontrar **const**, traduza essa palavra-chave como *constante*. Por exemplo, leia a definição:

```
int *const p;
```

como: *p é constante ponteiro para int*, que, rearranjando para o bom português, fica: *p é um ponteiro constante para int*. Por outro lado, a definição:

```
const int *p;
```

deve ser lida como: *p é um ponteiro para int constante*, que, rearranjando de forma mais precisa (mas prolixa), fica: *p é um ponteiro para um conteúdo do tipo int considerado constante*.

A definição:

```
int const *p;
```

é lida exatamente como a definição precedente: *p é um ponteiro para um conteúdo do tipo int considerado constante*.

Apesar de serem equivalentes, a definição:

```
const  int  *ponteiro;
```

é mais frequentemente usada do que a definição:

```
int  const  *ponteiro;
```

A palavra-chave **const** também pode ser usada para qualificar o conteúdo (i.e., os elementos) de um array como, por exemplo:

```
const int ar[] = {1, 2, 3, 4};
```

ou:

```
int const ar[] = {1, 2, 3, 4};
```

As duas últimas definições têm exatamente o mesmo significado e indicam que os elementos do array `ar[]` devem ser considerados constantes.

O principal propósito de **const** é assegurar que dados que não devem ser modificados não serão realmente modificados. Em particular, o uso de **const** é bastante útil quando ponteiros são passados para funções, pois a declaração de um ponteiro utilizado como parâmetro com a palavra **const** garante que o valor apontado pelo ponteiro não será modificado pela função, como será visto na Seção 8.9.5.

8.9 Uso de Arrays com Funções

8.9.1 Declarando Arrays como Parâmetros Formais

Na definição de uma função, um parâmetro formal que representa um array é declarado como um ponteiro para o elemento inicial do array. Existem duas formas alternativas de declarar tal parâmetro:

tipo-do-elemento ***parâmetro***

ou:

tipo-do-elemento parâmetro[]

Como exemplo de parâmetro formal que representa um array considere o seguinte cabeçalho de função:

```
void  MinhaFuncao(double *ar)
```

ou, alternativamente:

```
void  MinhaFuncao(double ar[])
```

Quando o segundo formato de declaração é utilizado, o compilador converte-o no primeiro formato. Por exemplo, no segundo cabeçalho acima, `double ar[]` é convertido em `double *ar`. Assim, os dois tipos de declarações são completamente equivalentes. Entretanto, em termos de legibilidade, a segunda declaração é melhor do que a primeira, pois enfatiza que o parâmetro será tratado como um array. Na primeira declaração, não existe, em princípio, uma maneira de se saber se o parâmetro é um ponteiro para uma única variável do tipo **double** ou para um array de elementos do tipo **double**.

A escolha entre declarar um parâmetro de função em forma de array ou de ponteiro não tem nenhum efeito na tradução feita pelo compilador. Para o compilador, o parâmetro `ar` do exemplo anterior é apenas um ponteiro para um espaço em memória contendo um valor **double** e não exatamente um array. Mas, em virtude da relação entre ponteiros e arrays (v. Seção 8.7), ainda é possível acessar os elementos do parâmetro `ar` usando colchetes.

Deve-se ressaltar que, em consequência do modo como arrays são tratados numa função, não é possível determinar o tamanho de um array por meio da aplicação do operador **sizeof** sobre um parâmetro formal que representa o array. Por exemplo, se a função `MinhaFuncao()` fosse definida como:

```
void  MinhaFuncao(double ar[])
{
   printf("O tamanho do array e': %d\n", sizeof(ar));
}
```

uma chamada dessa função apresentaria o número de bytes necessários para armazenar um ponteiro, e não o número de bytes necessários para armazenar o array, visto que o parâmetro `ar` é interpretado como um ponteiro.

Devido à impossibilidade de uma função determinar o tamanho de um array cujo endereço é recebido como parâmetro, é muitas vezes necessário incluir o tamanho do array na lista de parâmetros da função. Isso permite à função saber onde o array termina, conforme mostra o seguinte exemplo:

```
void ExibeArrayDoubles(const double ar[], int tamanho)
{
   int  i;

   for (i = 0; i < tamanho; i++) {
      printf("ar[%d] = %3.1f\n", i, ar[i]);
   }
}
```

Note o uso de **const** no cabeçalho da função desse exemplo:

```
void ExibeArrayDoubles(const double ar[], int tamanho)
```

Como na declaração de um parâmetro, `ar[]` é o mesmo que `*ar`, tem-se que:

```
const double ar[]
```

é o mesmo que:

```
const double *ar
```

Portanto, de acordo com o que foi exposto na Seção 8.8, o parâmetro `ar` é declarado como um ponteiro para **double** que se compromete a não alterar o conteúdo para o qual ele aponta. Faz sentido declarar o parâmetro `ar` desse modo porque o array que ele representa deve ser considerado um parâmetro de entrada e o uso de **const** garante que o array não é alterado acidentalmente pela função (v. Seção 8.9.5).

8.9.2 Arrays como Parâmetros Reais

Numa chamada de função que possui um parâmetro que representa um array, utiliza-se como parâmetro real o nome de um array compatível com o parâmetro formal correspondente. Esse nome, conforme já foi visto, será interpretado como o endereço do array, como ilustra o seguinte programa:

```
#include <stdio.h>

void ExibeArrayDoubles(const double ar[], int tamanho)
{
   int  i;

   for (i = 0; i < tamanho; i++) {
      printf("ar[%d] = %3.1f\n", i, ar[i]);
   }
}

double MediaArrayDoubles(const double notas[], int numNotas)
{
   int    i;
   double soma = 0.0;

   for (i = 0; i < numNotas; i++) {
      soma = soma + notas[i];
   }
   return soma / numNotas;
}

int main(void)
{
   double notas[] = {5.2, 6.6, 8.0, 4.0, 5.5, 4.8, 9.1},
          m;

   printf("\n\n****** Notas ******\n\n");
   ExibeArrayDoubles(notas, sizeof(notas)/sizeof(notas[0]));

   m = MediaArrayDoubles(notas, sizeof(notas)/sizeof(notas[0]));
   printf("\nMedia da turma: %3.2f\n", m);

   return 0;
}
```

Esse programa apresenta o seguinte na tela:

```
****** Notas ******

ar[0] = 5.2
ar[1] = 6.6
ar[2] = 8.0
ar[3] = 4.0
ar[4] = 5.5
ar[5] = 4.8
ar[6] = 9.1

Media da turma: 6.17
```

O último programa possui duas funções e cada uma delas recebe como parâmetro um array de elementos do tipo **double**. Os protótipos dessas funções são:

```
void ExibeArrayDoubles(const double ar[], int tamanho)
```

e

```
double MediaArrayDoubles(const double notas[], int numNotas)
```

Observe o uso de **const** nas duas funções. Em ambos os casos, o array é um parâmetro de entrada e o uso de **const** garante que ele não é acidentalmente modificado pela função.

As funções `ExibeArrayDoubles()` e `MediaArrayDoubles()` são chamadas no corpo da função **main()** como:

```
ExibeArrayDoubles(notas, sizeof(notas)/sizeof(notas[0]));
```

e

```
m = MediaArrayDoubles(notas, sizeof(notas)/sizeof(notas[0]));
```

Note que, nos dois casos, o primeiro parâmetro real passado para essas funções é **notas**, que é o nome do array definido na função **main**() como:

```
double notas[] = {5.2, 6.6, 8.0, 4.0, 5.5, 4.8, 9.1};
```

O segundo parâmetro nas chamadas das funções `ExibeArrayDoubles()` e `MediaArrayDoubles()` é representado pela expressão:

```
sizeof(notas)/sizeof(notas[0])
```

que, conforme visto na Seção 8.5, resulta no tamanho do array `notas[]`.

Considere agora o seguinte programa que contém uma função que copia o conteúdo de um array para outro:

```
#include <stdio.h>

void ExibeArrayInts(const int ar[], int tamanho)
{
   int i;

      /* Se o número de elementos for menor do que ou */
      /* igual a zero, escreve as chaves e retorna    */
   if (tamanho <= 0) {
      printf("\n\t{ }\n");
      return;
   }

   printf("\n\t{ "); /* Escreve abre-chaves */

      /* Escreve do primeiro ao penúltimo elemento */
   for (i = 0; i < tamanho - 1; ++i) {
      printf("%d, ", ar[i]);
   }

      /* Escreve o último elemento separadamente para */
      /* que não haja uma vírgula sobrando ao final   */
   printf("%d }\n",  ar[tamanho - 1]);
}

void CopiaArray( int destino[], const int origem[], int nElementos )
{
   int  i;

   for (i = 0; i < nElementos; i++) {
      destino[i] = origem[i];
   }
}

int main(void)
{
   int ar1[5] = {5, 2, 6, 8, 0},
       ar2[5];

   printf("\n\n   ****** Array Original ******\n\n");
   ExibeArrayInts(ar1, 5);

   CopiaArray(ar2, ar1, 5);
```

```
    printf("\n\n   ****** Array Copiado ******\n\n");
    ExibeArrayInts(ar2, 5);

    return 0;
}
```

Esse último programa é bastante simples e, se você vem acompanhando atentamente o material exposto até aqui, não terá dificuldade em entendê-lo.

Observando-se a definição da função `CopiaArray()`, nota-se que ela possui dois parâmetros de entrada e um parâmetro de saída:

- O parâmetro `destino` é um ponteiro para um array cujos elementos terão atribuídos valores oriundos do array apontado por `origem`. Por outro lado, os elementos do array apontado por `destino` não são consultados. Portanto `destino` é um parâmetro de saída.
- O parâmetro `origem` representa um parâmetro de entrada, pois os elementos do array para os quais ele aponta são acessados apenas para serem copiados para os elementos do array apontado por `destino`. Isto é, o array apontado por `origem` não é alterado; ele é apenas consultado e, portanto, é um parâmetro de entrada. A propósito, é suficiente examinar o cabeçalho da função para suspeitar que o array apontado por `origem` não é alterado em virtude da presença de **const** na declaração desse parâmetro.
- Claramente, o parâmetro `nElementos` também é um parâmetro de entrada (v. **Seção 5.5.1**).

Quando a função `CopiaArray()` é chamada no corpo da função **main()**, o primeiro parâmetro real passado para aquela função é o endereço do array `ar2[]`, que é um array do mesmo tamanho do array `ar1[]` que será copiado. Nessa circunstância, a função cumpre seu objetivo perfeitamente bem e copia o conteúdo do array `ar1[]` para o array `ar2[]`. Mas, o que aconteceria se o array `ar2[]` não tivesse o mesmo tamanho de `ar1[]`? Bem, se o tamanho do array `ar2[]` fosse maior do que o tamanho de `ar1[]`, não haveria problema, pois o resultado da cópia poderia ser acomodado com folga no array `ar2[]`. Agora, o que ocorreria se o array `ar2[]` tivesse um tamanho menor do que o tamanho do array a ser copiado? A resposta a essa última questão será apresentada na próxima seção.

8.9.3 Corrupção de Memória

Se, no último programa da **Seção 8.9.2**, o tamanho do array `ar2[]` fosse menor do que o tamanho do array a ser copiado, o programador estaria diante de uma situação catastrófica que acomete muitos programas escritos em C: **corrupção de memória**. Esse tipo de problema ocorre quando porções de memória que não estão reservadas para uma determinada variável são alteradas acidentalmente. Corrupção de memória pode causar aborto de um programa (melhor) ou fazer com que seu comportamento seja errático (pior). Para entender bem o problema, considere a seguinte versão ligeiramente modificada da função **main()** do programa da **Seção 8.9.2**.

```
int main(void)
{
    int ar2[3];
    int i = 25;
    int ar1[10] = {5, 2, 6, 8, 0};

    printf("\nValor de i: %d", i);

    printf("\n\n****** Array Original ******\n\n");
    ExibeArrayInts(ar1, 10);

    CopiaArray(ar2, ar1, 10);

    printf("\n\n****** Array Copiado ******\n\n");
    ExibeArrayInts(ar2, 10);
```

```
    printf("\nValor de i: %d\n", i);
    return 0;
}
```

As diferenças entre essa função **main()** e aquela do programa da Seção 8.9.2 são as seguintes:

- Os tamanhos dos arrays `ar1[]` e `ar2[]` foram alterados: agora, `ar1[]` tem dez elementos e `ar2[]` tem apenas três elementos.
- Foi introduzida a variável `i` com o objetivo de mostrar que essa variável é alterada *misteriosamente* (i.e., sem a aplicação de nenhum operador com efeito colateral sobre ela).
- O valor da variável `i` é apresentado antes e depois da chamada da função `CopiaArray()` para demonstrar o efeito nefasto da corrupção de memória.

Quando executado, o programa contendo a função **main()** acima produz o seguinte surpreendente resultado:

```
Valor de i: 25
        ****** Array Original ******
        { 5, 2, 6, 8, 0, 0, 0, 0, 0, 0 }
        ****** Array Copiado ******
        { 5, 2, 6, 8, 0, 0, 0, 0, 0, 0 }
Valor de i: 8
```

Observe no resultado do último programa que a variável `i` apresenta dois valores diferentes sem que ela tenha sido aparentemente modificada no programa. Esse tipo de erro é comumente denominado erro lógico (v. Seção 7.4.1) e é difícil de corrigir porque causa o mau funcionamento do programa sem abortá-lo, o que deixa o programador sem pista do que possa ter acontecido de errado.

Coincidentemente, esse último programa também causou um erro de execução (novamente, v. Seção 7.4.1) que fez com ele fosse abortado logo antes de encerrar em virtude de violação de memória. Erros de execução são um pouco mais fáceis de depurar do que erros lógicos porque eles deixam rastros, mas, mesmo assim, frequentemente, causam transtornos para o programador.

Enfim, todos os problemas apresentados pelo último programa são decorrentes da chamada de função:

```
CopiaArray(ar2, ar1, 10);
```

Ou seja, como o array `ar2[]` passado como parâmetro não tinha espaço suficiente para conter o resultado da cópia, ocorreu corrupção de memória. Portanto, para evitar problemas em seus programas decorrentes de corrupção de memória, siga sempre a seguinte recomendação:

Recomendação *Quando um array é passado como parâmetro de saída ou de entrada e saída, ele deve ter espaço suficiente para acomodar o resultado produzido pela função.*

Corrupção de memória não ocorre apenas em chamadas de funções, como mostra o primeiro programa apresentado como exemplo na Seção 8.3.

8.9.4 Retorno de Arrays e Zumbis

Do mesmo modo que uma função nunca recebe um array completo como parâmetro, também não é permitido a uma função retornar um array completo. Mas é permitido a uma função retornar o endereço de um array (ou de qualquer outro tipo de variável). Todavia, o programador deve tomar cuidado para não retornar um

endereço de um array de duração automática (i.e., definido no corpo da função sem o uso de **static**), pois tal array é liberado quando a função retorna. Aliás, esse conselho não se aplica apenas no caso de arrays; ele é mais abrangente e deve ser sempre seguido:

Recomendação *Nunca retorne o endereço de uma variável de duração automática.*

Para entender melhor a dimensão do problema que surge quando essa recomendação não é seguida, considere como exemplo o seguinte programa:

```
#include <stdio.h>

void ExibeArrayEmTabela(const int ar[], int tamanho)
{
   int i;
   int array[10] = {0}; /* Apenas para acordar o zumbi */

   printf("\nIndice\tElemento\n");

   for (i = 0; i < tamanho; ++i) {
      printf("\n  %d\t    %d", i, ar[i]);
   }
}

int *RetornaZumbi(void)
{
   int zumbi[5];
   int i;

   for (i = 0; i < 5; ++i)
      zumbi[i] = i;

   printf("\n*** No corpo de RetornaZumbi() ***\n");
   ExibeArrayEmTabela(zumbi, 5);

   return zumbi; /* Retorna o endereço de um array zumbi */
}

int main(void)
{
   int *ar;

   ar = RetornaZumbi();

   printf("\n\n*** No corpo de main() ***\n");
   ExibeArrayEmTabela(ar, 5);

   return 0;
}
```

Quando compilado com GCC e executado no Windows, esse programa produz o seguinte resultado:

```
*** No corpo de RetornaZumbi() ***

Indice  Elemento

  0         0
  1         1
  2         2
  3         3
  4         4
```

CONTINUA

```
*** No corpo de main() ***
Indice  Elemento
  0         0
  1         0
  2         0
  3         0
  4         0
```

O que programa acima faz é apenas exibir na tela o conteúdo do array `zumbi[]` duas vezes. A primeira exibição se dá no corpo da função `RetornaZumbi()`, no qual o referido array é definido, enquanto a segunda exibição do array ocorre no corpo da função **main()**, que usa o endereço do array retornado pela função `RetornaZumbi()`.

O que há de surpreendente no resultado apresentado pelo último programa é que os valores dos elementos do array `zumbi[]`, cujo endereço é retornado pela função `RetornaZumbi()`, parecem ter sido alterados como num passe de mágica. Quer dizer, com exceção da instrução **for** no corpo de `RetornaZumbi()`, que atribui valores aos elementos do array, não há nenhuma outra instrução do programa que altere explicitamente o valor de qualquer elemento desse array. Então, como os valores desse array foram alterados?

A função `RetornaZumbi()` apresentada nesse último exemplo retorna o endereço de um array de duração automática alocado no corpo da função e, conforme foi exposto na **Seção 5.9.1**, o espaço ocupado por esse array é liberado quando a função retorna. Quer dizer, o espaço reservado para a variável `zumbi[]` durante a chamada da função, obviamente, continua a existir quando ela retorna, mas, como esse espaço está liberado para alocação de outras variáveis ou parâmetros, ele poderá ser alterado (i.e., ele está *vivo*) sem que isso ocorra por intermédio da variável `zumbi[]` (que deveria estar *morta*) ou de seu endereço. Por isso, esse gênero de variável recebe a denominação de *zumbi*.

Claramente, o último exemplo apresentado é artificial e foi criado com o intuito de forçar a manifestação do efeito zumbi. Esse efeito é provocado pela singela função `ExibeArrayEmTabela()` que, como disfarce, até usa **const**, comprometendo-se a não alterar o array recebido como parâmetro. De fato, essa função não altera diretamente o array recebido como parâmetro, mesmo porque, se ela tentasse fazê-lo, o compilador indicaria um erro em virtude do uso de **const**. Agora, observando-se essa função nota-se que ela define um array de duração automática e inicia todos os elementos desse array com zero:

```
int array[10] = {0};
```

Esse array não tem nenhuma finalidade prática na função `ExibeArrayEmTabela()`, mas tem um efeito devastador sobre o array `zumbi[]`, pois uma parte de seus elementos é alocada exatamente no espaço ocupado por esse array, já que esse espaço está liberado. Isso explica o fato de os elementos do array `zumbi[]` apresentarem valores nulos na segunda exibição desse array.

Apesar de esse exemplo ter sido um tanto constrito, o efeito zumbi decorrente do retorno do endereço de uma variável de duração automática pode ocorrer em situações práticas e, nesse caso, provavelmente, será muito mais difícil descobrir a causa do problema.

Finalmente, deve-se salientar que o problema ocorrido com o último programa é decorrente do fato de o array `zumbi[]` retornado pela função `RetornaZumbi()` ter duração automática e não por causa do fato de ele ser local à função (como informa o compilador GCC). Ou seja, o problema aqui é de duração e não de escopo. Portanto, para corrigir o programa anterior, basta incluir **static** na definição do array cujo endereço é retornado pela função `RetornaZumbi()` para torná-lo uma variável de duração fixa:

```
static int naoEhZumbi[5];
```

Um bom compilador emite uma mensagem de advertência quando há iminência de retorno de zumbis. O compilador GCC, por exemplo, emite a seguinte mensagem quando compila o programa em discussão[2]:

```
C:\Programas\zumbi.c In function 'RetornaZumbi':
C:\Programas\zumbi.c 36 warning: function returns address of local variable
```

Apesar de essa mensagem de advertência ser imprecisa, ela indica que um desastre está prestes a acontecer durante a execução do programa. Portanto, mais uma vez, nunca menospreze uma mensagem de advertência emitida por um compilador antes de examiná-la cuidadosamente.

Infelizmente, nem todo compilador indica problemas de forma tão clara e precisa quanto Clang, disponível no Mac OS X. Após compilar o último programa em discussão, esse compilador emite uma mensagem de advertência descrevendo exatamente qual é o problema com o programa:

```
zumbi.c:36:11: warning: address of stack memory associated with
local variable 'zumbi' returned
   return zumbi;
          ^~~~~
```

Antes de encerrar a corrente discussão, é importante salientar ainda que parâmetros são tratados como variáveis de duração automática no sentido de que eles são alocados quando a função que os usa é chamada e são liberados quando a mesma função é encerrada. Assim, retornar o endereço de um parâmetro também produz efeito zumbi. Portanto siga mais esta recomendação adicional referente a zumbis:

Recomendação *Nunca retorne o endereço de um parâmetro.*

8.9.5 Qualificação de Parâmetros com const

Os exemplos de uso de **const** apresentados na Seção 8.8 tiveram caráter quase que exclusivamente didático. Quer dizer, eles têm pouca utilidade prática. O uso mais pragmático e efetivo de **const** é a qualificação de parâmetros formais e reais, notadamente quando eles representam estruturas (v. Capítulo 10) e arrays.

Com relação à qualificação com **const** de parâmetros reais e formais, a seguinte recomendação deve ser seguida:

Recomendação *Se uma variável for qualificada com const, seu endereço só deverá ser passado como parâmetro real se o respectivo parâmetro formal for um ponteiro para o tipo da variável e for qualificado com const.*

Se uma variável não for qualificada com **const**, a restrição acima não se aplica. Por exemplo, suponha a existência das funções `F1()` e `F2()` com os seguintes protótipos:

```
void F1(int *p)
void F2(const int *p)
```

Considere ainda as seguintes definições de variáveis:

```
const int x = 10;
int       y = 20;
```

Então, as seguintes chamadas são consideradas legais e válidas:

```
F2(&x); /* OK */
F1(&y); /* OK */
F2(&y); /* OK */
```

[2] O nome do programa-fonte é `zumbi.c` e ele se encontra alojado no diretório `C:\Programas`.

Mas, a chamada a seguir não é considerada válida:

```
F1(&x); /* Inválida */
```

Se o conteúdo de um array é qualificado com **const**, a seguinte norma deve ser obedecida:

Recomendação *Se o conteúdo de um array for qualificado com const, seu endereço só deverá ser passado como parâmetro real se o respectivo parâmetro formal for um ponteiro para o tipo do elemento do array e for qualificado com const.*

Como exemplo de aplicação dessa última recomendação, suponha a existência das funções `G1()` e `G2()` com os seguintes protótipos:

```
void G1(int ar[])
void G2(const int ar[])
```

Conforme foi visto na Seção 8.9.1, `int ar[]` é o mesmo que `int *ar` e `const int ar[]` é o mesmo que `const int *ar`.

Agora, se os arrays `ar1[]` e `ar2[]` forem definidos como:

```
const int ar1[] = {1, 2, 3, 4};
int       ar2[] = {5, 6, 7, 8};
```

então, as seguintes chamadas serão consideradas legais e válidas:

```
G2(ar1); /* OK */
G1(ar2); /* OK */
G2(ar2); /* OK */
```

Entretanto, a chamada a seguir será considerada inválida:

```
G1(ar1); /* Inválida */
```

Em geral, o uso de **const** protege contra erros de programação provocados por instruções que, inadvertidamente, alteram dados que não deveriam ser alterados. Uma função que qualifica um parâmetro formal com **const** deve receber um parâmetro real correspondente qualificado ou não com **const**. Entretanto, se um parâmetro formal *não* for qualificado com **const**, o parâmetro real correspondente não pode ser qualificado com **const**. Por exemplo:

```
void G(int *n)
{
   *n = 0;
}
int F(const int *n)
{
   G(n);  /* Inválido */
   return 0;
}
```

8.10 Arrays Multidimensionais

Array multidimensional é um array cujos elementos também são arrays. Os arrays vistos até aqui são, em contrapartida, denominados **unidimensionais**. Um array **bidimensional** é aquele cujos elementos são arrays unidimensionais, um array **tridimensional** é aquele cujos elementos são arrays bidimensionais e assim por

diante. Algumas áreas de conhecimento, como Física e Engenharia, usam arrays multidimensionais com frequência, mas em cursos da área de Ciência da Computação o estudo em profundidade desses arrays pode ser adiado para disciplinas mais avançadas de programação. Portanto, neste livro, serão apresentadas apenas noções básicas de arrays multidimensionais.

Um array multidimensional é definido com pares consecutivos de colchetes, cada um dos quais contendo o tamanho de cada dimensão:

$$\textit{tipo-do-elemento nome-do-array}[\textit{tamanho}_1][\textit{tamanho}_2]...[\textit{tamanho}_N]$$

Embora o padrão ISO determine que um compilador de C deve suportar pelo menos seis dimensões para arrays multidimensionais, raramente, mais de três ou quatro dimensões são utilizadas em aplicações práticas.

No exemplo abaixo, um array tridimensional de caracteres é definido:

```
char arrayDeCaracteres[3][4][5];
```

A variável `arrayDeCaracteres` desse exemplo é interpretada como um array com três elementos, sendo cada um dos quais um array com quatro elementos, cada um dos quais é um array com cinco elementos do tipo **char**.

Para acessar um elemento de um array multidimensional, utilizam-se tantos índices quanto forem as dimensões do array. Por exemplo, um array tridimensional, como o do último exemplo, requer três índices para o acesso de cada elemento, como por exemplo:

```
arrayDeCaracteres[1][0][2] = 'A';
```

Para iniciar um array multidimensional, devem-se colocar os valores dos elementos de cada dimensão do array entre chaves. Se, para uma dada dimensão, houver um número de valores menor do que o número especificado na definição do array, os elementos remanescentes receberão zero como valor. A seguir, é apresentado um exemplo de iniciação de um array bidimensional:

```
int  arBi[5][3] = { {1, 2, 3},
                    {4},
                    {5, 6, 7}
                  };
```

Frequentemente, a primeira e a segunda dimensões de um array bidimensional são denominadas, respectivamente, **linha** e **coluna**. Assim, no último exemplo, `arBi[]` é definido como um array com 5 linhas e 3 colunas, mas apenas suas três primeiras linhas são iniciadas explicitamente e apenas o primeiro elemento da sua segunda linha é iniciado explicitamente.

Para declarar um array multidimensional como parâmetro formal de uma função, devem-se especificar os tamanhos de todas as dimensões, exceto o tamanho da primeira dimensão. Por outro lado, para passar um array multidimensional como parâmetro real para uma função, deve-se proceder da mesma forma que com arrays unidimensionais. Isto é, apenas o nome do array deve ser utilizado na chamada. O programa apresentado a seguir ilustra esses pontos.

```
void ExibeArrayBi(int a[][2], int dim1, int dim2)
{
   int i, j;

   putchar('\n'); /* Embelezamento */

      /* Escreve o índice de cada coluna */
   for (i = 0; i < dim2; ++i) {
      printf("\t%d", i);
   }
```

```
   putchar('\n'); /* Embelezamento */

   for (i = 0; i < dim1; ++i) {
         /* Escreve o índice de cada linha */
      printf("\n%d", i);

         /* Escreve o valor de cada elemento */
      for (j = 0; j < dim2; ++j) {
         printf("\t%d", a[i][j]);
      }
   }

   putchar('\n'); /* Embelezamento */
}

int main ()
{
   int   ar[3][2] = {
                       {1, 2},
                       {3, 4},
                       {5, 6}
                     };

    ExibeArrayBi(ar, 3, 2);

    return 0;
}
```

Quando executado, o programa do último exemplo apresenta o seguinte na tela:

```
         0       1

0        1       2
1        3       4
2        5       6
```

8.11 Exemplos de Programação

8.11.1 Maior, Menor e Média de Elementos de um Array

Problema: Escreva um programa que determina o maior, o menor e as médias inteira e real dos elementos de um array de inteiros.

Solução:

```
#include <stdio.h>

/****
 * main(): Determina o maior, o menor e as médias inteira
 *         e real dos elementos de um array de inteiros
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int ar[] = {-1, 2, 0, -2, 3, 8, 10, -1, 11, 4};
   int i, /* Usada para acessar cada elemento do array */
       maior, /* Armazenará o maior valor */
       menor, /* Armazenará o menor valor */
```

```
      soma,  /* Soma dos elementos do array */
      nElem; /* Número de elementos do array */

   /* Assume-se inicialmente que o maior e o menor valores */
   /* são iguais ao primeiro elemento do array             */
  maior = menor = ar[0];

   /* Inicia a soma com zero, que é elemento neutro da soma */
  soma = 0;

   /* Calcula o número de elementos do array */
  nElem = sizeof(ar)/sizeof(ar[0]);

   /* Determina o maior, o menor e soma dos elementos do array ar[] */
  for (i = 0; i < nElem; ++i) {
      /* Se o elemento corrente for maior do que o valor da variável */
      /* 'maior', essa variável assume o valor do elemento corrente. */
   if (ar[i] > maior) {
      maior = ar[i];
   }

      /* Se o elemento corrente for menor do que o valor da variável */
      /* 'menor', essa variável assume o valor do elemento corrente. */
   if (ar[i] < menor) {
      menor = ar[i];
   }

      /* Acrescenta o corrente elemento à soma acumulada */
   soma = soma + ar[i];
  }

  printf("\n\t>>> Array <<<\n\n"); /* Embelezamento */

   /* Exibe o array */
  for (i = 0; i < nElem; ++i) {
   printf("%d  ", ar[i]);
  }

  printf("\n\n\t>>> Menor valor: %d", menor);
  printf("\n\t>>> Maior valor: %d", maior);
  printf("\n\t>>> Media Inteira: %d", soma/nElem);
  printf("\n\t>>> Media Real: %3.2f\n", (double) soma/nElem);

  return 0;
}
```

Resultado de execução do programa:

```
        >>> Array <<<

-1  2  0  -2  3  8  10  -1  11  4

        >>> Menor valor: -2
        >>> Maior valor: 11
        >>> Media Inteira: 3
        >>> Media Real: 3.40
```

8.11.2 Ocorrências de Valores num Array

Problema: (a) Escreva uma função que conta o número de ocorrências de um número inteiro num array de elementos do tipo **int**. (b) Escreva uma função **main()** que lê um número limitado de valores do tipo **int**, armazena-os num array e conta o número de ocorrências de cada valor.

Solução de (a):

```
/****
 *
 * Ocorrencias(): Conta o número de ocorrências de um
 *                número inteiro num array de inteiros
 *
 * Parâmetros: ar[] (entrada): o array que será pesquisado
 *             tam (entrada): número de elementos do array
 *             num (entrada): o número que será procurado
 *
 * Retorno: O número de ocorrências do número no array
 *
 ****/
int Ocorrencias(const int ar[], int tam, int num)
{
   int i,
       ocorr = 0; /* Conta o número de ocorrência */

      /* Acessa cada elemento do array e compara-o com 'num'. */
      /* Se forem iguais, incrementa o número de ocorrências. */
   for (i = 0; i < tam; ++i) {
      if (ar[i] == num) { /* Mais uma ocorrência */
         ++ocorr; /* Incrementa o número de ocorrências */
      }
   }

   return ocorr;
}
```

Solução de (b):

```
/****
 * main(): Determina o número de ocorrências de cada
 *         valor de elemento de um array de inteiros
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/

int main(void)
{
   int i, ar[MAX_ELEMENTOS], nElementos, nOcor;
      /* Apresenta o programa */
   printf( "\n\t>>> Este programa determina o numero de "
           "ocorrencias \n\t>>> de cada elemento de um array de inteiros\n" );

   /* Lê o número de elementos que serão introduzidos. Se esse valor for */
   /* menor do que MAX_ELEMENTOS, apenas parte do array ar[] será usada. */

   printf( "\nDigite o numero de elementos do array "
           "(min = 2, max = %d):\n\t> ", MAX_ELEMENTOS);

      /* O laço encerra quando for lido um valor maior do */
      /* que 1 e menor do que ou igual a MAX_ELEMENTOS    */
   while (1) {
      nElementos = LeInteiro();

      if (nElementos > 1 && nElementos <= MAX_ELEMENTOS) {
         break;
```

```
      } else {
         printf( "\nO valor deve estar entre 2 e %d\n\t> ", MAX_ELEMENTOS );
      }
   }

      /* >>> Lê os elementos <<< */

   printf("\nIntroduza os elementos do array\n");

   for (i = 0; i < nElementos; ++i) {
      printf("\t%d> ", i + 1); /* Prompt */
      ar[i] = LeInteiro(); /* Leitura */
   }

      /* Conta o número de ocorrências de cada elemento */
   for (i = 0; i < nElementos; ++i) {
      nOcor = Ocorrencias(ar, nElementos, ar[i]);
      printf( "\n\t>>> O numero %d aparece %d vez%s",
              ar[i], nOcor, nOcor > 1 ? "es" : "" );
   }
   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa determina o numero de ocorrencias
        >>> de cada elemento de um array de inteiros

Digite o numero de elementos do array (min = 2, max = 100):
        > 6

Introduza os elementos do array
        1> -1
        2> 2
        3> 2
        4> 1
        5> 3
        6> -1

        >>> O numero -1 aparece 2 vezes
        >>> O numero 2 aparece 2 vezes
        >>> O numero 2 aparece 2 vezes
        >>> O numero 1 aparece 1 vez
        >>> O numero 3 aparece 1 vez
        >>> O numero -1 aparece 2 vezes
```

Análise: Observe no exemplo de execução que o programa apresenta o inconveniente fato de ser repetitivo. Isto é, se um valor aparece duas vezes no array (como são os casos dos valores `2` e `-1` exibidos), o usuário será informado duas vezes sobre o mesmo fato. É fácil inferir que se um valor aparece `n` vezes num array, o usuário receberá a mesma respectiva informação `n` vezes. Essa inconveniência pode ser resolvida verificando se o valor de cada elemento já foi levado em consideração antes de apresentar seu número de ocorrências. Isso pode ser implementado alterando-se a última instrução **for** da função **main()**, conforme mostrado a seguir:

```
for (i = 0; i < nElementos; ++i) {
      /* Se o elemento tem ocorrência na porção do array que o   */
      /* antecede, ele já foi levado em consideração. Isso evita */
      /* que ele seja contado várias vezes.                      */
   if (Ocorrencias(ar, i, ar[i]) == 0) {
      nOcor = Ocorrencias(ar, nElementos, ar[i]);
      printf( "\nO numero %d aparece %d vez%s no array",
              ar[i], nOcor, nOcor > 1 ? "es" : "" );
   }
}
```

Nesse laço **for**, a condição da instrução **if**:

```
Ocorrencias(ar, i, ar[i]) == 0
```

deve ser satisfeita para que sejam contadas as ocorrências do valor do elemento de índice `i`. Nessa chamada da função `Ocorrencias()`, o número de elementos do array passado como segundo parâmetro é `i`, de modo que os elementos do array que se encontram entre `0` e `i - 1` serão analisados para verificar se o valor do elemento de índice `i` possui ocorrência nesse intervalo. Se esse for o caso, o número de ocorrências desse valor não será levado em conta novamente.

Fazendo a alteração do laço **for** proposta e executando o programa resultante com os mesmo dados de entrada do exemplo de execução anterior, o resultado obtido será:

```
        >>> Este programa determina o numero de ocorrencias
        >>> de cada elemento de um array de inteiros

Digite o numero de elementos do array (min = 2, max = 100):
        > 6

Introduza os elementos do array
        1> -1
        2> 2
        3> 2
        4> 1
        5> 3
        6> -1

        >>> O numero -1 aparece 2 vezes
        >>> O numero 2 aparece 2 vezes
        >>> O numero 1 aparece 1 vez
        >>> O numero 3 aparece 1 vez
```

8.11.3 Ocorrências de Dígitos num Número Inteiro Positivo

Problema: Escreva um programa que lê números inteiros positivos via teclado e determina a frequência de ocorrência de dígitos em cada um deles. Por exemplo, se o número introduzido for `2110`, o programa deverá informar que o dígito `1` aparece duas vezes, o dígito `2` aparece uma vez e o dígito `0` aparece uma vez.

Solução:

```
#include <stdio.h>   /* printf()    */
#include "leitura.h" /* LeInteiro() */

#define DIGITOS_BASE_DECIMAL  10 /* Número de dígitos na base decimal */

/****
 * main(): Determina a frequência de ocorrência de dígitos num número inteiro positivo
```

```
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int i,
       num,     /* Armazena o número introduzido  */
       digito;  /* Armazena cada dígito do número */

   int digitos[DIGITOS_BASE_DECIMAL] = {0};
         /* Os índices do array digitos[] representam os dígitos da base   */
         /* decimal e os elementos armazenarão as ocorrências dos dígitos. */
         /* É importante iniciar todos os elementos com zero.              */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa determina a frequencia de "
           "ocorrencia \n\t>>> de digitos num numero inteiro positivo\n" );

   /* Este programa não é amigável ao usuário. Ou seja, se o usuário */
   /* não digitar aquilo que o programa espera, ele é encerrado.     */

      /* Tenta ler um valor válido */
   printf("\n\t>>> Digite um numero positivo: ");
   num = LeInteiro();

      /* Este programa não tolera usuário chato */
   if (num <= 0) {
      printf( "\n\t>>> Voce deveria ter digitado um numero positivo. Bye.\n" );
      return 1;
   }

      /* Obtém cada dígito e incrementa o número de */
      /* ocorrências do dígito no array digitos[]   */
   for (i = num; i > 0; i = i/10) {
      digito = i%10; /* Obtém o próximo dígito */
      ++digitos[digito]; /* Incrementa o número de ocorrências do dígito obtido */
   }
   /*     >>>>> Apresenta o resultado <<<<<     */

   printf( "\n>>> Ocorrencias de digitos em %d <<<\n", num );

   for (i = 0; i < 10; ++i) {
         /* Dígitos que não ocorrem não são exibidos */
      if (digitos[i]) {
         printf( "\n\t>>> O digito %d ocorre %d vez%s", i,
                 digitos[i], digitos[i] > 1 ? "es" : "" );
      }
   }

   return 0;
}
```

Análise: Cada elemento do array `digitos[]` é um contador de ocorrência de um dígito da base decimal, de modo que o contador do dígito `i` (`0 ≤ i ≤ 9`) corresponde exatamente ao elemento de índice `i`. Assim, no corpo do primeiro laço **for**, para cada dígito obtido e armazenado na variável `digito`, sua ocorrência é levada em conta por meio do incremento do elemento que representa seu contador como: `++digitos[digito]`.

Exemplo de execução do programa:

```
        >>> Este programa determina a frequencia de ocorrencia
        >>> de digitos num numero inteiro positivo

        >>> Digite um numero positivo: 223453265

>>> Ocorrencias de digitos em 223453265 <<<

        >>> O digito 2 ocorre 3 vezes
        >>> O digito 3 ocorre 2 vezes
        >>> O digito 4 ocorre 1 vez
        >>> O digito 5 ocorre 2 vezes
        >>> O digito 6 ocorre 1 vez
```

8.11.4 Números de Fibonacci 2

Problema: Escreva um programa que verifica se um número inteiro introduzido via teclado faz parte de uma sequência de Fibonacci (v. Seção 2.10.4).

Solução: Esse problema é semelhante àquele discutido na Seção 7.6.7 e a solução apresentada a seguir produz essencialmente os mesmos resultados do citado exemplo. Entretanto, a presente solução é mais eficiente porque, nesse caso, apenas uma sequência de Fibonacci é gerada. Então, essa sequência é armazenada num array, de modo que verificar se um determinado número faz parte de uma sequência de Fibonacci reduz-se a verificar se ele é um elemento do array mencionado.

```
#include <stdio.h>   /* Entrada e saída */
#include "leitura.h" /* LeituraFacil    */

   /* Tamanho máximo do array que armazenará a sequência de Fibonacci */
#define MAX_ELEMENTOS 100

/****
 * EmArray(): Verifica se um número inteiro encontra-se num array de inteiros
 *
 * Parâmetros: ar[] (entrada): o array que será pesquisado
 *             tam (entrada): número de elementos do array
 *             num (entrada): o número que será procurado
 *
 * Retorno: Índice do número no array, se ele for encontrado -1, em caso contrário
 ****/
int EmArray(const int ar[], int tam, int num)
{
   int i;

      /* Compara cada elemento do array com o parâmetro 'num'. Se for */
      /* encontrado um elemento igual a 'num', retorna seu índice.    */
   for (i = 0; i < tam; ++i) {
      if (ar[i] == num) {
         return i; /* Encontrado um elemento igual */
      }
   }
   return -1; /* Valor recebido como parâmetro não foi encontrado no array */
}
/****
 * main(): Verifica se cada número inteiro introduzido pelo
 *         usuário faz parte de uma sequência de Fibonacci
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
```

```
int main(void)
{
   int fib[MAX_ELEMENTOS] = {1, 1}, /* Armazenará a maior sequência de Fibonacci */
       i, /* Indexador dos elementos da sequência */
       nElem, /* Número de elementos armazenados em fib[] */
       nTeste; /* Número que será testado */

      /* Encontra a maior sequência de Fibonacci possível para o tipo */
      /* int e armazena-a no array fib[]. O maior valor possível é o  */
      /* último encontrado antes da ocorrência de overflow.           */
   for (i = 2; i < MAX_ELEMENTOS; ++i) {
      fib[i] = fib[i - 1] + fib[i - 2];

         /* Verifica se ocorreu overflow */
      if (fib[i] <= fib[i - 1] || fib[i] <= fib[i - 2]) {
         nElem = i; /* Ocorreu overflow. O maior número de   */
         break;     /* Fibonacci do tipo int foi encontrado. */
      }
   }
            *** Aqui começa o programa para o usuário ***

      /* Se o laço for foi encerrado porque o índice do array atingiu seu valor */
      /* limite, talvez não tenha sido possível armazenar a maior sequência     */
      /* permitida. Logo, o programa precisa ser abortado voluntariamente para  */
      /* que o tamanho máximo do array seja acrescido. Esse tipo de problema    */
      /* pode ser contornado por meio de alocação dinâmica de memória (Cap. 12) */
   if (i == MAX_ELEMENTOS) {
      printf( "\n\t>>> Este programa contem um erro. Entre"
              "\n\t>>> em contato com o programador.\n\n" );
      return 1;
   }

   printf( "\n\t>>> Este programa verifica se cada numero "
           "inteiro\n\t>>> introduzido faz parte de uma "
           "sequencia de Fibonacci. \n\t>>> Digite um "
           "valor menor do que 1 para encerrar.\n" );

   printf( "\n\t>>> Numero de elementos da maior "
           "sequencia de Fibonacci:\n\t>>> %d\n", nElem);

   printf("\n\t>>> Maior numero de Fibonacci armazenado:\n\t>>> %d\n", fib[nElem-1]);

      /* A saída do laço ocorre quando o usuário   */
      /* digita um valor menor do que ou igual a 1 */
   while (1) {
      printf( "\n\nDigite o valor a ser testado:\n\t> ");
      nTeste = LeInteiro();

         /* Verifica se o valor digitado encerra o laço */
      if (nTeste < 1) {
         break;
      }

         /* Verifica se o valor digitado faz parte da  */
         /* sequência de Fibonacci armazenada em fib[] */
      i = EmArray(fib, nElem, nTeste);

         /* Apresenta o resultado da busca no array */
      if (i >= 0) {
         printf("\n\t>>> %d e' um numero de Fibonacci\n", nTeste);

            /* Se o índice retornado pelo array for 0,   */
```

```
            /* o valor introduzido pelo usuário foi 1, e a menor sequência */
            /* de Fibonacci que contém 1 possui 2 termos. Se o valor de i  */
            /* for diferente de 0, o número de termos da menor sequência   */
            /* será dada por i + 1.                                        */
         printf( "\n\t>>> A menor sequencia de Fibonacci"
                 "\n\t>>> que contem esse numero possui "
                 "%d elementos", !i ? 2 : i + 1 );
      } else {
         printf( "\n\t>>> %d NAO e' um numero de Fibonacci\n", nTeste );
      }
   }

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

   return 0;
}
```

Análise:

- O programa começa gerando e armazenando num array a maior sequência de Fibonacci possível, considerando que seus termos são do tipo **int**. O maior termo dessa sequência é o último valor gerado antes da ocorrência de overflow (v. Seção 4.11.7).
- Para essa abordagem funcionar, o array que armazena os termos da sequência deve ter tamanho suficiente para conter a maior sequência possível. Assim, faz-se uma estimativa inicial de tamanho por meio da definição da constante `MAX_ELEMENTOS`. Além disso, o programa inclui o teste:

```
if (i == MAX_ELEMENTOS) {
   printf( "\n\t>>> Este programa contem um erro. Entre"
           "\n\t>>> em contato com o programador.\n\n" );
   return 1;
}
```

  para checar se o laço **for** foi encerrado porque a variável `i` atingiu seu valor limite, e não porque ocorreu overflow. Se esse foi o caso, talvez não tenha sido possível armazenar a maior sequência esperada. Assim, o programa é abortado voluntariamente para que o tamanho máximo do array seja corrigido pelo programador. Essa limitação do programa pode ser contornada por meio de alocação dinâmica de memória (v. Capítulo 12).
- A função **main()** chama a função `EmArray()`, que verifica se um valor do tipo **int** faz parte de um array de elementos desse tipo. Essa última função é usada para checar se cada número introduzido pelo usuário faz parte do array que contém a maior sequência de Fibonacci.
- A abordagem adotada por esse programa é vantajosa com relação àquela utilizada pelo programa apresentado na Seção 7.6.7 quando é necessário testar se vários valores são números de Fibonacci, pois, na presente abordagem, todos os possíveis números de Fibonacci do tipo **int** são gerados uma única vez.

Exemplo de execução do programa:

```
>>> Este programa verifica se cada numero inteiro
>>> introduzido faz parte de uma sequencia de Fibonacci.
>>> Digite um valor menor do que 1 para encerrar.

>>> Numero de elementos da maior sequencia de Fibonacci:
>>> 46

>>> Maior numero de Fibonacci armazenado:
>>> 1836311903
```

CONTINUA

```
Digite o valor a ser testado:
        > 8

        >>> 8 e' um numero de Fibonacci

        >>> A menor sequencia de Fibonacci
        >>> que contem esse numero possui 6 elementos

Digite o valor a ser testado:
        > 0

        >>> Obrigado por usar este programa.
```

8.11.5 Ordenação de Arrays pelo Método da Bolha

Preâmbulo: Um **algoritmo de ordenação** consiste num procedimento que descreve como organizar os elementos de um array segundo certa ordem. Ordenação de dados é uma operação de grande importância em programação e, por isso, inúmeros algoritmos de ordenação têm sido propostos. Um dos algoritmos de ordenação mais simples (mas, também, um dos mais ineficientes) é conhecido como **método da bolha** (*Bubble Sort*, em inglês) e consiste no seguinte[3]:

- São efetuados vários acessos sequenciais aos elementos do array a ser ordenado.
- Em cada acesso, comparam-se, dois a dois, os elementos adjacentes do array.
- Se algum par de elementos estiver fora de ordem, esses elementos têm suas posições trocadas.
- O algoritmo encerra quando se faz um acesso sequencial do primeiro ao último elemento do array sem que ocorra nenhuma troca de posição entre dois elementos quaisquer.

Problema: (a) Escreva uma função que ordena um array de elementos do tipo **int** usando o método da bolha.
(b) Escreva um programa que testa a função do item (a).

Solução de (a):

```
/****
 * BubbleSort(): Ordena em ordem crescente um array de elementos do tipo int
 *               utilizando o método de ordenação da bolha (Bubble Sort)
 *
 * Parâmetros:
 *      ar (entrada e saída) - array que será ordenado
 *      nElementos (entrada) - número de elementos do array
 *
 * Retorno: Nada
 *
 * Observação: Esse é o método de ordenação mais simples de entender (e explicar),
 *             mas ele é um dos mais ineficientes.
 ****/
void BubbleSort(int ar[], int nElementos)
{
   int i, aux,
       ordenado = 0; /* Informará se o array está ordenado. É importante que */
                     /* essa variável seja iniciada com zero (v. abaixo).    */

   /*************************************************************************/
   /* Supõe-se que inicialmente o array está ordenado fazendo-se 'ordenado = 1'. */
   /* Então, se forem encontrados dois  elementos fora de ordem, atribui-se      */
   /* novamente zero a 'ordenado', trocam-se os elementos de posição e recomeça- */
   /* se a verificação de ordem do array.                                        */
   /*************************************************************************/
```

[3] A justificativa para a denominação desse método é a seguinte: os elementos menores (i.e., *mais leves*) vão aos poucos *subindo* para o início do array, como se fossem *bolhas*.

```
      /* O laço encerra quando a variável 'ordenado' for diferente de 0 */
   while (!ordenado) {
      ordenado = 1; /* Supõe que o array está ordenado */

      /* Compara cada elemento do array com o elemento seguinte. Se for */
      /* encontrado um elemento menor do que seu antecessor, os dois    */
      /* elementos trocam de posição e o array é considerado fora de    */
      /* ordem (i.e., 'ordenado' assume zero).                          */

      for (i = 0; i < nElementos - 1; i++){
            /* Compara cada elemento com o elemento seguinte */
         if (ar[i] > ar[i+1]){
               /* Pelo menos dois elementos estão fora de ordem */
            ordenado = 0;

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

Solução de (b):

```
#include <stdio.h>

/****
 * main(): Testa a função BubbleSort()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int array[] = {12, 55, 21, 1, 6, 8, 17, 220, 5, 83},
       i, tamanho;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa ordena um array de"
           "\n\t>>> inteiros usando o metodo da bolha.\n" );

      /* Calcula do tamanho do array */
   tamanho = sizeof(array)/sizeof(array[0]);

      /* Apresenta o array antes da ordenação */
   printf("\n\t>>> Array Original <<<\n\n\t\t> ");
   for (i=0; i < tamanho; i++) {
      printf("%d   ", array[i]);
   }

   BubbleSort(array, tamanho); /* Ordena o array */

      /* Apresenta o array depois da ordenação */
   printf("\n\n\t>>> Array Ordenado <<<\n\n\t\t> ");
   for (i=0; i < tamanho; i++) {
      printf("%d   ", array[i]);
   }
   return 0;
}
```

Resultado de execução do programa:

```
>>> Este programa ordena um array de
>>> inteiros usando o metodo da bolha.

>>> Array Original <<<

> 12   55   21   1   6   8   17   220   5   83

>>> Array Ordenado <<<

> 1   5   6   8   12   17   21   55   83   220
```

8.11.6 Invertendo um Array

Problema: Escreva um programa que lê um array de inteiros, inverte a ordem dos elementos e apresenta o resultado da operação na tela.

Solução:

```
/*********************** Includes *************************/

#include <stdio.h>   /* Entrada e saída */
#include "leitura.h" /* LeituraFacil    */
/***************** Constantes Simbólicas ******************/

#define MAX_ELEMENTOS   20 /* Número máximo de elementos do array */
#define ENCERRA_LEITURA  0 /* Valor que encerra a leitura de dados */

/************************ Alusões ************************/

extern int LeArray(int ar[], int max, int encerra);
extern void ExibeArrayInts(const int ar[], int tamanho);
extern void InverteArray(int ar[], int nElem);

/***************** Definições de Funções ******************/

/****
 * LeArray(): Lê valores inteiros e armazena-os num array. A leitura encerra
 *            quando o usuário digitar o valor que especifica o encerramento
 *            da leitura ou quando o número máximo de elementos for atingido.
 *
 * Parâmetros:
 *      ar (saída) - array que conterá os elementos lidos
 *      max (entrada) - número máximo de elementos lidos
 *      encerra (entrada) - valor que encerra a leitura
 *
 * Retorno: O número de elementos lidos
 ****/
int LeArray(int ar[], int max, int encerra)
{
   int nValoresLidos = 0, /* Número de valores lidos */
       elemento; /* Armazenará cada valor lido */

   printf("\a\n\t   *** Introduza os valores ***\n\n");

      /* O laço encerra quando o usuário digitar o valor que */
      /* especifica o encerramento da leitura ou quando o    */
      /* número máximo de elementos for atingido             */
   while (1) {
      if (nValoresLidos >= max) {/* Número máximo de      */
         break;                  /* elementos já foi lido */
      }
```

```
         /* Lê o próximo elemento do array */
      printf("\t>>> Valor: ");
      elemento = LeInteiro();

         /* Se o valor que identifica encerramento */
         /* de leitura foi lido, encerra o laço    */
      if (elemento == encerra) {
         break;
      }

      ar[nValoresLidos] = elemento; /* Armazena no array o elemento lido */

      ++nValoresLidos; /* Mas um elemento foi lido */
   }

   return nValoresLidos;
}

/****
 * InverteArray(): Inverte a ordem dos elementos de um array de ints
 *
 * Parâmetros:
 *      ar (entrada) - array que será invertido
 *      nElem (entrada) - número de elementos do array
 *
 * Retorno: Nada
 ****/
void InverteArray(int ar[], int nElem)
{
   int aux, i;

      /* Troca as posições do primeiro e último elementos, do */
      /* segundo e penúltimo elementos e assim por diante     */
   for (i = 0; i < nElem/2; ++i) {
      aux = ar[i];
      ar[i] = ar[nElem - i - 1];
      ar[nElem - i - 1] = aux;
   }
}

/****
 * main(): Lê um array de inteiros e inverte a ordem dos seus elementos
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int array[MAX_ELEMENTOS], n;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa le um conjunto de valores "
           "\n\t>>> inteiros e escreve-os em ordem inversa. "
           "\n\t>>> O valor %d encerra a leitura de dados e "
           "\n\t>>> o numero maximo de valores e' %d.\n",
           ENCERRA_LEITURA, MAX_ELEMENTOS );

      /* Lê os elementos do array e armazena */
      /* em 'n' o número de elementos lidos  */
```

```
   n = LeArray(array, MAX_ELEMENTOS, ENCERRA_LEITURA);

      /* Exibe o array em sua ordem original */
   printf("\n\t*** Ordem Original ***\n");
   ExibeArrayInts(array, n);

   InverteArray(array, n); /* Inverte a ordem dos elementos lidos */

      /* Exibe o array invertido */
   printf("\n\t*** Ordem Invertida ***\n");
   ExibeArrayInts(array, n);

   return 0;
}
```

Análise: A função **main()** chama a função `ExibeArrayInts()`, apresentada na Seção 8.9.2.

Exemplo de execução do programa:

```
>>> Este programa le um conjunto de valores
>>> inteiros e escreve-os em ordem inversa.
>>> O valor 0 encerra a leitura de dados e
>>> o numero maximo de valores e' 20.

   *** Introduza os valores ***

>>> Valor: 1
>>> Valor: 2
>>> Valor: 3
>>> Valor: 0

*** Ordem Original ***

{ 1, 2, 3 }

*** Ordem Invertida ***

{ 3, 2, 1 }
```

8.11.7 Permutações de Arrays

Preâmbulo: Uma **permutação** de um array consiste num rearranjo dos elementos do array de modo um novo array é constituído por simples alterações de posições dos elementos do array original. Isto é, um array é permutação de outro quando ambos são do mesmo tamanho, contêm os mesmos elementos, mas esses ocupam diferentes posições nos dois arrays. Por exemplo, os seguintes arrays são considerados permutações:

```
{1, 2, 3, 4}    {2, 1, 4, 3}
```

Uma técnica utilizada para obtenção de todas as permutações possíveis envolvendo os elementos de um array é denominada geração de permutações por **ordenação lexicográfica**. O algoritmo seguido por essa técnica requer que o array esteja inicialmente *em ordem crescente*. Assim, supondo que um array `ar[]` satisfaça esse requisito, a geração de permutações de acordo com esse algoritmo segue os seguintes passos:

1. Encontre o maior índice `i` do array `ar[]` tal que a expressão `ar[i-1] < ar[i]` seja satisfeita (i.e., resulte em `1`).
2. Se o índice `i` for igual a `0`, não existe nenhum elemento do array que seja maior do que seu antecessor; ou seja, o array está em ordem decrescente e sua configuração atual constitui a última permutação que pode ser gerada.
3. Atribua a `i` o maior o índice tal que a expressão `ar[i] < ar[i+i]` seja satisfeita. Esse valor é exatamente `i - 1`.

4. Encontre o maior índice `j` tal que a expressão `ar[i] < ar[j]` seja satisfeita. Como `ar[i]` é menor do que `ar[i+1]`, o índice `j` deve existir.
5. Troque os elementos `ar[i]` e `ar[j]`.
6. Inverta os elementos do array a partir do índice `i + 1` até o final.

Problema: (a) Escreva uma função, denominada `ProximaPermutacao()`, que recebe um array de elementos do tipo **int** e seu número de elementos como parâmetros e gera a *próxima permutação* em ordem lexicográfica, se essa existir. (b) Escreva uma função, denominada `GeraPermutacoes()`, que recebe um array de elementos do tipo **int** e seu número de elementos como parâmetros e gera e escreve na tela *todas as permutações* do array. (c) Escreva um programa que define e inicia um array de elementos do tipo **int** e invoca a função `GeraPermutacoes()` para gerar todas a permutações desse array.

Solução de (a):

```
/****
 * ProximaPermutacao(): Gera a próxima permutação de um array de elementos do tipo
 *                      int usando ordenação lexicográfica
 *
 * Parâmetros:
 *      ar (entrada/saída) - array cuja permutação será gerada
 *      n (entrada) - número de elementos do array
 *
 * Retorno: 1, se uma permutação foi gerada; 0, em caso contrário
 ****/
int ProximaPermutacao(int ar[], int n)
{
   int i, j;

      /*                    >>> Passo 1 <<<                     */
      /* Encontra o maior índice i tal que: ar[i - 1] < ar[i]  */
   for (i = n - 1; i != 0 && ar[i - 1] >= ar[i]; i--) {
      ; /* Vazio */
   }

      /*                 >>> Passo 2 <<<                */
      /* Se o índice i for igual a 0, não existe nenhum */
      /* elemento do array que seja maior do que seu    */
      /* antecessor. I.e., o array está em ordem        */
      /* decrescente e sua configuração atual constitui */
      /* a última permutação que pode ser gerada.       */
   if (!i) {
      return 0; /* Não há mais permutação a ser gerada */
   }
      /*                 >>> Passo 3 <<<                 */
      /* Faz i assumir o maior o índice tal que:         */
      /* ar[i] < ar[i + i]. Esse valor é exatamente i-1. */
   --i;

      /*            >>> Passo 4 <<<             */
      /* Encontra o maior índice j tal que:     */
      /* ar[i] < ar[j]. Como ar[i] < ar[i + 1], */
      /* j deve existir.                        */
   for (j = n - 1; ar[j] <= ar[i]; j--) {
      ; /* Vazio */
   }

      /*    >>> Passo 5 <<<    */
```

```
      /* Troca ar[i] com ar[j] */
   TrocaElementos(ar, i, j);

      /*      >>> Passo 6 <<<       */
      /* Inverte os elementos do    */
      /* array de i + 1 até o final */
   for (i++, j = n - 1; j > i; j--, i++) {
      TrocaElementos(ar, i, j);
   }

   return 1; /* Uma permutação foi gerada */
}
```

Análise: A função `ProximaPermutacao()` chama a função `TrocaElementos()`, que troca as posições de dois elementos de um array. Essa última função é definida como:

```
/****
 *
 * TrocaElementos(): Troca as posições de dois elementos
 *                   de um array de elementos do tipo int
 *
 * Parâmetros:
 *      ar[] (entrada/saída) - array contendo os elementos que trocarão de posições
 *      i, j (entrada) - índices dos elementos que serão trocados
 *
 * Retorno: Nenhum
 *
 ****/
void TrocaElementos(int ar[], int i, int j)
{
   int aux = ar[i]; /* Guarda o valor do elemento de índice i */

      /* O valor do elemento ar[i] está guardado */
   ar[i] = ar[j];

      /* Atribui a ar[j] valor do elemento ar[i] */
   ar[j] = aux; /* O valor de ar[i] foi guardado em 'aux' */
}
```

Comentários adicionais sobre a função `ProximaPermutacao()`:

A função `ProximaPermutacao()` é crucial para entendimento da técnica de geração de permutações descrita no preâmbulo. Portanto é importante que os passos do algoritmo implementado por essa função sejam examinados detalhadamente.

Suponha que a função `ProximaPermutacao()` recebe como parâmetro o array:

```
{1, 2, 4, 3}
```

Então, após a execução do **Passo 1** do algoritmo, os conteúdos do array e da variável `i` podem ser representados como na **Figura 8–8**.

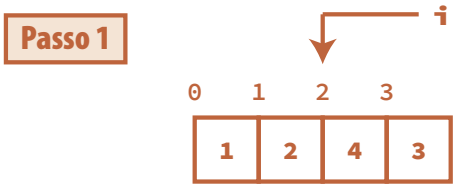

Figura 8–8: Passo 1 do Algoritmo de Permutações de Arrays

O Passo 2 do algoritmo verifica se o valor de `i` é zero (o que não é o caso), enquanto o Passo 3 decrementa o valor da variável `i`. Portanto, após a execução desse último passo do algoritmo, a situação pode ser representada como na Figura 8–9.

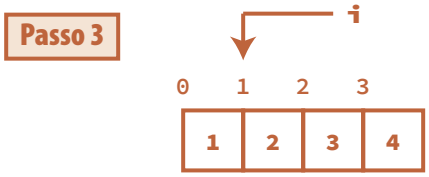

FIGURA 8–9: PASSO 3 DO ALGORITMO DE PERMUTAÇÕES DE ARRAYS

No Passo 4 do algoritmo, deve-se encontrar o maior índice `j` que satisfaz a relação `a[i] < a[j]`. Após a execução desse passo, a situação do array e das variáveis `i` e `j` pode ser representada como na Figura 8–10.

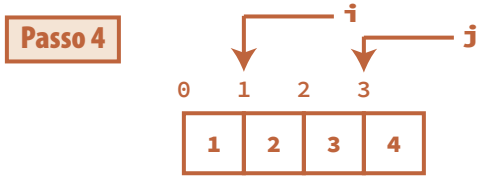

FIGURA 8–10: PASSO 4 DO ALGORITMO DE PERMUTAÇÕES DE ARRAYS

No Passo 5, os valores de `a[i]` e `a[j]`são trocado e o resultado é apresentado na Figura 8–11.

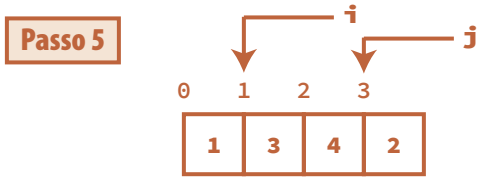

FIGURA 8–11: PASSO 5 DO ALGORITMO DE PERMUTAÇÕES DE ARRAYS

Finalmente, no Passo 6 do algoritmo, os elementos de índices `2` a `3` são invertidos, resultando na Figura 8–12.

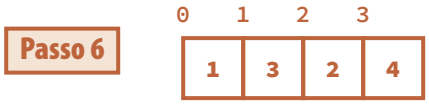

FIGURA 8–12: PASSO 6 DO ALGORITMO DE PERMUTAÇÕES DE ARRAYS

Solução de (b):

```
/****
 *
 * GeraPermutacoes(): Gera e exibe na tela todas as permutações de um array de
 *                    elementos do tipo int usando ordenação lexicográfica
 * Parâmetros:
 *      ar (entrada/saída): array cujas permutações serão geradas
 *      n (entrada): número de elementos do array
 *
 * Retorno: Nenhum
 *
 ****/
void GeraPermutacoes(int ar[], int n)
{
        /* Para a técnica de ordenação lexicográfica funcionar, é essencial */
        /* que a geração de permutações seja iniciada com o array ordenado  */
        /* em ordem crescente.                                              */
        /*                                                                  */
```

```
   BubbleSort(ar, n); /* Ordena o array ar[] em ordem crescente */

   printf("\n     >>> Permutacoes <<<\n");

      /* Exibe todas as permutações na tela. O laço termina */
      /* quando a função ProximaPermutacao() retornar 0,    */
      /* indicando que não há mais permutação a ser gerada. */
   do {
      ExibeArrayInts(ar, n); /* Exibe a configuração corrente do array */
   } while (ProximaPermutacao(ar, n));
}
```

Análise: Inicialmente, a função `GeraPermutacoes()` chama a função `BubbleSort()`, apresentada na Seção 8.11.5, para ordenar o array recebido como parâmetro. Então, a função `GeraPermutacoes()` executa um laço que chama a função `ExibeArrayInts()` (v. Seção 8.9.2) para apresentar a configuração corrente do array e, em seguida, chama a função `ProximaPermutacao()`, que gera a próxima configuração do array.

Solução de (c): O complemento do programa é deixado como exercício para o leitor. (A solução completa deste problema encontra-se no site do livro: *www.ulysseso.com/ip*.)

8.11.8 Mega-sena Gratuita (mas sem Premiação)

Problema: Escreva um programa que simula o jogo de azar (legalizado) Mega-sena. Isto é, o programa deve ler uma aposta efetuada pelo usuário, realizar o sorteio dos seis números da loteria, comparar a aposta do usuário com os números sorteados e, finalmente, informar o usuário se ele ganhou ou perdeu.

Solução: É lamentável saber que programadores desperdiçam dinheiro à toa jogando na Mega-sena, pois é tão simples implementar esse jogo. O que o programa proposto deve implementar é o seguinte:

1. **Ler a quantidade de números que o usuário apostará**. Na mega-sena oficial, essa quantidade varia entre `6` e `15`.
2. **Ler cada número que integra a aposta do usuário**. Cada número numa aposta pode ser um valor entre `1` e `60`.
3. **Fazer o sorteio da loteria.** Sorteio em programação em C, conforme foi discutido na Seção 4.10, pode ser efetuado com auxílio das funções **srand()** e **rand()**, mas é necessário limitar o valor sorteado ao intervalo compreendido entre `1` e `60`. Ou seja, a fórmula usada para o sorteio de números da Mega-sena é: `rand()%60 + 1` (v. Seção 4.10).
4. **Comparar a aposta do usuário com os números sorteados**. É conveniente armazenar tanto os números sorteados quanto a aposta do usuário em arrays. Assim, para verificar se a aposta do usuário foi premiada, basta verificar se cada elemento do array que contém os números sorteados faz parte do array que contém a aposta do usuário. Logo, quando for detectado que um número sorteado não consta na aposta do usuário, os demais números não precisam mais ser verificados, porque pode-se imediatamente concluir que o usuário perdeu. (Esse programa não verifica se o usuário ganhou algum prêmio secundário.) A função `EmArray()`, apresentada na Seção 8.11.4 pode ser usada com essa finalidade.
5. **Informar o usuário se ele ganhou ou perdeu**. Essa parte do programa é trivial, mas testar o programa com valores qualitativamente diferentes de modo que ele apresente todas as saídas possíveis, conforme foi preconizado no Capítulo 2, é complicado pelo fato de ser muito difícil acertar os números sorteados. Isto é, se o sorteio resultante da função **rand()** for tão aleatório quanto aquele da Mega-sena real e o programador decidir testar o programa com apostas de seis números (quem tem paciência de testar com 15 números?), a probabilidade de ele acertar e, consequentemente, obter a respectiva saída do programa é da ordem de 1 para 50 milhões. Assim, para testar o programa quando a aposta é ganhadora,

a melhor opção para o programador é trapacear (no bom sentido). Ou seja, durante a fase de testes, o programa apresenta os números sorteados para o usuário antes de ele apostar, de modo que ele pode fazer uma aposta contendo os números sorteados e verificar se o programa funciona nessa situação. Outra opção seria comentar a instrução contendo a chamada de **srand()** que alimenta o gerador de números aleatórios, de modo que os mesmos números fossem sorteados a cada execução do programa.

Para evitar o uso de números mágicos, o programa proposto deve incluir em seu início as seguintes definições de constantes simbólicas:

```
#define NUMEROS_LOTO       6 /* Números sorteados           */
#define MENOR_NUMERO_LOTO  1 /* O menor número da Mega-sena */
#define MAIOR_NUMERO_LOTO 60 /* O maior número da Mega-sena */
#define MAX_APOSTAS       15 /* Número máximo de apostas    */
```

Além disso, todos os valores que o programa necessita são inteiros positivos que se encontram dentro de um determinado intervalo. A função `LeIntEntre()`, apresentada a seguir, serve para ler números com esse perfil:

```
/****
 * LeIntEntre(): Lê um número inteiro entre os valores especificados pelos parâmetros
 *
 * Parâmetros: menor (entrada): o menor valor permitido
 *             maior (entrada): o maior valor permitido
 *
 * Retorno: O número lido
 *
 * Observação: Esta função assume que o primeiro parâmetro
 *             é o menor e o segundo parâmetro é o maior
 ****/
int LeIntEntre(int menor, int maior)
{
   int num;

   /* Volta para cá a cada tentativa frustrada de leitura */
inicioLeitura:
      /* Apresenta prompt e faz uma tentativa de leitura */
   printf( "\n\t>>> Digite um valor inteiro entre %d e %d\n\t\t> ", menor, maior );
   num = LeInteiro();

      /* Verifica se o valor introduzido satisfaz os critérios especificados. */
      /* Se não for o caso, apresenta uma mensagem informando o erro ao       */
      /* usuário, e faz uma nova tentativa de leitura.                        */
   if (num < menor || num > maior) {
      printf("\n\t>>> Valor invalido\n");
      goto inicioLeitura; /* Não cria código espaguete */
   }

   return num; /* Seguramente, o número retornado satisfaz as especificações */
}
```

A função **main()** que completa o programa é a seguinte:

```
/****
 * main():
 *   * Lê a quantidade de números que o usuário apostará
 *   * Lê cada número que integra a aposta do usuário
 *   * Faz o sorteio da loteria
 *   * Compara a aposta do usuário com os números sorteados
 *   * Informa o usuário se ele ganhou ou perdeu
 *
```

```
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int    sorteio[NUMEROS_LOTO], /* Conterá os números sorteados */
          aposta[MAX_APOSTAS], /* Conterá a aposta do usuário */
          i,
          nApostas, /* Quantidade de números que o usuário apostará */
          num,  /* Um número de aposta do usuário */
          perdeu = 0; /* Indicará se o usuário foi sorteado ou não */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa permite que voce jogue na Mega-sena sem"
           "\n\t>>> gastar um centavo. Em compensacao, a possibilidade de"
           "\n\t>>> voce ficar rico e' zero.\n" );

     /* Lê a quantidade de números que o usuário apostará */
   printf("\n\t>>> Quantos numeros voce ira' apostar?");
   nApostas = LeIntEntre(NUMEROS_LOTO, MAX_APOSTAS);

   printf("\n\t>>> Digite sua aposta <<<\n");

      /* Lê aposta do usuário */
   for (i = 0; i < nApostas; ) {
         /* Lê um número da aposta */
      num = LeIntEntre(MENOR_NUMERO_LOTO, MAIOR_NUMERO_LOTO);

         /* Verifica se o usuário já apostou nesse número */
      if ( i > 0 && EmArray(aposta, i, num) >= 0 ) {
         printf("\n\t>>> Voce ja' apostou neste numero\n");
      } else {
         aposta[i] = num; /* Número ainda não apostado */
            /* O incremento de i é colocado aqui, em vez de como  */
            /* expressão do laço for, porque ele  só deve ocorrer */
            /* quando o número não for repetido                   */
         ++i;
      }
   }

   srand(time(NULL)); /* Inicia o gerador de números aleatórios */

   for (i = 0; i < NUMEROS_LOTO; ++i) {/* Faz o sorteio */
      sorteio[i] = rand()%MAIOR_NUMERO_LOTO + MENOR_NUMERO_LOTO;
   }

      /* Verifica se o usuário foi sorteado */
   for (i = 0; i < nApostas; ++i) {
         /* Verifica se um número sorteado faz parte da aposta do usuário */
      if (EmArray(sorteio, NUMEROS_LOTO, aposta[i]) < 0) {
            /* Se o usuário não acertou um dado número sorteado, ele já   */
            /* perdeu e não adianta verificar os demais números sorteados */
         perdeu = 1;
         break;
      }
   }

         /* >>> Apresenta os números sorteados <<< */

   printf("\n>>> Os numeros sorteados foram:\n\t");
```

```
   for (i = 0; i < NUMEROS_LOTO; ++i) {
      printf("\t%2d", sorteio[i]);
   }

         /* >>> Apresenta os números do usuário <<< */
   printf("\n\n>>> Sua aposta foi:\n\t");

   for (i = 0; i < nApostas; ++i) {
      printf("\t%2d", aposta[i]);
   }

   if (perdeu) { /* Informa o usuário se ele ganhou ou perdeu */
      printf("\n\n\t>>> Infelizmente voce nao ganhou.\n");
   } else {
      printf("\n\n\t>>> Parabens! Voce ganhou!"
          "\n\t>>> Agora, seja generoso e divida o premio com o criador do programa.\n" );
   }
   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa permite que voce jogue na Mega-sena sem
        >>> gastar um centavo. Em compensacao, a possibilidade de
        >>> voce ficar rico e' zero.

        >>> Quantos numeros voce ira' apostar?
        >>> Digite um valor inteiro entre 6 e 15
                > 6

        >>> Digite sua aposta <<<

        >>> Digite um valor inteiro entre 1 e 60
                > 12

        >>> Digite um valor inteiro entre 1 e 60
                > 66
        >>> Valor invalido
       >>> Digite um valor inteiro entre 1 e 60
                > 12
       >>> Voce ja' apostou neste numero

        >>> Digite um valor inteiro entre 1 e 60
                > 3

        >>> Digite um valor inteiro entre 1 e 60
                > 55

        >>> Digite um valor inteiro entre 1 e 60
                > 33

        >>> Digite um valor inteiro entre 1 e 60
                > 22

        >>> Digite um valor inteiro entre 1 e 60
                > 44

>>> Os numeros sorteados foram:
                23      29      48      36      39      33

>>> Sua aposta foi:
                12       3      55      33      22      44

        >>> Infelizmente voce nao ganhou.
```

8.11.9 Qual É o Troco?

Problema: Escreva um programa que apresenta na tela os números de cédulas e moedas que devem ser entregues a um cliente como troco de uma compra. O programa encerra quando o usuário introduz zero como valor de uma compra. Além disso, antes de encerrar, o programa apresenta o número total de cédulas e moedas de cada tipo que forem entregues durante sua execução. **Observação**: Para simplificar o problema, considere R$1 como cédula e não como moeda.

Solução:

```
#include <stdio.h>   /* Entrada e saída */
#include <math.h>    /* Função fabs()   */
#include "leitura.h" /* LeituraFacil    */

#define MAIOR_TROCO 99.99  /* Maior troco recomendado por este programa */
#define N_CEDULAS  6       /* Número de tipos de cédulas */
#define N_MOEDAS   4       /* Número de tipos de moedas  */
   /* Precisão utilizada na comparação de números reais. Esse valor  */
   /* também é adicionado a valores reais para evitar erros de       */
   /* truncamento na conversão desses valores para inteiros.         */
#define DELTA 1.0E-14

/****
 *
 * main(): Apresenta na tela os números de cédulas e moedas que
 *         devem ser cedidas a clientes como trocos de compras
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   double valorCompra, /* Valor da compra */
          pago, /* Valor pago pelo cliente */
          troco, /* Valor do troco */
          trocoCorrigido; /* Correção para evitar erro de truncamento */
             /* Array que contém as cédulas disponíveis */
   int    cedulas[N_CEDULAS] = {50, 20, 10, 5, 2, 1},
            /* Array que contém as moedas disponíveis */
          moedas[N_MOEDAS] = {50, 25, 10, 5},
            /* Total de cédulas de cada tipo que forem    */
            /* entregues durante uma execução do programa */
          totalCedulas[N_CEDULAS] = {0},
            /* Total de moedas de cada tipo que forem     */
            /* entregues durante uma execução do programa */
          totalMoedas[N_MOEDAS] = {0},
          parcial, /* Número parcial de cédulas/moedas */
                   /* a serem entregues ao cliente     */
          pInteira,     /* Parte inteira do troco     */
          pFracionaria, /* Parte fracionária do troco */
          i;

      /*************************************************************************/
      /* Os arrays cedulas[] e moedas[] foram ordenados em ordem decrescente   */
      /* para que o troco reunisse o menor número possível de cédulas e moedas */
      /*************************************************************************/
```

```
   /* Apresenta o programa */
printf( "\n\t>>> Este programa apresenta numeros de cedulas"
        "\n\t>>> ou moedas que devem ser entregues a clientes"
        "\n\t>>> como trocos de compras. Digite 0 como valor"
        "\n\t>>> da compra para encerrar o programa." );

   /* O laço encerra quando o usuário introduzir 0 como valor de uma compra */
while (1) {
      /* Lê o valor da compra */
   printf("\n\n\t>>> Valor da compra (R$)? ");
   valorCompra = LeReal();

      /* Se o usuário digitou 0, encerra o laço */
   if (!ComparaDoubles(valorCompra, 0.0)) {
      break;
   }

      /* Lê o valor pago pelo cliente */
   printf("\t>>> Valor pago (R$)? ");
   pago = LeReal();

   troco = pago - valorCompra; /* Calcula o valor do troco */

      /* Verifica se há troco a ser entregue */
   if (ComparaDoubles(troco, 0.0) <= 0) {
      printf("\a\n\t>>> Nao ha' troco!\n");
   } else if (ComparaDoubles(troco, MAIOR_TROCO) > 0) {
         /* O valor pago excede limite máximo de troco */
      printf("\n\t>>> O valor pago deveria ser menor\n");
   } else { /* Há troco a ser entregue */
      /* Agora, deve-se calcular as partes inteira e fracionária do troco. */
      /* Para evitar que uma variável inteira receba um valor incorreto    */
      /* por causa de truncamento de um valor real, deve-se arredondar o   */
      /* valor real. Para obter o efeito desejado sem introduzir um novo   */
      /* erro, acrescenta-se um pequeno valor (DELTA) ao valor real antes  */
      /* que ele seja convertido.                                          */

      trocoCorrigido = troco + DELTA;

         /* Calcula as partes inteira e fracionária do troco */
      pInteira = trocoCorrigido;
      pFracionaria = (trocoCorrigido - pInteira)*100;

         /* Apresenta o valor do troco */
      printf("\n\t>>> Troco: R$%0.2f <<<\n", troco);

         /* Determina e apresenta na tela o número  */
         /* de cédulas a serem entregues ao cliente */
      for (i = 0; i < N_CEDULAS; ++i) {
            /* Calcula o número de cédulas com o */
            /* valor determinado por cedulas[i]  */
         parcial = pInteira/cedulas[i];

            /* Se houver cédulas com o valor dado por cedulas[i], apresenta- */
            /* as na tela e reduz a parte inteira do valor já fornecido      */
         if (parcial) {
            printf( "\n\t>>> %d cedula%s de R$%d",
                    parcial, parcial > 1 ? "s" : "", cedulas[i] );

               /* Atualiza o número total de cédulas entregues do */
               /* tipo armazenado no índice i do array cedulas[]  */
            totalCedulas[i] =  totalCedulas[i] + parcial;
```

```
                  /* A parte inteira do troco precisa  */
                  /* ser reduzida do valor apresentado */
               pInteira = pInteira - parcial*cedulas[i];
            }
         }
            /* Determina e apresenta na tela o número */
            /* de moedas a serem entregues ao cliente */
         for (i = 0; i < N_MOEDAS; ++i) {
            parcial = pFracionaria/moedas[i];

               /* Se houver moedas com o valor dado por moedas[i], apresenta-  */
               /* as na tela e reduz a parte fracionária do valor já fornecido */
            if (parcial) {
               printf("\n\t>>> %d moeda%s de %d centavos",
                      parcial, parcial > 1 ? "s" : "", moedas[i]);

                  /* Atualiza o número total de moedas entregues do */
                  /* tipo armazenado no índice i do array moedas[]  */
               totalMoedas[i] =  totalMoedas[i] + parcial;

                  /* A parte fracionária do troco precisa */
                  /* ser reduzida do valor apresentado    */
               pFracionaria = pFracionaria - parcial*moedas[i];
            }
         }
            /* Se sobraram alguns centavos cujos valores são menores do  */
            /* que o menor valor de moeda e o cliente exigir, sugere que */
            /* o atendente entregue mais uma moeda com o menor valor     */
         if (pFracionaria) {
            printf("\n\t>>> Se o cliente exigir, de-lhe"
                   "\n\t    mais uma moeda de %d centavos", moedas[--i]);
         }
      }
   }
         /*** Apresenta um resumo das cédulas e moedas ***/
         /*** entregues durante a execução do programa ***/

   printf( "\n>>> Total de cedulas entregues <<<\n" );

   for (i = 0; i < N_CEDULAS; ++i) {
         /* Se o número total de cédulas do tipo armazenado no índice i */
         /* do array cedulas[] for diferente de 0, apresenta esse total */
      if (totalCedulas[i]) {
         printf( "\n\t>>> %d cedula%s de R$%d", totalCedulas[i],
                 totalCedulas[i] > 1 ? "s" : "", cedulas[i] );
      }
   }

   printf( "\n\n>>> Total de moedas entregues <<<\n" );

   for (i = 0; i < N_MOEDAS; ++i) {
         /* Se o número total de moedas do tipo armazenado no índice i do array */
         /* moedas[] for diferente de 0, apresenta esse total                   */
      if (totalMoedas[i]) {
         printf( "\n\t>>> %d moeda%s de R$%d", totalMoedas[i],
                 totalMoedas[i] > 1 ? "s" : "", moedas[i] );
      }
   }
   return 0;
}
```

Análise:

- A função `ComparaDoubles()`, apresentada na Seção 5.11.6, é utilizada pelo programa para comparar números reais. Essa função não aparece na listagem do programa apresentada.
- Note o uso da constante `DELTA`, que é adicionada ao valor do troco para evitar erros de truncamento na obtenção das suas partes inteira e fracionária, conforme foi discutido na Seção 7.5. Essa mesma constante também é usada pela função `ComparaDoubles()`.
- Os arrays `cedulas[]` e `moedas[]` foram ordenados em ordem decrescente para que o troco reunisse o menor número possível de cédulas e moedas. Ou seja, inicialmente, o número de cédulas é calculado dividindo-se a parte inteira do troco pelo valor da maior cédula; então, o restante da parte inteira do troco, se houver, é dividido pelo segundo maior valor de cédula para determinar a quantidade dessa cédula no troco e assim por diante. O número de moedas que, eventualmente, fará parte do troco é calculado de modo semelhante.

Exemplo de execução do programa:

```
        >>> Este programa apresenta numeros de cedulas
        >>> ou moedas que devem ser entregues a clientes
        >>> como trocos de compras. Digite 0 como valor
        >>> da compra para encerrar o programa.

        >>> Valor da compra (R$)? 9.90
        >>> Valor pago (R$)? 10

        >>> Troco: R$0.10 <<<

        >>> 1 moeda de 10 centavos

        >>> Valor da compra (R$)? 25.6
        >>> Valor pago (R$)? 30

        >>> Troco: R$4.40 <<<
        >>> 2 cedulas de R$2
        >>> 1 moeda de 25 centavos
        >>> 1 moeda de 10 centavos
        >>> 1 moeda de 5 centavos

        >>> Valor da compra (R$)? 0

>>> Total de cedulas entregues <<<

        >>> 2 cedulas de R$2

>>> Total de moedas entregues <<<

        >>> 1 moeda de R$25
        >>> 2 moedas de R$10
        >>> 1 moeda de R$5
```

8.12 Exercícios de Revisão

Introdução (Seção 8.1)

1. (a) O que é uma variável estruturada? (b) O que é uma variável estruturada homogênea? (c) O que é uma variável estruturada heterogênea?
2. O que é um array?

Definições de Arrays (Seção 8.2)

3. Qual é a sintaxe usada na definição de arrays?

4. Existe alguma restrição em relação ao tipo de cada elemento de um array?
5. Quais são as vantagens obtidas ao se declarar o tamanho de um array usando-se uma constante simbólica em vez de usando-se o valor da constante em si?

Acesso a Elementos de um Array (Seção 8.3)

6. (a) O que é índice de um array? (b) Quais são os valores válidos de um índice de array? (c) O que ocorre quando o programador usa um índice inválido para acesso a um elemento de array?
7. O que há de errado com o seguinte trecho de programa?

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

for (j = 1; j <= 5; ++j) {
   printf("%d\n", ar[j]);
}
```

8. (a) Qual é o problema com o seguinte trecho de programa? (b) Qual é a gravidade desse problema?

```
int i, ar[5];

for (i = 1; i <= 5; ++i) {
   ar[i] = i;
}
```

9. O que escreve na tela o seguinte programa?

```
#include <stdio.h>

#define MAX_ELEMENTOS 10

int main(void)
{
   int i, ar[MAX_ELEMENTOS] = {1, 1};

   for (i = 2; i < MAX_ELEMENTOS; ++i) {
      ar[i] = ar[i - 1] + ar[i - 2];
   }

   for (i = 0; i < MAX_ELEMENTOS; ++i) {
      printf("\nar[%d] = %2d", i, ar[i]);
   }

   return 0;
}
```

10. No seguinte fragmento de programa, o conteúdo do array `ar1[]` é copiado para o array `ar2[]`. (a) Explique por que ele não é portável. (b) Sugira uma forma de tornar esse fragmento de programa portável. [**Sugestão**: Se não conseguir resolver esse exercício, consulte a **Seção 10.4**.]

```
int i = 0, ar1[5], ar2[5];

...

while (i < 5) {
   ar2[i] = ar1[i++];
}
```

11. O que faz o seguinte programa?

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

#define TAMANHO 5

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

   printf("\nDigite %d numeros inteiros: \n", TAMANHO);

   for (i = 0; i < TAMANHO; i++) {
      printf("\t> ");
      ar[i] = LeInteiro();
   }

   putchar('\n');

   for (i = TAMANHO - 1; i >= 0; i--) {
      printf("%d\t", ar[i]);
   }

   putchar('\n');

   return 0;
}
```

Iniciações de Arrays (Seção 8.4)

12. (a) Como deve ser escrita a iniciação de um array unidimensional? (b) É obrigatória a iniciação explícita de todos os elementos de um array?

13. (a) Qual é a maneira mais simples de iniciar com zero todos os elementos de um array de duração automática? (b) Isso é necessário quando o array é de duração fixa?

14. Qual é a diferença em termos de iniciação entre arrays de duração fixa e arrays de duração automática?

15. É ilegal incluir numa iniciação um número de valores maior do que o tamanho do array? Explique.

16. Quando todos os elementos de um array são iniciados, é necessário incluir o número de elementos do array em sua definição?

Operador sizeof (Seção 8.5)

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

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

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

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

21. Considere um array `ar[]`. (a) Como se determina o número de bytes ocupados pelo array `ar[]`? (b) Como se determina o número de bytes ocupados por um elemento do array `ar[]`? (c) Como o número de elementos do array `ar[]` pode ser determinado sem que se tenha que recorrer à sua definição?

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

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

24. Na Seção 8.5, foi explicada a razão pela qual o programa abaixo não consegue apresentar os elementos do array `ar[]`:

```
#include <stdio.h>

int main(void)
{
   int    i, ar[] = {1, 2, 3, 4, 5, 6, 7},
   size_t nElementos;

   nElementos = sizeof(ar)/sizeof(ar[0]);

   for(i = -1; i <= nElementos - 2; i++) {
      printf("%d\t", ar[i + 1]);
   }

   return 0;
}
```

O uso de conversão explícita como na instrução:

```
nElementos = (int) sizeof(ar)/sizeof(ar[0]);
```

resolve o problema apresentado por esse programa? Explique.

25. Por que a aplicação do operador **sizeof** num parâmetro formal que representa um array não resulta no tamanho em bytes do array?

26. O que exibe na tela o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int tamanho, x = 0;

   tamanho = sizeof(x = 25);

   printf("\nTamanho = %d \t x = %d\n", tamanho, x);

   return 0;
}
```

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

28. Qual é a saída do seguinte programa?

```
#include <stdio.h>

int main(void)
{
   int i = 10;

   printf("i = %d\n", i);
   printf("sizeof(i++) = %d\n", sizeof(i++));
   printf("i = %d\n", i);

   return 0;
}
```

29. O que o programa abaixo apresenta na tela?

```
#include <stdio.h>

int Tamanho(double ar[])
{
   return sizeof(ar);
}

int main( void )
{
   double array[20] = {0.0};

   printf("Tamanho = %d bytes", Tamanho(array));

   return 0;
}
```

Aritmética de Ponteiros (Seção 8.6)

30. Quais são as operações aritméticas permitidas sobre ponteiros?

31. Quando um inteiro é adicionado ou subtraído a um ponteiro, como a operação é interpretada?

32. O que é fator de escala em aritmética de ponteiros?

33. Suponha que um ponteiro `p` possui num determinado instante o valor `1240`. Se o valor de `p + 1` for `1241`, o que se pode concluir em relação ao tipo de `p`?

34. Suponha que um ponteiro `p` possui num determinado instante o valor `1240`. Se o valor de `p + 1` for `1244`, é possível inferir qual é o tipo de `p`?

35. Que operação aritmética sobre ponteiros não é influenciada por fator de escala, independente do tipo de ponteiro envolvido?

36. Suponha que `p1` e `p2` são ponteiros. (a) Quando a operação `p1 - p2` ou `p1 - p2` é ilegal? (b) Quando ela é legal, mas não faz sentido?

37. O que há de errado com o seguinte programa?

```
#include<stdio.h>

int main()
{
   int  x = 5, y = 10, diferenca;
   int *p1 = &x, *p2 = &y;

   diferenca = p1 - p2;

   printf("\nDiferenca = %d\n" , diferenca);

   return 0;
}
```

38. Considerando as seguintes definições de variáveis:

```
double *p1, *p2;
int     j;
char   *p3;
```

quais das seguintes expressões são válidas?

(a) `p2 = p1 + 4;`
(b) `j = p2 - p1;`
(c) `j = p1 - p2;`
(d) `p1 = p2 - 2;`
(e) `p3 = p1 - 1;`
(f) `j = p1 - p3;`

39. Suponha que `p` seja um ponteiro para **int**. (a) Interprete cada uma das expressões a seguir consideradas legais. (b) Quais delas são ilegais?

(a) `*p`
(b) `++p`
(c) `++*p`
(d) `*++p`
(e) `*p++`
(f) `p++*`
(g) `p*++`

Relações entre Ponteiros e Arrays (Seção 8.7)

40. Seja `ar[]` um array unidimensional. (a) Descreva duas formas diferentes de especificar o endereço do elemento de índice `i` desse array. (b) Descreva duas formas diferentes de acessar o valor do elemento de índice `i` desse array.

41. Um programa em C contém a seguinte definição de array:

```
int  a[8] = {10, 20, 30, 40, 50, 60, 70, 80}
```

(a) O que representa `a`?
(b) O que representa `a + 2`?
(c) Qual é o valor de `*a`?
(d) Qual é o valor de `*a + 2`?
(e) Qual é o valor de `*(a + 2)`?

42. Dadas as seguintes iniciações:

```
int  ar[] = {10, 15, 4, 25, 3, -4};
int *p = &ar[2];
```

quais são os resultados das avaliações das seguintes expressões:

(a) `*(p + 1)`
(b) `p[-1]`
(c) `ar - p`
(d) `ar[*p++]`
(e) `*(ar + ar[2])`

43. Por que o seguinte programa não consegue ser compilado?

```
#include <stdio.h>

int main(void)
{
   int  ar[10] = {-2, 0, -2, 3, 8, -1, 11, 4};
   int  i;

   printf("\nIndice Valor");

   for (i = 0; i < 10; ++i) {
      printf("\n  %d\t%d", i, *ar);
      ++ar;
   }

   return 0;
}
```

44. Considere a seguinte iniciação do array `ar[]`:

```
int ar[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
```

(a) Escreva um trecho de programa contendo um laço **for** responsável pela apresentação na tela dos valores do array `ar[]` utilizando índices.

(b) Repita a tarefa do item anterior utilizando aritmética de ponteiros, em vez de índices.

45. O que justifica o fato de indexação de arrays em C começar em zero, e não em um, como seria mais natural?

Uso de const (Seção 8.8)

46. Para que serve a palavra-chave **const**?

47. Uma variável qualificada com **const** pode ser alterada?

48. Em que situações práticas a palavra-chave **const** deve ser utilizada?

49. Qual é a diferença entre usar **const** e **#define** para definir uma constante?

50. Qual é a diferença entre as seguintes definições de variáveis?

```
const char *p1;
```

e

```
char* const p2;
```

51. Interprete cada uma das seguintes definições de ponteiros. [**Sugestão**: Use a leitura árabe descrita na **Seção 8.8**.]

(a) `const int *p;`
(b) `int const *p;`
(c) `int *const p;`
(d) `const int *const p;`
(e) `int const *const p;`

52. Considerando os protótipos de função a seguir, é possível dizer se `x` é um parâmetro de entrada, de saída ou de entrada e saída?

(a) `void F(int x)`
(b) `void F(int *x)`
(c) `void F(const int *x)`

53. Por que não faz muito sentido usar **const** com um parâmetro que não é ponteiro?

Uso de Arrays com Funções (Seção 8.9)

54. (a) Como um parâmetro formal que representa um array deve ser declarado? (b) Como deve ser passado um parâmetro real correspondente a um parâmetro formal que representa um array?

55. Como o nome de um array passado como parâmetro real para uma função é interpretado?

56. (a) Se um array é passado para uma função e um de seus elementos é alterado, essa alteração é reconhecida na porção do programa que chamou a função? (b) Se esse for o caso, como se poderia garantir que os elementos de um array não são modificados por uma função?

57. O tipo de retorno de uma função pode ser um array?

58. (a) Por que um array de duração automática cujo endereço é retornado por uma função é denominado *zumbi*? (b) Por que erros causados por zumbis são difíceis de detectar? (c) Por que arrays de duração fixa nunca são zumbis?

59. (a) Quais das seguintes funções retornam zumbis? (b) Qual delas retorna um valor incompatível com tipo de retorno da função?

(i)
```
int *F1(int n)
{
   ...
   return &n;
}
```

(ii)
```
int *F2(int *p)
{
   ...
   return &p;
}
```

(iii)
```
int *F3(int *p)
{
   ...
   return p;
}
```

(iv)
```
int F4(int *p)
{
   ...
   return *p;
}
```

60. (a) O que há de errado com o seguinte fragmento de programa? (b) Por que as chances de este programa ser abortado são maiores se o parâmetro `p` da função `F()` representar um array do que se ele representar um ponteiro para uma única variável?

```
int F(double *p, int n)
{
   ...
   return 0;
}

int main(void)
{
   int     x;
   double *ptr;

   x = F(ptr, 10);
   ...
}
```

61. Na **Seção 8.9.4**, recomenda-se que não se deve retornar o endereço de um parâmetro. Considerando essa recomendação, se uma função recebe um endereço como parâmetro, é seguro retorná-lo, como faz a seguinte função?

```
int *F(int ar[])
{
   ...
   return ar;
}
```

62. O parâmetro `ar` no protótipo de função abaixo necessariamente representa um array?

```
int F(double ar[], int n)
```

63. O parâmetro `p` no protótipo de função a seguir pode representar um array?

```
int F(double *p, int n)
```

Arrays Multidimensionais (Seção 8.10)

64. (a) O que é um array bidimensional? (b) O que é um array tridimensional?

65. Por que o uso de arrays com mais de quatro dimensões deve ser evitado?

66. Por que arrays unidimensionais são qualificados como *unidimensionais*?

67. Considerando arrays bidimensionais, (a) o que é linha? (b) O que é coluna?

8.13 Exercícios de Programação

8.13.1 Fácil

EP8.1 (a) Escreva uma função que exibe um array de inteiros na tela, de tal modo que elementos repetidos sejam escritos uma única vez. (b) Escreva um programa que testa a função requerida no item (a). [**Sugestão**: Use a função `EmArray()` apresentada na **Seção 8.11.4**.]

EP8.2 Suponha que se deseje processar um conjunto de valores representando altura e sexo (`'M'` ou `'F'`) de um grupo de 10 pessoas. Escreva um programa em C que:

(a) Leia esse conjunto de dados e armazene-o em dois arrays vinculados, um dos quais contém as alturas e o outro contém os sexos dos indivíduos. [**Sugestão**: Use a função `LeOpcao()` para leitura do sexo e `LeReal()` para ler a altura de cada indivíduo. Ambas funções fazem parte da biblioteca LeituraFacil.]

(b) Determine a maior e a menor altura dentre esses indivíduos, indicando o sexo do indivíduo de maior altura e o sexo do indivíduo de menor altura. [**Sugestão**: Siga o exemplo apresentado na **Seção 8.11.1**.]

(c) Encontre a média de altura entre os indivíduos do sexo feminino (representados no programa pelo caractere `'F'`) e a média de altura entre os indivíduos do sexo masculino (representados no programa pelo caractere `'M'`). [**Sugestão**: Siga o exemplo apresentado na **Seção 8.11.1**.]

EP8.3 (a) Escreva uma função que retorna `1` se um array de elementos do tipo **int** estiver ordenado em ordem crescente ou zero, em caso contrário. [**Sugestões**: (1) Use um laço de repetição no corpo do qual elementos adjacentes do array são comparados dois a dois, como no exemplo apresentado na **Seção 8.11.5**. (2) Se forem encontrados dois elementos fora de ordem, a função retorna `0`. (3) Na instrução seguinte ao laço citado, a função deve retornar `1`, já que não foram encontrados elementos fora de ordem.] (b) Escreva uma função **main()** que lê via teclado valores do tipo **int** até um limite máximo estipulado por uma constante simbólica, armazena esses valores num array na ordem em que eles são introduzidos e usa a função especificada no item (a) para testar se o array está ordenado em ordem crescente. [**Sugestão**: Use a função `LeArray()` definida na **Seção 8.11.6**.]

EP8.4 (a) Escreva uma função que retorna `1` se um array de elementos do tipo **int** estiver ordenado em ordem decrescente ou zero, em caso contrário. (b) Escreva uma função **main()** que lê via teclado valores do tipo **int** até um limite máximo estipulado por uma constante simbólica, armazena esses valores num array na ordem em que eles são introduzidos e usa a função especificada no item (a) para testar se o array está ordenado em ordem decrescente. [**Sugestão**: Siga as sugestões apresentadas no exercício **EP8.3**.]

EP8.5 (a) Escreva uma função que retorna `-1` se um array de elementos do tipo **int** estiver ordenado em ordem decrescente; `1`, se o array estiver ordenado em ordem crescente; ou `0`, se o array estiver desordenado. (b) Escreva uma função **main()** que lê via teclado valores do tipo **int** até um limite máximo estipulado por uma constante simbólica, armazena esses valores num array na ordem em que eles são

introduzidos e usa a função especificada no item (a) para informar se o array está ordenado em ordem crescente ou decrescente. [**Sugestão**: Use as funções definidas nos exercício **EP8.3** e **EP8.4**.]

EP8.6 (a) Escreva uma função que simula um determinado número de lançamentos de um dado e exibe na tela o percentual de ocorrências de cada face do dado. Essa função deve usar um array local para armazenar os números de ocorrências das faces do dado. [**Sugestão**: Como arrays são indexados a partir de zero, para tornar a função mais eficiente, defina o array com um elemento a mais do que o número de faces de um dado. Então, utilize os elementos indexados de `1` a `6` para contar as respectivas ocorrências das faces do dado.] (b) Escreva um programa que, repetidamente, lê um valor inteiro introduzido pelo usuário e, ele for positivo, a função solicitada no item (a) é chamada para simular o lançamento do dado o número de vezes especificado. Se o usuário introduzir um valor negativo ou zero, o programa deverá ser encerrado. [**Sugestão**: Esse programa deve semelhante àquele solicitado no exercício **EP5.20**.]

EP8.7 O programa sobre Mega-sena apresentado na **Seção 8.11.8** não verifica se o usuário ganhou algum prêmio secundário; i.e., quina ou quadra. Altere esse programa, de modo que ele seja capaz de fazer esse tipo de verificação. [**Sugestões**: (1) Substitua a variável `perdeu` por uma variável que conta quantos números o usuário acertou e inicie essa variável com zero. (2) No laço que verifica se o usuário foi sorteado, incremente essa variável sempre que a função `EmArray()` retornar um valor maior do que ou igual a zero. (3) Após o final do laço, verifique se o valor da referida variável é `4`, `5` ou `6` e, se for o caso, informe o usuário qual foi seu prêmio.]

8.13.2 Moderado

EP8.8 **Preâmbulo**: Uma **distribuição de frequência** é uma tabela que mostra os números de ocorrências de dados que se encaixam em determinados intervalos (ou **classes**). As ocorrências em cada classe são denominadas **frequências de classes**. Ou seja, a frequência de um valor é o número de vezes que ele ocorre no conjunto de dados e uma classe armazena a soma das frequências dos valores que a classe representa. Uma **distribuição relativa de frequência** apresenta a percentagem de observações de cada classe com respeito ao número total de observações. Um **histograma** é um gráfico de barras no qual a altura de cada barra é proporcional ao número de ocorrências na classe que a barra representa. Histogramas são usados para representar distribuições de frequência, bem como distribuições relativas de frequência. **Problema**: Escreva um programa que sorteia valores entre dois limites especificados e classifica-os em determinados intervalos. Então, o programa deve exibir na tela um histograma representando a distribuição de frequência dos valores sorteados. O exemplo de execução a seguir ilustra como deve ser esse programa:

```
        >>> Este programa sorteia 800 valores
        >>> entre 1 e 50 e classifica-os
        >>> nos intervalos: 1-10, 11-20, 21-30
        >>> 31-40, 41-50 e desenha um histograma
        >>> usando o fator de escala 1:5.

               **********************************
        00-10  **********************************  174
               **********************************
...
               ******************************
        40-50  ******************************  153
               ******************************
```

Sugestões:

[1] Utilize as seguintes definições de constantes simbólicas:

```
#define MENOR_VALOR     1 /* Menor valor sorteado */
#define MAIOR_VALOR    50 /* Maior valor sorteado */
#define SORTEIOS      800 /* Número de sorteios   */
#define INTERVALOS      5 /* Número de intervalos */
#define ESCALA          5 /* Fator de escala      */
```

[2] Defina o array que armazenará a distribuição de frequência como:

```
int frequencias[INTERVALOS] = {0}
```

[3] Defina o array que armazenará os intervalos da distribuição como:

```
intervalos[] = {0, 10, 20, 30, 40, 50}
```

[4] Inicie o gerador de números aleatórios (v. **Seção 4.10**).

[5] Utilize um laço de contagem que sorteia os valores e constrói a distribuição de frequência. Isto é, após sortear um número entre os valores especificados, deve-se verificar em que intervalo o valor sorteado se encaixa e incrementar o respectivo elemento do array que representa a distribuição.

[6] Use um laço de contagem para desenhar cada barra do histograma (v. exemplo de execução acima). A altura (i.e., o número de asteriscos) de cada barra corresponde ao número de ocorrências que a barra representa dividido pelo fator de escala.

EP8.9 **Preâmbulo**: **Rotação à direita de ordem k** de um array com `n` elementos (`k < n`) consiste em mover cada elemento do array `k` posições adiante, de tal modo que o último elemento passe a ocupar a posição `k - 1`, o penúltimo elemento passe a ocupar a posição `k - 2` e assim por diante. Se `k = n`, a rotação à direita de ordem `k` é equivalente à rotação à direita de ordem `k%n`. Essa equivalência também vale quando `k < n`, pois, nesse caso, `k%n` é igual a `k`. **Rotação à esquerda de ordem k** de um array é definida de modo análogo, mas, agora, os elementos são deslocados para trás. **Problema**: (a) Escreva uma função que provoca a rotação dos elementos de um array um número determinado de vezes para a direita (rotação positiva) ou para a esquerda (rotação negativa). (b) Escreva um programa que define um array e, repetidamente, apresenta-o antes e depois de sofrer as rotações especificadas pelo usuário. Exemplo de execução do programa:

```
>>> Este programa provoca a rotacao de
 >>> um array o numero de vezes que voce
 >>> especificar. Digite zero para
 >>> encerrar o programa.

 >>> Estado atual do array:

 { 1, 2, 3, 4, 5 }

 >>> Numero de rotacoes (0 encerra o programa): 3

 >>> Estado atual do array:

 { 3, 4, 5, 1, 2 }

 >>> Numero de rotacoes (0 encerra o programa): -8

 >>> Estado atual do array:

 { 1, 2, 3, 4, 5 }

 >>> Numero de rotacoes (0 encerra o programa): 23

 >>> Estado atual do array:

 { 3, 4, 5, 1, 2 }
```

CONTINUA

```
>>> Numero de rotacoes (0 encerra o programa): 0
>>> Obrigado por usar este programa.
```

[**Sugestões**: (1) Existem diversos algoritmos para rotação de arrays e um dos mais fáceis de entender e implementar usa um array auxiliar para armazenar os elementos do array que está passando por uma rotação. Portanto defina um array auxiliar local à função que implementa rotações para copiar os elementos do array original em suas novas posições. Antes de retornar, essa função deve copiar o conteúdo do array auxiliar para o array original. (2) Se o valor de **k** for negativo, representando uma rotação à esquerda, converta-o numa rotação à direita substituindo esse valor por: **n + k%n**.]