CAPÍTULO 7

ARRAYS E PONTEIROS

7.1 Introdução

Uma **variável estruturada** 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 tipos diferentes. Arrays, que serão discutidos neste capítulo, são variáveis estruturadas e homogêneas, enquanto estruturas e uniões, discutidas no **Capítulo 9**, são variáveis estruturadas e heterogêneas.

Este capítulo enfoca arrays e ponteiros, bem como as relações entres eles. Aconselha-se cautela ao leitor nos tópicos relacionados à aritmética de ponteiros, visto que este assunto contém algumas sutilezas e, ao mesmo tempo, é essencial para o domínio da linguagem C. Várias novidades e alterações introduzidas pelo padrão C99, como arrays de tamanhos variáveis e com índices estáticos, iniciadores designados de arrays e ponteiros restritos, também são expostas aqui.

7.2 Arrays

Um **array** é uma coleção de variáveis de mesmo tipo armazenadas contiguamente em memória. Cada variável que compõe um array é denominada **elemento** do array e pode ser acessada usando o nome do array e um **índice** (ou **subscrito**). Em C, o elemento inicial de um array sempre tem índice igual a zero.

O propósito típico de um array é armazenar valores de um mesmo tipo relacionados entre si.

7.2.1 Definições de Arrays

A definição de uma variável de um tipo array, em sua forma mais simples, consiste no seguinte:

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

onde, ***tipo*** é o tipo de cada elemento do array ***e tamanho-do-array*** consiste em uma expressão inteira, constante e positiva que especifica o número de elementos do array[1]. Note que, diferentemente de algumas outras linguagens (por exemplo, Pascal), na declaração de um array não aparece a definição do tipo do índice a ser utilizado para acesso aos elementos do array, pois, em C, este tipo é sempre um inteiro não-negativo (**unsigned int** ou **unsigned long int**, dependendo do compilador)**.**

7.2.2 Acessando Elementos de um Array

Os elementos de um array são **acessados** por meio de índices que indicam a posição do elemento com relação ao elemento inicial do array. Mais precisamente, o elemento inicial possui índice 0, o próximo tem índice 1 e assim por diante. Note que, como a indexação dos elementos começa com 0, o último elemento do array possui índice igual ao número de elementos do array menos 1. Por exemplo:

```
float   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; /* Esta referência é problemática, pois */
                 /* ultrapassa o limite superior do array */
```

[1] O padrão C99 permite que ***tamanho-do-array*** possa conter variáveis, conforme apresentado na **Seção 7.7**.

O fato de C iniciar a indexação de arrays sempre com 0 é devido à maior eficiência do código produzido por esta opção em relação à indexação começando em 1, que seria mais natural. Além disso, a indexação começando em 0 facilita o acesso a elementos de um array por meio de ponteiros, conforme será visto na **Seção 7.4**.

Nem todo compilador de C faz verificação de acesso além dos limites de um array (o padrão ISO não requer isso). Portanto, o programador pode acidentalmente acessar porções de memória que não foram alocadas para o array e isso pode trazer conseqüências 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. Freqüentemente, este tipo de erro é causado porque o programador excede em 1 o teste de final de um array acessado num laço de repetição **for**, como mostra o exemplo a seguir:

```
int main()
{
   int  ar[10], j;

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

   return 0;
}
```

No último exemplo, o array `ar` foi declarado com capacidade para conter 10 elementos e, portanto, ele pode 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 `j`. Isso é provável porque a declaração da variável `j` foi feita logo em seguida à declaração de `ar`. Logo, este erro poderá causar um laço de repetição infinito, uma vez que `j` é reiniciado com zero sempre que se atinge o final previsto para o laço.

O tamanho, em bytes, de um array pode ser determinado utilizando o operador **sizeof**, como, por exemplo:

```
float   notas[4*10 + 5];

sizeof(notas);/* Resultaria no valor equivalente a 45 vezes o
número de */
            /* bytes ocupado por um valor do tipo float */
```

Note, entretanto, que, para obter o tamanho de um dado array, 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). Por exemplo:

```
sizeof(notas[0]); /* Resulta no espaço ocupado  */
                  /* por um valor do tipo float */
```

7.2.3 Iniciações de Arrays

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

▶ Arrays de Duração Fixa

Como padrão, arrays com duração fixa têm todos os seus elementos iniciados com zero, mas pode-se iniciar todos ou alguns elementos 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 declaração do array. Por exemplo:

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

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

É ilegal incluir numa iniciação um número de valores maior do que o permitido pelo tamanho do array, mas não é necessário atribuir valores

a todos os elementos do array. Isto é, se houver um número de valores de iniciação menor do que o número de elementos do array, os elementos remanescentes terão o valor zero atribuído.

Quando todos os elementos de um array são iniciados, o tamanho do array pode ser omitido, pois, neste 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};
```

▶ **Arrays de Duração Automática**

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

7.2.4 Iniciadores Designados de Arrays (C99)

O padrão C99 introduziu o conceito de **iniciadores designados** para arrays. Esta forma de iniciação permite que elementos específicos de um array sejam iniciados, conforme mostrado no exemplo a seguir:

```
int ar[20] = {[4] = -2, [9] = 5, [12] = 1, [17] = -5};
```

No exemplo apresentado, os elementos `ar[4]`, `ar[9]`, `ar[12]` e `ar[17]` são explicitamente iniciados, enquanto os elementos remanescentes são iniciados implicitamente com 0. Note que este tipo de iniciação não era possível antes de ser introduzido pelo padrão C99.

É importante ressaltar ainda que a ordem na qual os elementos são iniciados é irrelevante. Por exemplo, a iniciação designada a seguir:

```
int ar[20] = {[12] = 1, [9] = 5, [17] = -5, [4] = -2};
```

tem o mesmo efeito daquela apresentada no exemplo anterior. Em termos de legibilidade, entretanto, a primeira iniciação é mais recomendada.

Um mesmo elemento pode ser iniciado mais de uma vez. Neste caso, o valor que prevalece é o último valor atribuído. Por exemplo:

```
int ar[20] = {[4] = -2, [9] = 5, [12] = 1, [17] = -5, [4] = 8};
```

No último exemplo, o elemento `ar[4]` é iniciado duas vezes e o valor atribuído a este elemento é `8`, que é o último valor atribuído a ele[2].

A iniciação designada de arrays pode ser utilizada com arrays de tamanhos variáveis (v. **Seção 7.7**):

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

Neste exemplo, se `N ≥ 5`, os elementos indexados entre `3` e `N − 3` serão iniciados com `0`; se `N < 5`, algumas das primeiras iniciações serão sobrescritas.

7.3 Aritmética de Ponteiros

Antes de introduzir uma importante relação entre ponteiros e arrays, é necessário apresentar a noção de aritmética de ponteiros. A linguagem C permite que as operações aritméticas apresentadas na **Tabela 29** sejam executadas com ponteiros.

OPERAÇÃO	EXEMPLO
Soma de um inteiro a um ponteiro	`p + 2`
Subtração de um inteiro a 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 29: Operações aritméticas sobre ponteiros

[2] Apesar de legalmente correto, não parece haver nenhuma razão de natureza prática que justifique a iniciação de um elemento mais de uma vez.

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

```
int  *p;
```

a expressão:

```
p + 3
```

deve ser interpretada como o endereço da posição de memória que está três objetos do tipo **int** adiante do endereço do objeto para o qual `p` aponta. Isto é, como `p` é um endereço, `p + 3` também será um endereço. Mas, em vez de simplesmente adicionar 3 ao valor do endereço armazenado em `p`, o compilador adiciona 3 multiplicado pelo tamanho (i.e., número de bytes) do objeto para o qual `p` aponta. Neste contexto, o tamanho do objeto apontado pelo ponteiro é denominado **fator de escala**. Com exceção da última operação apresentada na **Tabela 27**, todas as demais operações sobre ponteiros envolvem a aplicação de um fator de escala.

Suponha, por exemplo, que o endereço correntemente contido no ponteiro `p` definido acima seja `e` e que o tipo **int** seja armazenado 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 24** ilustra esse argumento.

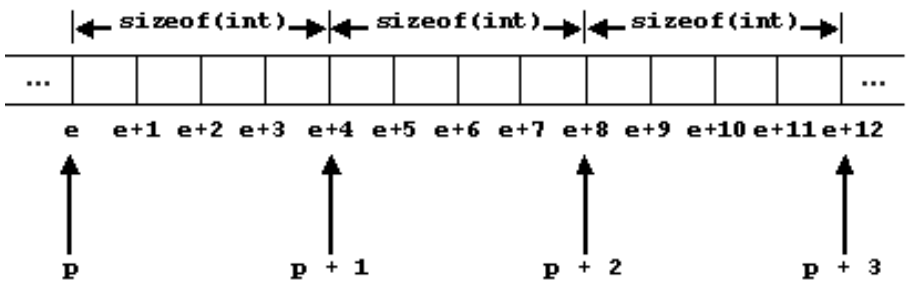

Figura 24: Soma de um inteiro a um ponteiro

Se, no exemplo anterior, o ponteiro `p` tivesse sido declarado como `char *p`, então `p + 3` significaria `e + 3`. Concluindo, `p + 3` sempre significa o endereço do terceiro objeto do tipo apontado pelo ponteiro após aquele correntemente apontado por `p`.

Subtrair um inteiro de um ponteiro tem uma interpretação semelhante. Por exemplo, `p - 3` representa o endereço do terceiro objeto do tipo apontado pelo ponteiro `p` que precede o objeto correntemente apontado por ele, como mostra a **Figura 25**.

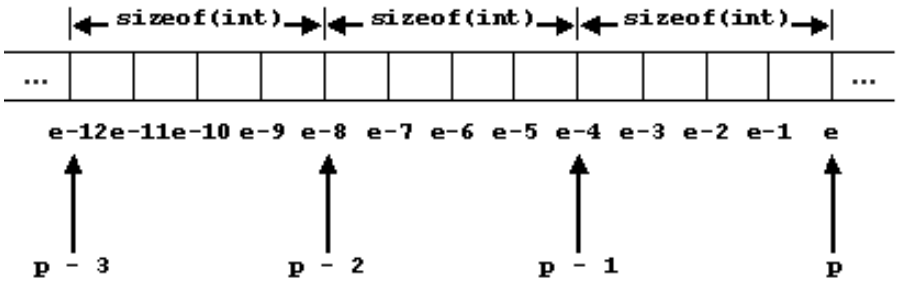

Figura 25: Subtração de um inteiro a um ponteiro

Operações de incremento e decremento de ponteiros também são bastante comuns em C. Nesses casos, os operadores de incremento e decremento são utilizados para fazer um ponteiro apontar para o objeto posterior e anterior, respectivamente, à sua posição atual. Em qualquer caso, o fator de escala é aplicado à operação (i.e., a multiplicação pelo tamanho do objeto).

A subtração de dois ponteiros é legal, desde que os ponteiros apontem para valores do mesmo tipo. Esta operação resulta num número inteiro cujo valor absoluto representa o número de objetos do tipo para o qual os ponteiros apontam entre os dois ponteiros mais um. Por exemplo, supondo que `ar` seja um array previamente definido:

```
&ar[3] - &ar[0]
```

resulta em 3, significando que existem 2 objetos do tipo dos elementos do array `ar` entre `ar[3]` e `ar[0]`.

A seguir, serão apresentados alguns exemplos de expressões aritméticas legais e ilegais envolvendo ponteiros

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

p2 = p1 + 4;  /* Legal */
j = p2 - p1;  /* Legal - resultado: j recebe 4  */
j = p1 - p2;  /* Legal - resultado: j recebe -4 */
p1 = p2 - 2;  /* Legal - os ponteiros são compatíveis */
p3 = p1 - 1;  /* Legal, mas os ponteiros não são compatíveis */
j = p1 - p3;  /* ILEGAL - os ponteiros não são compatíveis */
```

Note que, os dois últimos exemplos envolvem operações com ponteiros incompatíveis (v. **Seção 3.2.5**). Contudo, apenas no último caso o compilador indica erro de compilação, o que está de acordo com o que foi exposto nesta seção. No penúltimo caso, o compilador apresenta apenas uma mensagem de advertência, pois se está atribuindo o valor de um ponteiro do tipo **long *** a um ponteiro do tipo **char ***; isso está de acordo com o que foi apresentado na **Seção 3.2.5**.

7.4 Relação entre Ponteiros e Arrays Unidimensionais

Conforme foi visto anteriormente, os elementos de um array são acessados por meio de índices. Aqui, mostrar-se-á que ponteiros provêm outra forma de acesso a elementos de um array. Suponha, por exemplo, a existência das seguintes definições:

```
long   ar[4];
long   *p;
```

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

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

Portanto, a indireção do ponteiro `p`: `*p`
resulta no valor de `ar[0]`. 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, de modo que, em geral, se `p` aponta para o início do array `ar`, então a seguinte relação é válida:

`*(p + i)` é o mesmo que `ar[i]`

para qualquer `i` inteiro. Em outras palavras, esta relação pode ser traduzida como ***adicionar um inteiro a um ponteiro que aponta para o início de um array e então aplicar o operador de indireção nesta expressão é o mesmo que utilizar o inteiro como índice do array***.

Outra relação importante decorre do fato de o nome de um array considerado isoladamente ser interpretado como um ponteiro para o início do array (i.e., o endereço do elemento de índice 0). Ou seja,

`ar` é o mesmo que `&ar[0]`

Combinando as duas relações anteriores, chega-se à seguinte eqüivalência:

`*(ar + i)` é o mesmo que `ar[i]`

Esta última relação é uma das características mais importantes da linguagem C. Em conseqüência desta relação, quando um compilador de C encontra uma referência com índice a um elemento de um array (por exemplo, `ar[2]`), ele adiciona o índice ao endereço do array (por exemplo, `ar + 2`) e aplica o operador de indireção a este endereço [por exemplo, `*(ar + 2)`], obtendo, assim, o valor do elemento do array.

Devido às relações apresentadas acima, ponteiros e arrays podem ser utilizados de modo equivalente para fazer acessar elementos de arrays. É importante lembrar, entretanto, que valores de variáveis do tipo ponteiro podem ser modificados, enquanto o valor atribuído a um nome de array não o pode. Isto ocorre porque o nome de um array não é realmente um nome de variável: ele representa o endereço do array (variável), e o endereço de uma variável não pode ser modificado (v. **Seção 3.2**). Em termos práticos,

isto significa, por exemplo, que o nome de um array (sem índice) não pode aparecer no lado esquerdo de uma instrução de atribuição ou sofrer a ação de um operador de incremento ou decremento.

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

```
float  ar[5], *p;

p = ar; /* Legal: é o mesmo que p = &ar[0] */
ar = p;  /* ILEGAL: não se pode alterar o endereço de um array  */
&p = ar; /* ILEGAL: não se pode alterar o endereço de um ponteiro
*/
ar++;      /* ILEGAL: não é permitido alterar o endereço de um array
*/
p++;  /* Legal: ponteiros podem ser incrementados */
ar[1] = *(p + 3); /* Legal: ar[1] e *(p + 3) são do tipo float
*/
```

As relações e diferenças entre ponteiros e arrays são muito importantes para o domínio da linguagem C. Portanto, convença-se de que realmente entendeu todos os conceitos e exemplos apresentado nesta seção antes de prosseguir.

7.5 Arrays como Parâmetros de Funções

7.5.1 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 fazer isso:

tipo-do-elemento ***parâmetro***

ou

tipo-do-elemento ***parâmetro***[ ]

Como exemplos de parâmetros formais que representam arrays, considere o seguinte cabeçalho de função:

```
void  MinhaFuncao(float *ar)
```

ou, alternativamente:

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

Quando o segundo formato de declaração é utilizado, o compilador converte-o no primeiro formato. Por exemplo, no segundo cabeçalho acima, `float ar[]` é convertido em `float *ar`. Conseqüentemente, os dois tipos de declarações são funcionalmente eqüivalentes. 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 argumento é um ponteiro para um único **float** ou para um array de elementos do tipo **float**.

Pode-se ainda incluir o tamanho do array no segundo tipo de declaração acima (por exemplo, `float  ar[80]`), mas, neste caso, o compilador utiliza esta informação apenas para verificar se uma dada referência a um elemento do array está dentro dos limites do array (se o compilador suportar esta facilidade).

A escolha entre declarar um argumento de função em forma de array ou como ponteiro não tem nenhum efeito na tradução feita pelo compilador[3]. Para o compilador, o argumento `ar` do exemplo anterior é simplesmente um ponteiro para um **float** e não propriamente um array. Mas, devido à eqüivalência entre ponteiros e arrays, ainda é possível acessar os elementos de `ar` como se fossem um array.

De acordo com o padrão C99, há situações nas quais é permitido apenas o formato que utiliza colchetes na declaração de um parâmetro formal que representa um array (v. **Seções 7.5.3**, **7.7** e **7.8**).

7.5.2 Arrays como Parâmetros Reais

Numa chamada de função que possui um parâmetro representando um array, utiliza-se o nome de um array como parâmetro real. Este nome, conforme já foi visto, será interpretado como o endereço do primeiro

[3] A não ser que se utilizem arrays com índices estáticos, conforme apresentado na **Seção 7.5.3**.

elemento do array. Por exemplo, considerando as definições do array `meuArray` e da função a seguir:

```
float  meuArray[10];

void  MinhaFuncao(float ar[])
{
   ...
}
```

A função `MinhaFuncao()` poderá ser chamada como:

```
MinhaFuncao(meuArray);
```

que é o mesmo que:

```
MinhaFuncao(&meuArray[0]);
```

É importante ressaltar que não se pode determinar o tamanho de um array fora do escopo onde o array é definido. Conseqüentemente, é impossível calcular 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(float ar[])
{
   printf("O tamanho do array e': %d\n", sizeof(ar));
}
```

uma chamada desta função imprimiria o número de bytes necessários para armazenar um ponteiro, e não o número de bytes necessários para armazenar o array, visto que apenas seu endereço é passado para a função quando ela é chamada.

Devido à impossibilidade de uma função determinar o tamanho de um array cujo endereço é recebido como parâmetro, é muitas vezes uma boa idéia incluir o tamanho do array na lista de argumentos da função. Isto permite à função saber onde o array termina, conforme mostra o exemplo a seguir:

```
void  ImprimeArray(float  ar[], unsigned  tamanhoDoArray)
{
   unsigned  i;

   for (i = 0; i < tamanhoDoArray; i++) {
      printf("ar[%u] = %3.2f\n", i, ar[i]);
   }
}
```

Do lado da função que faz a chamada, onde é definido o array passado como argumento, o número de elementos do array pode ser calculado dividindo-se o tamanho total do array em bytes pelo tamanho de um elemento do array em bytes. Por exemplo, considerando a definição de array a seguir:

```
float meuArray[] = {1.2, 3.14, 1.69, 10.1, 0.5, 0, 3.0, 12.42};
```

a função `ImprimeArray()` do último exemplo poderia ser invocada por meio de:

```
ImprimeArray(meuArray, sizeof(meuArray)/sizeof(meuArray[0]));
```

Note que a expressão `sizeof(meuArray)/sizeof(meuArray[0])` resulta no número de elementos do array `meuArray` para qualquer que seja o tipo dos elementos do array `meuArray`.

7.5.3 Arrays com Índices Estáticos e Qualificadores de Tipos (C99)

Conforme visto anteriormente, quando um array é declarado como argumento de uma função, ele é interpretado pelo compilador como um ponteiro para o primeiro elemento do array, e um compilador que segue o padrão ISO pode ignorar qualquer informação a respeito do tamanho do array.

O padrão C99 permite que informação sobre o tamanho de um array seja incluída na declaração do array numa lista de argumentos formais utilizando a seguinte sintaxe:

tipo nome [**static** *N*];

onde:

- ***tipo*** – é o tipo de cada elemento do array
- ***nome*** – é o nome do parâmetro que representa um array
- *N* – é o número mínimo de elementos do array que o parâmetro representa

Por exemplo:

```
void F1(int ar[static 10]); // ar representa um array com,
                      // no mínimo,10 elementos do tipo int

unsigned int n;
...
void F2(int ar[static n]); // ar representa um array com,
                           // no mínimo, n elementos, onde
                           // o valor de n é conhecido em
                           // tempo de execução
```

De acordo com o padrão C99, os qualificadores **const**, **volatile** e **restrict**[4] também podem ser utilizados numa declaração de array como argumento formal de modo similar a **static**. Estes qualificadores alteram a interpretação do ponteiro que representa o array com os mesmos significados vistos antes para estas palavras-chaves, mas não alteram a interpretação do conteúdo do array. Por exemplo:

```
void F3(int ar[const]);
```

é o mesmo que:

```
void F3(int *const ar);
```

Os qualificadores **const**, **volatile** e **restrict** também podem ser utilizados de modo combinado, como, por exemplo:

```
void F4(int ar[restrict static 10]);
```

[4] O significado de **restrict** será apresentado mais adiante, na **Seção 7.8**.

No último exemplo, o parâmetro `ar` é interpretado como um ponteiro restrito (v. **Seção 7.8**) para o primeiro elemento de um array com pelo menos 10 elementos do tipo **int**.

7.5.4 Retorno de Arrays

Uma pergunta que freqüentemente intriga programadores é: ***É permitido que uma função retorne um array em C?*** Rigorosamente, a resposta a esta questão é ***não***, apesar de ser permitido que uma função retorne um ponteiro para um array. Do mesmo modo, estritamente falando, não é possível passar um array para uma função. Ou seja, conforme foi visto nas seções precedentes, é permitido apenas passar um ponteiro para um array.

Para dirimir dúvidas, considere o seguinte programa-exemplo:

```
#include <stdio.h>

typedef float tArray[20];

float F1(tArray ar)
{
   return ar[0];
}

tArray F2(tArray a) /* NÃO COMPILA! */
{
   return a;
}

int main ()
{
   tArray ar = {-5};
   float  f;
   float  *p = ar;

        printf("Tamanho  do  array  em  main():  %u\n",
sizeof(ar));
   f = F1(ar);

   f = F1(p);

   printf("Primeiro elemento do arranjo: %f\n", f);

   return 0;
}
```

No exemplo acima, `tArray` é definido como um tipo que representa arrays de 20 elementos do tipo **float**. Aparentemente, a função `F1()` recebe um array do tipo `tArray`, o que seria considerado ilegal, mas, na realidade, o compilador interpreta uma declaração como aquela do parâmetro `ar` como um ponteiro para um array de elementos do tipo **float**. Ou seja, o compilador interpreta o cabeçalho da função `F1()` como:

```
float F1(float *ar)
```

Por outro lado, o caso da função `F2()` é diferente. Esta função tem `tArray` como tipo de retorno e o compilador considera esta declaração ilegal, pois se está tentando retornar um array, o que não é permitido em C.

Conforme foi afirmado, uma função pode retornar um ponteiro para um array. Todavia, o programador deve tomar cuidado para não retornar um ponteiro para um array de duração automática (i.e., criado dentro da função sem o uso de **static**), pois tal array deixa de existir quando a função retorna[5]. Considere, por exemplo, a seguinte função:

```
float RetornaZumbi(void)
{
   float z[10];
   ...
   return z; /* Retorna um ponteiro para um array zumbi */
}
```

A função apresentada neste último exemplo retorna um ponteiro para um array de duração automática criado no interior da função e, conforme foi visto na **Seção 4.2**, esta variável (array) deixa de ser válida quando a função retorna. Precisamente, a expressão ***deixa de ser válida*** significa o seguinte:

> 1. Como a variável `z` tem duração automática, ela é alocada (i.e., tem espaço em memória reservado para ela e ***apenas*** ela) num ***stack frame*** (v. **Seção 6.13.12**) quando a função `RetornaZumbi()` é chamada.

[5] Aliás, este conselho não se aplica apenas no caso de arrays; ele é mais abrangente e significa ***nunca retorne um ponteiro para uma variável de duração automática***.

2. Quando a função `RetornaZumbi()` retorna, o ***stack frame*** alocado para sua chamada é liberado (novamente, v. **Seção 6.13.12**). Isto que dizer que o espaço que estava alocado exclusivamente para a variável `z` poderá ser alocado para outras variáveis ou parâmetros quando um novo ***stack frame*** for criado (numa chamada subseqüente de função).

Concluindo, , o espaço reservado anteriormente para a variável `z` continua a existir fisicamente, mas, é claro, como este espaço poderá estar alocado para outras variáveis ou parâmetros, ele poderá ser alterado (i.e., ele está ***vivo***) sem ser por meio de uso da variável `z` (que deveria estar ***morta***). Por isso, este gênero de variável recebe a denominação de ***zumbi***. Outra justificativa para esta denominação é que este tipo de erro é muito difícil de ser descoberto (i.e., ele parecer ser ***sobrenatural***).

Finalmente, deve-se salientar que o problema que ocorre com a função `RetornaZumbi()` é decorrente do fato de o array `z` ter duração automática e não devido ao fato de ele ser local à função. Ou seja, o problema aqui é de duração e não de escopo. Se o array `z` tivesse sido declarado com **static**, o problema não ocorreria, pois variáveis de duração fixa não são armazenadas em ***stack frames***.

7.6 Arrays Multidimensionais

Array multidimensional é um array cujos elementos também são arrays. Os arrays vistos aqui são, em contrapartida, denominados **unidimensionais**. Um array **bidimensional** é aquele cujos elementos são arranjos unidimensionais; um array **tridimensional** é aquele cujos elementos são arranjos bidimensionais e assim por diante. Algumas áreas de conhecimento, como a Física e a Engenharia, utilizam arrays multidimensionais com freqüência.

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

tipo-do-elemento nome-do-array [*tamanho1*][*tamanho2*]...[*tamanhoN*]

Embora o padrão ISO determine que um compilador de C deva suportar pelo menos seis dimensões para arrays multidimensionais, raramente mais de três ou quatro dimensões são utilizadas em aplicações práticas. No exemplo a seguir, um array tridimensional de caracteres é definido:

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

A variável `arrayDeCaracteres` do exemplo acima é interpretada como um array de três elementos, sendo cada um dos quais um array de quatro elementos e um array de 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.

7.6.1 Iniciações de Arrays Multidimensionais

Convencionalmente, uma **linha** de um array multidimensional corresponde à sua primeira dimensão, enquanto uma **coluna** corresponde às demais dimensões. Apesar de esta terminologia ser mais intuitiva no caso de arrays bidimensionais (devido à analogia com matrizes em Matemática), ela é também utilizada para arrays com dimensões maiores que dois.

Para iniciar um array multidimensional, deve-se colocar os valores dos elementos de cada linha do array entre chaves. Se, para uma dada linha, houver um número de valores menor do que o número especificado na definição do array, os elementos remanescentes receberão o valor 0. Considere o seguinte exemplo[6]:

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

Nesse 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.

[6] Observe como a endentação torna a iniciação mais legível.

Em forma de tabela, esse array poderia ser visualizado como:

1	2	3
4	0	0
5	6	7
0	0	0
0	0	0

Se as chaves internas não tivessem sido incluídas na iniciação do array, como no exemplo a seguir:

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

o resultado da iniciação seria visualizado como:

1	2	3
4	5	6
7	0	0
0	0	0
0	0	0

A iniciação nesse último exemplo é confusa, pois um programador menos experiente poderia pensar que o resultado desta iniciação seria aquele do penúltimo exemplo. Portanto, para melhorar a legibilidade em iniciações de arrays multidimensionais, é sempre recomendável colocar cada linha entre chaves.

Se a especificação de tamanho da primeira dimensão de um array multidimensional for omitida, o compilador automaticamente deduz o tamanho dela baseado no número de valores de iniciação presentes. Neste caso, os especificadores de tamanho das outras dimensões devem estar presentes. Por exemplo,

```
int  arA[][3][2] = { { {1, 2}, {0, 0}, {1, 1} },
                     { {0, 1}, {1, 0}, {0, 0} } };
```

resulta num array 2x3x2, pois existem 12 valores, e cada elemento do array `arA` é um array 3x2 (12 dividido por 3*2 resulta em 2, que é o tamanho omitido da primeira dimensão).

Por outro lado, a definição a seguir:

```
int  arB[][] = { 1, 2, 3, 4, 5, 6 };   /* ILEGAL */
```

é ilegal porque o compilador não consegue determinar os tamanhos das dimensões do array (i.e., ele não seria capaz de decidir se o array é 2x3 ou 3x2). Entretanto, se o tamanho da segunda dimensão for especificado, a declaração deixa de ser ambígua e torna-se legal.

7.6.2 Acesso a Elementos de Arrays Multidimensionais

O acesso a um elemento de um array multidimensional é obtido utilizando-se um número de índices igual ao número de dimensões do array. Cada índice deve ser envolvido por um par de colchetes. Por exemplo:

```
int  ar[2][3][4];
int  x;

x = ar[1][1][2];
```

Um erro freqüente cometido por programadores iniciantes em C, especialmente entre aqueles acostumados a lidar com arrays multidimensionais em outras linguagens de programação, é utilizar vírgula para separar índices. Por exemplo, um erro comum é utilizar:

```
a[1,2] = 0;    /* ILEGAL */
```

em vez de:

```
a[1][2] = 0;   /* Correto */
```

A primeira instrução em Pascal, por exemplo, é exatamente equivalente à segunda em C. Mas, em C, a primeira expressão tem um significado diferente porque a vírgula, neste caso, é considerada um operador (e não um separador, como em Pascal). Conforme já foi visto, em C, a expressão

1,2 resulta em 2. Portanto, a referência a[1,2] resulta em a[2]. Agora, se a é um array bidimensional de elementos do tipo **int**, a[2] é o endereço do elemento a[2][0] (v. **Seção 7.6.3**) e, portanto, não pode ser modificado.

7.6.3 Relação entre Ponteiros e Arrays Multidimensionais

A relação entre ponteiros e arrays multidimensionais é um pouco mais complexa do que aquela envolvendo ponteiros e arrays unidimensionais. Esta relação pode ser melhor entendida por meio de um exemplo, como o apresentado a seguir.

Suponha que o tipo **int** ocupe 4 bytes e considere a seguinte definição do array ar:

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

Uma referência a um elemento deste array, tal como:

```
ar[2][1]
```

é interpretada pelo compilador como[7]:

```
*(ar[2] + 1)
```

Esta última expressão representa o elemento situado um elemento além daquele armazenado em ar[2]. Mas, o que significa ar[2], se ar representa um array bidimensional? Para responder a esta pergunta, recorde-se que int ar[3][2] pode ser visto como um array unidimensional de tamanho 3, cujos elementos são arrays de elementos do tipo **int** de tamanho 2. Portanto, ar[2] representa o terceiro elemento do array ar, que, de acordo com a iniciação de ar, é o array contendo os valores 5 e 6. Além disso, 1 na expressão *(ar[2] + 1) deve ser interpretado, de acordo com a aritmética de ponteiros, como uma vez o tamanho em bytes de cada

[7] Esta expressão é obtida do seguinte modo: ar[2][1] é o mesmo que (ar[2])[1] que, de acordo com a relação entre arrays e ponteiros vista na Seção 7.4, é equivalente a *(ar[1] + 2).

elemento do array de elementos do tipo **int** (i.e., `1*4`, pois se supõe aqui que um **int** ocupe 4 bytes).

Utilizando novamente a relação entre ponteiros e arrays unidimensionais vista na **Seção 7.4**, a última expressão acima resulta em:

```
*(*(ar + 2) + 1)
```

Mas `ar` é um array cujos elementos também são arrays, de modo que `2` nesta última expressão representa duas vezes o tamanho (em bytes) de cada um destes elementos, que são arrays de dois elementos do tipo **int**. Logo, o `2,` na expressão acima, deve ser multiplicado por duas vezes o tamanho (em bytes) de um **int**, que, hipoteticamente, é 4. Portanto, após a aplicação do fator de escala, o `2,` na última expressão, transforma-se em `2*2*4`. Assim, a última expressão obtida é eqüivalente a:

```
*(int *)( ( (char *)ar + (2*2*4) ) + (1*4) )
```

A razão pela qual o operador de conversão **(char *)** foi colocado antes de `ar` é que isso evita que o compilador aplique o fator de escala, o que já foi feito manualmente na expressão. O uso de **(int *)** assegura que a expressão sendo referenciada (i.e., toda a expressão seguindo o primeiro asterisco) é um ponteiro para o tipo **int**. Esta última conversão é necessária porque o operador **(char *)** transforma a expressão:

```
( (char *)ar + (2*2*4) ) + (1*4)
```

num ponteiro para o tipo **char**. Portanto, sem a conversão `(int *)`, a indireção desta expressão resultaria no conteúdo de apenas um byte de memória, em vez de 4 bytes.

Após resolver as operações na última expressão, obtém-se, finalmente:

```
*(int *)( (char *)ar + 20 )
```

Ao valor `20` na última expressão já foi aplicado o fator de escala, de modo que ele representa o número de bytes a ser atravessados a partir do início do array a fim de se atingir o elemento `ar[2][1]` que gerou a

expressão. Em outras palavras, a última expressão indica que o elemento `ar[2][1]` está 20 bytes adiante do elemento `ar[0][0]`.

É interessante notar que quando, numa expressão, se especifica um número de índices menor do que o número de dimensões de um array, o resultado é sempre um ponteiro. No caso específico de um array bidimensional, este ponteiro aponta para o tipo base do array. Por exemplo, dado o array bidimensional `ar` declarado no último exemplo, a referência:

```
ar[1]
```

é o mesmo que:

```
&ar[1][0]
```

que é um ponteiro para o tipo **int**.

7.6.4 Arrays Multidimensionais como Parâmetros de Funções

Para declarar um array multidimensional como parâmetro formal de uma função, deve-se especificar os tamanhos de todas as dimensões, exceto o da primeira dimensão, que pode ou não ser especificado. Por outro lado, para passar um array multidimensional como argumento 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 estes pontos.

```
#include <stdio.h>

void ImprimeArrayBi(int a[][2], int dim1, int dim2)
{
   int i, j;
   for (i = 0; i < dim2; ++i) /* Imprime o índice de cada coluna
*/
      printf("\t%d", i);

   putchar('\n');

   for (i = 0; i < dim1; ++i) {
    printf("\n%d", i); /* Imprime o índice de cada linha */
      for (j = 0; j < dim2; ++j)
         printf("\t%d", a[i][j]);
```

```
   }
}

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

    ImprimeArrayBi(ar, 3, 2);

    return 0;
}
```

Quando executado, o programa do último exemplo imprime o seguinte no meio de saída padrão:

```
            0       1

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

7.7 Arrays de Tamanhos Variáveis (C99)

Um **array de tamanho variável** (C99) é um array de duração automática cujo número de elementos é representado por uma variável. Assim, o tamanho dele só é conhecido em tempo de execução do programa. Apesar desta propriedade, uma vez definido, o tamanho do array não pode ser alterado. Por exemplo:

```
int F(int N)
{
  int arv[N]; // O array arv tem duração automática e
              // terá N elementos enquanto existir
   ...
   N = 10; // Alterar o valor de N não altera o
           // tamanho do array arv
   ...
}
```

Numa alusão de função usa-se * entre colchetes para indicar que um parâmetro é um array de comprimento variável. Por exemplo:

```
extern void F2(float arv[*], int tamanho);
```

Na definição da função, entretanto, não se deve usar esta notação. Por exemplo:

```
          void F2(float arv[tamanho], int tamanho)
{
   ...
}
```

A função `F2()` acima seria chamada como mostrado a seguir:

```
int F3(int N)
{
   float arv[N]; //
   ...
   F2(arv, N);
   ...
}
```

Outros exemplos apresentados a seguir ilustram situações válidas e inválidas de uso de arrays de comprimento variável:

```
void F(void)
{
   int M = 10;
   int N = 30;

   static int arv[N]; // ILEGAL: arrays de comprimento variável
não
                      // podem ter duração fixa
 int arvBi[M][N]; // OK: arrays multidimensionais de duração
            // automática podem ter comprimento variável

   struct est { // ILEGAL: arrays de comprimento variável
  int arMembro[N]; // não podem ser membros de estruturas
         };

 int arvErrado[*]; // ILEGAL: arrays de comprimento variável só
                 // podem usar * em alusões de funções
```

```
   int K = -1;
   int arvZerado[K]; // Legal, mas o tamanho do array será 0
}
```

7.8 Ponteiros Restritos (C99)

O padrão C99 introduziu o qualificador de tipo **restrict** que pode ser utilizado com ponteiros para informar o compilador de que o objeto apontado pelo ponteiro não possui nenhum outro ponteiro apontando para ele. O uso de **restrict** permite que o compilador execute otimizações de código relacionadas a ponteiros que utilizam esta palavra-chave. A sintaxe de declaração de um ponteiro qualificado com **restrict** é:

tipo ***restrict** nome-do-ponteiro*;

onde:

- ***tipo*** é o tipo para o qual o ponteiro aponta
- ***nome-do-ponteiro*** é o identificador associado ao ponteiro

Outros especificadores de classe de armazenamento e qualificadores de tipos apresentados no **Capítulo 4** podem ser utilizados em conjunto com a palavra-chave **restrict**. Por exemplo:

```
const int *restrict p;
```

O programador é responsável por assegurar que o objeto apontado por um ponteiro qualificado com **restrict** não possua nenhum outro ponteiro (imitador) apontando para ele. Para entender o que pode acontecer quando a palavra-chave **restrict** é usada indevidamente, considere a seguinte definição de função:

```
void F(int *a, int *b, int nElementos)
{
   int i;

   for (i = 0; i < nElementos; ++i)
      b[i] = a[i]*a[i];
}
```

Suponha ainda que esta função seja chamada como:

```
F(A, B, 10);
```

onde A e B são ponteiros para dois arrays com 10 elementos compartilhados, conforme mostrado na **Figura 26** a seguir[8].

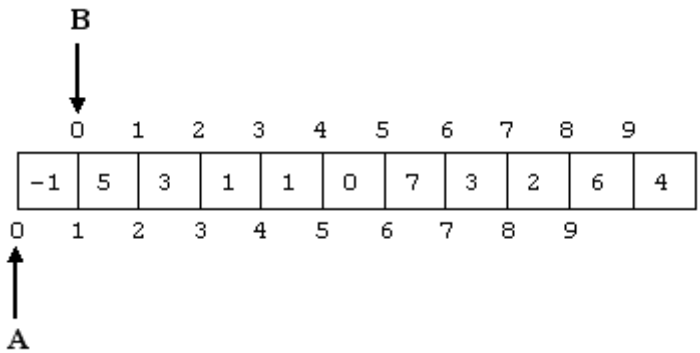

Figura 26: Exemplo de uso incorreto do qualificador restrict

Conforme mostra a **Figura 26**, os elementos dos arrays apontados por A e B são tais que vale a relação:

$$B[i] = A[i+1], \text{ quando } 0 \leq i \leq 8$$

Do modo como a função F() foi definida, o compilador não faz nenhuma suposição com relação aos elementos dos arrays A e B, e o resultado da execução desta função satisfaz a expectativa. No entanto, se o cabeçalho da função for alterado para:

```
void F(int *restrict a, int *restrict b, int nElementos)
```

pode ser que o resultado de uma execução desta função não mais corresponda ao esperado. Isto ocorre porque, agora, o programador

[8] Criar uma situação como esta é trivial. Basta definir o array A com 11 elementos (i.e., `int A[11];`), definir B como um ponteiro para **int** e iniciá-lo com o endereço do segundo elemento de A (i.e., `int *B = &A[1];`).

informa ao compilador que os valores para onde `a` e `b` apontam são acessados com exclusividade por cada um destes ponteiros. Mas isto não ocorre quando a função é chamada com os ponteiros `A` e `B` da situação descrita aqui. Assim, um possível resultado inesperado será decorrente do fato de, usando a informação implícita na palavra-chave **restrict**, o compilador poder otimizar o laço encontrado no corpo da função usando paralelismo e executar incorretamente, por exemplo, a atribuição `b[5] = a[5]*a[5]` antes da atribuição `b[4] = a[4]*a[4]`[9].

De acordo com o padrão C99, o compilador pode ignorar completamente o significado da palavra-chave **restrict** apresentado aqui.

7.9 Imitações de Ponteiros (C99)

Um ponteiro é um **imitador** de outro quando ambos apontam para a mesma posição de memória. Um ponteiro pode imitar outro e acessar o conteúdo apontado sem restrições quando ambos são do mesmo tipo. Em C99, um ponteiro também pode imitar outro mesmo quando os tipos destes ponteiros não são compatíveis, mas, neste caso, ele não pode acessar, por meio de indireção, o valor apontado, a não ser em raras exceções especificadas pelo padrão (v. a seguir). Por exemplo, em versões prévias do padrão de C, o ponteiro `p` do exemplo a seguir, que é um imitador do endereço da variável `x`, podia acessar o conteúdo desta variável, mas, de acordo com C99, isso é ilegal:

```
long x = 10L;
int *p = (int *) &x;
...
*p = 2; // Ilegal em C99, mas OK em outras versões do padrão de C
```

A garantia de que ponteiros de tipos incompatíveis não podem acessar a mesma região de memória permite que compiladores executem otimizações que não seriam possíveis em caso contrário.

Conforme foi afirmado, existem algumas poucas exceções à regra de imitação imposta por C99, mas, por simplicidade, apenas algumas destas exceções serão mencionadas aqui. Uma tal exceção é que ponteiros para tipos que diferem apenas em termos de qualificação usando **const** ou **volatile** ou o tipo apontado ter ou não sinal podem imitar uns aos outros

com acesso irrestrito aos conteúdos apontados. Ponteiros dos tipos **void*** e **char*** também podem imitar qualquer tipo de ponteiro e acessar os conteúdos apontados e vice-versa, no caso de **void***, mas não vice-versa, no caso de **char***. Por exemplo:

```
int            x;
int* const    p1 = &x; // p1 pode acessar o conteúdo de x
int* volatile p2 = &x; // p2 pode acessar o conteúdo de x
int*          p3 = (int *)&x; // p3 pode acessar o conteúdo de x
unsigned*   p4 = (unsigned *)&x; // p4 pode acessar o conteúdo de
x
char*          p5 = (char *)&x; // p5 pode acessar o conteúdo de
x
void*         p6 = &x; // p6 pode acessar o conteúdo de x
```

Como regra prática, evite tentar acessar a mesma região de memória com ponteiros de tipos diferentes em casos que não sejam semelhantes àqueles exemplificados aqui.

7.10 Exercícios de Revisão

1. (a) Que problemas podem ser decorrentes do uso de números mágicos em definições de arrays (v. **Seção 6.6**)?

(b) Qual é a vantagem que se obtém ao declarar-se o tamanho de um array em termos de uma constante simbólica ao invés de em termos do valor da constante em si?

2. Escreva uma macro que receba o nome de um array como argumento e, quando expandida, resulte no número de elementos do array.

3. (a) Como deve ser escrita a iniciação de um array unidimensional?

(b) É obrigatória a iniciação de todos os componentes do array?

4. O que é iniciação designada de array?

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

6. (a) Como deve ser escrito um parâmetro real que representa um array unidimensional numa chamada de função?

(b) Como deve ser escrito o parâmetro formal correspondente na declaração da função?

7. Como um nome de array passado como argumento real para uma função é interpretado?

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

9. (a) Se um array é passado para uma função e um de seus elementos é alterado, esta alteração é reconhecida na porção do programa que chamou a função?

(b) Se este for o caso, como se poderia garantir que os elementos de um array não são modificados por uma função?

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

11. (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***?

12. Quando se atribuem valores a elementos de um array multidimensional, qual é a vantagem de se incluírem chaves em torno de grupos de valores?

13. Quando um array multidimensional é utilizado como parâmetro de uma função, como ele deve ser declarado?

14. Explique a relação existente entre o nome de um array unidimensional e um ponteiro.

15. (a) Descreva duas formas diferentes de se especificar o endereço de um elemento de um array unidimensional.

(b) Descreva duas formas diferentes de se acessar o valor de um elemento de um array unidimensional.

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

17. O que é fator de escala?

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

```
static  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)`?

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

```
static  double  tabela[2][3] = { {1.1, 1.2, 1.3},
                                 {2.1, 2.2, 2.3} }
```

(a) O que representa `tabela`?
(b) O que representa `(tabela + 1)`?
(c) O que representa `*(tabela + 1)`?
(d) O que representa `(*(tabela + 1) + 1)`?
(e) O que representa `(*tabela + 1)`?
(f) Qual é o valor de `*(*(tabela + 1) + 1)`?
(g) Qual é o valor de `*(*tabela + 1)`?
(h) Qual é o valor de `*(*tabela + 1) + 1`?

20. Dadas as seguintes definições e atribuição:

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

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])`

21. Coloque chaves em torno de cada linha da iniciação a seguir e utilize uma forma de endentação que melhore a legibilidade dela.

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

22. 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]);
```

23. 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 impressão dos valores do array `ar` utilizando índices.

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

24. Quais das seguintes expressões são eqüivalentes a `a[i][j]`?

```
(a) *(a[i] + j)
(b) **(a[i + j)
(c) (*(a + i))[j]
(d) (*(a + j))[i]
(e) *((*(a + i)) + j)
(f) **(a + i) + j
(g) *(&a[0][0] + i + j)
```

7.11 Exercícios de Programação

EP7.1) Suponha que se deseje processar um conjunto de valores representado altura e sexo (M/F) de um grupo de 10 pessoas. Escreva um programa em C que:

(a) Leia este conjunto de dados e armazene-o em dois arrays vinculados, um dos quais contendo as alturas e o outro contendo os sexos dos indivíduos.

(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.

(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`').

(d) Determine o número total de indivíduos de cada sexo.

EP7.2) Uma empresa fez uma pesquisa para saber se as pessoas gostaram ou não de um novo produto. Um número `N` de pessoas de ambos os sexos foi entrevistado, e o questionário consistia em apenas duas perguntas: (i) o sexo do entrevistado (M/F) e (ii) sua opinião sobre o produto (gostou/não gostou). Suponha que, no máximo, 200 pessoas poderiam ser entrevistadas. Escreva um programa em C que:

(a) Leia as respostas contidas nos questionário e armazene-as em dois arrays vinculados, um deles contendo a resposta para a primeira pergunta e o outro contendo a resposta para a segunda pergunta.

(b) Encontre a porcentagem de pessoas do sexo feminino que responderam que gostaram do produto.

(c) Determine a porcentagem de pessoas do sexo masculino que responderam que ***não*** gostaram do produto.

EP7.3) Mesmo que seu compilador ainda não implemente a palavra-chave **restrict**, você pode escrever um programa que simule a situação descrita na **Figura 26** da **Seção 7.8**. Para isto, implemente duas funções `F1()` e `F2()`, ambas com o mesmo protótipo da função `F()` apresentada na **Seção 7.8**

(sem o uso de **restrict**). Faça com que a variável de contagem `i` do laço **for** da função `F1()` varie de 0 (inclusive) a `nElementos/2` (exclusive), enquanto a variável `i` do laço **for** da função `F2()` deve variar de `nElementos/2` (inclusive) a `nElementos` (exclusive). Na função **main()** de seu programa, crie e inicie os dois arrays de modo a refletir a situação descrita na **Figura 26**. A primeira parte do experimento consiste em imprimir o array `B` após chamar `F1()` e, em seguida, chamar `F2()`. Na segunda parte do experimento, reinicie o array `A` e inverta a ordem das chamadas de `F1()` e `F2()`. A primeira combinação de chamadas (i.e., chamar `F1()` e, em seguida, `F2()`) corresponde ao resultado correto; i.e., o que ocorreria na chamada da função `F()` apresentada na **Seção 7.8** sem o uso de **restrict**. A segunda combinação de chamadas (i.e., chamar `F1()` e, em seguida, `F2()`) corresponde a um possível resultado incorreto; i.e., o que poderia ocorrer na chamada da função `F()` com o uso indevido de **restrict**.

EP7.4) Escreva um programa em C que solicite ao usuário a introdução de `n` valores inteiros, leia estes números e apresente logo antes de encerrar, o seguinte:

(i) Todos os números introduzidos
(ii) O menor valor introduzido
(iii) O maior valor introduzido
(iv) A média dos valores introduzidos.

O valor `n` deve ser o primeiro dado introduzido pelo usuário, mas você deve estipular em seu programa um valor máximo para `n` (não use números mágicos). Veja um exemplo de interação com o programa (**negrito** corresponde à entrada do usuário):

```
[Apresentação do programa]
Quantos números você irá introduzir? 3
Introduza o próximo número: 5
Introduza o próximo número: -2
Introduza o próximo número: 0

Os números introduzidos foram: 3, 5, -2 e 0.
Menor valor introduzido: -2
Maior valor introduzido: 5
Média dos valores introduzidos: 1.5
```

SUGESTÃO

Este problema é simples e similar àquele do exercício **EP2.12**, proposto no **Capítulo 2**. A diferença é que, aqui, pede-se também para apresentar todos os números introduzidos pelo usuário. Para guardar todos estes valores, use um array de tamanho igual ao número máximo de valores que você deverá especificar.

EP7.5) Escreva um programa que lance um dado `n` vezes e imprima o percentual de surgimento de cada face do dado. O valor `n` é introduzido pelo usuário, sendo que 0 encerra o programa. Seu programa deverá utilizar um array para armazenar os números de aparecimento de cada face. Aproveite-se deste fato para evitar o uso de instruções condicionais (**switch** ou **if**) no cálculo dos percentuais. (**Sugestão**: O funcionamento de seu programa deve ser exatamente igual àquele do programa proposto no exercício **EP3.8** do **Capítulo 3**. Entretanto, aqui pede-se que seja utilizado um array para armazenar os aparecimentos das faces do dado. Lembre-se de alimentar o gerador de números aleatórios apenas uma vez.)