Capítulo 3

ARRANJOS E PONTEIROS

3.1 Arranjos

Um **arranjo** é uma coleção de variáveis de mesmo tipo armazenadas contiguamente em memória. Cada uma destas variáveis componentes de um arranjo é denominada um **elemento** do arranjo, e pode ser acessada por meio do nome do arranjo seguido de um **índice** (também chamado **subscrito**). Em C, o elemento inicial de um arranjo sempre tem subscrito (índice) igual a zero.

O propósito típico de arranjos é armazenar quantidades de dados de um mesmo tipo relacionados entre si. Por exemplo, se você desejasse armazenar as notas de 50 alunos utilizando variáveis convencionais, você teria que declarar todas as 50 variáveis necessárias; utilizando arranjos, entretanto, a declaração de uma única variável (neste caso, com 50 elementos) do tipo arranjo seria suficiente.

3.1.1 Declaração de Arranjos

A declaração de uma variável de um tipo arranjo em C, em sua forma mais simples, consiste no seguinte:

<tipo> <nome do arranjo> [<tamanho do arranjo>];

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

3.1.2 Acessando Elementos de um Arranjo

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

```
float   notas[50];

notas[0] = 5.0;  /* Atribui o valor 5.0 ao elemento inicial do arranjo */
      .
      :
notas[49] = 7.5; /* Atribui o valor 7.5 ao último elemento do arranjo */
notas[50] = 9.0; /* Esta referência é ILEGAL pois ultrapassa o limite */
                 /* superior do arranjo                               */
```

O fato de C iniciar a indexação de arranjos sempre com 0 é devido à maior eficiência do código produzido por esta opção em relação à indexação começando em 1 de outras linguagens. Além disso, a indexação começando em 0 facilita o acesso a elementos de um arranjo por meio de ponteiros, conforme será visto mais adiante.

Nem todo compilador C faz verificação de acesso além dos limites de um arranjo (o padrão ANSI/ISO não requer isso). Portanto, o programador pode tentar (acidentalmente) acessar porções de memória que não foram alocadas para o arranjo, 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 a *quebra* (i.e., a ocorrência de erro em tempo de execução) do programa. Freqüentemente, este tipo de erro é causado porque o programador excede em 1 o teste de final de um arranjo acessado num laço de repetição **for**, como mostra o exemplo a seguir.

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

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

No último exemplo, o arranjo `ar[]` foi declarado com capacidade para conter 10 elementos. Portanto, ele pode ser acessado apenas com índices variando de 0 a 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 arranjo `ar[]`, mas que, provavelmente, pertence à variável `j`[1]. Portanto, este *bug* irá causar um laço de repetição infinito, uma vez que `j` é reinicializado com zero sempre que se atinge o final previsto para o laço.

É interessante notar ainda que se pode determinar o tamanho, em bytes, de um arranjo por meio do uso do operador **sizeof**, como por exemplo:

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

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

Note entretanto que, para obter o tamanho de um dado arranjo, como acima, não se deve utilizar nenhum subscrito; caso contrário, o tamanho retornado será aquele de um único elemento do arranjo (ao invés do tamanho de todo o arranjo). Por exemplo:

```
sizeof(notas[0]); /* Resultaria no espaço ocupado pelo tipo float */
```

3.1.3 Inicialização de Arranjos

Como padrão, arranjos com duração fixa têm todos os seus elementos inicializados com zero, mas podem-se inicializar todos ou alguns elementos com outros valores. A inicialização de elementos de um arranjo é feita por meio do uso de expressões constantes, separadas por vírgulas e entre chaves, seguindo a declaração do arranjo. Por exemplo:

```
static  int  meuArranjo1[5];
static  int  meuArranjo2[5] = {1, 2, 3.14, 4, 5};
```

No primeiro caso, todos os elementos de `meuArranjo1` serão inicializados com `0`, enquanto que, no segundo caso, `meuArranjo2[0]` recebe o valor `1`, `meuArranjo2[1]` recebe o valor `2`, `meuArranjo2[2]` recebe o valor `3` (aqui ocorre uma conversão de tipo), `meuArranjo2[3]` recebe o valor `4`, e `meuArranjo2[4]` recebe o valor `5`.

É ilegal incluir numa inicialização um número de valores maior do que o permitido pelo tamanho do arranjo, mas não é necessário atribuir valores a todos os elementos do arranjo. Isto é, se houver um número de valores de inicialização menor do que o número de elementos do arranjo, os elementos remanescentes terão o valor zero atribuído.

---

[1] Isto é provável porque a declaração da variável `j` foi feita logo em seguida à declaração de `ar[]`.

Quando valores iniciais são atribuídos a todos os elementos de um arranjo, o tamanho do arranjo pode ser omitido, pois, neste caso, o compilador deduz o tamanho do arranjo baseado no número de valores iniciais. Por exemplo:

```
static  int  meuArranjo2[] = {1, 2, 3.14, 4, 5};
```

é o mesmo que:

```
static  int  meuArranjo2[5] = {1, 2, 3.14, 4, 5};
```

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

3.2 Aritmética de Ponteiros

Antes de introduzir uma importante relação entre ponteiros e arranjos, é necessário apresentar a noção de aritmética de ponteiros. A linguagem C permite adição e subtração entre ponteiros e inteiros, e entre ponteiros entre si. Por exemplo, se `p` é um ponteiro declarado como:

```
int  *p;
```

a expressão:

```
p + 3
```

é perfeitamente legal, e deve ser interpretada como o endereço da posição de memória que está três objetos adiante do endereço do objeto para o qual `p` aponta. Isto é, como `p` é um endereço, `p + 3` também será um endereço. Entretanto, ao invés de simplesmente adicionar 3 ao valor do endereço de `p`, o compilador adiciona o valor 3 multiplicado pelo tamanho do objeto para o qual `p` aponta[2]. Suponha, por exemplo, que o endereço correntemente contido em `p` é `1000`, e que o tipo **int** é armazenado em 4 bytes. Então, `p + 3` significa `1000 + 3*4`, que é igual ao endereço `1012`. Por outro lado, se `p` tivesse sido declarado como `char *p`, então `p + 3` significaria `1003`. Concluindo, `p + 3` sempre significa o endereço do terceiro objeto após `p`, independentemente do tipo de objeto para o qual `p` aponta.

Subtrair um inteiro de um ponteiro tem uma interpretação semelhante. Por exemplo, `p - 3` significa o endereço do terceiro objeto antes de `p`.

Operações de incremento e decremento de ponteiros também são bastante comuns em C. Neste caso, os operadores de incremento e decremento são utilizados para fazer um ponteiro apontar para o elemento posterior e anterior, respectivamente, à sua posição atual.

A subtração de dois ponteiros é legal, desde que os ponteiros apontem para objetos 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. Por exemplo:

```
&a[3] - &a[0]
```

resulta em 3, significando que existem 3 objetos (do tipo dos elementos do arranjo `a[]`) entre `a[3]` e `a[0]`.

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

---

[2] Isto é denominado *scaling*, em Inglês.

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

3.3 O Ponteiro Nulo

Um **ponteiro nulo** é um ponteiro que não aponta para nenhum objeto válido. Um ponteiro torna-se nulo quando recebe o valor inteiro 0. Por exemplo,

```
char  *p;

p = 0;  /* Torna p um ponteiro nulo */
```

A seguinte macro pode tornar expressões de atribuição e comparação envolvendo ponteiros nulos mais legíveis[3]:

```
#define  NULL  0
```

Também é comum escreverem-se laços de repetição envolvendo um ponteiro que termina quando o valor do ponteiro resulta em nulo. Por exemplo:

```
while (p){
     .
     :
}
```

ou, alternativamente:

```
 while (p != NULL){
     .
     :
}
```

3.4 Relação entre Ponteiros e Arranjos Unidimensionais

Conforme foi visto anteriormente, uma forma de se acessarem os elementos de um arranjo é através de subscritos. Ponteiros provêem outra forma de acesso a elementos de um arranjo. Suponha, por exemplo, a existência das seguintes declarações:

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

A instrução a seguir pode ser utilizada para fazer com que `p` aponte para o início do arranjo (i.e., para o elemento `ar[0]`):

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

Portanto, a referenciação do ponteiro `p`:

---

[3] De acordo com o padrão ISO corrente, 0 é o único valor inteiro compatível com qualquer ponteiro. Entretanto, alguns compiladores mais antigos requerem que a macro `NULL` seja definida como: `#define NULL ((void *)0)` para que este valor seja compatível com qualquer tipo de ponteiro. De qualquer modo, `NULL` é definida em `stdlib.h` (na maioria dos compiladores) e o programador não precisa se preocupar em defini-la.

`*p`

resulta no valor de `ar[0]`. Além disso, utilizando-se aritmética de ponteiros pode-se ter acesso aos outros elementos do arranjo. 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 arranjo `ar[]`, então a seguinte relação é válida:

`*(p + i)` é o mesmo que `ar[i]`

para qualquer `i` inteiro. Em palavras, esta relação pode ser traduzida como: *adicionar um inteiro a um ponteiro que aponta para o início de um arranjo e então referenciar esta expressão, é o mesmo que utilizar o inteiro como índice do arranjo.*

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

`ar` é o mesmo que `&ar[0]`

Combinando as duas relações vistas anteriormente, 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 C encontra uma referência com subscrito a um elemento de um arranjo, ele adiciona o subscrito ao endereço do arranjo, obtendo, assim, o endereço do elemento. Então, o compilador referencia este endereço e obtém o valor do próprio elemento. Por exemplo, quando o compilador encontra uma expressão tal como `ar[2]`, ele a interpreta como o conteúdo do endereço `ar + 2` [i.e., `*(ar + 2)`], onde `ar` representa o endereço inicial do arranjo.

Devido às relações apresentadas acima, ponteiros e arranjos podem ser utilizados de modo eqüivalente para referenciar elementos de arranjos. É importante lembrar, entretanto, que valores de variáveis do tipo ponteiro podem ser modificados, enquanto que o valor atribuído a um nome de arranjo não pode ser modificado. Isto ocorre porque o nome de um arranjo não é realmente um nome de variável: ele simplesmente representa o ***endereço da variável arranjo***, e o endereço de uma variável não pode ser modificado (v. **Seção 2.1**). Em termos práticos, isto significa, por exemplo, que o nome de um arranjo (sem subscrito) não pode aparecer no lado esquerdo de uma instrução de atribuição.

A seguir estão apresentados mais alguns exemplos de instruções legais e ilegais envolvendo arranjos e ponteiros:

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

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

As relações e diferenças entre ponteiros e arranjos são muito importantes para o domínio da linguagem C. Convença-se de que entendeu cada exemplo apresentado nesta seção antes de prosseguir.

3.5 Arranjos como Parâmetros de Funções

Em C, o nome de um arranjo que aparece como parâmetro real numa chamada de função é interpretado como o endereço do primeiro elemento do arranjo. Por exemplo:

```
float  meuArranjo[10];

MinhaFuncao(meuArranjo);
```

é o mesmo que:

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

Por outro lado, na declaração de uma função, um parâmetro formal de tipo arranjo é declarado como um ponteiro para o elemento inicial do arranjo. Existem duas formas de fazer isso:

```
void  MinhaFuncao(float  *arranjo)
```

ou

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

No segundo exemplo, `arranjo` é declarado como um arranjo de tamanho indeterminado. O tamanho do arranjo pode ser omitido porque o arranjo na realidade deve ser criado pela função que chama esta função aqui declarada, e o argumento que será passado é de fato o endereço do primeiro elemento do arranjo. Portanto, o compilador converte o segundo tipo de declaração no primeiro (i.e., `float  arranjo[]` é convertido para `float *arranjo`). Conseqüentemente, os dois tipos de declarações são funcionalmente eqüivalentes. Entretanto, em termos de legibilidade, a segunda declaração é melhor do que a primeira, pois a segunda enfatiza que o parâmetro será tratado como um arranjo. Na primeira declaração, não existe, em princípio, uma maneira de se saber se o argumento é um ponteiro para um único **float** ou para o primeiro elemento de um arranjo de **float**s.

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

A escolha entre declarar um argumento de função como arranjo ou como ponteiro não tem nenhum efeito na tradução feita pelo compilador (a não ser que o compilador suporte verificação de acesso dentro dos limites de arranjos). Para o compilador, o argumento `arranjo[]` do exemplo anterior é simplesmente um ponteiro para um **float** e não propriamente um arranjo. Mas, devido à eqüivalência entre ponteiros e arranjos, ainda é possível acessar os elementos de `arranjo[]` como se fosse um arranjo.

É interessante notar ainda que não se pode determinar o tamanho de um arranjo dentro de uma função através da aplicação do operador **sizeof** sobre o argumento-arranjo. Por exemplo, se `MinhaFuncao()` fosse definida como:

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

uma chamada a esta função imprimiria o número de bytes necessários para armazenar um ponteiro, e não o número de bytes necessários para armazenar o arranjo passado para a função durante a chamada[4].

Devido à impossibilidade de uma função determinar o tamanho de um arranjo passado como parâmetro real, é muitas vezes uma boa idéia incluir o tamanho do arranjo na lista de argumentos da função. Isto permite à função saber aonde o arranjo termina, conforme esquematizado no exemplo a seguir:

---

[4] O tamanho do arranjo em bytes pode, obviamente, ser determinado utilizando o operador **sizeof** com o nome do arranjo ***dentro do escopo do arranjo*** conforme foi visto anteriormente.

```
#define  TAMANHO_MAXIMO  100

void  MinhaFuncao(float  arranjo[], unsigned  tamanhoDoArranjo)
{
    unsigned  i;
    ...

    if (tamanhoDoArranjo > TAMANHO_MAXIMO){
       printf("O arranjo e' excessivamente grande");
       exit(1);
    }

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

Do lado da função que faz a chamada, aonde é definido o arranjo passado como argumento, o tamanho do arranjo (i.e., o número de elementos do mesmo) pode ser calculado dividindo-se o tamanho total do arranjo em bytes pelo tamanho de um elemento do arranjo em bytes. Por exemplo, considerando a declaração de arranjo a seguir:

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

a função `MinhaFuncao()` do último exemplo poderia ser invocada através de:

```
MinhaFuncao(meuArranjo, sizeof(meuArranjo)/sizeof(meuArranjo[0]));
```

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

3.6 Arranjos Multidimensionais

Um arranjo multidimensional é um arranjo de arranjos, e é declarado com pares consecutivos de colchetes contendo os tamanhos de cada dimensão:

<tipo do elemento> <nome do arranjo> [<tamanho 1>][<tamanho 2>]...[<tamanho N>]

Embora o padrão ANSI/ISO determine que um compilador C deve suportar pelo menos seis dimensões para arranjos multidimensionais, raramente mais de três dimensões são utilizadas em aplicações práticas. No exemplo a seguir, um arranjo de caracteres de três dimensões é declarado:

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

A variável `arranjoDeCaracteres` do exemplo acima pode ser interpretada como um arranjo de três elementos, cada um dos quais é um arranjo de quatro elementos, cada um dos quais é um arranjo de cinco elementos do tipo **char**.

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

3.6.1 Inicialização de Arranjos Multidimensionais

Para inicializar um arranjo multidimensional, deve-se colocar cada linha[5] do arranjo entre chaves. Se houver um número de valores menor numa dada linha, zeros serão acrescentados. Considere o seguinte exemplo (observe como a endentação torna a inicialização mais legível):

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

Este exemplo declara um arranjo com 5 linhas e 3 colunas, mas apenas as três primeiras linhas são inicializadas, e apenas o primeiro elemento da segunda linha é inicializado. Em forma de tabela, este arranjo poderia ser visualizado como a seguir:

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

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

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

o resultado na inicialização seria:

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

Obviamente, a inicialização deste último exemplo é confusa, pois um programador menos experiente poderia pensar que o resultado desta inicialização seria aquele do penúltimo exemplo. Portanto, para melhorar a legibilidade em inicializações de arranjos multidimensionais, sempre coloque cada linha entre chaves.

Exercício: Coloque chaves em torno de cada linha da inicialização do último exemplo e utilize uma forma de endentação que melhore a legibilidade da mesma.

Se a especificação de tamanho da primeira dimensão de um arranjo multidimensional for omitida, o compilador automaticamente deduz o tamanho desta dimensão baseado no número de valores de inicialização 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 arranjo 2x3x2, pois existem 12 valores, e cada elemento do arranjo `arA` é um arranjo 3x2 (12 dividido por 3*2 resulta em 2, que é o tamanho omitido da primeira dimensão).

Por outro lado, a declaração a seguir:

---

[5] Uma **linha** de um arranjo multidimensional corresponde à sua primeira dimensão, enquanto que uma **coluna** corresponde às demais dimensões. Apesar de esta terminologia ser mais intuitiva no caso de arranjos bidimensionais (devido à analogia com matrizes em Matemática), a mesma é também utilizada para arranjos com dimensões maiores que dois.

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

é ilegal porque o compilador não consegue determinar os tamanho das dimensões do arranjo (i.e., 2x3 ou 3x2?). Entretanto, se o tamanho da segunda dimensão for especificado, a declaração deixa de ser ambígua e, portanto, legal.

3.6.2 Acesso a Elementos de Arranjos Multidimensionais

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

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

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

atribui a `x` o valor do terceiro elemento **int** do segundo arranjo de **int**s do segundo arranjo de arranjos de **int**s.

Um erro comum cometido por programadores iniciantes em C (especialmente entre aqueles acostumados com outra linguagem de programação, tal como Pascal) é utilizar vírgula para separar índices. Por exemplo, um erro comum é utilizar:

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

ao invés de:

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

A primeira instrução é exatamente eqüivalente à segunda em Pascal, mas em C ela tem um significado diferente porque a vírgula é um operador em C (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 arranjo bidimensional de **int**s, `a[2]` é o endereço do elemento `a[2][0]` (v. abaixo) e portanto não pode ser modificado.

3.6.3 Relação entre Ponteiros e Arranjos Multidimensionais

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

Suponha que o tipo **int** ocupe 4 bytes e considere a seguinte declaração:

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

Uma referência a este arranjo tal como:

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

é interpretada pelo compilador como:

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

Esta última expressão representa o elemento situado dois elementos além do elemento `ar[1]`. Mas, o que significa `ar[1]`, se `ar[][]` representa um arranjo bidimensional? Para responder esta pergunta, recorde-se que **`int`** `ar[2][3]` pode ser visto como um arranjo unidimensional de tamanho 2, cujos elementos são arranjos de **int**s de tamanho 3. Portanto, `ar[1]` representa o segundo elemento do arranjo `ar[][]`. Este elemento, de acordo com a inicialização de `ar[][]`, é o arranjo consistindo dos valores 4, 5 e 6. Além disso, `2` na

expressão `*(ar[1] + 2)` deve ser interpretado, de acordo com a aritmética de ponteiros, como duas vezes o tamanho (em bytes) de cada elemento do arranjo de **int**s (i.e., `2*4`, pois se supõe aqui que um **int** ocupa 4 bytes).

Prosseguindo com o exemplo, a última expressão obtida pode ainda ser expandida como:

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

Agora, lembre-se novamente que `ar` é um arranjo cujos elementos são arranjos, de modo que `1` nesta última expressão representa uma vez o tamanho (em bytes) de cada um destes elementos, que são arranjos de três **int**s. Portanto, `1` na expressão acima deve ser multiplicado por três vezes o tamanho (em bytes) de um **int**, que é 4. Portanto, após *scaling*, `1` na última expressão transforma-se em `1*3*4`.

Continuando com o exemplo, a última expressão obtida é eqüivalente a:

```
*(int *)( ( (char *)ar + (1*3*4) ) + (2*4) )
```

A razão pela qual o ***casting*** `(char *)` foi colocado antes de `ar` é evitar que o compilador faça o *scaling* que já foi feito explicitamente na expressão[6]. O *casting* `(int *)` assegura que a expressão sendo referenciada (i.e., toda a expressão seguindo o primeiro asterisco) é um ponteiro para o tipo **int**. Este último ***casting*** é necessário porque o *casting* `(char *)` transforma a expressão:

```
( (char *)ar + (1*3*4) ) + (2*4)
```

num ponteiro para o tipo **char**. Portanto, sem o *casting* `(int *)`, a referenciação desta expressão resultaria no conteúdo de apenas um byte de memória (ao invés de 4 bytes).

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

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

O valor 20 na última expressão já passou por *scaling*, de modo que este número representa o número de bytes a serem atravessados a partir do início do arranjo a fim de se atingir o elemento `ar[1][2]`, que gerou esta expressão. Em outras palavras, a última expressão indica que o elemento `ar[1][2]` está 20 bytes adiante do elemento `ar[0][0]`.

É interessante notar ainda que quando se especifica um número de subscritos menor do que o número de dimensões de um arranjo numa expressão, o resultado é um ponteiro para o tipo-base do arranjo. Por exemplo, dado o arranjo bidimensional `ar` declarado no último exemplo, a referência:

```
ar[1]
```

seria o mesmo que:

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

que é um ponteiro para um **int**.

3.6.4 Passagem de Arranjos Multidimensionais como Argumentos

Para passar um arranjo multidimensional como argumento real para uma função deve-se proceder da mesma forma que com arranjos unidimensionais. Isto é, apenas o nome do arranjo deve ser utilizado na chamada. Por outro lado, para declarar um arranjo multidimensional como parâmetro formal de uma função, deve-se especificar os tamanhos de todas as dimensões, exceto o tamanho da primeira dimensão que pode ou não ser especificado.

Por exemplo:

---

[6] O *scaling* para um ponteiro para o tipo **char** é 1, pois o este tipo ocupa apenas 1 byte.

```
void  F1()
{
   int   ar[5][6][7];
     .
     :
   F2(ar);
   ...
}

void  F2(int arranjo[][6][7])
{
    ...
}
```

3.7 Strings

Strings são **cadeias** (ou seqüências) **de caracteres**. Em C, um *string* é representado por um arranjo de caracteres terminado pelo **caracter nulo**, representado pela seqüência de escape '`\0`'. Um *string* constante é qualquer seqüência de caracteres entre aspas, e seu tipo de dados é um arranjo de elementos do tipo **char**. O compilador sempre acrescenta o caractere nulo para designar o final de um *string* constante.

3.7.1 Declaração e Inicialização de Strings

Pode-se armazenar um *string* em memória declarando-se um arranjo do tipo **char**. Pode-se ainda iniciar um arranjo de caracteres com um *string* constante. Por exemplo:

```
char  str[] = "Isto eh um string.";
```

Devido ao caractere terminal do *string*, o tamanho do arranjo que o armazena é sempre um a mais do que o número de caracteres visíveis no *string*. Assim, o arranjo `str` do exemplo acima possui tamanho igual a 19 (= 18 caracteres visíveis mais o caractere terminal).

Se o tamanho do arranjo for especificado, este tamanho deve ser no mínimo igual ao número de caracteres presentes na inicialização mais um; os elementos remanescentes no arranjo, se for o caso, serão inicializados com 0. Por exemplo, na declaração a seguir:

```
char  str[10] = "ola";
```

`str[0]` assume o valor '`o`', `str[1]` assume o valor '`l`', `str[2]` assume o valor '`a`', e `str[3]` assume o valor '`\0`'. Os elementos restantes (i.e., de `str[4]` a `str[9]`) recebem o valor 0. Por outro lado, a declaração:

```
char  str[4] = "Tudo bem.";
```

seria ilegal (por que?).

Em C, ainda é permitida a inicialização de um ponteiro para **char** com um *string* constante. Por exemplo:

```
char  *ptr = "Isto eh um string constante."
```

Entretanto, esta última declaração é ligeiramente diferente das declarações precedentes que utilizam arranjos. Uma diferença entre esta última declaração e a declaração:

```
char  str[] = "Isto eh um string constante."
```

é que no primeiro caso, além do espaço reservado para conter o *string*, também é alocado espaço para conter o ponteiro `ptr`. Além disso, apesar de `ptr` e `str` apontarem para o elemento inicial do *string* (i.e., para o caractere '`I`'), o valor de `ptr` pode ser modificado,

enquanto o endereço `str` não pode (v. **Seção 2.1**). Note, entretanto, que se o valor de `ptr` for modificado, o endereço com o qual este ponteiro foi iniciado será perdido (i.e., o string para onde `ptr` estava apontando não mais poderá ser acessado).

A razão pela qual se pode inicializar um ponteiro para **char** com um *string* constante é que um *string* é um arranjo de caracteres, de modo que um *string* constante representa um ponteiro para o primeiro caractere do arranjo de caracteres que contém o *string*. Isto significa que, conforme já foi visto, pode-se atribuir um *string* constante a um ponteiro que aponta para um **char**[7].

Os exemplos apresentados a seguir chamam a atenção para algumas dúvidas que um programador inexperiente em C pode ter com o uso de ponteiros, arranjos e *strings*.

```
char  ar[10];
char  *ptr = "10 espacos";

ar = "errado";    /* ILEGAL */
```

Esta instrução é ***ilegal*** porque `ar` representa o endereço do elemento inicial do arranjo `ar[]`, e este endereço não pode ser modificado.

```
ar[2] = 'a';
```

Esta instrução é ***legal***: ela representa simplesmente a atribuição do caractere 'a' ao terceiro elemento do arranjo `ar[]`, que é um arranjo de caracteres.

```
ptr[5] = 'b';
```

Esta instrução é ***legal*** devido à relação entre ponteiros e arranjos; esta atribuição modifica o valor do elemento de índice 5 do *string* `"10 espacos"` para 'b', de modo que este *string* se torna `"10 esbacos"`. Note que o valor do ponteiro `ptr` em si ***não é modificado***. **Importante**: apesar de ser legal, esta instrução pode causar problemas se o string for armazenado numa área de memória apenas para leitura (v. última nota de rodapé). Portanto, é melhor evitar instruções que tentem modificar strings considerados constantes.

```
*(ptr + 5) = 'b';
```

Esta instrução é ***legal*** e é exatamente eqüivalente à instrução anterior (e pode causar o mesmo problema da instrução anterior).

```
ptr = "OK";
```

Esta instrução é obviamente legal. Talvez menos óbvio é o fato que `ptr` agora aponta para outra posição de memória: aquela que contém o *string* `"OK"`. O endereço do *string* antigo para onde `ptr` apontava fica perdido, e aquele *string* não pode mais ser acessado.

```
ptr[5] = 'b';
```

Como visto anteriormente, esta instrução é ***sintaticamente legal***. Entretanto, agora, esta instrução irá provavelmente causar problemas quando for executada. O problema é que, devido à atribuição no item anterior, `ptr` aponta para o *string* `"OK"`, que possui apenas três posições de memória alocadas, e esta última instrução atribui o valor 'b' à terceira posição de memória além do *string* `"OK"`. Portanto, se está modificando uma posição de memória que é

[7] IMPORTANTE: Uma outra diferença entre estes dois tipos de declarações é o fato de alguns compiladores armazenarem strings constantes em posições de memória cujos conteúdos não podem ser modificados (i.e., são apenas para leitura). Em tal situação, qualquer tentativa de modificar o string para onde o ponteiro `ptr` aponta gera um erro em tempo de execução do programa. Para não correr riscos, considere os conteúdos de strings cujos endereços são atribuídos a ponteiros *realmente constantes*.

desconhecida. Em outras palavras, esta instrução não irá causar um erro de compilação, mas certamente causará um erro durante a execução do programa.

```
*ptr = "ilegal";  /* ILEGAL */
```

Esta instrução é ***ilegal***. Aqui, está-se tentando atribuir um valor para o caractere para o qual `ptr` aponta. O *string* constante `"ilegal"` é interpretado como o endereço do elemento inicial do arranjo que contém este *string* em memória. Portanto, esta atribuição representa, na realidade, a tentativa de atribuição de um endereço a um objeto do tipo **char**, o que é ilegal.

Novamente, é extremamente importante que você entenda todos os exemplos apresentados nesta seção antes de prosseguir.

3.7.2 Comparação entre Strings e Caracteres

É importante entender bem as diferenças entre *strings* constantes e caracteres constantes. A primeira diferença refere-se ao espaço ocupado por um caractere constante e um *string* constante consistido de apenas um caractere: no primeiro caso, apenas um espaço é alocado em memória, enquanto que, no segundo caso, dois espaços são alocados devido ao caractere nulo de terminação do *string*. Por exemplo, o caractere constante `'a'` ocupa apenas um byte, enquanto que o *string* constante `"a"` ocupa dois bytes.

Outra diferença é que é legal atribuir um caractere constante ao conteúdo de um ponteiro para o tipo **char**, mas o mesmo não é verdade com relação a um *string* constante. Por exemplo, se `p` é um ponteiro para **char**, a atribuição:

```
*p = 'a';  /* Legal */
```

é legal, mas a atribuição seguinte não é legal:

```
*p = "a";   /* ILEGAL */
```

Como um *string* é interpretado como um ponteiro para **char** e um ponteiro referenciado tem o mesmo tipo do objeto para o qual o ponteiro aponta, esta última instrução tenta atribuir o valor de um ponteiro (i.e., um endereço) a uma variável do tipo **char**, o que é ilegal. De modo similar, é legal atribuir um *string* a um ponteiro (sem referenciação), mas é incorreto atribuir um caractere constante a um ponteiro. Por exemplo, se `p` é um ponteiro para **char**:

```
p = "a";    /* Legal */
p = 'a';    /* ILEGAL */
```

É importante ainda chamar a atenção para uma confusão comum entre iniciantes em C: inicializações e atribuições não são eqüivalentes. Por exemplo, a inicialização:

```
char  *p = "string";
```

é legal, pois se está atribuindo um endereço a um ponteiro. Entretanto, a instrução:

```
*p = "string";
```

não é legal, pois se está tentando atribuir um endereço a uma variável do tipo `char` `(i.e., *p)`.

3.7.3 Funções de Biblioteca para Tratamento de Strings

A biblioteca da linguagem C possui várias funções para tratamento de *strings*. A maioria dessas funções encontra-se no módulo de biblioteca ***string*** (para utilizá-las, use `#include <string.h>` em seu programa).

3.7.3.1 Entrada e Saída de Strings

Pode-se ler e imprimir *strings* utilizando as funções **scanf()** e **printf()** em conjunto com o especificador de formato `%s`. O argumento a ser utilizado com **scanf()** deve ser um arranjo de caracteres ***com tamanho suficiente para conter o string de entrada***. Após a leitura do *string* de entrada, **scanf()** automaticamente acrescenta o caractere nulo no final do *string*. Por exemplo[8]:

```
char  str[30];

scanf("%s", str);
```

Esta última instrução seria capaz de ler um *string* de no máximo 29 (por que não 30?) caracteres introduzidos pelo usuário do programa.

O argumento utilizado com **printf()** para impressão de *strings* no meio de saída deve ser um ponteiro para um arranjo de caracteres contendo o caractere nulo. A função **printf()** imprime todos os caracteres no arranjo até que o caracter nulo seja encontrado. Por exemplo:

```
char  str[] = "isto é um string"

printf("O string a ser impresso eh: %s.\n", str);
```

Esta última instrução causaria a impressão do seguinte no meio de saída padrão:

```
O string a ser impresso eh: isto é um string.
```

Uma desvantagem do uso de **scanf()** para entrada de *strings* é que cada *string* termina quando o usuário digita um espaço em branco, o que significa que *strings* contendo caracteres em branco não podem ser introduzidos usando essa função. Uma forma de superar esta deficiência é utilizar a função de entrada **gets()** que é específica para entrada de *strings*.

A função **gets()** tem um único argumento que é um arranjo de caracteres[9]. Caracteres são lidos do teclado até que `<ENTER>` ou `<RETURN>` seja encontrado. Então, a função adiciona o caractere nulo e os caracteres obtidos são armazenados a partir do endereço recebido como argumento. Exemplo:

```
char  str[30];

gets(str);
```

Existe ainda a função **puts()** para saída de *strings*, mas esta não oferece nenhuma facilidade adicional que não seja oferecida pela função **printf()**.

3.7.3.2 Comprimento de Strings

A função de tratamento de *strings* mais simples é aquela que calcula o comprimento de um *string*. A função **strlen()** retorna o comprimento do *string* que recebe como argumento. Para utilizar esta função, inclua o arquivo de cabeçalho ***`string`***`.h`. Por exemplo,

```
char *ptr = "um string de comprimento 27"
int  j;

j = strlen(ptr);  /* Resulta na atribuição de 27 a j */
```

---

[8] O uso do operador **&** precedendo `str` não é necessário aqui, pois `str` já é um endereço, mas seu uso não causa erro num compilador ANSI/ISO C.

[9] Não esqueça: o arranjo deve ser suficientemente grande para conter o string introduzido.

3.7.3.3 Cópia de Strings

A função **strcpy**() recebe dois *strings* como argumentos e, após sua execução, o segundo *string* é copiado no primeiro. Para utilizar esta função, inclua o arquivo de cabeçalho `string.h`. Por exemplo,

```
char *origem = "String a ser copiado";
char destino[5];

strcpy(destino, origem);
```

Ao final desta chamada, `destino` apontará para o *string* `"String a ser copiado"`. Note que o primeiro argumento deve ter espaço suficiente para conter o *string* a ser copiado.

A função **strcpy**() retorna o primeiro argumento (no último exemplo, esta função retornaria `destino`).

3.7.3.4 Concatenação de Strings

A função **strcat()** tem dois *strings* como argumentos e serve para concatenar uma cópia do segundo *string* ao final do primeiro. O caractere terminal do primeiro *string* é substituído pelo primeiro caractere do segundo *string*. Para utilizar a função **strcat()**, inclua o arquivo de cabeçalho `string.h`.

Exemplo de uso de **strcat()**:

```
char *str[20] = "Bom ";
const char *ptr = "dia";

strcat(str, ptr);
```

Ao final desta chamada, `str` apontará para o *string* `"Bom dia"`. Note (novamente) que o primeiro argumento deve ter espaço suficiente para conter o resultado da concatenação.

A função **strcat**() retorna o primeiro argumento (no último exemplo, esta função retornaria `str`).

Uma forma de concatenar *strings* constantes é simplesmente colocá-los justapostos. Neste caso, o número de espaços em branco entre os *strings* não faz diferença. Por exemplo, a instrução:

```
printf("primeiro string.."  "segundo string.."  "terceiro string");
```

é tratada pelo compilador C como se tivesse sido escrita:

```
printf("primeiro string..segundo string..terceiro string");
```

3.7.3.5 Casamento de Strings

A função **strstr()** recebe dois *strings* como entrada e retorna um ponteiro para a posição da primeira ocorrência do segundo *string* no primeiro. Se nenhum ***casamento*** for encontrado, esta função retorna **`NULL`**. Por exemplo:

```
char *ptr1 = "bom dia";
char *ptr2 = "dia";
char *posicao;

posicao = strstr(ptr1, ptr2); /* posicao passa a apontar para a */
                              /* posição aonde se encontra d em */
                              /* "bom dia"                      */
```

Para utilizar a função **strstr()**, inclua o arquivo de cabeçalho `string.h`.

3.7.3.6 Comparação de Strings

A função **strcmp()** recebe dois *strings* como entrada e retorna:

- 0, se os strings são iguais
- um valor ***negativo***, se o primeiro string for ***menor*** do que o segundo
- um valor ***positivo***, se o primeiro string for ***maior*** do que o segundo

Exemplos:

```
char ar1[20] = "Maria";
char ar2[10] = "Maria";
char *str1 = "maria";

strcmp(ar1, ar2); /* Retorna 0 ("Maria" = "Maria") */
strcmp(ar1, str1); /* Retorna um valor negativo ("Maria" < "maria") */
strcmp(ar1, "MARIA"); /* Retorna um valor positivo ("Maria" > "MARIA") */
```

3.7.3.7 Outras Funções de Tratamento de Strings

Além das funções para tratamento de *strings* apresentadas acima, existem muitas outras funções com essa finalidade na biblioteca padrão de C. Estas funções são resumidas na **Tabela 1** a seguir:

FUNÇÃO	OPERAÇÃO
strncpy()	Copia uma porção de um *string* para um arranjo de caracteres
strncmp()	Compara dois *strings* até um número especificado de caracteres
strchr()	Encontra a primeira ocorrência de um caractere especificado num *string*
strcoll()	Compara dois *strings* baseada numa ordenação específica de caracteres
strcspn()	Calcula o comprimento de um *string* excluindo caracteres especificados
strerror()	Mapea um número de erro com uma mensagem de erro (*string*)
strpbrk()	Encontra a primeira ocorrência num *string* de caracteres especificados
strchr()	Encontra a última ocorrência num *string* de caracteres especificados
strspn()	Calcula o comprimento de um *string* que contém apenas os caracteres especificados
strtok()	Divide um *string* numa seqüência de partes (*tokens*)
strxfrm()	Transforma um *string* de modo que ele pode ser utilizado como argumento para **strcmp()**

Tabela 1: Outras Funções de Tratamento de Strings

Maiores detalhes sobre como utilizar cada uma das funções acima podem ser encontrados no **Apêndice C** ou utilizando ajuda on-line do ambiente de programação utilizado.

3.8 Arranjos de Ponteiros

Existem situações em programação aonde é desejável utilizar um arranjo de ponteiros. Considere, por exemplo, a seguinte declaração:

```
char   *arranjoDePonteiros[5];
```

A variável `arranjoDePonteiros` é um arranjo de ponteiros para caracteres (e não um ponteiro para um arranjo de caracteres), pois `[]` tem maior precedência do que `*`, conforme será visto mais adiante.

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

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

A função `ImprimeMes()` tem funcionamento bastante simples. A variável `mes` é um arranjo de ponteiros para **char**s, e em conformidade com sua inicialização, cada ponteiro aponta para o início de um *string*. A única coisa que pode parecer estranha à primeira vista com relação a este exemplo é que a variável `mes` foi declarada como sendo um arranjo de 13 ponteiros, ao invés de 12, como parece ser mais sensato. A razão para o acréscimo do *string* `""` como endereço do elemento inicial do arranjo é que ele torna o código mais eficiente ao custo de apenas um byte de memória, que é o espaço ocupado por este *string* (por que?), mais o espaço suficiente para armazenar um ponteiro. Em outras palavras, a razão pela qual um ponteiro extra com um valor inútil foi utilizado é que não se torna necessário subtrair 1 do índice `m` quando da impressão do nome do mês na segunda chamada de **printf()** acima. Mais precisamente, se o arranjo fosse declarado com 12 ponteiros, ao invés de 13, a segunda chamada de **printf()** deveria ser escrita como:

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

Assim, este exemplo ilustra uma prática comum em programação em C, que consiste em descartar o elemento inicial (i.e., de índice 0) de um arranjo quando os índices começam naturalmente em 1 (como a ordem dos meses do ano). A única desvantagem de fazer isto é que deve-se alocar espaço para um elemento que nunca é utilizado. Entretanto, a subtração que se economiza também tem seu preço. Operações aritméticas adicionais tornam a execução do programa mais lenta. Além disso, operações adicionais também aumentam o tamanho do código gerado pelo compilador, de modo que talvez nem haja economia de memória.

A função `ImprimeMes()` poderia, ao invés de imprimir o nome do mês, retornar este valor de modo que a função chamadora possa fazer aquilo que desejar. As únicas mudanças necessárias para produzir esta transformação seriam:

(1) Trocar o cabeçalho da função para:

```
char *ImprimeMes(unsigned  int  m);
```

(2) Substituir a primeira chamada de **printf()** por:

```
return  NULL;
```

que indica um mês inválido e

(3) Substituir a segunda chamada de **printf()** por:

```
return  mes[m];
```

Exercício: Implemente essas mudanças na função `ImprimeMes()` e escreva uma função `main()` para testá-la.

3.9 Ponteiros para Ponteiros

Ponteiro para ponteiro é uma construção utilizada com freqüência em programação mais sofisticada. Por exemplo, o *Sistema 7* da *Apple* utiliza ponteiros para ponteiros (denominados ***handles***) num esquema de gerenciamento de memória do *Macintosh* que tenta reduzir os efeitos de fragmentação de ***heap***. Uma variável-ponteiro que aponta para um ponteiro pode ser declarada precedendo-se o nome da variável com dois asteriscos. Por exemplo:

```
char  **p;
```

declara `p` como sendo um ponteiro para um ponteiro para um **char**. Para acessar o caractere apontado por `p`, é necessária uma referência dupla. Por exemplo:

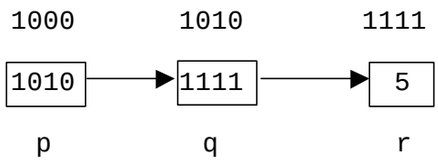
```
char c = **p;
```

atribui um valor **char** a `c`.

Agora, considere as seguintes declarações:

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

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



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

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

```
r = 10;
*q = 10;
**p = 10;
```

Ponteiros para ponteiros são usados com freqüência como parâmetros formais quando se deseja modificar o valor do próprio ponteiro (e não o valor do *conteúdo* apontado); i.e., quando o ponteiro representa um parâmetro de saída ou de entrada/saída. Por exemplo,

```
void F1(char *p)  /* O conteúdo apontado por p pode ser */
                  /* modificado, mas  o valor de p não  */
                  /* pode ser modificado pela função F1 */

void F2(char **p) /* Tanto o conteúdo apontado por p    */
                  /* quando o próprio p podem ser       */
```

```
                /* modificados pela função F2         */
```

3.10 Ponteiros para Funções

Ponteiros para funções constituem uma ferramenta poderosa e uma forma elegante de chamada de funções diferentes com base em dados de entrada. Antes de introduzir este conceito, entretanto, é necessário que se examine mais profundamente o modo como o compilador C interpreta uma chamada de função.

Não apenas dados (i.e., variáveis) são armazenados na memória de um computador durante a execução de um programa. O próprio código do programa compilado em linguagem de máquina também é armazenado em memória. Conseqüentemente, do mesmo modo que cada variável possui um endereço em memória, cada instrução (em linguagem de máquina) de um programa também possui um endereço. Quando uma função é definida, o compilador C trata o nome da função como sendo o endereço da função em memória (ou, mais precisamente, o nome da função é tratado como um ponteiro para a primeira instrução da função em linguagem de máquina). Uma chamada desta função dentro do programa implica na transferência do fluxo de execução do programa para este endereço.

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

Para declarar uma variável-ponteiro para uma função deve-se preceder o nome da variável com asterisco, como na declaração de qualquer ponteiro, mas aqui deve-se também colocar o asterisco e o ponteiro entre parênteses. A declaração termina com um par de parênteses contendo os tipos dos argumentos das funções para onde o ponteiro pode apontar[10]. Portanto, uma declaração de ponteiro para função tem a seguinte forma:

```
<tipo> (*<nome do ponteiro>)(<tipos dos argumentos>);
```

onde <*tipo*> denota o tipo de retorno das funções para as quais o ponteiro pode apontar e <*tipos dos argumentos*> representa os tipos dos argumentos destas funções.

Por exemplo:

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

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

Note ainda que, se os parênteses em torno de `*pf` na declaração do último exemplo forem omitidos, o compilador tratará a declaração como se fosse uma alusão a uma suposta função `pf()` cujo argumento é do tipo **float** e cujo tipo de retorno é um ponteiro para o tipo **int**.

3.10.1 Atribuição de Valores a um Ponteiro para Função

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

---

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

```
extern  int  F1(float);
int  (*pf)(float);

pf = F1;
```

Outras possibilidades de atribuição a `pf` são incorretas como exemplificado a seguir:

```
pf = F1(2.5);/* Ilegal, pois F1() é uma chamada de função que retorna int. */
             /* Portanto, se está tentando atribuir um valor do tipo int a */
             /* um ponteiro, o que é ilegal.                                */

pf = &F1(2.5); /* Ilegal, pois se está tentando obter o endereço do */
               /* valor retornado por F1                             */

pf = &F1;    /* Estritamente falando, é incorreto, pois se está     */
             /* tentando obter o endereço de um endereço, mas       */
             /* compiladores ANSI/ISO não irão considerar isto      */
             /* um erro. Eles simplesmente ignorarão o operador &. */
```

Outros pontos importantes que o programador deve considerar quando atribui um valor a um ponteiro para função são: (1) os tipos de retorno na declaração da função e na declaração do ponteiro para função devem ser os mesmos e (2) os tipos dos argumentos na declaração da função e na declaração do ponteiro para função devem ser os mesmos. Por exemplo, se foi declarado um ponteiro para função com argumento **float** e que retorna um valor **int**, deve-se atribuir a este ponteiro o endereço de uma função cujos tipos de argumentos e de retorno sejam respectivamente iguais a estes tipos. Por exemplo, as seguintes atribuições seriam ilegais:

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

pf = F2; /* Ilegal, pois os tipos de retorno de pf e F2 são diferentes. */
pf = F3; /* Ilegal, pois os tipos de argumentos de pf e F3 são diferentes. */
```

3.10.2 Chamada de uma Função por meio de um Ponteiro

Uma chamada de uma função através de um ponteiro ao qual se atribuiu o endereço da função é similar a uma chamada utilizando o próprio nome da função, exceto pelo fato de o nome da função ser substituído pelo nome do ponteiro precedido do operador *; então, estes são colocados entre parênteses (como na declaração do ponteiro). Por exemplo:

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

pf = F1;

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

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

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

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

Na realidade, compiladores ANSI/ISO C não requerem esta sintaxe de chamada de função através de ponteiros (embora a admitam). Num tal compilador, a chamada de função do último exemplo poderia ser feita simplesmente com:

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

Compiladores antigos, entretanto podem não aceitar este último formato de chamada. Além disso, o primeiro formato pode ser mais legível pois chama a atenção para o fato de que `pf` é um ponteiro para função, e não um nome de função.

3.10.3 Retornando Ponteiros para Funções

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

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

int  (*EscolheFuncao2(unsigned condicao))(float)
{
   if (condicao)
      return  F1;

   return  F2;
}
```

A função `EscolheFuncao()` do exemplo acima, retorna um ponteiro para a função `F1()` se o argumento `condicao` é diferente de zero, e um ponteiro para a função `F2()` em caso contrário (lembre-se que `F1` e `F2` são endereços de funções, e, portanto, são ponteiros). A função `EscolheFuncao()` poderia ser chamada conforme ilustrado a seguir:

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

O uso de definições de tipo facilita a definição de funções que retornam ponteiros para funções e favorece a legibilidade. Por exemplo, usando uma definição de tipo, a função `EscolheFuncao()` poderia ser redefinida como:

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

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

   return  F2;
}
```

3.10.4 Ponteiros para Funções como Parâmetros de Funções

Algumas vezes, é bastante útil utilizar ponteiros para funções como parâmetros de funções. Como parâmetro formal na definição de uma função, um ponteiro para função deve ser declarado exatamente da mesma forma apresentada no início desta seção. Na chamada de uma função que tem um ponteiro para função como um de seus argumentos, deve-se utilizar como parâmetro real correspondente a este argumento apenas o nome de uma função compatível com este argumento (i.e., conforme foi visto anteriormente nesta seção). Por exemplo, suponha que se tenha a seguinte definição de função num dado programa:

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

e que se tenham as seguintes declarações no programa:

```
float   x;
```

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

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

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

Então, as seguintes chamadas seriam legais:

```
FuncaoComPonteiroParaFuncao(x, F1);

FuncaoComPonteiroParaFuncao(x, F2);
```

Entretanto, a chamada:

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

seria considerada ilegal, pois a função `F3()` não é compatível com o segundo argumento (ponteiro para função) da função `FuncaoComPonteiroParaFuncao()`.

A **Seção 3.12** apresenta um exemplo prático de uso de ponteiros para funções como parâmetros de funções.

3.11 Construções Complexas

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

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

Esta declaração define a variável `x` como sendo um ponteiro para uma função que retorna um ponteiro para um arranjo com 5 ponteiros para **char**s. Complicado, não é?

Uma forma de se evitarem declarações complexas como a desse exemplo é criar definições de tipos intermediários, como mostrado a seguir:

```
typedef  char  *tAPC[5];   /* Arranjo com 5 ponteiros para chars            */
typedef  tAPC  *tFPAPC();  /* Função retornando um ponteiro para arranjo  */
                           /* com 5 ponteiros para chars.                   */
tFPAPC          *x;     /* Eqüivalente à declaração char  *(*(*x)())[5];   */
```

Na maioria das vezes, declarações difíceis de serem decifradas são compostas com os operadores de ponteiro (`*`), arranjo (`[]`) e função (`()`). Portanto, antes da apresentação de um procedimento para entendimento e composição de declarações complexas envolvendo estes operadores, é importante recordar as regras de precedência e associatividade destes operadores[11]:

- Os operadores de arranjo (`[]`) e de função (`()`) têm a mesma precedência, e esta precedência é maior do que a precedência do operador de ponteiro (`*`).

- Os operadores `[]` e `()` são associados da esquerda para a direita, enquanto que o operador `*` é associado da direita para a esquerda.

[11] Os símbolos "*", "[]" e "()" que aparecem no último exemplo não são na realidade operadores, mas sim ***declaradores***. Isto é, eles servem unicamente para determinar (declarar) o tipo da variável x. No contexto da discussão que se segue, entretanto, eles são tratados com as mesmas regras de precedência e associatividade dos operadores correspondentes.

3.11.1 Interpretando Declarações Complexas

A melhor estratégia para decifrar declarações complexas é começar com o próprio nome da variável sendo declarada, e, então, acrescentar cada parte da declaração a partir do operador de maior precedência que esteja mais próximo da variável (exceto, é claro, na presença de parênteses que modifiquem a precedência dos operadores). Considere, por exemplo, a seguinte declaração:

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

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

1. `x[]` é um arranjo;
2. `*x[]` é um arranjo de ponteiros;
3. **`char`** `*x[]` é um arranjo de ponteiros para **char**s.

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

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

que poderia ser decifrada por meio dos seguintes passos:

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

Se esta última declaração tivesse sido escrita sem parênteses como:

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

ela seria interpretada como:

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

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

3.11.2 Compondo Declarações Complexas

Instruções complexas são compostas utilizando a mesma estratégia descrita acima para decifrá-las. Suponha, por exemplo, que se deseje um ponteiro para um arranjo de ponteiros para funções que retornam ponteiros para arranjos de **char**s. Uma declaração com este significado poderia ser composta utilizando os seguintes passos:

1. `(*x)` é um ponteiro;
2. `(*x)[]` é um ponteiro para um arranjo;
3. `(*(*x)[])` é um ponteiro para um arranjo de ponteiros;
4. `(*(*x)[])()` é um ponteiro para um arranjo de ponteiros para funções;
5. `(*(*(*x)[])())` é um ponteiro para um arranjo de ponteiros para funções que retornam ponteiros;
6. `(*(*(*x)[])())[]` é um ponteiro para um arranjo de ponteiros para funções que retornam ponteiros para arranjos;
7. **`char`** `(*(*(*x)[])())[]` é um ponteiro para um arranjo de ponteiros para funções que retornam ponteiros para arranjos de **char**s.

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

Em termos de legibilidade, a melhor estratégia é mesmo evitar o uso de instruções complexas, como as apresentadas aqui, por meio do uso de definições de tipos intermediários conforme foi descrito no início desta seção.

Exercício: Utilize definições intermediárias de tipos para tornar a declaração `char (*(*(*x)[])())[]` do último exemplo mais legível.

3.12 Algoritmos de Classificação

Classificar (ou ordenar) uma lista de objetos em ordem alfabética ou numérica é uma operação bastante comum em programação. Embora o processo manual de classificar objetos seja relativamente fácil, muitos algoritmos de classificação automática não são triviais. Existem vários algoritmos de classificação e aquele que será apresentado aqui é um dos mais ineficientes em termos de uso de recursos computacionais. Por outro lado, este algoritmo, denominado de *método da bolha* (***bubble sort***), é talvez o mais fácil de ser entendido[12].

A idéia que norteia o algoritmo *bubble sort* é a de comparar elementos adjacentes, a partir dos primeiros dois elementos do arranjo, e trocá-los quando o elemento de índice menor é maior do que o elemento de índice maior (ou o contrário, quando se deseja classificação em ordem decrescente, ao invés de em ordem crescente). Após comparar os dois primeiros elementos, compara-se o segundo elemento com o terceiro, então o terceiro com o quarto, e assim por diante até que o final do arranjo seja atingido. Uma ***passagem*** pelo arranjo consiste de uma seqüência de comparações entre elementos adjacentes, do início ao final do arranjo. Caso haja pelo menos uma troca entre elementos, uma nova passagem se faz necessária. O algoritmo *bubble sort* requer que sejam feitas passagens pelo arranjo até que o mesmo esteja completamente ordenado (i.e., até que se consiga uma passagem pelo arranjo sem que haja troca de elementos). A função `BubbleSort()` a seguir implementa este algoritmo para um arranjo de inteiros do tipo **int**.

```
#define  FALSE  0
#define  TRUE   1

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

   while (!ordenada){
      ordenada = TRUE; /* Supõe que a lista está ordenada */

      for (i = 0; i < tamanhoDaLista - 1; i++){
        if (lista[i] > lista[i+1]){     /* Compara elementos adjacentes */
          ordenada = FALSE;/* Pelo menos um elemento esta fora de ordem */
                           /* Troca elementos adjacentes */
          aux = lista[i];
          lista[i] = lista[i+1];
          lista[i+1] = aux;
        }  /* final do if */
      } /* final do for */
   } /* final do while */
} /* final da funcao BubbleSort */
```

O programa a seguir chama a função `BubbleSort` com um arranjo de 10 elementos inteiros:

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

[12] A biblioteca de C oferece uma função mais eficiente, denominada **qsort()**, que implementa o método de classificação conhecido como ***quick sort***.

```
    for (i=0; i<10; i++) {
        printf(" %d ", arranjoDeInts[i]);
    }

    BubbleSort(arranjoDeInts, sizeof(arranjoDeInts)/sizeof(arranjoDeInts[0]));

    printf("\n\n\n");
    for (i=0; i<10; i++) {
        printf(" %d ", arranjoDeInts[i]);
    }

    return 0;
}
```

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

Exercício: Acrescente esse laço adicional **for** na função `BubbleSort()`, implemente-a juntamente com o programa do último exemplo e examine cuidadosamente as posições dos elementos do arranjo a cada passagem do algoritmo.

Generalizando BubbleSort

Freqüentemente, ponteiros para funções são utilizados como um mecanismo para executar várias operações similares sem a necessidade de duplicação de código. Suponha que você deseje classificar um arranjo de **int**s em ambas as ordens crescente e decrescente. Obviamente, uma forma de se fazer isto seria escrever duas funções de classificação: uma para classificar em ordem crescente (como a função `BubbleSort()` apresentada acima), e outra para classificar em ordem decrescente.

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

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

Portanto, uma função para classificar em ordem decrescente seria quase igual à função `BubbleSort()` apresentada acima. A única diferença seria a substituição da comparação:

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

por:

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

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

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

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

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

e

```
int   ComparaDecrescente(int   a, int  b)
```

```
{
    return (a < b);
}
```

Agora, utilizando ponteiros para funções, o problema poderá ser finalmente resolvido. Mais especificamente, precisa-se tornar `Compara()` um ponteiro para uma função capaz de apontar para `ComparaCrescente()` ou para `ComparaDecrescente()`, de acordo com a classificação desejada. Portanto, um novo argumento será acrescentado à lista de parâmetros de `BubbleSort()` que indicará se a classificação será em ordem crescente ou decrescente. A função `BubbleSort()` generalizada desta maneira pode então ser rescrita como apresentado a seguir:

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

   while (!ordenada){
      ordenada = TRUE;     /* Supõe que a lista está ordenada */

       for (i = 0; i < tamanhoDaLista - 1; i++){
        if ((*Compara)(lista[i], lista[i+1])){/* Compara elementos adjacentes*/
          ordenada = FALSE; /* Pelo menos um elemento esta fora de ordem */

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

O programa a seguir chama a função `BubbleSort` generalizada com um arranjo de 10 elementos inteiros para classificá-lo nas ordens crescente e decrescente:

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

   printf("Ordem original:\n");
   for (i=0; i<10; i++) {
     printf(" %d ", arranjoDeInts[i]);
   }

   BubbleSort(arranjoDeInts, sizeof(arranjoDeInts)/sizeof(arranjoDeInts[0]),
               ComparaCrescente);

   printf("\n\nOrdem crescente:\n");
   for (i=0; i<10; i++) {
     printf(" %d ", arranjoDeInts[i]);
   }

   BubbleSort(arranjoDeInts, sizeof(arranjoDeInts)/sizeof(arranjoDeInts[0]),
               ComparaDecrescente);

   printf("\n\nOrdem decrescente:\n");
   for (i=0; i<10; i++) {
     printf(" %d ", arranjoDeInts[i]);
   }

   return 0;
}
```

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

3.13 A Função main()

A função **main()**, que já foi apresentada informalmente no capítulo precedente, é uma função com certas características especiais. A presença desta função num programa em C é obrigatória e ela é sempre a primeira função a ser executada no programa. Também, quando ocorre o retorno desta função, o programa estará encerrado. Outro fato interessante sobre a função **main()** é que ela pode receber dois argumentos do sistema operacional no qual o programa é executado. Estes argumentos estão associados com valores que acompanham a chamada do programa e são denominados **parâmetros de linha de comando**.

O primeiro argumento recebido pela função **main()** quando da execução do programa é do tipo **int** e usualmente denominado **argc**. Este argumento representa o número de parâmetros que presentes na linha de comando do sistema operacional quando o programa foi invocado. O segundo argumento fornecido pelo sistema operacional, usualmente denominado **argv**, consiste de um arranjo de ponteiros para os argumentos presentes na linha de comando. Mais precisamente, **argv** é um arranjo de ponteiros para os *strings* presentes na linha de comando; cada *string* neste arranjo representa um parâmetro de linha de comando passado para a função **main()**. Portanto, a definição de uma função **main()** contendo estes dois argumentos poderia ser representada pelo seguinte esquema:

```
main(int  argc, char  *argv[])
{
      .
      :
}
```

Muitas vezes, a execução de um programa é iniciada por meio da especificação do nome do programa na linha de comandos do sistema operacional (notadamente, em sistemas como **DOS** e **UNIX**). Sendo um arquivo executável, o nome do programa é interpretado como um **comando** (ou uma **extensão**) do sistema operacional. Para que parâmetros de linha de comando possam ser passados para o programa, estes devem seguir o nome do programa na linha de comando, de acordo com o seguinte formato:

<nome do programa> <parâmetro 1> <parâmetro 2> ... <parâmetro N>

Cada *string* na chamada de programa acima deve ser separado do outro por meio de um ou mais espaços em branco.

O nome do programa será armazenado como o primeiro item no arranjo `argv[]`, e em seguida, os parâmetros de linha de comando serão armazenados neste arranjo. O valor de `argc` será automaticamente atribuído como sendo o número total de elementos no arranjo `argv[]`. Por exemplo, se houver três parâmetros na linha de comando, `argv[]` terá quatro elementos (os três parâmetros mais o nome do programa), e `argc` assumirá automaticamente o valor 4.

Como exemplo mais concreto, considere o seguinte programa:

```
#include <stdio.h>

int main(int  argc, char  *argv[])
{
    int  i;

    printf("argc  = %d\n",  argc);

    for(i = 0;  i < argc; i++)
        printf("argv[%d]  = %s\n",  i,  argv[i]);
}
```

Quando executado, o programa acima apresenta no meio de saída padrão o valor de `argc` e os *strings* armazenados em `argv`. Por exemplo, suponha que o nome deste programa após compilado é `exemplo.exe`. Então, como resultado da seguinte chamada:

```
exemplo  azul preto branco
```

o programa produziria como saída:

```
argc = 4
argv[0] = exemplo.exe
argv[1] = azul
argv[2] = preto
argv[3] = branco
```

Note que o nome completo do programa[13] (e não apenas o nome abreviado utilizado na chamada) é introduzido como parâmetro para o programa.

3.14 Exercícios de Revisão

1. Qual é a vantagem que se obtém ao se declarar o tamanho de um arranjo em termos de uma constante simbólica ao invés de em termos do valor da constante em si?

2. (a) Como deve ser escrita a inicialização de um arranjo unidimensional? (b) É obrigatória a inicialização de todos os componentes do arranjo?

3. (a) Como deve ser escrito um parâmetro real que é um arranjo unidimensional numa chamada de função? (b) Como deve ser escrito o parâmetro formal correspondente na declaração da função?

4. Como um nome de arranjo passado como argumento para uma função é interpretado?

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

6. O tipo de retorno de uma função pode ser um arranjo?

7. Quando se atribui valores a um arranjo multidimensional, qual é a vantagem de se incluírem chaves em torno de grupos de valores?

8. Quando um arranjo multidimensional é utilizado como parâmetro de uma função, como o mesmo deve ser declarado?

9. Em que situações é aconselhável utilizar um ponteiro como argumento para uma função?

10. Explique a relação existente entre o nome de um arranjo unidimensional e um ponteiro.

11. (a) Descreva duas formas diferentes de se especificar o endereço de um elemento de um arranjo unidimensional. (b) Descreva duas formas diferentes de se acessar o valor de um elemento de um arranjo unidimensional.

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

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

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

---

[13]Este nome também poderia incluir, dependendo do sistema operacional utilizado, o caminho completo até o diretório aonde se encontra o programa executável.

15. Escreva uma declaração apropriada para cada uma das seguintes situações:
(a) Uma função que recebe outra função como parâmetro e retorna um ponteiro para um caractere. A função que é parâmetro recebe um parâmetro do tipo **int** e retorna um valor do tipo **long**.
(b) Um ponteiro para uma função que recebe três argumentos do tipo **int** e retorna um valor do tipo **float**.
(c) Um ponteiro para uma função que recebe três ponteiros para o tipo **int** como argumentos e retorna um ponteiro para um valor do tipo float.

16. Um programa em C contém a seguinte declaração:

```
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)`?

17. Um programa em C contém a seguinte declaração:

```
static  float  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`?

18. Dadas as seguintes declarações e atribuições:

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

p = &ar[2];
```

quais os resultados das avaliações das seguintes expressões:
a) `*(p + 1);`
b) `p[-1];`
c) `(ar - p);`
d) `ar[*p++];`
e) `*(ar + ar[2])`

19. O que está 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]);
```

20. Dadas as seguintes declarações e atribuições:

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

quais são os resultados das avaliações das seguintes expressões:

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

21. Quais das seguintes expressões são eqüivalentes a `a[j][k]`?
a) `*(a[j] + k);`
b) `**(a[j+k);`
c) `(*(a + j))[k];`
d) `(*(a + k))[j];`
e) `*((*(a + j)) + k);`
f) `**(a + j) + k;`
g) `*(&a[0][0] + j + k)`

22. Interprete as seguintes declarações e diga quais são legais e quais são ilegais:
a) `**p;`
b) `(*pa)[];`
c) `*ap[];`
d) `(*pf)();`
e) `af[]();`
f) `fa()[];`
g) `ff()();`
h) `(**ppa)[];`
i) `(**ppf)();`
j) `*(*pap)[];`
k) `(*ppa)[][];`
l) `(*paf)[]();`
m) `*(*pfp)();`
n) `(*pfa)()[];`
o) `(*pff)()();`
p) `**app[];`
q) `(*apa[])[];`
r) `(*apf[])();`
s) `*aap[][];`
t) `aaf[][]();`
u) `*afp[]();`
v) `afa[]()[];`
x) `(*fpa())[];`
y) `(*fpf())();`
w) `*fap()[];`
z) `faf()[]()`.

3.15 Exercícios de Programação

EP3.1) 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 em dois arranjos vinculados, um dos quais contém as alturas, e o outro contém o sexo 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) Determine 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) O número total de indivíduos de cada sexo.

EP3.2) 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 questionário consistia de apenas duas perguntas: (i) o sexo do entrevistado (`'M'`/`'F'`) e (ii) sua opinião sobre o produto (*gostou/não gostou*). Suponha que um número máximo de 200 pessoas poderiam ser entrevistadas. Escreva um programa em C que:

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

(b) Determine a percentagem de pessoas do sexo feminino que responderam que gostaram do produto.

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

EP3.3) Escreva funções em C que implementem as seguintes operações sobre *strings*:
(a) Calcule o comprimento de um *string*;
(b) Copie um *string* em outro *string*;
(c) Concatene dois *strings* resultando num terceiro *string*;
(d) Verifique se um dado *string* ocorre dentro de um outro *string*. Se este for o caso, a função deve retornar a posição no segundo *string* onde a ocorrência se verifica; caso contrário, esta função deve retornar -1.

(**OBSERVAÇÃO**: Todas as operações deste exercício são implementadas em funções da biblioteca padrão de C. Obviamente, não será instrutivo se você utilizar essas funções neste exercício.)

EP3.4) Modifique a função `BubbleSort()` apresentada neste capítulo de modo que, ao invés de rearranjar os elementos de um arranjo, ela armazene a ordem correta de classificação num novo arranjo denominado `listaOrdenada[]`. Escreva um programa para testar esta nova versão de `BubbleSort()`.

EP3.5) Escreva uma função que calcule o comprimento de um *string* que utilize apenas ponteiros e operadores de incremento para torná-la tão eficiente quanto possível.

EP3.6) Escreva uma função ***recursiva*** que calcule o comprimento de um *string*. Compare esta função com aquelas escritas nos exercício **EP3.3**, letra (a), e **EP3.5** em termos de eficiência (espaço ocupado em memória, rapidez) e legibilidade.