# 4

# Matrizes e Strings

Uma *matriz* é uma coleção de variáveis do mesmo tipo que é referenciada por um nome comum. Um elemento específico em uma matriz é acessado por meio de um índice. Em C, todas as matrizes consistem em posições contíguas na memória. O endereço mais baixo corresponde ao primeiro elemento e o mais alto, ao último elemento. Matrizes podem ter de uma a várias dimensões. A matriz mais comum em C é a de string, que é simplesmente uma matriz de caracteres terminada por um nulo. Essa abordagem a strings dá a C maior poder e eficiência que às outras linguagens.

Em C, matrizes e ponteiros estão intimamente relacionados; uma discussão sobre um deles normalmente refere-se ao outro. Este capítulo focaliza matrizes, enquanto o Capítulo 5 examina mais profundamente os ponteiros. Você deve ler ambos para entender completamente essas construções importantes de C.

## Matrizes Unidimensionais

A forma geral para declarar uma matriz unidimensional é

*tipo nome_var[tamanho];*

Como outras variáveis, as matrizes devem ser explicitamente declaradas para que o compilador possa alocar espaço para elas na memória. Aqui, *tipo* declara o tipo de base da matriz, que é o tipo de cada elemento da matriz; *tamanho* define quantos elementos a matriz irá guardar. Por exemplo, para declarar um matriz de 100 elementos, chamada **balance**, e do tipo **double**, use este comando:

```
double balance[100];
```

Em C, toda matriz tem 0 como o índice do seu primeiro elemento. Portanto, quando você escreve

```
char p[10];
```

você está declarando uma matriz de caracteres que tem dez elementos, **p[0]** até **p[9]**. Por exemplo, o seguinte programa carrega uma matriz inteira com os números de 0 a 99:

```
void main(void)
{
  int x[100];  /* isto reserva 100 elementos inteiros */
  int t;

  for(t=0; t<100; ++t) x[t] = t;
}
```

A quantidade de armazenamento necessário para guardar uma matriz está diretamente relacionada com seu tamanho e seu tipo. Para uma matriz unidimensional, o tamanho total em bytes é calculado como mostrado aqui:

total em bytes = sizeof(tipo) * tamanho da matriz

C não tem verificação de limites em matrizes. Você poderia ultrapassar o fim de uma matriz e escrever nos dados de alguma outra variável ou mesmo no código do programa. Como programador, é seu trabalho prover verificação dos limites onde for necessário. Por exemplo, este código compilará sem erros, mas é incorreto, porque o laço **for** fará com que a matriz **count** ultrapasse seus limites.

```
int count[10], i;

/* isto faz com que count seja ultrapassada */
for(i=0; i<100; i++) count[i]= i;
```

Matrizes unidimensionais são, essencialmente, listas de informações do mesmo tipo, que são armazenadas em posições contíguas da memória em uma ordem de índice. Por exemplo, a Figura 4.1 mostra como a matriz **a** apareceria na memória se ela começasse na posição de memória 1000 e fosse declarada como mostrado aqui:

```
char a[7];
```

| Elemento | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] |
|---|---|---|---|---|---|---|---|
| Endereço | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 |

**Figura 4.1** Uma matriz de sete elementos começando na posição 1000.

## Gerando um Ponteiro para uma Matriz

Você pode gerar um ponteiro para o primeiro elemento de uma matriz simplesmente especificando o nome da matriz, sem nenhum índice. Por exemplo, dado

```
int sample[10];
```

você pode gerar um ponteiro para o primeiro elemento simplesmente usando o nome **sample**. Por exemplo, o seguinte fragmento atribui a **p** o endereço do primeiro elemento de **sample**:

```
int *p;
int sample[10];

p = sample;
```

Você também pode especificar o endereço do primeiro elemento de uma matriz usando o operador **&**. Por exemplo, **sample** e **&sample[0]** produzem os mesmos resultados. Porém, em códigos C escritos profissionalmente, você quase nunca verá algo como **&sample[0]**.

## Passando Matrizes Unidimensionais para Funções

Em C, você não pode passar uma matriz inteira como um argumento para uma função. Você pode, porém, passar um ponteiro para uma matriz para uma função, especificando o nome da matriz sem um índice. Por exemplo, o seguinte fragmento de programa passa o endereço de **i** para **func1()**:

```
void main(void)
{
  int i[10];
```

```
    func1(i);
    .
    .
    .
}
```

Se uma função recebe uma matriz unidimensional, você pode declarar o parâmetro formal em uma entre três formas: como um ponteiro, como uma matriz dimensionada ou como uma matriz não-dimensionada. Por exemplo, para receber **i**, uma função chamada **func1()** pode ser declarada como

```
void func1(int *x) /* ponteiro */

{
    .
    .
    .
}
```

ou

```
void func1(int x[10]) /* matriz dimensionada */

{
    .
    .
    .
}
```

ou finalmente como

```
void func1(int x[]) /* matriz não-dimensionada */

{
    .
    .
    .
}
```

Todos os três métodos de declaração produzem resultados idênticos, porque cada um diz ao compilador que um ponteiro inteiro vai ser recebido. A primeira declaração usa, de fato, um ponteiro. A segunda emprega a declaração de matriz padrão. Na última versão, uma versão modificada de uma declaração de matriz simplesmente especifica que uma matriz do tipo **int**, de algum tamanho, será recebida. Como você pode ver, o comprimento da matriz não importa à função, porque C não realiza verificação de limites. De fato, até onde diz respeito ao compilador,

```
void func1(int x[32])
{
      .
      .
      .
}
```

também funciona, porque o compilador C gera um código que instrui **func1()** a receber um ponteiro — ele não cria realmente uma matriz de 32 elementos.

## Strings

O uso mais comum de matrizes unidimensionais é como string de caracteres. Lembre-se de que, em C, uma string é definida como uma matriz de caracteres que é terminada por um nulo. Um nulo é especificado como '\0' e geralmente é zero. Por essa razão, você precisa declarar matrizes de caracteres como sendo um caractere mais longo que a maior string que elas devem guardar. Por exemplo, para declarar uma matriz **str** que guarda uma string de 10 caracteres, você escreveria

```
char str[11];
```

Isso reserva espaço para o nulo no final da string.

Embora C não tenha o tipo de dado string, ela permite constantes string. Uma *constante string* é uma lista de caracteres entre aspas. Por exemplo,

"alo aqui"

Você não precisa adicionar o nulo no final das constantes string manualmente — o compilador C faz isso por você automaticamente.

C suporta uma ampla gama de funções de manipulação de strings. As mais comuns são:

| Nome | Função |
|---|---|
| strcpy(s1, s2) | Copia s2 em s1. |
| strcat(s1, s2) | Concatena s2 ao final de s1. |
| strlen(s1) | Retorna o tamanho de s1. |
| strcmp(s1, s2) | Retorna 0 se s1 e s2 são iguais; menor que 0 se s1<s2; maior que 0 se s1>s2. |
| strchr(s1, ch) | Retorna um ponteiro para a primeira ocorrência de ch em s1. |
| strstr(s1, s2) | Retorna um ponteiro para a primeira ocorrência de s2 em s1. |

Essas funções usam o cabeçalho padrão **STRING.H**. (Essas e outras funções de string são discutidas em detalhes na Parte 2.) O seguinte programa ilustra o uso dessas funções de string:

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

void main(void)
{
  char s1[80], s2[80];

  gets(s1);

  gets(s2);

  printf("comprimentos: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf ("As strings são iguais\n");
  strcat(s1, s2);
  printf("%s\n", s1);

  strcpy(s1, "Isso é um teste.\n");
  printf(s1);
  if(strchr("alo", 'o')) printf("o está em alo\n");
  if(strstr("ola aqui", "ola")) printf("ola encontrado");
}
```

Se você rodar esse programa e digitar as strings "**alo**" e "**alo**", a saída será

```
comprimentos: 33
As string são iguais
aloalo
Isso é um teste.
o está em alo
ola encontrado
```

Lembre-se de que **strcmp()** retorna falso se as strings são iguais. Assegure-se de usar o operador ! para reverter a condição, como mostrado, se você estiver testando igualdade.

## Matrizes Bidimensionais

C suporta matrizes multidimensionais. A forma mais simples de matriz multidimensional é a matriz bidimensional — uma matriz de matrizes unidimensionais. Para declarar uma matriz bidimensional de inteiros **d** de tamanho 10,20, você escreveria

```
int d[10][20];
```

Preste bastante atenção à declaração. Muitas linguagens de computador usam vírgulas para separar as dimensões da matriz; C, em contraste, coloca cada dimensão no seu próprio conjunto de colchetes.

Similarmente, para acessar o ponto 1,2 da matriz **d**, você usaria

```
d[1][2];
```

O seguinte exemplo carrega uma matriz bidimensional com os números de 1 a 12 e escreve-os linha por linha.

```
#include <stdio.h>

void main(void)
{
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

/* agora escreva-os */
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }
}
```

Neste exemplo, **num[0][0]** tem o valor 1, **num[0][1]**, o valor 2, **num[0][2]**, o valor 3 e assim por diante. O valor de **num[2][3]** será 12. Você pode visualizar a matriz **num** como mostrada aqui:

num [t] [i]

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 2 | 9 | 10 | 11 | 12 |

Matrizes bidimensionais são armazenadas em uma matriz linha-coluna, onde o primeiro índice indica a linha e o segundo, a coluna. Isso significa que o índice mais à direita varia mais rapidamente do que o mais à esquerda quando acessamos os elementos da matriz na ordem em que eles estão realmente armazenados na memória. Veja a Figura 4.2 para uma representação gráfica de uma matriz bidimensional na memória. Você pode pensar no primeiro índice como um "ponteiro" para a linha correta.

No caso de uma matriz bidimensional, a seguinte fórmula fornece o número de bytes de memória necessários para armazená-la:

bytes = tamanho do 1º índice * tamanho do 2º índice * sizeof(tipo base)

Portanto, assumindo inteiros de dois bytes, uma matriz de inteiros com dimensões 10,5 teria

10 * 5 * 2

ou 100 bytes alocados.

Quando uma matriz bidimensional é usada como um argumento para uma função, apenas um ponteiro para o primeiro elemento é realmente passado. Porém, uma função que recebe uma matriz bidimensional como um parâmetro deve definir pelo menos o comprimento da segunda dimensão. Isso ocorre porque o compilador C precisa saber o comprimento de cada linha para indexar a matriz corretamente. Por exemplo, uma função que recebe uma matriz bidimensional de inteiros com dimensões 10,10 é declarada desta forma:

```
void func1(int x[][10])
{
     .
     .
     .
}
```

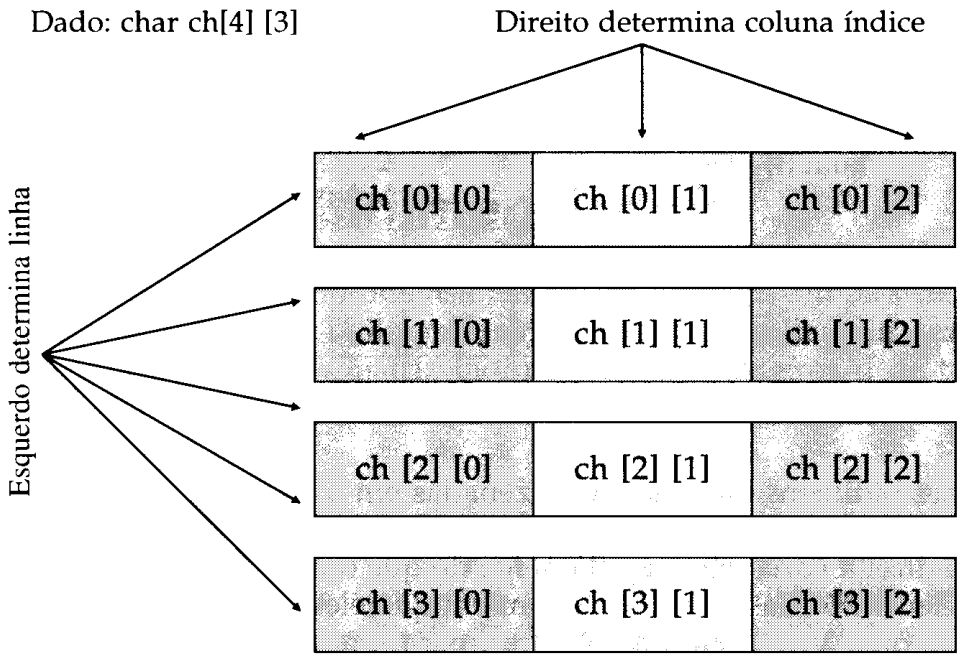


**Figura 4.2** Uma matriz bidimensional na memória.

Você pode especificar a primeira dimensão, se quiser, mas não é necessário. O compilador C precisa saber a segunda dimensão para trabalhar em sentenças como

```
x[2][4]
```

dentro da função. Se o comprimento das linhas não é conhecido, o compilador não pode determinar onde a terceira linha começa.

O seguinte programa usa uma matriz bidimensional para armazenar as notas numéricas de cada aluno de uma sala de aula. O programa assume que o professor tem três turmas e um máximo de 30 alunos por turma. Note a maneira como a matriz **grade** é acessada em cada uma das funções.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

/* Um banco de dados simples para notas de alunos. */

#define CLASSES  3
```

```
#define GRADES  30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

void main(void)
{
  char ch, str[80];

  for(;;) {
    do {
      printf("(D)igitar notas\n");
      printf("(M)ostrar notas\n");
      printf("(S)air\n");
      gets(str);
      ch = toupper(*str);
    } while(ch!='D' && ch!='M' && ch!='S');

    switch(ch) {
      case 'D':
        enter_grades();
        break;
      case 'M':
        disp_grades(grade);
        break;
      case 'S':
        exit(0);
    }
  }
}

/* Digita a nota dos alunos. */
void enter_grades(void)
{
  int t, i;

  for(t=0; t<CLASSES; t++) {
    printf("Turma # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade [t][i]= get_grade(i);
  }
}
```

```
/* Lê uma nota. */
get_grade(int num)
{
  char s[80];

  printf("Digite a nota do aluno # %d:\n",num+1);
  gets(s);
  return(atoi(s));
}

/* Mostra as notas. */
void disp_grades(int g[][GRADES])
{
  int t, i;

  for(t=0; t<CLASSES; ++t) {
    printf("Turma # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("Aluno #%d é %d\n", i+1, g[t][i]);
  }
}
```

## Matrizes de Strings

Não é incomum, em programação, usar uma matriz de strings. Por exemplo, o processador de entrada de um banco de dados pode verificar os comandos do usuário com base em uma matriz de comandos válidos. Para criar uma matriz de strings, use uma matriz bidimensional de caracteres. O tamanho do índice esquerdo indica o número de strings e o tamanho do índice do lado direito especifica o comprimento máximo de cada string. O código seguinte declara uma matriz de 30 strings, cada qual com um comprimento máximo de 79 caracteres:

```
char str_array[30][80];
```

É fácil acessar uma string individual: você simplesmente especifica apenas o índice esquerdo. Por exemplo, o seguinte comando chama **gets()** com a terceira string em **str_array**.

```
gets(str_array[2]);
```

O comando anterior é funcionalmente equivalente a

```
gets(&str_array[2][0]);
```

mas a primeira das duas formas é muito mais comum em códigos C escritos profissionalmente.

Para entender melhor como matrizes de string funcionam, estude o programa seguinte, que usa uma matriz de string como base para um editor de texto muito simples:

```
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

/* Um editor de texto muito simples. */
void main(void)


{
  register int t, i, j;

  printf("Digite uma linha vazia para sair.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* sai com linha em branco */


}

  for(i=0; i<t; i++) {
   for(j=0; text[i][j]; j++) putchar(text[i][j]);
   putchar('\n');
  }
}
```

Este programa recebe linhas de texto até que uma linha em branco seja inserida. Então, ele mostra novamente cada linha, um caractere por vez.

# Matrizes Multidimensionais

C permite matrizes com mais de duas dimensões. O limite exato, se existe, é determinado por seu compilador. A forma geral da declaração de uma matriz multidimensional é

*tipo nome*[*Tamanho1*][*Tamanho2*][*Tamanho3*]...[*TamanhoN*];

Matrizes de três ou mais dimensões não são freqüentemente usadas devido à quantidade de memória de que elas necessitam. Por exemplo, uma matriz de quatro dimensões do tipo caractere e com tamanhos 10,6,9,4 requer

10 * 6 * 9 * 4

ou 2.160 bytes. Se a matriz guardasse inteiros de 2 bytes, 4.320 bytes seriam necessários. Se a matriz guardasse **double** (assumindo 8 bytes por **double**), 17.280 bytes seriam necessários. O armazenamento necessário cresce exponencialmente com o número de dimensões. Grandes matrizes multidimensionais são geralmente alocadas dinamicamente, uma parte por vez, com as funções de alocação dinâmica de C e ponteiros. Essa abordagem é chamada de *matriz esparsa* e é discutida no Capítulo 21.

Em matrizes multidimensionais, toma-se tempo do computador para calcular cada índice. Isso significa que acessar um elemento em uma matriz multidimensional é mais lento do que acessar um elemento em uma matriz unidimensional.

Quando passar matrizes multidimensionais para funções, você deve declarar todas menos a primeira dimensão. Por exemplo, se você declarar a matriz **m** como

```
int m[4][3][6][5];
```

uma função, **func1()**, que recebe **m**, se pareceria com isto:

```
void func1(int d[][3][6][5])
{
  .
  .
  .
}
```

Obviamente, você pode incluir a primeira dimensão, se quiser.

# Indexando Ponteiros

Em C, ponteiros e matrizes estão intimamente relacionados. Como você sabe, um nome de matriz sem um índice é um ponteiro para o primeiro elemento da matriz. Por exemplo, considere a seguinte matriz.

```
char p[10];
```

As seguintes sentenças são iguais:

```
p
&p[0]
```

Colocando de outra forma,

```
p == &p[0]
```

é avaliado como verdadeiro, porque o endereço do primeiro elemento de uma matriz é o mesmo que o da matriz.

Reciprocamente, qualquer ponteiro pode ser indexado como se uma matriz fosse declarada. Por exemplo, considere este fragmento de programa:

```
int *p, i[10];
p = i;
p[5] = 100; /* atribui usando o índice */
*(p+5) = 100; /*atribui usando aritmética de ponteiros */
```

Os dois comandos de atribuição colocam o valor 100 no sexto elemento de **i**. O primeiro elemento indexa **p**; o segundo usa aritmética de ponteiro. De qualquer forma, o resultado é o mesmo. (O Capítulo 5 discute ponteiros e aritmética de ponteiros.)

Esse processo também pode ser aplicado a matrizes de duas ou mais dimensões. Por exemplo, assumindo que **a** é uma matriz de inteiros 10 por 10, estas duas sentenças são equivalentes:

```
a
&a[0][0]
```

Além disso, o elemento 0,4 de **a** pode ser referenciado de duas formas: por indexação de matriz, **a[0][4]**, ou por ponteiro, ***(a+4)**. Similarmente, o elemento 1,2 é **a[1][2]** ou ***(a+12)**. Em geral, para qualquer matriz bidimensional

a[j][k] é equivalente a *(a+(j*comprimento das linhas)+k)

Às vezes, ponteiros são usados para acessar matrizes porque a aritmética de ponteiros é geralmente mais rápida que a indexação de matrizes.

De certa forma, uma matriz bidimensional é semelhante a uma matriz de ponteiros que apontam para matrizes de linhas. Por isso, usar uma variável de ponteiro separada torna-se uma maneira fácil de utilizar ponteiros para acessar os elementos de uma matriz bidimensional. A função seguinte imprimirá o conteúdo da linha especificada da matriz global inteira **num**:

```
int num[10][10];
  .
  .
  .
void pr_row(int j)
{
  int *p, t;

  p = &num[j][0]; /* obtém o endereço do primeiro
                     elemento na linha j */

  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

Você pode generalizar essa rotina usando como argumentos de chamada a linha, o comprimento das linhas e um ponteiro para o primeiro elemento da matriz, como mostrado aqui:

```
void pr_row(int j, int row_dimension, int *p)
{
  int t;

  p = p + (j * row_dimension);

  for(t=0; t<row_dimension; ++t)
    printf("%d ", *(p+t));
}
```

Matrizes de dimensões maiores que dois podem ser reduzidas de forma semelhante. Por exemplo, uma matriz tridimensional pode ser reduzida a um ponteiro para uma matriz bidimensional, que pode ser reduzida a um ponteiro para uma

matriz unidimensional. Genericamente, um matriz *n*-dimensional pode ser reduzida a um ponteiro para uma matriz (*n*-1)-dimensional. Essa nova matriz pode ser reduzida novamente com o mesmo método. O processo termina quando uma matriz unidimensional é produzida.

## Inicialização de Matriz

C permite a inicialização de matrizes no momento da declaração. A forma geral de uma inicialização de matriz é semelhante à de outras variáveis, como mostrado aqui:

*especificador_de_tipo nome_da_matriz[tamanho1]...[tamanhoN] = {lista_valores};*

A *lista_valores* é uma lista separada por vírgulas de constantes cujo tipo é compatível com *especificador_de_tipo*. A primeira constante é colocada na primeira posição, da matriz, a segunda, na segunda posição e assim por diante. Observe o ponto-e-vírgula que segue o }.

No exemplo seguinte, uma matriz inteira de dez elementos é inicializada com os números de 1 a 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Isso significa que **i[0]** terá o valor 1 e **i[9]** terá o valor 10.

Matrizes de caracteres que contêm strings permitem uma inicialização abreviada que toma a forma:

char *nome_da_matriz[tamanho] = "string";*

Por exemplo, este fragmento de código inicializa **str** com a frase "Eu gosto de C".

```
char str[14] = "Eu gosto de C";
```

Isso é o mesmo que escrever

```
char str[14]= {'E','u',' ','g','o','s','t','o','','d','e',
               ' ','C','\0'};
```

Como todas as strings em C terminam com um nulo, você deve ter certeza de que a matriz a ser declarada é longa o bastante para incluir o nulo. Isso explica porque **str** tem comprimento de 14 caracteres, muito embora "Eu gosto de C" tenha apenas 13. Quando você usa uma constante string, o compilador automaticamente fornece o terminador nulo.

Matrizes multidimensionais são inicializadas da mesma forma que matrizes unidimensionais. Por exemplo, o código seguinte inicializa **sqrs** com os números de 1 a 10 e seus quadrados.

```
int sqrs[10][2] = {
  1,1,
  2,4,
  3,9,
  4,16,
  5,25,
  6,36,
  7,49,
  8,64,
  9,81,
  10,100
};
```

## Inicialização de Matrizes Não-Dimensionadas

Imagine que você esteja usando inicialização de matrizes para construir uma tabela de mensagens de erro, como mostrado aqui:

```
char e1[17] = "erro de leitura\n";
char e2[17] = "erro de escrita\n";
char e3[29] = "arquivo não pode ser aberto\n";
```

Como você poderia supor, é tedioso contar os caracteres em cada mensagem, manualmente, para determinar a dimensão correta da matriz. Você pode deixar C calcular automaticamente as dimensões da matriz, usando matrizes não dimensionadas. Se, em um comando de inicialização de matriz, o tamanho da matriz não é especificado, o compilador C cria uma matriz grande o bastante para conter todos os inicializadores presentes. Isso é chamado de *matriz não-dimensionada*. Usando essa abordagem, a tabela de mensagens torna-se

```
char e1[] = "Erro de leitura\n";
char e2[] = "Erro de escrita\n";
char e3[] = "Arquivo não pode ser aberto\n";
```

Dadas essas inicializações, este comando

```
printf("%s tem comprimento %d\n", e2, sizeof e2);
```

mostrará

erro de escrita tem comprimento 17

Além de ser menos tediosa, a inicialização de matrizes *não-dimensionadas* permite a você alterar qualquer mensagem sem se preocupar em usar uma matriz de dimensões incorretas.

Inicializações de matrizes não-dimensionadas não estão restritas a matrizes unidimensionais. Para matrizes multidimensionais, você deve especificar todas, exceto a dimensão mais à esquerda, para que o compilador possa indexar a matriz de forma apropriada. Desta forma, você pode construir tabelas de comprimentos variáveis e o compilador aloca automaticamente armazenamento suficiente para guardá-las. Por exemplo, a declaração de **sqrs** como uma matriz não-dimensionada é mostrada aqui:

```
int sqrs[][2] = {
  1,1,
  2,4,
  3,9,
  4,16,
  5,25,
  6,36,
  7,49,
  8,64,
  9,81,
  10,100
};
```

A vantagem dessa declaração sobre a versão que especifica o tamanho é que você pode aumentar ou diminuir a tabela sem alterar as dimensões da matriz.

## Um Exemplo com o Jogo-da-Velha

O exemplo que segue ilustra muitas das maneiras pelas quais você pode manipular matrizes em C. Matrizes multidimensionais são comumente usadas para simular as matrizes de jogos de tabuleiro. Essa seção desenvolve um programa simples de jogo da velha.

O computador joga de forma simples. Quando é a vez do computador, ela usa **get_computer_move()** para varrer a matriz, procurando por uma célula desocupada. Quando encontra uma, ele põe um **O** nesta posição. Se ele não pode encontrar uma célula vazia, ele indica um jogo empatado e termina. A função **get_player_move()** pergunta-lhe onde você quer colocar um **X**. O canto esquerdo superior é a posição 1,1; o canto direito inferior é a posição 3,3.

A matriz do tabuleiro é inicializada para conter espaços. Isso torna mais fácil apresentar a matriz na tela.

Toda vez que é feito um movimento, o programa chama a função **check()**. Essa função devolve um espaço se ainda não há vencedor, um X se você ganhou ou um O se o computador ganhou. Ela varre as linhas, as colunas e, em seguida, as diagonais, procurando uma que contenha tudo X's ou tudo O's.

A função **disp_matrix()** apresenta o estado atual do jogo. Observe como a inicialização com espaços simplifica essa função.

Todas as rotinas, neste exemplo, acessam a matriz **matrix** de forma diferente. Estude-as para ter certeza de que você compreendeu cada operação com matriz.

```
/* Um exemplo de jogo-da-velha simples. */
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3]; /* a matriz do jogo */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void));

void main(void)
{
  char done;

  printf("Este é o jogo-da-velha.\n");
  printf("Você estará jogando contra o computador.\n");

  done = ' ';
  init_matrix();
  do{
    disp_matrix();
    get_player_move();
    done = check();  /* verifica se há vencedor */
    if(done!=' ') break;  /* vencedor! */
    get_computer_move();
    done = check();  /* verifica se há vencedor */
  } while(done== ' ');
  if(done=='X') printf("Você ganhou!\n");
  else printf("Eu ganhei!!!!\n");
```

```
  disp_matrix();  /* mostra as posições finais */
}

/* Inicializa a matriz. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
   for(j=0; j<3; j++) matrix[i][j] = ' ';
}

/* Obtém a sua jogada. */
void get_player_move(void)
{
  int x, y;

  printf("Digite as coordenadas para o X: ");
  scanf("%d%d", &x, &y);

  x--; y--;

  if(matrix[x][y]!=' ') {
    printf("Posição inválida, tente novamente. \n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}

/* Obtém uma jogada do computador. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++) {
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
    if(matrix[i][j]==' ') break;
  }

  if(i*j==9) {
    printf("empate\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
```

```
}

/* Mostra a matriz na tela. */
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {
    printf(" %c ¦ %c ¦ %c ", matrix[t][0], matrix[t][1],
         matrix[t][2]);
    if(t!=2) printf("\n---¦---¦---\n");
  }
  printf("\n");
}

/* Verifica se há um vencedor. */
char check(void)
{
  int i;

  for(i=0; i<3; i++) /* verifica as linhas */
    if(matrix[i][0]==matrix[i][1] &&
       matrix[i][0]==matriz[i][2]) return matrix[i][0];

  for(i=0; i<3; i++) /* verifica as colunas */
    if(matrix[0][i]==matrix[1][i] &&
       matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* testa as diagonais */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matriz[2][2])
        return matrix[0][0];
  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matriz[2][0])
        return matrix[0][2];

  return ' ';
}
```