

8. Matrizes

W. Celes e J. L. Rangel

Já discutimos em capítulos anteriores a construção de conjuntos unidimensionais através do uso de vetores. A linguagem C também permite a construção de conjuntos bi ou multidimensionais. Neste capítulo, discutiremos em detalhe a manipulação de matrizes, representadas por conjuntos bidimensionais de valores numéricos. As construções apresentadas aqui podem ser estendidas para conjuntos de dimensões maiores.

8.1. Alocação estática versus dinâmica

Antes de tratarmos das construções de matrizes, vamos recapitular alguns conceitos apresentados com vetores. A forma mais simples de declararmos um vetor de inteiros em C é mostrada a seguir:

```
int v[10];
```

ou, se quisermos criar uma constante simbólica para a dimensão:

```
#define N 10  
int v[N];
```

Podemos dizer que, nestes casos, os vetores são declarados “estaticamente”¹. A variável que representa o vetor é uma *constante* que armazena o endereço ocupado pelo primeiro elemento do vetor. Esses vetores podem ser declarados como variáveis globais ou dentro do corpo de uma função. Se declarado dentro do corpo de uma função, o vetor existirá apenas enquanto a função estiver sendo executada, pois o espaço de memória para o vetor é reservado na pilha de execução. Portanto, não podemos fazer referência ao espaço de memória de um vetor local de uma função que já retornou.

O problema de declararmos um vetor estaticamente, seja como variável global ou local, é que precisamos saber de antemão a dimensão máxima do vetor. Usando alocação dinâmica, podemos determinar a dimensão do vetor em tempo de execução:

```
int* v;  
...  
v = (int*) malloc(n * sizeof(int));
```

Neste fragmento de código, *n* representa uma variável com a dimensão do vetor, determinada em tempo de execução (podemos, por exemplo, capturar o valor de *n* fornecido pelo usuário). Após a alocação dinâmica, acessamos os elementos do vetor da mesma forma que os elementos de vetores criados estaticamente. Outra diferença importante: com alocação dinâmica, declaramos uma variável do tipo ponteiro que posteriormente recebe o valor do endereço do primeiro elemento do vetor, alocado dinamicamente. A área de memória ocupada pelo vetor permanece válida até que seja explicitamente liberada (através da função `free`). Portanto, mesmo que um vetor seja

¹ O termo “estático” aqui refere-se ao fato de não usarmos alocação dinâmica.

criado dinamicamente dentro da função, podemos acessá-lo depois da função ser finalizada, pois a área de memória ocupada por ele permanece válida, isto é, o vetor não está alocado na pilha de execução. Usamos esta propriedade quando escrevemos a função que duplica uma cadeia de caracteres (*string*): a função *duplica* aloca um vetor de *char* dinamicamente, preenche seus valores e retorna o ponteiro, para que a função que chama possa acessar a nova cadeia de caracteres.

A linguagem C oferece ainda um mecanismo para re-alocarmos um vetor dinamicamente. Em tempo de execução, podemos verificar que a dimensão inicialmente escolhida para um vetor tornou-se insuficiente (ou excessivamente grande), necessitando um re-dimensionamento. A função *realloc* da biblioteca padrão nos permite re-alocar um vetor, preservando o conteúdo dos elementos, que permanecem válidos após a re-alocação (no fragmento de código abaixo, *m* representa a nova dimensão do vetor).

```
v = (int*) realloc(v, m*sizeof(int));
```

Vale salientar que, sempre que possível, optamos por trabalhar com vetores criados estaticamente. Eles tendem a ser mais eficientes, já que os vetores alocados dinamicamente têm uma indireção a mais (primeiro acessa-se o valor do endereço armazenado na variável ponteiro para então acessar o elemento do vetor).

8.2. Vetores bidimensionais – Matrizes

A linguagem C permite a criação de vetores bidimensionais, declarados estaticamente. Por exemplo, para declararmos uma matriz de valores reais com 4 linhas e 3 colunas, fazemos:

```
float mat[4][3];
```

Esta declaração reserva um espaço de memória necessário para armazenar os 12 elementos da matriz, que são armazenados de maneira contínua, organizados linha a linha.

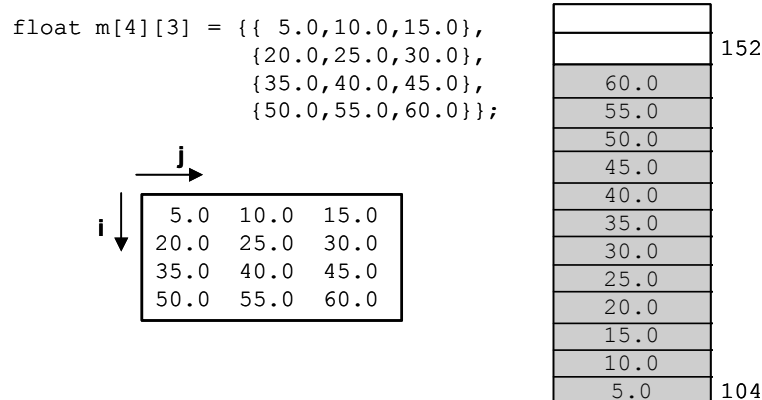


Figura 8.1: Alocação dos elementos de uma matriz.

Os elementos da matriz são acessados com indexação dupla: `mat[i][j]`. O primeiro índice, `i`, acessa a linha e o segundo, `j`, acessa a coluna. Como em C a indexação começa em zero, o elemento da primeira linha e primeira coluna é acessado por `mat[0][0]`. Após a declaração estática de uma matriz, a variável que representa a matriz, `mat` no exemplo acima, representa um ponteiro para o primeiro “vetor-linha”, composto por 3 elementos. Com isto, `mat[1]` aponta para o primeiro elemento do segundo “vetor-linha”, e assim por diante.

As matrizes também podem ser inicializadas na declaração:

```
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
```

Ou podemos inicializar seqüencialmente:

```
float mat[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

O número de elementos por linha pode ser omitido numa inicialização, mas o número de colunas deve, obrigatoriamente, ser fornecido:

```
float mat[][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Passagem de matrizes para funções

Conforme dissemos acima, uma matriz criada estaticamente é representada por um ponteiro para um “vetor-linha” com o número de elementos da linha. Quando passamos uma matriz para uma função, o parâmetro da função deve ser deste tipo. Infelizmente, a sintaxe para representar este tipo é obscura. O protótipo de uma função que recebe a matriz declarada acima seria:

```
void f (... , float (*mat)[3], ...);
```

Uma segunda opção é declarar o parâmetro como matriz, podendo omitir o número de linhas²:

```
void f (... , float mat[][3], ...);
```

De qualquer forma, o acesso aos elementos da matriz dentro da função é feito da forma usual, com indexação dupla.

Na próxima seção, examinaremos formas de trabalhar com matrizes alocadas dinamicamente. No entanto, vale salientar que recomendamos, sempre que possível, o uso de matrizes alocadas estaticamente. Em diversas aplicações, as matrizes têm dimensões fixas e não justificam a criação de estratégias para trabalhar com alocação dinâmica. Em aplicações da área de Computação Gráfica, por exemplo, é comum trabalharmos com matrizes de 4 por 4 para representar transformações geométricas e projeções. Nestes casos, é muito mais simples definirmos as matrizes estaticamente (`float mat[4][4];`), uma

² Isto também vale para vetores. Um protótipo de uma função que recebe um vetor como parâmetro pode ser dado por: `void f (... , float v[], ...);`.

vez que sabemos de antemão as dimensões a serem usadas. Nestes casos, vale a pena definirmos um tipo próprio, pois nos livramos das construções sintáticas confusas explicitadas acima. Por exemplo, podemos definir o tipo `Matrix4`.

```
typedef float Matrix4[4][4];
```

Com esta definição podemos declarar variáveis e parâmetros deste tipo:

```
Matrix4 m;                                /* declaração de variável */
...
void f (... , Matrix4 m, ...);            /* especificação de parâmetro */
```

8.3. Matrizes dinâmicas

As matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores: precisamos saber de antemão suas dimensões. O problema que encontramos é que a linguagem C só permite alocarmos dinamicamente conjuntos unidimensionais. Para trabalharmos com matrizes alocadas dinamicamente, temos que criar abstrações conceituais com vetores para representar conjuntos bidimensionais. Nesta seção, discutiremos duas estratégias distintas para representar matrizes alocadas dinamicamente.

Matriz representada por um vetor simples

Conceitualmente, podemos representar uma matriz num vetor simples. Reservamos as primeiras posições do vetor para armazenar os elementos da primeira linha, seguidos dos elementos da segunda linha, e assim por diante. Como, de fato, trabalharemos com um vetor unidimensional, temos que criar uma disciplina para acessar os elementos da matriz, representada conceitualmente. A estratégia de endereçamento para acessar os elementos é a seguinte: se quisermos acessar o que seria o elemento `mat[i][j]` de uma matriz, devemos acessar o elemento `v[i*n+j]`, onde `n` representa o número de colunas da matriz.

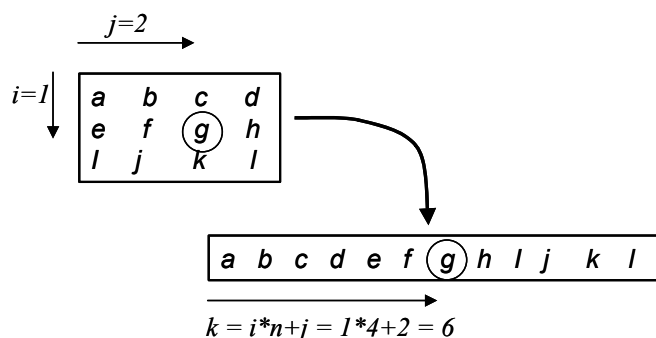


Figura 8.2: Matriz representada por vetor simples.

Esta conta de endereçamento é intuitiva: se quisermos acessar elementos da terceira ($i=2$) linha da matriz, temos que pular duas linhas de elementos ($i*n$) e depois indexar o elemento da linha com j .

Com esta estratégia, a alocação da “matriz” recai numa alocação de vetor que tem $m \times n$ elementos, onde m e n representam as dimensões da matriz.

```
float *mat;          /* matriz representada por um vetor */
...
mat = (float*) malloc(m*n*sizeof(float));
...
```

No entanto, somos obrigados a usar uma notação desconfortável, $v[i \times n + j]$, para acessar os elementos, o que pode deixar o código pouco legível.

Matriz representada por um vetor de ponteiros

Nesta segunda estratégia, faremos algo parecido com o que fizemos para tratar vetores de cadeias de caracteres, que em C são representados por conjuntos bidimensionais de caracteres. De acordo com esta estratégia, cada linha da matriz é representada por um vetor independente. A matriz é então representada por um vetor de vetores, ou vetor de ponteiros, no qual cada elemento armazena o endereço do primeiro elemento de cada linha. A figura abaixo ilustra o arranjo da memória utilizada nesta estratégia.

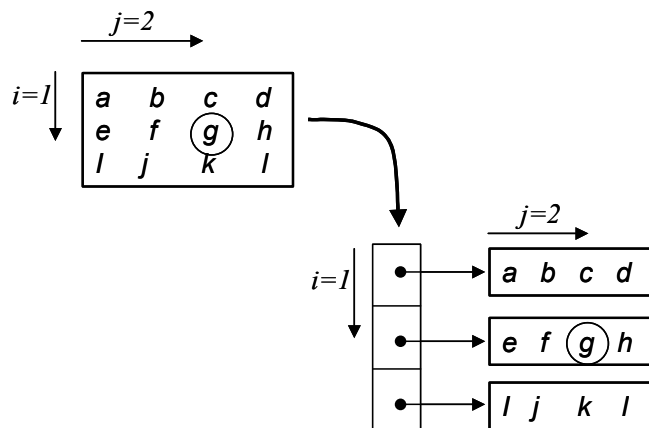


Figura 8.3: Matriz com vetor de ponteiros.

A alocação da matriz agora é mais elaborada. Primeiro, temos que alocar o vetor de ponteiros. Em seguida, alocamos cada uma das linhas da matriz, atribuindo seus endereços aos elementos do vetor de ponteiros criado. O fragmento de código abaixo ilustra esta codificação:

```
int i;
float **mat;      /* matriz representada por um vetor de ponteiros */
...
mat = (float**) malloc(m*sizeof(float*));
for (i=0; i<m; i++)
    m[i] = (float*) malloc(n*sizeof(float));
```

A grande vantagem desta estratégia é que o acesso aos elementos é feito da mesma forma que quando temos uma matriz criada estaticamente, pois, se `mat` representa uma matriz

alocada segundo esta estratégia, `mat[i]` representa o ponteiro para o primeiro elemento da linha `i`, e, conseqüentemente, `mat[i][j]` acessa o elemento da coluna `j` da linha `i`.

A liberação do espaço de memória ocupado pela matriz também exige a construção de um laço, pois temos que liberar cada linha antes de liberar o vetor de ponteiros:

```
...
for (i=0; i<m; i++)
    free(mat[i]);
free(mat);
```

8.4. Representação de matrizes

Para exemplificar o uso de matrizes dinâmicas, vamos discutir a escolha de um tipo para representar as matrizes e um conjunto de operações implementadas sobre o tipo escolhido. Podemos considerar, por exemplo, a implementação de funções básicas, sobre as quais podemos futuramente implementar funções mais complexas, tais como soma, multiplicação e inversão de matrizes.

Vamos considerar a implementação das seguintes operações básicas:

- `cria`: operação que cria uma matriz de dimensão `m` por `n`;
- `libera`: operação que libera a memória alocada para a matriz;
- `acessa`: operação que acessa o elemento da linha `i` e da coluna `j` da matriz;
- `atribui`: operação que atribui o elemento da linha `i` e da coluna `j` da matriz.

A seguir, mostraremos a implementação dessas operações usando as duas estratégias para alocar dinamicamente uma matriz, apresentadas na seção anterior.

Matriz com vetor simples

Usando a estratégia com um vetor simples, o tipo matriz pode ser representado por uma estrutura que guarda a dimensão da matriz e o vetor que armazena os elementos.

```
struct matriz {
    int lin;
    int col;
    float* v;
};

typedef struct matriz Matriz;
```

A função que cria a matriz dinamicamente deve alocar a estrutura que representa a matriz e alocar o vetor dos elementos:

```
Matriz* cria (int m, int n)
{
    Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
    mat->lin = m;
    mat->col = n;
    mat->v = (float*) malloc(m*n*sizeof(float));
}
```

```

    return mat;
}

```

Poderíamos ainda incluir na criação uma inicialização dos elementos da matriz, por exemplo atribuindo-lhes valores iguais a zero.

A função que libera a memória deve liberar o vetor de elementos e então liberar a estrutura que representa a matriz:

```

void libera (Matriz* mat)
{
    free(mat->v);
    free(mat);
}

```

A função de acesso e atribuição pode fazer um teste adicional para garantir que não haja invasão de memória. Se a aplicação que usa o módulo tentar acessar um elemento fora das dimensões da matriz, podemos reportar um erro e abortar o programa. A implementação destas funções pode ser dada por:

```

float acessa (Matriz* mat, int i, int j)
{
    int k;    /* índice do elemento no vetor */

    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    k = i*mat->col + j;
    return mat->v[k];
}

void atribui (Matriz* mat, int i, int j, float v)
{
    int k;    /* índice do elemento no vetor */

    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    k = i*mat->col + j;
    mat->v[k] = v;
}

```

Matriz com vetor de ponteiros

O módulo de implementação usando a estratégia de representar a matriz por um vetor de ponteiros é apresentado a seguir. O tipo que representa a matriz, neste caso, pode ser dado por:

```

struct matriz {
    int lin;
    int col;
    float** v;
};

```

```
typedef struct matriz Matriz;
```

As funções para criar uma nova matriz e para liberar uma matriz previamente criada podem ser dadas por:

```
Matriz* cria (int m, int n)
{
    int i;
    Matriz mat = (Matriz*) malloc(sizeof(Matriz));
    mat->lin = m;
    mat->col = n;
    mat->v = (float**) malloc(m*sizeof(float*));
    for (i=0; i<m; i++)
        mat->v[i] = (float*) malloc(n*sizeof(float));
    return mat;
}

void libera (Matriz* mat)
{
    int i;
    for (i=0; i<mat->lin; i++)
        free(mat->v[i]);
    free(mat->v);
    free(mat);
}
```

As funções para acessar e atribuir podem ser implementadas conforme ilustrado abaixo:

```
float acessa (Matriz* mat, int i, int j)
{
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    return mat->v[i][j];
}

void atribui (Matriz* mat, int i, int j, float v)
{
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    mat->v[i][j] = v;
}
```

Exercício: Escreva um programa que faça uso das operações de matriz definidas acima. Note que a estratégia de implementação não deve alterar o uso das operações.

Exercício: Implemente uma função que, dada uma matriz, crie dinamicamente a matriz transposta correspondente, fazendo uso das operações básicas discutidas acima.

Exercício: Implemente uma função que determine se uma matriz é ou não simétrica quadrada, também fazendo uso das operações básicas.

8.5. Representação de matrizes simétricas

Em uma matriz simétrica n por n , não há necessidade, no caso de $i \neq j$, de armazenar ambos os elementos `mat[i][j]` e `mat[j][i]`, porque os dois têm o mesmo valor. Portanto, basta guardar os valores dos elementos da diagonal e de metade dos elementos restantes – por exemplo, os elementos abaixo da diagonal, para os quais $i > j$. Ou seja, podemos fazer uma economia de espaço usado para alocar a matriz. Em vez de n^2 valores, podemos armazenar apenas s elementos, sendo s dado por:

$$s = n + \frac{(n^2 - n)}{2} = \frac{n(n+1)}{2}$$

Podemos também determinar s como sendo a soma de uma progressão aritmética, pois temos que armazenar um elemento da primeira linha, dois elementos da segunda, três da terceira, e assim por diante.

$$s = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

A implementação deste tipo abstrato também pode ser feita com um vetor simples ou um vetor de ponteiros. A seguir, discutimos a implementação das operações para criar uma matriz e para acessar os elementos, agora para um tipo que representa uma matriz simétrica.

Matriz simétrica com vetor simples

Usando um vetor simples para armazenar os elementos da matriz, dimensionamos o vetor com apenas s elementos. A estrutura que representa a matriz pode ser dada por:

```
struct matsim {
    int dim;           /* matriz obrigatoriamente quadrada */
    float* v;
};

typedef struct matsim MatSim;
```

Uma função para criar uma matriz simétrica pode ser dada por:

```
MatSim* cria (int n)
{
    int s = n*(n+1)/2;
    MatSim* mat = (MatSim*) malloc(sizeof(MatSim));
    mat->dim = n;
    mat->v = (float*) malloc(s*sizeof(float));
    return mat;
}
```

O acesso aos elementos da matriz deve ser feito como se estivéssemos representando a matriz inteira. Se for um acesso a um elemento acima da diagonal ($i < j$), o valor de retorno é o elemento simétrico da parte inferior, que está devidamente representado. O endereçamento de um elemento da parte inferior da matriz é feito saltando-se os elementos das linhas superiores. Assim, se desejarmos acessar um elemento da quinta linha ($i=4$),

devemos saltar $1+2+3+4$ elementos, isto é, devemos saltar $1+2+\dots+i$ elementos, ou seja, $i*(i+1)/2$ elementos. Depois, usamos o índice j para acessar a coluna.

```
float acessa (MatSim* mat, int i, int j)
{
    int k;    /* índice do elemento no vetor */

    if (i<0 || i>=mat->dim || j<0 || j>=mat->dim) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    if (i>=j)
        k = i*(i+1)/2 + j;
    else
        k = j*(j+1)/2 + i;
    return mat->v[k];
}
```

Matriz simétrica com vetor de ponteiros

A estratégia de trabalhar com vetores de ponteiros para matrizes alocadas dinamicamente é muito adequada para a representação matrizes simétricas. Numa matriz simétrica, para otimizar o uso da memória, armazenamos apenas a parte triangular inferior da matriz. Isto significa que a primeira linha será representada por um vetor de um único elemento, a segunda linha será representada por um vetor de dois elementos e assim por diante. Como o uso de um vetor de ponteiros trata as linhas como vetores independentes, a adaptação desta estratégia para matrizes simétricas fica simples.

O tipo da matriz pode ser definido por:

```
struct matsim {
    int dim;
    float** v;
};

typedef struct matsim MatSim;
```

Para criar a matriz, basta alocarmos um número variável de elementos para cada linha. O código abaixo ilustra uma possível implementação:

```
MatSim* cria (int n)
{
    int i;
    MatSim* mat = (MatSim*) malloc(sizeof(MatSim));
    mat->dim = n;
    mat->v = (float**) malloc(n*sizeof(float*));
    for (i=0; i<n; i++)
        mat->v[i] = (float*) malloc((i+1)*sizeof(float));
    return mat;
}
```

O acesso aos elementos é natural, desde que tenhamos o cuidado de não acessar elementos que não estejam explicitamente alocados (isto é, elementos com $i<j$).

```

float acessa (MatSim* mat, int i, int j)
{
    if (i<0 || i>=mat->dim || j<0 || j>=mat->dim) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    if (i>=j)
        return mat->v[i][j];
    else
        return mat->v[j][i];
}

```

Finalmente, observamos que exatamente as mesmas técnicas poderiam ser usadas para representar uma matriz “triangular”, isto é, uma matriz cujos elementos acima (ou abaixo) da diagonal são todos nulos. Neste caso, a principal diferença seria na função `acessa`, que teria como resultado o valor zero em um dos lados da diagonal, em vez acessar o valor simétrico.

Exercício: Escreva um código para representar uma matriz triangular inferior.

Exercício: Escreva um código para representar uma matriz triangular superior.