Capítulo 8: Matrizes

Waldemar Celes e Roberto Ierusalimschy

29 de Fevereiro de 2012

1 Conjuntos bidimensionais

Além de conjuntos unidimensionais de valores (vetores), muitas aplicações necessitam armazenar e manipular conjuntos de dimensões maiores. A representação de *matrizes*, por exemplo, é muito usada em diversas aplicações. Uma matriz representa um conjunto bi-dimensional de valores. Matrizes são fundamentais em áreas como a Algébrica e são frequentemente usadas na representação de tabelas. Matrizes podem também ser usadas para representar mapas discretos. Muitos jogos de computador, por exemplo, representam seus "mundos" por meio de matrizes; exemplos clássicos são jogos de tabuleiro. Conjunto com dimensões maiores (3, 4, 5, ...) também são possíveis de serem representados em programas de computador, mas neste texto vamos nos limitar a discutir a representação de matrizes. A representação de conjunto com dimensões maiores segue as mesmas regras.

Similar a variáveis simples e vetores, matrizes devem ser declaradas para que o espaço de memória apropriado seja reservado. Como matriz representa um conjunto bi-dimensional, devemos especificar as duas dimensões na declaração: o *número de linhas* e o *número de colunas*. Assim, uma declaração de uma matriz de reais pode ser feita como mostrada a seguir:

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

Este trecho de código declara uma matriz com 3 linhas e 4 colunas, e reserva um espaço de memória suficiente para armazenar 12 ($= 3 \times 4$) valores do tipo `float`. O nome da variável, `mat`, representa uma referência para o espaço de memória reservado. Para acessar um elemento da matriz, utilizamos indexação dupla: `mat[i][j]`. Para uma matriz com m linhas e n colunas, os índices usados no acesso aos elementos devem satisfazer: $0 \leq i < m$ e $0 \leq j < n$. Assim, no caso da matriz declarada no trecho de código mostrado, o elemento da primeira linha e primeira coluna é acessado por `mat[0][0]`, e o elemento da última linha e última coluna é acessado por `mat[2][3]`.

Como sabemos, a memória do computador é linear. Para que se tenha a representação de conjuntos bi-dimensionais, cria-se um artifício. Uma matriz declarada em C é armazenada na memória linha por linha. Os primeiros espaços de memória associados à matriz são reservados para os elementos da primeira linha, os espaços seguintes são associados aos elementos da segunda linha, e assim por diante. Para que o compilador identifique o espaço de memória associado a um determinado elemento `m[i][j]`, é feita uma conta de endereçamento: o elemento com índices i e j é armazenado na posição k do espaço de memória associado à matriz, onde $k = in + j$. Este é um detalhe interno, que é calculado automaticamente pelo compilador. Nos nossos programas, nos preocupamos apenas em acessar os elementos escrevendo `mat[i][j]`. O importante é observarmos que o compilador necessita saber o número de colunas da matriz, n, para fazer a conta de endereçamento. Esta informação nos será útil adiante.

2 Laços aninhados

Para acessarmos todos os elementos de um conjunto bi-dimensional, em geral usamos a construção de laços aninhados, isto é, codificamos um laço dentro de outro laço. Assim, podemos fazer um laço para percorrer as linhas de uma matriz e, para cada linha, podemos fazer um laço para percorrer os elementos da linha. Antes de apresentar exemplos que manipulam matrizes, vamos ilustrar a codificação de laços aninhados.

Como vimos, a construção de laços nos permite executar um bloco de comandos repetidas vezes. Nos exemplos anteriores, temos usado laços simples, mas podemos construir laços dentro de laços, criando laços aninhados. Como exemplo simples, vamos considerar o código a seguir:

```
#include <stdio.h>

int main (void)
{
   int i, j;
   for (i=0; i<3; i++) {
      for (j=0; j<4; j++) {
         printf("%d %d\n",i,j);
      }
   }
   return 0;
}
```

Se este código for executado, serão exibidos 12 pares de valores, pois o bloco de comandos associado ao segundo `for` será executado 3×4 vezes. Isto porque o laço controlado pela variável `i` (primeiro `for`) repete a execução de seu bloco 3 vezes. Cada vez que esse bloco for executado, todas as iterações do laço controlado pela variável `j` são processadas, pois este laço está definido dentro do bloco do laço anterior. Isto é, para cada iteração do laço externo, o laço interno executa todas as suas iterações. Como resultado, para o valor da variável `i` igual a 0, a variável `j` assume os valores 0, 1, 2 e 3. Em seguida, para `i` igual a 1, `j` assume os valores 0, 1, 2 e 3; e assim por diante. Desta forma, a saída deste programa exibe a seqüência de números mostrada a seguir, cada par de números em uma linha. Cada par é exibido em uma iteração do laço mais interno.

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

Da mesma forma, podemos codificar laços aninhados fazendo uso das outras formas de construção de laços, como `while` e `do-while`. É comum inclusive misturar as formas de construção: por exemplo, usar um `while` para percorrer as linhas de uma matriz e um `for` para percorrer os elementos de cada linha.

3 Exemplo com matriz

Para ilustrar o uso de matrizes, vamos considerar inicialmente uma aplicação que manipula as notas obtidas por alunos de uma disciplina. Vamos considerar a existência de um arquivo que apresenta, em cada linha, as três notas obtidas por cada aluno, conforme ilustrado a seguir:

7.5	8.5	7.8
8.4	9.2	6.8
9.1	10.0	9.5
4.0	5.2	4.6
5.7	3.4	4.3
4.3	6.0	5.8

Vamos considerar que nosso objetivo seja ler as notas do arquivo e armazená-las na memória do computador para que, posteriormente, seja possível processarmos as notas: calcular a média de cada aluno, a média da disciplina, verificar quantos alunos foram aprovados etc. Assumindo que não exista mais do que 50 alunos na disciplina, podemos armazenar as notas usando vetores. Para tanto, precisamos declarar três vetores, um para cada nota:

```
float p1[50];
float p2[50];
float p3[50];
```

Assim, se quisermos processar as notas, devemos acessar os elementos dos três vetores. Outra alternativa é usar apenas uma estrutura para armazenar todas as notas de todos os alunos, fazendo uso de matrizes. Precisamos declarar apenas uma matriz, com 3 colunas:

```
float notas[50][3];
```

Desta forma, as notas do i-ésimo aluno são representadas por `notas[i][0]`, `notas[i][1]` e `notas[i][2]`. A vantagem é que temos todos os dados armazenados em uma única estrutura. Tendo acesso à matriz, podemos acessar todas as notas. É fácil imaginar que em aplicações onde se tem vários valores associados a cada linha, o uso de matriz torna-se mais adequado.

Para ilustrar códigos de programas que usam matrizes, vamos considerar este exemplo de notas. À princípio, vamos escrever um programa que leia os dados do arquivo, armazenando-os em uma matriz e então calcule a média geral obtida pelos alunos na disciplina. Nosso programa inicialmente declara a matriz e então preenche os elementos da matriz com os valores existentes no arquivo "notas.txt". Para tanto, devemos usar um laço aninhado lendo as notas de cada linha do arquivo. Ao final deste procedimento, sabemos as notas de quantos alunos estão cadastradas no arquivo. Em seguida, para calcular a média na disciplina, podemos calcular o somatório de todas as notas armazenadas na matriz, novamente usando laços aninhados, e então calcular a média, exibindo-a na tela. Um implementação deste código é mostrada a seguir:

```
#include <stdio.h>

int main (void)
{
  int i, j;
  int lidos;
  int nalunos;
  float media = 0.0;
  float notas[50][3];
  FILE *f = fopen("notas.txt","r");
  /* lê valores do arquivo */
  i = 0;
  do {
    lidos = fscanf(f,"%f %f %f", &notas[i][0],&notas[i][1], &notas[i][2]);
    i++;
  } while (i < 50 && lidos == 3);
  nalunos = i-1;
```

```
  fclose(f);
  /* calcula média */
  for (i=0; i<nalunos; i++) {
    for (j=0; j<3; j++) {
      media = media + notas[i][j];
    }
  }
  media = media / (3*nalunos);
  /* exibe média calculada */
  printf("Media da disciplina: %.2f\n", media);
  return 0;
}
```

4 Passagem de matrizes para funções

Para que nosso programa fique mais estruturado, podemos re-arrumá-lo usando funções auxiliares. Podemos prever uma função para ler os valores do arquivo e armazená-los na matriz, e uma outra função para calcular e retornar a média. A função que faz a leitura dos valores pode retornar o número de alunos cujas notas estão no arquivo.

Estas funções recebem como parâmetro uma matriz. Passar uma matriz para uma função é análogo a passar um vetor. Passa-se na verdade uma referência para a matriz. Assim, a função chamada acessa (e, portanto, pode modificar) o mesmo espaço de memória que a função que tem a declaração da matriz. Uma diferença importante com relação a vetores é que o parâmetro que representa a matriz deve ter especificado o número de colunas da matriz. Isso impossibilita a codificação de uma função para manipular uma matriz de dimensão qualquer. O máximo que podemos fazer é codificar uma função que manipula uma matriz com qualquer número de linhas, mas o número de colunas deve ser especificado por uma constante no código. Isso é necessário pois o compilador precisa conhecer o número de colunas da matriz para fazer a conta de endereçamento, conforme discutido.

O código de uma função auxiliar que lê os valores do arquivo e armazena-os na matriz, retornando o número de alunos, é mostrado a seguir:

```
int le_valores (float mat[ ][3])
{
  int i;
  FILE *f = fopen("notas.txt", "r");

  /* lê valores do arquivo */
  i = 0;
  do {
    lidos = fscanf(f,"%f %f %f", &mat[i][0],&mat[i][1], &mat[i][2]);
    i++;
  } while (i < 50 && lidos == 3);
  fclose(f);

  return i - 1;
}
```

Note a declaração do parâmetro que representa uma matriz com 3 colunas, `float mat[ ][3]`. Note ainda que, como vetores, podemos alterar os valores dos elementos da matriz dentro de uma função.

A função que calcula a média dos valores armazenados na matriz é mostrada a seguir. Neste caso, a função recebe como parâmetros o número de linhas da matriz e a matriz propriamente

dita (mais uma vez, a função é limitada para trabalhar com matrizes de 3 colunas):

```
float media (int n, float mat[ ][3])
{
  int i, j;
  float soma = 0.0;
  for (i=0; i<n; i++) {
    for (j=0; j<3; j++) {
      soma = soma + mat[i][j];
    }
  }
  return soma / (3*n);
}
```

A função *main* pode então ser re-escrita da forma mostrada a seguir:

```
int main (void)
{
  int n;
  float m;
  float notas[50][3];

  n = le_valores(notas);
  m = media(n,notas);

  printf("Media da disciplina: %.2f", m);

  return 0;
}
```

5 Representação de tabelas

Muitas aplicações precisam organizar informações na forma de tabelas, e matrizes são naturalmente estruturas de dados adequadas para representação de tabelas. Para exemplificar, vamos considerar uma tabela de um campeonato de futebol. Uma tabela deste tipo é ilustrada a seguir, onde cada linha armazena as informações de um determinado time do campeonato: número de pontos ganhos (PG), número de jogos (J), número de vitórias (V), saldo de gols (SG) e gols próprios (GP).

Time	PG	J	V	SG	GP
Time 0	10	8	3	-4	12
Time 1	17	8	5	10	19
Time 2	10	8	3	-5	11
Time 3	11	8	3	-1	15
Time 4	19	8	6	13	23

A tabela ilustrada pode ser representada por uma matriz de valores inteiros de dimensão $N \times 5$, onde N representa o número de times:

10	8	3	-4	12
17	8	5	10	19
10	8	3	-5	11
11	8	3	-1	15
19	8	6	13	23

Assim, para acessar o número de pontos ganhos de um determinado time `i`, podemos escrever `t[i][0]`, onde `t` representa a variável declarada como matriz. Analogamente, o saldo de gols pode ser acessado por `t[i][3]`. Para o código ficar mais legível, podemos definir constantes simbólicas como:

```
#define PG  0
#define J   1
#define V   2
#define SG  3
#define GC  4
```

Com isso, podemos acessar o número de pontos ganhos escrevendo `t[i][PG]`, o número de jogos escrevendo `t[i][J]` e assim por diante. Na verdade, quando temos a definição de várias constantes simbólicas relacionadas, em geral optamos por definir as constantes usando uma *enumeração*:

```
enum {
   PG = 0,
   J,
   V,
   SG,
   GC
};
```

tendo o mesmo efeito prático que as definições anteriores.

Voltando à discussão de uma aplicação que manipula a tabela do campeonato, podemos por exemplo classificar os times segundo um determinado critério. Um critério usualmente usado para classificação dos times é: número de pontos ganhos, número de vitórias, saldo de gols e, finalmente, número de gols próprios. Assim, o líder do campeonato é o time que tem o maior número de pontos; se dois times tem o mesmo número de pontos, usa-se o maior número de vitórias como critério de desempate; se o número de vitórias também for igual, usa-se o saldo de gols; por fim, usa-se o número de gols próprios.

Podemos então codificar uma função que recebe como parâmetros o número de times e a matriz representando a tabela do campeonato e retorna o número do time que é líder. Uma possível codificação desta função é mostrada a seguir. A função consiste em um cálculo de máximo de um conjunto, com critérios de desempates.

```
int lider (int n, int t[][5])
{
   int l = 0;  /* assume inicialmente time 0 como líder */
   for (i = 1; i < n; i++) {
      if (t[i][PG] > t[l][PG]) {
         l = i;
      }
      else if (t[i][PG] == t[l][PG]) {
         if (t[i][V] > t[l][V]) {
            l = i;
         }
         else if (t[i][V] == t[l][V]) {
            if (t[i][SG] > t[l][SG]) {
               l = i;
            }
            else if (t[i][SG] == t[l][SG] && t[i][GP] > t[l][GP]) {
               l = i;
            }
```

```
            }
        }
    }
    return l;
}
```

Uma outra função que podemos considerar consiste em, dado o resultado de um jogo, atualizar a tabela do campeonato. A função pode receber, além da tabela em si que será atualizada, quatro parâmetros: os índices dos times que jogarão, a e b, e o número de gols de cada time na partida (placar), n_a e n_b. Uma possível implementação desta função é mostrada a seguir. O time vencedor ganha 3 pontos e o perdedor não ganha ponto algum. Se houver empate, ambos os times recebem 1 ponto cada. Além disso, devemos atualizar o número de jogos, o saldo de gols e o número de gols próprios. O saldo de gols do time a é atualizado computando $sg_a = sg_a + (na - nb)$ e do tipo b computando $sg_b = sg_b + (nb - na)$.

```
void atualiza (int n, int t[][5], int a, int b, int na, int nb)
{
    /* Atualiza pontos ganhos e número de vitórias */
    if (na > nb) {        /* time a foi o vencedor */
        t[a][PG] += 3;
        t[a][V] += 1;
    }
    else if (na < nb) {   /* time b foi vencedor */
        t[b][PG] += 3;
        t[b][V] += 1;
    }
    else {                /* houve empate */
        t[a][PG] += 1;
        t[b][PG] += 1;
    }

    /* Atualiza número de jogos, saldo de gols e gols próprios */
    t[a][J] += 1;
    t[b][J] += 1;
    t[a][SG] += na - nb;
    t[b][SG] += nb - na;
    t[a][GP] += na;
    t[b][GP] += nb;
}
```

Podemos pensar em diversas outras funções que manipulam este tipo de tabela, incluindo funções que fazem acesso a dados armazenados em arquivos. Os exercícios listados ao final do capítulo ilustram tais funções.

6 Funções algébricas

Em muitas aplicações computacionais, fazemos uso de *matrizes quadradas*, isto é, matrizes em que o número de linhas é igual ao número de colunas. Neste caso, como o número de colunas, e consequentemente o número de linhas, tem que ser pré-estabelecido, nossos códigos ficam particularizados para determinada dimensão de matriz. Para ilustrar, vamos considerar a implementação de algumas funções algébricas que trabalham com matrizes quadradas. Na implementação destas funções, vamos optar por usar o tipo real de dupla precisão. Estas funções podem ser agrupadas em um arquivo e servir como uma biblioteca de funções algébricas.

Nos códigos, vamos assumir que a dimensão das matrizes é `N x N`, onde `N` é uma constante

simbólica. Por exemplo, se quisermos que nosso código seja usado para matrizes `4 x 4`, fazemos:

```
#define N 4
```

6.1 Matriz simétrica

Como primeiro exemplo, vamos considerar uma função para verificar se uma dada matriz é simétrica. Uma matriz quadrada, M, é dita *simétrica* se $M_{ij} = M_{ji}$ para qualquer elemento da matriz. Isto é, elementos em lados opostos da diagonal principal da matriz devem ser iguais. Para concluir que uma matriz não é simétrica, basta percorrer os elementos abaixo da diagonal da matriz e verificar se o elemento simétrico (oposto, em relação à diagonal) é diferente. Uma possível implementação desta função é mostrada a seguir. A função retorna 1 se a matriz for simétrica e 0 caso contrário. Note que não é necessário verificar todos os elementos para concluir que a matriz não é simétrica.

```
int simetrica (double A[ ][N])
{
  int i, j;
  for (i=0; i<N; i++) {
    for (j=0; j<i; j++) {
      if (A[i][j] != A[j][i]) {
        return 0;
      }
    }
  }
  return 1;
}
```

Como na prática esta função também pré-estabelece um valor constante de número de linhas, muitos programadores optam por escrever o parâmetro como `double A[N][N]`, mas o primeiro `N` não é considerado na compilação. O código é limitado para matrizes com `N` linhas por causa do laço (e porque, no caso, só faz sentido falar em matriz simétrica se esta for quadrada).

6.2 Matriz transposta

Uma função que também pode ser útil calcula a transposta de uma dada matriz. Se M é uma dada matriz, sua *transposta* é definida por: $T_{ji} = M_{ij}$. No nosso exemplo, vamos limitar o código para calcular a transposta de uma matriz quadrada.

Nossa primeira função recebe duas matrizes como parâmetros. O primeiro representa a matriz de entrada e o segundo a transposta preenchida dentro da função. Uma possível implementação desta função é mostrada a seguir: percorre-se todos os elementos da matriz de entrada e preenche o elemento simétrico da matriz de saída.

```
void cria_transposta (double A[ ][N], double T[ ][N])
{
  int i, j;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      T[j][i] = A[i][j];
    }
  }
}
```

Como estamos trabalhando com matrizes quadradas, podemos também pensar em uma função que recebe uma matriz e a transforma na sua transposta, isto é, a matriz transposta resultante é armazenada no mesmo espaço de memória da matriz de entrada. Neste caso, precisamos apenas percorrer os elementos abaixo da diagonal principal da matriz, trocando seu valor com o do elemento simétrico. Uma possível implementação desta função é mostrada a seguir. Note a necessidade da variável auxiliar para efetuar a troca de valores entre dois elementos da matriz.

```
void transpoe (double A[ ][N])
{
  int i, j;
  for (i=0; i<N; i++) {
    for (j=0; j<i; j++) {
      double t = A[i][j];
      A[i][j] = A[j][i];
      A[j][i] = t;
    }
  }
}
```

6.3 Funções de multiplicação

Nesta seção, vamos mostrar a implementação de funções que efetuam operações de multiplicação envolvendo matrizes. A multiplicação de uma matriz por um escalar, $B = sA$, é simples e pode ser codificada como mostrada a seguir:

```
void mult_matriz_escalar (double A[ ][N], double s, double B[ ][N])
{
  int i, j;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      B[i][j] = s * A[i][j];
    }
  }
}
```

A multiplicação de uma matriz por um vetor, $w = Av$, também é simples, resultando em um novo vetor. Sabe-se que os elementos do vetor resultante são definidos como: $w_i = \sum_{j=0}^{N-1} A_{ij} v_j$. Como a matriz é quadrada, as dimensões dos vetores são iguais a `N`. Uma possível implementação desta função é mostrada a seguir:

```
void mult_matriz_vetor (double A[ ][N], double v[ ], double w[ ])
{
  int i, j;
  for (i=0; i<N; i++) {
    w[i] = 0.0;
    for (j=0; j<N; j++) {
      w[i] = w[i] + A[i][j] * v[j];
    }
  }
}
```

Um pouco mais elaborada é a operação que efetua a multiplicação entre duas matrizes, $C = AB$, resultando em uma outra matriz. Como estamos assumindo matrizes quadradas,

as três matrizes têm a mesma dimensão. Os elementos da matriz resultante são dados por: $C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$. Uma implementação desta função é mostrada a seguir:

```
void mult_matriz_matriz (double A[ ][N], double B[ ][N], double C[ ][N])
{
  int i, j, k;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      C[i][j] = 0.0;
      for (k=0; k<N; k++) {
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
    }
  }
}
```

Exercícios

1. Considerando o exemplo de notas de alunos armazenadas em um arquivo e carregadas para uma matriz, implemente uma função que receba a matriz com as notas dos alunos e retorne o número de alunos reprovados, isto é, alunos com média menor que 5.0. A função deve ter o cabeçalho:

   ```
   int reprovados (int n, float notas[ ][3])
   ```

   Modifique a função *main* apresentada ao final da Seção 3 para testar sua função.

2. Ainda considerando o exemplo de notas dos alunos, implemente uma função que receba a matriz de notas e preencha um vetor com a média dos alunos. Cada elemento do vetor deve armazenar a média de cada aluno. A função deve seguir o cabeçalho mostrado a seguir. Assume-se que a dimensão do vetor é igual ao número de linhas da matriz.

   ```
   void media_dos_alunos (int n, float notas[ ][3], float media[ ])
   ```

3. Escreva um programa que leia uma tabela de um campeonato de futebol, como ilustrada na Seção 5, armazenada em um arquivo denominado "tabela.txt". O programa deve então acessar o arquivo denominado "jogos.txt". Neste arquivo, cada linha reporta o resultado de uma partida; por exemplo, a linha com os números a b n_a n_b indica que o time a jogou com o time b e o placar foi $n_a \times n_b$. O programa deve então atualizar a tabela e salvá-la no arquivo "tabela.txt", sobre-escrevendo a tabela antiga. Ao final, após a atualização da tabela, o programa deve exibir na tela o número do time que lidera o campeonato.

4. Escreva uma função que verifique se uma dada matriz quadrada de dimensão `N x N`, onde `N` representa uma constante simbólica, é uma *matriz triangular inferior*. Em uma matriz triangular inferior, todos os elementos acima da diagonal principal são iguais a 0.0. Os elementos da diagonal ou abaixo da diagonal podem assumir valores quaisquer. A função deve retornar 1 se a matriz dada for triangular inferior e 0 caso contrário, e deve seguir o seguinte cabeçalho:

   ```
   int triangular_inferior (double A[ ][N])
   ```

5. Escreva uma função que verifique se uma dada matriz quadrada de dimensão `N x N`, onde `N` representa uma constante simbólica, é uma *matriz identidade*. Em uma matriz identidade, os elementos da diagonal principal são iguais a 1.0 e os demais são iguais a 0.0. A função deve retornar 1 se a matriz dada for identidade e 0 caso contrário, e deve seguir o seguinte cabeçalho:

   ```
   int identidade (double A[ ][N])
   ```

6. Escreva uma função que, dadas duas matrizes, A e B, verifique se B é a *inversa* de A, isto é, se B é igual a A^{-1}. Se B for a inversa, a multiplicação de A por B resulta em uma matriz identidade. A função deve retornar 1 se B é a inversa de A e 0 caso contrário, e deve seguir o seguinte cabeçalho:

   ```
   int inversa (double A[ ][N], double B[ ][N])
   ```