

Capítulo 7

Alocação dinâmica

Até aqui, na declaração de um vetor, foi preciso dimensioná-lo. Isso nos obrigava a saber, de antemão, quanto espaço seria necessário, isto é, tínhamos que prever o número máximo de elementos no vetor durante a codificação. Este pré-dimensionamento do vetor é um fator limitante. Por exemplo, se desenvolvemos um programa para calcular a média e a variância de um conjunto de valores, teremos que prever o número máximo de valores. Uma solução é dimensionar o vetor com um número arbitrariamente alto para não termos limitações quando da utilização do programa. No entanto, isso levaria a um desperdício de memória inaceitável em diversas aplicações. Se, por outro lado, formos modestos no pré-dimensionamento do vetor, o uso do programa fica muito limitado, pois não conseguiríamos tratar conjuntos com um número de valores maior do que o previsto.

Felizmente, a linguagem C oferece meios de requisitar espaços de memória em tempo de execução. Dizemos que podemos alocar memória dinamicamente. Com este recurso, nosso suposto programa para o cálculo da média e da variância pode, em tempo de execução, consultar o número de valores no conjunto e então fazer a alocação do vetor dinamicamente, sem desperdício de memória.

7.1 Uso da memória

Informalmente, podemos dizer que existem três maneiras de reservar espaço de memória para o armazenamento de informações. A primeira é através do uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado. A segunda maneira é pelo uso de variáveis locais. Neste caso, como já discutimos, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Por este motivo, a função que chama não pode fazer referência ao espaço local da função chamada. As variáveis globais ou locais podem ser simples ou vetores. Para os vetores, precisamos informar o número máximo de elementos, caso contrário, o compilador não saberia o tamanho do espaço a ser reservado.

A terceira maneira de reservar memória é requisitando ao sistema, em tempo de execução, um espaço de determinado tamanho. Este espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado. Por isso, podemos alocar dinamicamente um espaço de memória numa função e acessá-lo em outra. A partir do momento que liberamos o espaço, ele estará disponibilizado para outros usos e não podemos mais acessá-lo.

Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar. De qualquer forma, é boa prática de programação liberar todos os espaços alocados antes da finalização do programa. Quando não liberamos uma memória alocada, dizemos que houve “vazamento de memória” (*memory leak*). Em programas que manipulam grandes volumes de informações, esse vazamento pode ser um problema sério, pois a memória do sistema pode se esgotar desnecessariamente (grande parte da memória pode não mais estar em uso pelo programa).

Na Figura 7.1, apresentamos um esquema didático que ilustra a distribuição do uso da memória pelo sistema operacional. Quando requisitamos ao sistema operacional para executar um programa, o código em linguagem de máquina do programa deve ser carregado na memória, conforme discutido no Capítulo 1. O sistema operacional reserva também os espaços necessários para armazenar as variáveis globais (e estáticas) existentes no programa. O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Cada vez que uma função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Este espaço pertence à pilha de execução e, quando a função termina, é desempilhado. A parte da memória não ocupada pela pilha de execução pode ser requisitada dinamicamente. Se a pilha tentar crescer mais do que o espaço disponível existente, dizemos que ela “estourou” (*stack overflow*) e o programa é abortado com erro. Similarmente, se o espaço de memória livre for menor que o espaço requisitado dinamicamente, a alocação não é feita e o programa pode prever um tratamento de erro adequado (por exemplo, podemos imprimir a mensagem “Memória insuficiente” e interromper a execução do programa).



Figura 7.1: Alocação esquemática de memória do computador.

7.2 Alocação de vetores

Existem funções, presentes na biblioteca padrão `stdlib.h`, que permitem alocar e liberar memória dinamicamente. A função básica para alocar memória é `malloc`. Ela recebe

como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Para exemplificar, vamos considerar a alocação dinâmica de um vetor de inteiros com 10 elementos. Como a função `malloc` retorna o endereço da área alocada e, neste exemplo, desejamos armazenar valores inteiros nessa área, devemos declarar um ponteiro de inteiro para receber o endereço inicial do espaço alocado.

O trecho de código então seria:

```
int *v = malloc(10*sizeof(int));
```

lembmando que o operador `sizeof`, aplicado a um tipo, retorna o número de bytes ocupado pelo tipo. Após este comando, se a alocação for bem-sucedida, `v` armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros. Podemos, então, tratar `v` como tratamos um vetor declarado estaticamente, pois, se `v` aponta para o início da área alocada, sabemos que `v[0]` acessa o espaço do primeiro elemento, `v[1]` acessa o segundo e assim por diante, até `v[9]`.

Além disso, devemos salientar que a função `malloc` é usada para alocar espaço para armazenar valores de qualquer tipo. Por este motivo, `malloc` retorna um ponteiro genérico, para um tipo qualquer, representado por `void*`. Este tipo pode ser convertido automaticamente pela linguagem C para o tipo ponteiro na atribuição. No entanto, é comum fazermos a conversão explicitamente, utilizando o operador de molde de tipo (*cast*)¹. O comando para a alocação do vetor de inteiros fica então:

```
int *v = (int *) malloc(10*sizeof(int));
```

A Figura 7.2 ilustra de maneira esquemática o que ocorre na memória do computador quando este código é executado.

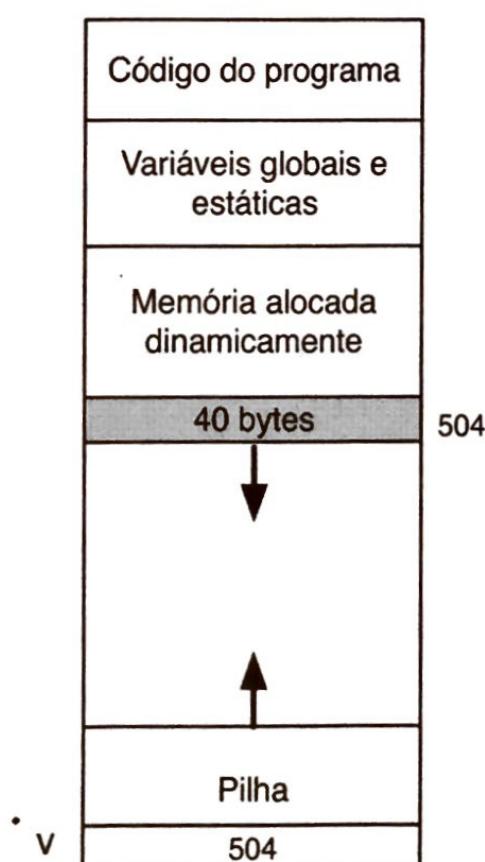


Figura 7.2: Alocação dinâmica de vetor.

¹A linguagem C++ exige a conversão explícita de `void*` para um ponteiro específico.

Se, porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo `NULL`, definido em `stdlib.h`). Devemos checar, e tratar se for o caso, o erro na alocação da memória verificando o valor de retorno da função `malloc`. Por exemplo, podemos imprimir uma mensagem e abortar o programa com a função `exit`, também definida na `stdlib.h`.

```
...
int *v = (int*) malloc(10*sizeof(int));
if (v == NULL)
{
    printf("Memoria insuficiente.\n");
    exit(1); /* aborta o programa */
}
...
...
```

Para liberar um espaço de memória previamente alocado, usamos a função `free` que recebe como parâmetro o ponteiro da memória a ser liberada. Assim, para liberar o vetor `v`, fazemos:

```
free (v);
```

Só podemos passar para a função `free` um endereço de memória que tenha sido alocado dinamicamente. Devemos lembrar ainda que não podemos acessar o espaço de memória depois que o liberamos.

Para exemplificar o uso da alocação dinâmica, alteraremos o programa para o cálculo da média e da variância mostrado anteriormente. Agora, o programa lê o número de valores que serão fornecidos, aloca um vetor dinamicamente, captura os valores e faz os cálculos. Somente a função `main` precisa ser alterada, pois as funções para capturar os valores, e calcular a média e a variância anteriormente apresentadas independem do fato de o vetor ter sido alocado estática ou dinamicamente.

```
int main (void)
{
    int n; /* número de valores */

    /* lê número de valores */
    printf("Entre com o numero de valores: ");
    scanf("%d", &n);
    float *x = (float*) malloc(n*sizeof(float));
    if (x == NULL)
    {
        printf("Memoria insuficiente.\n");
        exit(1); /* aborta o programa */
    }
    captura(n,x);
    float m = media(n,x);
    float v = variancia(n,x,m);

    printf("Media: %f\n Variancia: %f\n", m, v);
    return 0;
}
```

7.2.1 Vetores locais a funções

Em geral, reservamos o uso de alocação dinâmica para os casos em que a dimensão do vetor é desconhecida. O uso de vetores declarados localmente deve ser preferido sempre que soubermos de antemão a dimensão do vetor. No entanto, devemos mais uma vez salientar que a área de memória de uma variável local só existe enquanto a função que declara a variável estiver sendo executada. Este fato requer cuidado quando da utilização de vetores locais dentro de funções.

Para exemplificar, vamos considerar uma aplicação que manipula vetores algébricos no espaço tridimensional. Um vetor algébrico em 3D é representado pelos três componentes x , y , e z . Podemos então representar um vetor algébrico por um vetor (de C) de dimensão 3.

Vamos agora considerar a implementação de uma função que calcula o produto vetorial de dois vetores. O produto vetorial é dado por:

$$\vec{u} \times \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y & u_z v_x - u_x v_z & u_x v_y - u_y v_x \end{bmatrix}$$

Podemos pensar numa função que recebe dois vetores como parâmetros e retorna o resultado do produto vetorial. Uma forma *incorrecta* de implementar essa função é:

```
/* função INCORRETA */
float* prod_vetorial (float* u, float* v)
{
    float p[3];
    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];
    return p; /* ERRO: não podemos retornar endereço de área local */
}
```

O erro nesta implementação consiste no fato de retornar o valor de um endereço de memória que não estará mais disponível quando a função terminar. A variável p é declarada localmente, portanto esta área de memória deixa de ser válida quando a função termina. Assim, a função que chama não pode acessar a área apontada pelo valor retornado.

Uma possível solução para este problema consiste em usar alocação dinâmica. A implementação da função seria então dada por:

```
float* prod_vetorial (float* u, float* v)
{
    float *p = (float*) malloc(3*sizeof(float));
    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];
    return p;
}
```

Neste caso, a implementação é válida, pois a área apontada por p , alocada dinamicamente, permanece válida mesmo após o término da função. Assim, a função que chama

poderia acessar o ponteiro retornado. O único problema nesta solução é que fazemos uma alocação dinâmica para cada chamada da função. Isso, em geral, é ineficiente do ponto de vista computacional e requer que a função que chama seja responsável pela liberação do espaço alocado.

Outra solução consiste em requisitar que o espaço de memória para o armazenamento do resultado já seja passado pela função que chama. Assim, a função para o cálculo do produto recebe três vetores, dois com dados de entrada e um para armazenar o resultado. Uma implementação desta estratégia é ilustrada a seguir:

```
void prod_vetorial (float* u, float* v, float* p)
{
    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];
}
```

Para este problema, esta solução é, em geral, mais adequada, pois não envolve alocação dinâmica. Quando discutirmos tipos estruturados, vamos verificar que existe mais uma alternativa, que nos permite retornar explicitamente os três componentes do vetor.

7.3 Alocação de matrizes

As matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores: precisamos saber de antemão suas dimensões. Se as dimensões só são conhecidas em tempo de execução, devemos utilizar alocação dinâmica. O problema que encontramos é que a linguagem C só permite alocar dinamicamente conjuntos unidimensionais (a função *malloc* aloca um espaço contíguo de memória). Para trabalhar 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.

7.3.1 Matriz representada por um vetor simples

Concretamente, para representar uma matriz, precisamos de um espaço de memória suficiente para armazenar seus elementos. Podemos então adotar a estratégia de armazenar os elementos da matriz num vetor simples. Assim, reservamos as primeiras posições do vetor para armazenar os elementos da primeira linha, seguidos dos elementos da segunda linha e assim por diante. Conceitualmente, trabalharemos com um conjunto bidimensional, mas, de fato, temos um vetor unidimensional. Portanto, 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 fazer a conta de endereçamento explicitamente e acessar o elemento $v[k]$, com $k = i * n + j$, onde n representa o número de colunas da matriz, conforme ilustrado na Figura 7.3.

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 índice j .

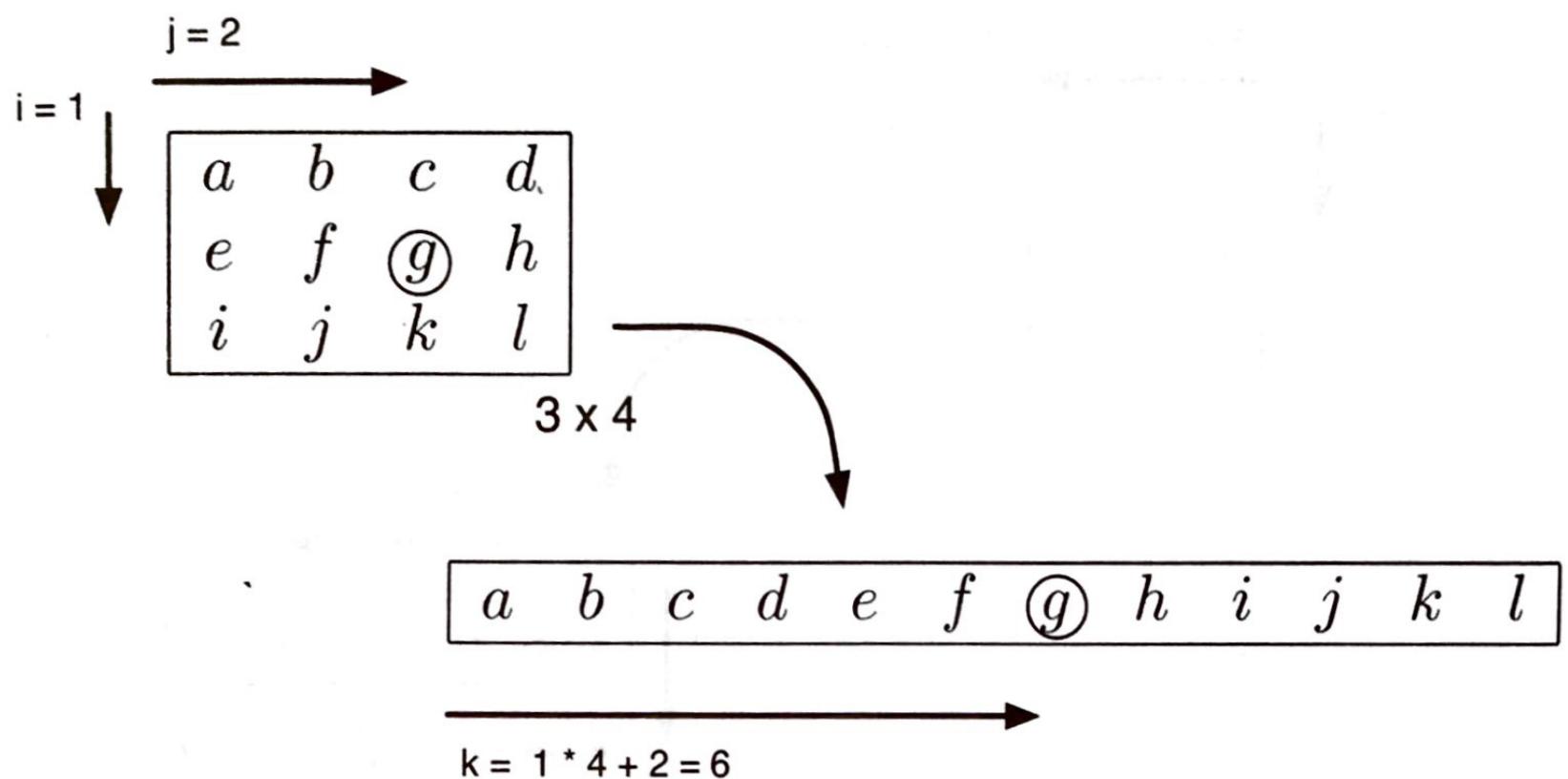


Figura 7.3: Matriz representada por vetor simples.

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

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

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

7.3.2 Matriz representada por um vetor de ponteiros

Vamos agora apresentar outra estratégia para trabalhar com matrizes dinâmicas implementadas usando vetores simples. Nela, 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 vetor-linha. A Figura 7.4 ilustra o arranjo da memória utilizado nesta estratégia.

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 a seguir ilustra essa codificação:

```
...
/* matriz representada por um vetor de ponteiros */
float** mat = (float**) malloc(m*sizeof(float*));
for (int i=0; i<m; i++)
    mat[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 assim alocada, $mat[i]$ representa o ponteiro para o primeiro elemento da linha i , e, consequentemente, $mat[i][j]$ acessa o elemento da coluna j da linha i .

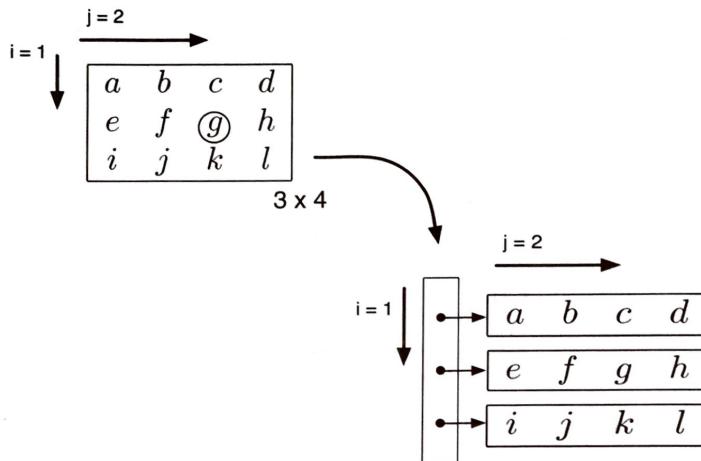


Figura 7.4: Matriz com vetor de ponteiros.

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

7.3.3 Operações com matrizes

Para exemplificar o uso de matrizes alocadas dinamicamente, vamos considerar a implementação de uma função que, dada uma matriz, cria dinamicamente a matriz transposta correspondente, isto é, a função retorna uma nova matriz, representando a transposta da matriz original. Vamos exemplificar o uso das duas estratégias para representação de matrizes discutidas.

Matriz com vetor simples

Usando a estratégia de representar a matriz através de um vetor simples, podemos considerar que o protótipo da função para criar a matriz transposta é dado por:

```
float* transposta (int m, int n, float* mat);
```

onde m e n representam, respectivamente, o número de linhas e colunas da matriz mat , cuja transposta queremos criar. A função tem como valor de retorno o ponteiro do vetor que representa a matriz transposta criada. A implementação dessa função pode ser dada por:

```

float* transposta (int m, int n, float* mat)
{
    /* aloca matriz transposta */
    float* trp = (float*) malloc(n*m*sizeof(float));

    /* preenche matriz */
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
            trp[j*m+i] = mat[i*n+j];

    return trp;
}

```

Matriz com vetor de ponteiros

O mesmo problema pode ser resolvido usando a estratégia de alocar a matriz através de um vetor de ponteiros. Neste caso, o protótipo da função tem que mudar ligeiramente, pois as matrizes passam a ser representadas por vetores de ponteiros:

```
float** transposta (int m, int n, float** mat);
```

Uma implementação para essa estratégia é mostrada a seguir. Devemos notar que, neste caso, a complexidade adicional na alocação da matriz nos permitiu acessar e atribuir os elementos usando a sintaxe convencional de acesso a conjuntos bidimensionais.

```

float** transposta (int m, int n, float** mat)
{
    /* aloca matriz transposta: n linhas, m colunas */
    float** trp = (float**) malloc(n*sizeof(float*));
    for (int i=0; i<n; i++)
        trp[i] = (float*) malloc(m*sizeof(float));

    /* preenche matriz */
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
            trp[j][i] = mat[i][j];

    return trp;
}

```

7.3.4 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.

A representação de matrizes com essa economia de memória pode ser feita com um vetor simples ou um vetor de ponteiros. A seguir, discutiremos a implementação de duas funções: uma para criar uma matriz quadrada simétrica e outra para, dada uma matriz já criada, acessar seus elementos.

Matriz simétrica com vetor simples

A função para criar a matriz dinamicamente usando um vetor simples não apresenta nenhuma dificuldade, pois basta dimensionar o vetor com apenas s elementos. Uma função para realizar essa tarefa é mostrada a seguir. Note que a matriz é obrigatoriamente quadrada e, portanto, só precisamos passar uma dimensão.

```
float* cria (int n)
{
    int s = n*(n+1)/2;
    float* mat = (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. Dessa forma, isolamos dentro do código que manipula a matriz diretamente o fato de a matriz não estar explicitamente toda armazenada. Usando essa função de acesso, podemos escrever outras funções que operam sobre matrizes simétricas sem nos preocupar com a forma de representação interna dos elementos.

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 (int n, float* mat, int i, int j)
{
    int k; /* índice do elemento no vetor */
    if (i>=j)
        k = i*(i+1)/2 + j; /* acessa elemento representado */
    else
        k = j*(j+1)/2 + i; /* acessa elemento simétrico */
    return mat[k];
}
```

7.3.5 Matriz simétrica com vetor de ponteiros

A estratégia de trabalhar com vetores de ponteiros para matrizes alocadas dinamicamente é muito adequada para a representação de matrizes simétricas. Conforme já discutido, para otimizar o uso da memória, armazenamos apenas a parte triangular inferior da matriz (incluindo os elementos da diagonal). 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.

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

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

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

```
float acessa (int n, float** mat, int i, int j)
{
    if (i>=j)
        return mat[i][j]; /* acessa elemento representado */
    else
        return mat[j][i]; /* acessa elemento simétrico */
}
```

Por fim, 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 de acessar o valor simétrico.

Exercícios

1. Escreva uma função que receba um vetor de números reais e tenha como valor de retorno um novo vetor, alocado dinamicamente, com os elementos do vetor original em ordem reversa. A função deve ter como valor de retorno o ponteiro do vetor alocado, seguindo o protótipo:

```
float* reverso (int n, float* v);
```

Faça uma função *main* para testar sua função. Na função *main*, não esqueça de liberar a memória alocada pela função auxiliar.

2. Escreva uma função que receba um vetor de inteiros e tenha como valor de retorno um novo vetor, alocado dinamicamente, apenas com os elementos pares do vetor original. A função deve ter como valor de retorno o ponteiro do vetor alocado e preencher o endereço de memória passado com a dimensão do novo vetor, seguindo o protótipo:

```
int* somente_pares (int n, int* v, int* npares);
```

Faça uma função *main* para testar sua função. Na função *main*, não esqueça de liberar a memória alocada pela função auxiliar.

3. Escreva funções para lidar com matrizes triangulares inferiores de dimensão $n \times n$, onde todos os elementos abaixo da diagonal são iguais a zero e não devem ser alocados. No entanto, um acesso a um elemento abaixo da diagonal deve retornar o valor zero. Escreva as seguintes funções, usando a estratégia de vetor de ponteiros para armazenar a matriz.

- (a) Função para criar uma matriz, onde n representa a dimensão da matriz, inicialmente com os valores todos iguais a zero:

```
float** ti_cria (int n);
```

- (b) Função para atribuir o valor de um elemento da matriz, assumindo que $i \geq j$:

```
void ti_atribui (int i, int j, float x);
```

- (c) Função para acessar o valor de um elemento da matriz, inclusive elementos acima da diagonal:

```
float ti_acessa (int i, int j);
```

- (d) Função para liberar a memória da matriz alocada:

```
void ti_libera (int n, float** mat);
```

Escreva uma função *main* para testar as funções implementadas.

4. Repita as implementações do exercício anterior para uma matriz triangular superior. *Nota:* é mais fácil alocar os elementos abaixo da diagonal, e usar o acesso transposto aos elementos.

5. Escreva:

- (a) Uma função para converter uma matriz representada por um vetor numa matriz representada por um vetor de ponteiros:

```
float** converte_a (int m, int n, float* mat);
```

- (b) Uma função para converter uma matriz representada por um vetor de ponteiros numa matriz representada por um vetor:

```
float* converte_b (int m, int n, float** mat);
```

Escreva também as funções para criar as matrizes e implemente uma função *main* para testar as conversões.