

Capítulo 6

Vetores

Nos exemplos anteriores, usamos variáveis simples para armazenar valores usados por nossos programas. Em várias situações, no entanto, precisamos armazenar não alguns poucos valores, mas um conjunto de valores. Para exemplificar, vamos considerar o cálculo do valor médio de um conjunto de valores. A média de um conjunto com n valores x_i é definida como sendo:

$$m = \frac{\sum x_i}{n}$$

Considerando que os valores serão fornecidos pelo usuário via teclado, podemos fazer um programa que lê inicialmente o número de valores que será fornecido e, em seguida, para cada valor capturado, acumula o valor numa variável que representa o somatório de todos os valores. Por fim, o valor da média é calculado e exibido:

```
#include <stdio.h>
int main (void)
{
    int n;          /* número de valores */
    float xi;        /* cada um dos valores a ser lido */
    float s = 0.0f;   /* somatório dos valores */

    /* lê número de valores */
    printf("Entre com o numero de valores: ");
    scanf("%d", &n);

    /* captura e acumula cada valor lido */
    printf("Entre com os valores:\n");
    for (int i=0; i<n; ++i) {
        scanf("%f", &xi);
        s += xi;
    }

    /* calcula e exibe valor da média */
    float m = s / n;
    printf("Media: %f\n", m);

    return 0;
}
```

Este é um exemplo simples pois foi possível processar o conjunto de valores fornecido sem armazená-lo na memória – o somatório era atualizado a cada valor capturado. Vamos agora considerar que, além da média, também estamos interessados em calcular a variância do conjunto dos números. A variância v de um conjunto de valores x_i é definida como sendo:

$$v = \frac{\sum (x_i - m)^2}{n}$$

onde m representa a média dos valores. Neste caso, a estratégia de acumular a soma à medida que lemos os valores deixa de ser adequada, pois precisamos do valor da média para poder fazer o somatório para o cálculo da variância. Uma alternativa seria ler o mesmo conjunto de dados duas vezes, uma para o cálculo da média e outra para o cálculo da variância; mas isso é impraticável (o usuário teria que entrar com a mesma sequência de valores duas vezes).

A solução para esse problema é usar um mecanismo que nos permita armazenar um conjunto de valores na memória do computador. Desta forma, podemos ler os valores e armazená-los na memória. Posteriormente, estes valores podem ser livremente processados de forma eficiente, pois já estarão na memória do computador.

6.1 Vetores

Podemos armazenar um conjunto de valores na memória do computador com o uso de *vetores* (*arrays*, em inglês). O vetor é a forma mais simples para organizar dados na memória do computador. Num vetor, os valores são armazenados em sequência, um após o outro, e podemos livremente acessar qualquer valor do conjunto. Na linguagem C, quando *declaramos um vetor* (conceito análogo ao de declaração de uma variável simples) devemos informar a *dimensão do vetor*, isto é, o número *máximo* de elementos que poderá ser armazenado no espaço de memória que é reservado para o vetor. Devemos também informar o tipo dos valores que serão armazenados no vetor (`int`, `float`, `double` etc.). Num vetor, só podemos armazenar valores de um mesmo tipo. Assim, se declararmos um vetor de `int`'s, só podemos armazenar valores inteiros; se declararmos um vetor de `float`'s, só podemos armazenar valores reais de simples precisão e assim por diante.

Em C, a sintaxe usada para declarar um vetor é similar à sintaxe para declaração de variáveis simples, mas devemos especificar a dimensão do vetor entre colchetes logo após o nome da variável que representa o vetor. Assim, para declarar uma variável que representa um vetor de até 10 inteiros com o nome v , fazemos:

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

Essa declaração reserva um espaço de memória para armazenar 10 valores inteiros e este espaço de memória é referenciado pelo nome v . Após a declaração de um vetor, podemos escrever e ler cada valor do conjunto. A linguagem C não oferece mecanismos para processar todos os valores do conjunto ao mesmo tempo – só podemos acessar e processar elemento a elemento. O primeiro elemento do conjunto representado pelo vetor é acessado pelo índice 0 (zero). Consequentemente, o último elemento de um vetor com dimensão n é acessado pelo índice $n - 1$. No caso da declaração mostrada, o último elemento do vetor é acessado pelo índice 9.

Após a declaração de um vetor, podemos atribuir valores a alguns elementos do vetor:

```
int v[10];

v[0] = 5;
v[1] = 11;
v[4] = 0;
v[9] = 3;
```

Este trecho de código atribui o valor 5 ao primeiro elemento do vetor, o valor 11 ao segundo, o valor 0 ao quinto e o valor 3 ao último elemento (décimo elemento do vetor). Os demais elementos do vetor permanecem com valores indefinidos. A Figura 6.1(a) ilustra a alocação do vetor na memória e a atribuição de valores dos seus elementos; a Figura 6.1(b) mostra como em geral representamos graficamente um vetor, com seus índices e seus valores.

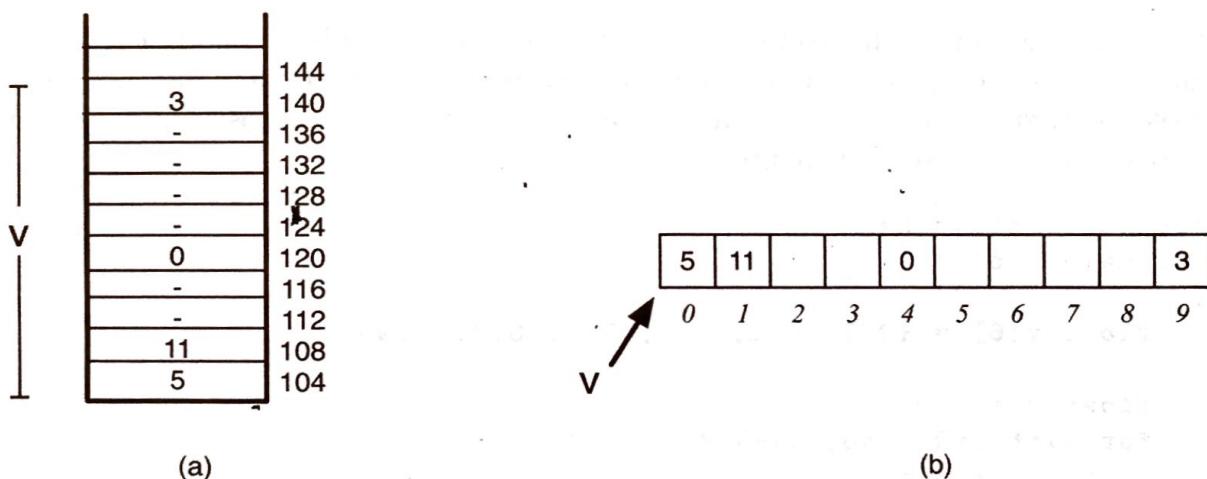


Figura 6.1: Representação de vetor na memória do computador: (a) representação na pilha; (b) representação gráfica.

Como dissemos, a partir da declaração de um vetor, podemos acessar livremente seus elementos. É válido declarar um vetor com uma dimensão n e armazenar valores nas primeiras m posições, desde que m seja menor ou igual a n . Se tentarmos acessar elementos além da dimensão do vetor, acessaremos uma área de memória inválida, pois não está reservada para o vetor em questão. Neste caso, dizemos que estamos *invadindo* memória, isto é, acessando uma memória que não está reservada para nosso uso – um grave erro de programação. Assim, se declaramos um vetor com n elementos, acessar o elemento de índice n é uma operação inválida; só podemos acessar os elementos com índices entre 0 e $n - 1$.

Podemos declarar vetores de diferentes tipos. O trecho de código a seguir ilustra declarações válidas de variáveis simples e variáveis vetor.

```
int a, b[20];           /* declara uma variável simples e um vetor */
float c[10];            /* declara um vetor */
double d[30], e, f[5];  /* declara dois vetores e uma simples */
```

Ainda, assim como podemos inicializar os valores das variáveis simples na declaração, podemos também inicializar os valores dos elementos dos vetores na declaração. Os valores iniciais dos elementos dos vetores devem ser listados entre abre e fecha chaves, separados por vírgula como ilustrado a seguir:

```
int v[5] = {12, 5, 34, 32, 9};
```

Se inicializamos todos os elementos do vetor, podemos omitir a dimensão na declaração (a dimensão é inferida pelo número de valores iniciais).

```
int v[] = {12, 5, 34, 32, 9};
```

Podemos ainda dimensionar explicitamente o vetor, mas só inicializar os primeiros elementos:

```
int v[10] = {12, 5, 34, 32, 9};
```

Para acessar, um a um, todos os elementos do vetor, podemos codificar uma construção de repetição, usando uma variável inteira como índice do elemento do vetor. Por exemplo, o programa a seguir declara e inicializa um vetor de números reais e, em seguida, exibe o somatório de seus elementos:

```
#include <stdio.h>
int main (void)
{
    float v[6] = {2.3, 5.4, 1.0, 7.6, 8.8, 3.9};

    float s = 0.0;
    for (int i=0; i<6; i++) {
        s = s + v[i];
    }
    printf("Somatorio: %f\n", s);
    return 0;
}
```

6.1.1 Cálculo da média e da variância

Agora que conhecemos um mecanismo para armazenar conjuntos de valores na memória do computador, podemos implementar o código que exibe a média e a variância de um conjunto de valores capturados via teclado. Vamos considerar que o número de valores não ultrapassará 100, que usaremos para dimensionar nosso vetor.

```
#include <stdio.h>
#define N 100          /* dimensão do vetor */

int main (void)
{
    int n;           /* número de valores */
    float x[N];      /* vetor dos valores */
```

```

/* lê número de valores */
printf("Entre com o numero de valores: ");
scanf("%d", &n);
if (n > N) {
    printf("Valor ultrapassa o limite de %d.\n", N);
    return 1; /* finaliza a execução da função main */
}

/* captura e armazena valores */
printf("Entre com os valores:\n");
for (int i=0; i<n; ++i) {
    scanf("%f", &x[i]);
}

/* calcula o valor da média */
float m = 0.0f;
for (int i=0; i<n; ++i) {
    m += x[i];
}
m /= n;

/* calcula o valor da variância */
float v = 0.0f;
for (int i=0; i<n; ++i) {
    v += (x[i]-m) * (x[i]-m);
}
v /= n;

/* exibe valores */
printf("Media: %f\nVariancia: %f\n", m, v);

return 0;
}

```

Devemos observar que passamos para a função `scanf` o endereço de cada elemento do vetor (`&x[i]`), pois desejamos que os valores capturados sejam armazenados nos elementos do vetor. Se `x[i]` representa o $(i + 1)$ -ésimo elemento do vetor, `&x[i]` representa o endereço de memória em que esse elemento está armazenado.

6.2 Vetores e ponteiros

Na linguagem C, existe uma forte associação entre vetores e ponteiros. Na declaração:

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

o símbolo `v`, que representa o vetor, é uma constante com o valor do *endereço inicial* do vetor, isto é, podemos pensar que `v`, sem indexação, aponta para o primeiro elemento do vetor. O tipo de um vetor é, portanto, um ponteiro para o tipo do elemento do vetor; no caso, o tipo de `v` é `int*`.

Com isso, como `v` representa um endereço de memória, se quisermos armazenar o valor de `v` em outra variável, esta deve ser declarada como ponteiro para inteiro:

```
int v[10];
...
int *u = v;
u[0] = 4;
u[1] = v[0] + 2;      /* note: v[0] armazena 4 */
```

A partir da atribuição `int *u = v;`, ambos, `v` e `u`, armazenam o endereço inicial do vetor; portanto, é possível acessar os elementos de `v` via variável `u`. A diferença entre `v` e `u` é que `v` é constante e `u` é variável: não podemos atribuir outro valor de endereço a `v`, mas podemos a `u`. De fato, podemos escrever:

```
int v[10];
...
int *u = v;           /* u aponta para o primeiro elemento de v */
u[0] = 4;             /* primeiro elemento de v é alterado */
u = &v[1];            /* u aponta agora para o segundo elemento de v */
u[0] = v[0] + 5;      /* segundo elemento de v é alterado */
```

Note ainda que escrever `u = v` é equivalente a escrever `u = &v[0]`; de forma similar, escrever `v[0] = 4` é equivalente a escrever `*v = 4`, já que `v` é o ponteiro para o primeiro elemento. Portanto, não existe diferença de tipo entre ponteiro para uma variável simples e um vetor, pois o vetor representa o ponteiro para o primeiro elemento. A diferença existe apenas no espaço de memória; o vetor aponta para o primeiro elemento de uma área maior de memória. Podemos pensar que um ponteiro para uma variável simples é um vetor com um único elemento.

A linguagem C também suporta *aritmética de ponteiros*. Podemos somar e subtrair ponteiros, desde que o valor do ponteiro resultante aponte para dentro da área reservada para o vetor. Se `p` representa um ponteiro para um inteiro, `p + 1` representa um ponteiro para o próximo inteiro armazenado na memória, isto é, o valor de `p` é incrementado de 4 (mais uma vez assumindo que um inteiro tem 4 bytes). Com isso, num vetor `v`, temos as seguintes equivalências:

- `v + 0` → aponta para o primeiro elemento do vetor
- `v + 1` → aponta para o segundo elemento do vetor
- `v + 2` → aponta para o terceiro elemento do vetor
- ...
- `v + 9` → aponta para o décimo elemento do vetor

Portanto, escrever `&v[i]` é equivalente a escrever `(v+i)`. De maneira análoga, escrever `v[i]` é equivalente a escrever `*(v+i)`. Devemos notar que o uso da aritmética de ponteiros aqui é perfeitamente válido, pois os elementos dos vetores são armazenados de forma contínua na memória.

6.2.1 Passagem de vetores para funções

Agora que conhecemos a associação entre vetores e ponteiros, podemos analisar de que maneira vetores são passados como argumentos para funções. Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor. Se passamos um valor de endereço, a função chamada deve ter um parâmetro do tipo ponteiro para armazenar este valor. Assim, se passarmos para uma função um vetor de `int`, devemos ter um parâmetro do tipo `int*`, capaz de armazenar endereços de inteiros. Salientamos que a expressão “passar um vetor para uma função” deve ser interpretada como “passar o endereço inicial do vetor”. Os elementos do vetor não são copiados para a função, o argumento copiado é apenas o endereço do primeiro elemento.

Para exemplificar, vamos modificar o código do exemplo anterior para cálculo de média e variância, usando três funções auxiliares: função para capturar os valores, função para calcular a média e função para calcular a variância:

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

#define N 100      /* dimensão do vetor */

void captura (int n, float* x)
{
    printf("Entre com os valores:\n");
    for (int i=0; i<n; ++i)
        scanf("%f", &x[i]);
}

float media (int n, float* x)
{
    float m = 0.0f;
    for (int i=0; i<n; ++i)
        m += x[i];
    return m / n;
}

float variancia (int n, float* x, float m)
{
    float v = 0.0f;
    for (int i=0; i<n; ++i)
        v += (x[i]-m) * (x[i]-m);
    return v / n;
}

int main (void)
{
    int n;          /* número de valores */
    float x[N];     /* vetor dos valores */

    /* lê número de valores */
    printf("Entre com o numero de valores: ");
    scanf("%d", &n);
```

```

if (n > N) {
    printf("Valor ultrapassa o limite de %d.\n", N);
    return 1;      /* finaliza a execução da função main */
}
captura(n, x);
float m = media(n, x);
float v = variancia(n, x, m);
printf("Media: %f\nVariância: %f\n", m, v);
return 0;
}

```

Observamos que, como é passado para a função o endereço do primeiro elemento do vetor (e não os elementos propriamente ditos), podemos alterar os valores dos elementos do vetor dentro da função, o que é feito na função `captura`. Para sinalizar que as funções `media` e `variancia` não alteram os elementos do vetor (apenas acessam seus valores), podemos declarar o parâmetro com o modificador de tipo `const` (constante):

```

float media (int n, const int* v)
{
    ...
}

```

O uso apropriado do modificador `const`, na verdade, é uma ótima prática de programação, em especial em programas maiores. Outro código que chama a função, olhando o protótipo com `const`, sabe que o vetor garantidamente não é alterado dentro da função, sem precisar olhar o código. Nos códigos ilustrados neste texto, no entanto, vamos evitar usar `const` para deixar o código conciso e mais fácil de ler.

Uma alternativa de implementação do nosso exemplo é fazer uma função única para calcular a média e a variância:

```

void media_variancia (int n, float* x, float* pm, float* pv)
{
    *pm = 0.0f;
    for (int i=0; i<n; ++i)
        *pm += x[i];
    *pm /= n;

    *pv = 0.0f;
    for (int i=0; i<n; ++i)
        *pv += (x[i]-(*pm)) * (x[i]-(*pm));
    *pv /= n;
}

```

Na função `main`, a chamada desta função seria:

```

...
float m, v;
media_variancia(n, x, &m, &v);
...

```

Neste caso, a função não tem valor de retorno, mas recebe os endereços de memória nos quais armazenar os valores de média e variância. Note que os parâmetros `x`, `pm` e `pv` são do mesmo tipo: `x` aponta para o primeiro elemento do vetor; `pm` e `pv` apontam para as variáveis que armazenarão os valores computados. A linguagem C permite definir um parâmetro do tipo `int *x` escrevendo `int x[]`. Pode-se então diferenciar ponteiros de variáveis e ponteiros de vetores, visualmente, usando sintaxes diferentes:

```
void media_variancia (int n, float x[], float* pm, float* pv)
```

Entretanto, é importante notar que a semântica é *exatamente* a mesma.

Muitos programadores preferem evitar o acesso indireto às variáveis ao longo da função, pois o código fica pesado. Uma alternativa é computar os valores em variáveis auxiliares e, no final, preencher os endereços de memória passados:

```
void media_variancia (int n, float* x, float* pm, float* pv)
{
    float m = 0.0f;
    for (int i=0; i<n; ++i)
        m += x[i];
    pm /= n;

    float v = 0.0f;
    for (int i=0; i<n; ++i)
        v += (x[i]-m) * (x[i]-m);
    v /= n;

    *pm = m
    *pv = v;
}
```

Tal estratégia até pode trazer um pequeno ganho de desempenho em programas maiores, já que minimiza o número de acessos indiretos a variáveis.

6.2.2 Recursão com vetores

Vamos agora considerar uma função que determina o valor máximo entre os elementos de um vetor. Esta função pode ser implementada da seguinte forma. Inicialmente, assumimos que o primeiro elemento do vetor é o valor máximo. Em seguida, percorremos os elementos restantes. Para cada elemento, se maior que o valor máximo assumido, o valor máximo é atualizado e o procedimento continua, até que todos os elementos tenham sido considerados. Ao final, teremos o valor máximo dos elementos:

```
float maximo (int n, float* v)
{
    float m = v[0]; /* armazena valor máximo */
    for (int i=1; i<n; ++i) {
        if (v[i] > m)
            m = v[i];
    }
    return m;
}
```

Esta função para cálculo do valor máximo é simples e eficiente. Vamos ver agora uma implementação alternativa, a fim de introduzir implementações recursivas com vetor.

Vamos definir vetor de uma forma recursiva. Um vetor pode ser definido como:

- um conjunto com um único elemento; ou
- um conjunto composto por um primeiro elemento seguido de um subvetor que começa no segundo elemento, tendo um elemento a menos.

A Figura 6.2 mostra, de forma esquemática, essa definição recursiva de vetor.

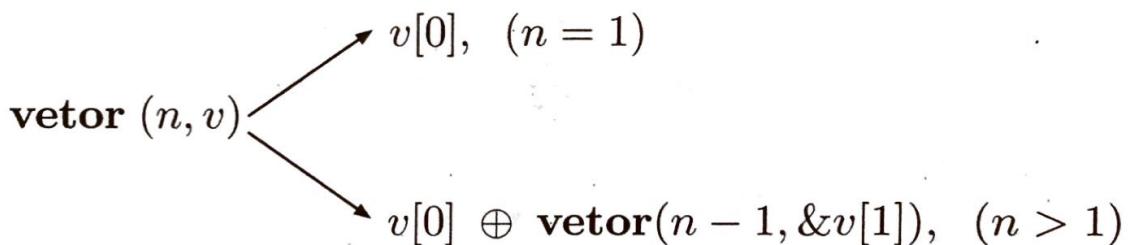


Figura 6.2: Definição recursiva de vetor.

Com base nesta definição, podemos implementar uma função recursiva para calcular o valor máximo no vetor. Se o vetor tem um único elemento, seu valor máximo é o único elemento que possui; caso contrário, o valor máximo será o máximo entre o primeiro elemento e o máximo valor do subvetor:

```

float maximo (int n, float* x)
{
    if (n == 1)
        return x[0];
    else {
        float msub = maximo(n-1, &x[1]); /* máximo do subvetor */
        return x[0] > msub ? x[0] : msub; /* máximo entre os dois */
    }
}
  
```

Como discutimos, uma implementação recursiva deve sempre considerar a condição de contorno. Neste caso, a condição de contorno ocorre quando a função é chamada para um vetor com um único elemento. Note que se o vetor tiver mais que um elemento, a mesma função é chamada para o subvetor (que tem um elemento a menos), até que se chegue a um vetor com um único elemento. Cada chamada da função `maximo` é independente. A melhor maneira de entender uma função recursiva é considerar a definição conceitual e verificar se o código trata o problema conceitualmente de forma correta. Assim, neste caso, se o vetor tiver um único elemento, a função retorna o valor máximo corretamente; se o vetor tiver mais que um elemento, calculamos o valor máximo presente no subvetor e o comparamos com o primeiro valor. Este é o ponto-chave: assumimos que a função calcula corretamente o valor máximo do subvetor e, então, calculamos o máximo do vetor inteiro, incluindo o teste com o primeiro elemento.

Se ainda não estamos convencidos de que a função executa com correção, podemos considerar um exemplo e verificar o funcionamento da função passo a passo. Vamos considerar que queremos calcular o valor máximo do vetor $\{4, 5, 3, 1\}$. Veja:

- O máximo do vetor $\{1\}$ é 1, pois só tem um elemento.
- O máximo do vetor $\{3, 1\}$ é o valor máximo entre 3 e máximo de $\{1\}$, que é 1, como visto, resultando em 3.
- O máximo do vetor $\{5, 3, 1\}$ é o valor máximo entre 5 e máximo de $\{3, 1\}$, que é 3, como visto, resultando em 5.
- O máximo do vetor $\{4, 5, 3, 1\}$ é o valor máximo entre 4 e máximo de $\{5, 3, 1\}$, que é 5, como visto, resultando em 5.

Ao longo do texto, iremos consolidar estes conceitos e aplicar implementações recursivas em diferentes problemas. Neste caso específico de cálculo de máximo valor de um vetor, a implementação recursiva serve apenas para introduzir conceitos; a implementação iterativa é muito mais eficiente e simples. Note ainda que outras definições recursivas de vetor são possíveis, dentre as quais considerar o último elemento e o subvetor que vai do primeiro elemento ao penúltimo.

6.3 Aplicação: Representação de polinômios

Nesta seção, como exemplo adicional de uso de vetores, vamos explorar a representação de polinômios. Um polinômio é uma função matemática definida por:

$$a(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_gx^g$$

onde $a_0, a_1, a_2, a_3, \dots, a_g$ são números reais, denominados *coeficientes do polinômio*, e g é um número inteiro, que representa o *grau do polinômio*. Um polinômio é, portanto, representado por seu grau e seus coeficientes. Se tivermos que manipular polinômios em um programa de computador, podemos usar vetores para representá-los. Um polinômio de grau g pode ser representado por um vetor v com $g + 1$ elementos, cujos valores representam os coeficientes do polinômio: $v[i] = a_i$. Assim, o polinômio $2x^3 + 8x + 5$ pode ser representado por um vetor com quatro elementos: $v[0]=5$, $v[1]=8$, $v[2]=0$ e $v[3]=2$.

Assumindo esta representação de polinômios com vetores, podemos considerar a implementação de diversas funções que operam sobre polinômios. Estas funções podem ser agrupadas num arquivo e servir como uma biblioteca para manipular polinômios, como veremos adiante. Vamos discutir a implementação de algumas delas.

6.3.1 Avaliação de polinômios

Avaliar um polinômio significa avaliar o valor numérico do polinômio, $y = a(x)$, para determinado x . O valor numérico de um polinômio pode ser expresso matematicamente por:

$$y = \sum_{i=0}^g a_i x^i$$

Portanto, a codificação para a avaliação de um polinômio se traduz no cálculo de um somatório. A função para avaliar um polinômio deve receber como entrada o polinômio e o valor de x , tendo como retorno o valor avaliado. Nos nossos exemplos, vamos assumir que os parâmetros que representam o polinômio são o número de coeficientes, n , e os valores destes coeficientes armazenados em um vetor, v . Portanto, estamos assumindo que o polinômio tem grau $n - 1$. Uma possível implementação desta função é mostrada a seguir:

```
float avalia (int n, float *v, float x)
{
    float y = 0.0f;
    for (int i=0; i<n; i++)
        y = y + v[i] * pow(x,i);
    return y;
}
```

6.3.2 Igualdade de polinômios

Dois polinômios de graus g , $a(x) = a_0 + a_1x + a_2x^2 + \dots + a_gx^g$ e $b(x) = b_0 + b_1x + b_2x^2 + \dots + b_gx^g$, são iguais se os valores de seus coeficientes forem iguais, isto é, a_i tem que ser igual a b_i para qualquer $i : 0 \leq i \leq g$.

Uma função para testar se dois polinômios de mesmo grau são iguais pode receber como parâmetros o número de coeficientes dos polinômios e os vetores com seus valores. Uma possível implementação desta função é mostrada a seguir. A função retorna 0 se os polinômios forem diferentes e 1 se eles forem iguais.

```
int igualdade (int n, float *a, float *b)
{
    for (int i=0; i<n; i++) {
        if (a[i] != b[i])
            return 0;
    }
    return 1;
}
```

6.3.3 Soma de polinômios

A soma de dois polinômios de graus g , $a(x) = a_0 + a_1x + a_2x^2 + \dots + a_gx^g$ e $b(x) = b_0 + b_1x + b_2x^2 + \dots + b_gx^g$, é dada por $c(x) = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + \dots + (a_g + b_g)x^g$. Esta operação pode ser matematicamente expressa por:

$$c(x) = a(x) + b(x) = \sum_{i=0}^g c_i x^i, \text{ onde } c_i = (a_i + b_i)$$

A implementação de uma função que efetua essa operação é simples – basta codificar a soma de dois vetores. A função deve receber como parâmetros o número de coeficientes dos polinômios e três vetores que armazenam os valores destes coeficientes: dois vetores representando os polinômios de entrada e um representando o polinômio de saída.

Na chamada desta função, o programador deve garantir que o vetor que armazenará os coeficientes da soma é dimensionado com número de elementos suficiente. Uma implementação desta função é mostrada a seguir:

```
void soma (int n, float *a, float *b, float *c)
{
    for (int i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```

6.3.4 Produto de polinômios

Um exemplo mais elaborado consiste em codificar uma função que efetua o produto entre dois polinômios. O produto de dois polinômios, $a(x)$ e $b(x)$, com grau g , é dado por:

$$c(x) = a(x)b(x) = c_0 + c_1x + c_2x^2 + \dots + c_{2g}x^{2g}$$

onde:

$$c_k = a_0b_k + a_1b_{k-1} + a_2b_{k-2} + \dots + a_{k-1}b_1 + a_kb_0$$

ou, de forma mais concisa, por:

$$c_k = \sum_{i=0}^k a_i b_{k-i}$$

sendo que assume-se $a_i = 0$ e $b_i = 0$ para $i > g$.

Podemos então codificar uma função que efetua o produto entre dois polinômios. Esta função pode receber como parâmetros o número de coeficientes dos polinômios de entrada e três vetores de coeficientes: dois de entrada e um de saída. Novamente, o programador é responsável por passar para a função um vetor com dimensão suficiente para armazenar os coeficientes do produto. Como o grau do polinômio resultante é $2g$ e sabemos que n , o número de coeficientes dos polinômios de entrada, é igual a $g + 1$, o número de coeficientes do vetor de saída é $2g + 1 = 2(n - 1) + 1 = 2n - 1$. Uma possível implementação desta função é mostrada a seguir. Note que na avaliação de cada termo c_k acessamos os valores a_i e b_{k-i} . Para estes acessos serem válidos, devemos garantir que $i < n$ e $k - i < n$. Quando estamos fora desta condição, o termo assume valor zero e não contribui para o somatório.

```
void produto (int n, float *a, float *b, float *c)
{
    int m = 2*n - 1; /* número de coeficientes de saída */
    for (int k=0; k<m; k++) {
        c[k] = 0.0;
        for (int i=0; i<=k; i++) {
            if (i<n && k-i<n)
                c[k] = c[k] + a[i]*b[k-i];
        }
    }
}
```

6.3.5 Derivada de polinômio

Também podemos codificar uma função que gera a derivada de um polinômio. Se o polinômio de entrada tem grau g , sua derivada é um polinômio de grau $g - 1$. Se o polinômio de entrada é dado por:

$$a(x) == a_0 + a_1x + a_2x^2 + \dots + a_gx^g$$

sua derivada é expressa por:

$$d(x) = a_1 + 2a_2x + 3a_3x^2 + \dots + ga_gx^{g-1}$$

Os termos da derivada podem então ser expressos por:

$$d_i = (i+1)a_{i+1}, \quad 0 \leq i < g$$

A função que implementa o cálculo da derivada pode receber como parâmetros o número de coeficientes do polinômio de entrada, n , e dois vetores de coeficientes: um de entrada e um de saída. O programador é responsável por passar para esta função um vetor de saída dimensionado com pelo menos $n - 1$ elementos. Uma possível implementação desta função é mostrada a seguir:

```
void derivada (int n, float *a, float *d)
{
    for (int i=0; i<n-1; i++) {
        d[i] = (i+1)*a[i+1];
    }
}
```

Fica como exercício a implementação de funções de testes para estas funções. Para os testes, é útil a implementação de uma função que exibe na tela os coeficientes de um dado polinômio.

6.4 Matrizes

Além de conjuntos unidimensionais de valores, muitas aplicações necessitam armazenar e manipular conjuntos de dimensões maiores. A representação de conjuntos bidimensionais, *matrizes*, por exemplo, é muito usada em diversas áreas, como em aplicações que envolvem cálculos algébricos. Matrizes podem também ser usadas para representar mapas discretos. Muitos jogos de computador, por exemplo, representam seus “mundos” por meio de matrizes de células. Conjuntos 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 bidimensional, 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 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 da primeira coluna é acessado por `mat[0][0]`, e o elemento da última linha e da última coluna é acessado por `mat[2][3]`.

Como sabemos, a memória do computador é linear. Para que se tenha a representação de conjuntos bidimensionais, 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 determinado elemento `mat[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 feito automaticamente pelo compilador. No momento, vamos nos preocupar apenas em acessar os elementos escrevendo `mat[i][j]`. O importante é observar que o compilador necessita saber o número de colunas da matriz, n , para fazer a conta de endereçamento. Esta informação nos será útil adiante.

Similar a vetor, podemos inicializar os elementos na declaração da matriz:

```
float mat [3][4] = {  
    {1.0f, 5.0f, 2.0f, 0.0f},  
    {8.0f, 2.0f, 3.0f, 1.0f},  
    {1.0f, 6.0f, 7.0f, 2.0f}  
};
```

Na declaração com inicialização, o número de colunas é obrigatório, mas o número de linhas pode ser omitido (e inferido baseado na inicialização). A Figura 6.3 ilustra o armazenamento de matrizes na memória do computador.

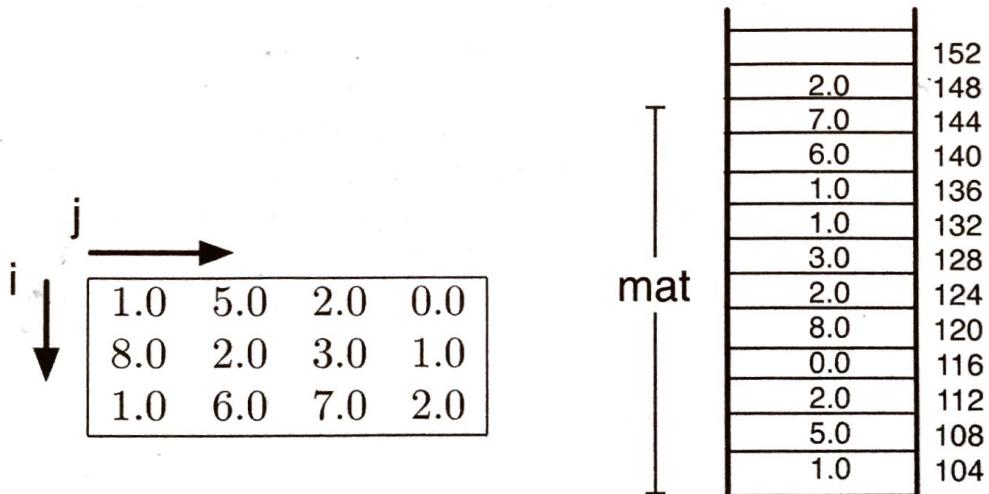


Figura 6.3: Armazenamento de matrizes na memória do computador.

6.5 Laços aninhados

Para acessar todos os elementos de um conjunto bidimensional, em geral usamos a construção de laços aninhados, isto é, codificamos um laço dentro de outro laço. Assim, podemos fazer um laço percorrer as linhas de uma matriz e, para cada linha, percorremos os elementos da linha. Um programa que declara e exibe na tela uma matriz pode ser:

```
#include <stdio.h>

int main (void)
{
    float mat [3][4] = {
        {1.0f, 5.0f, 2.0f, 0.0f},
        {8.0f, 2.0f, 3.0f, 1.0f},
        {1.0f, 6.0f, 7.0f, 2.0f}
    };
    for (int i=0; i<3; i++) {
        for (int j=0; j<4; j++) {
            printf("M(%d,%d)=% .1f ", i, j, mat[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Se executado, serão exibidos os 12 valores da matriz:

```
M(0,0)=1.0  M(0,1)=5.0  M(0,2)=2.0  M(0,3)=0.0
M(1,0)=8.0  M(1,1)=2.0  M(1,2)=3.0  M(1,3)=1.0
M(2,0)=1.0  M(2,1)=6.0  M(2,2)=7.0  M(2,3)=2.0
```

O laço controlado pela variável *i* (primeiro *for*) repete a execução de seu bloco três 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, 3 e assim por diante.

Da mesma maneira, podemos codificar laços aninhados fazendo uso das outras formas de construção de laços, como *while* e *do-while*. É válido 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.

6.5.1 Passagem de matrizes para funções

Após a declaração estática de uma matriz, como em

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

a variável *mat* representa um ponteiro para o primeiro “vetor-linha”, composto por quatro elementos no caso. Com isso, *mat[1]* aponta para o primeiro elemento do segundo “vetor-

linha” e assim por diante. Em C, o tipo que representa um “vetor-linha” com quatro elementos float é expresso como: float (*mat)[4]) ou float mat[] [4].

Quando passamos uma matriz para uma função, o parâmetro da função deve ser do tipo “vetor-linha”:

```
#include <stdio.h>
void imprime (int m, float mat[][])
{
    for (int i=0; i<m; i++) {
        for (int j=0; j<4; j++) {
            printf("M(%d,%d)=% .1f ", i, j, mat[i][j]);
        }
        printf("\n");
    }
}

int main (void)
{
    float mat [3][4] = {
        {1.0f, 5.0f, 2.0f, 0.0f},
        {8.0f, 2.0f, 3.0f, 1.0f},
        {1.0f, 6.0f, 7.0f, 2.0f}
    };
    imprime(3, mat);
    return 0;
}
```

Note que a função `imprime` codificada serve para imprimir qualquer matriz de float com quatro colunas. O número de linhas pode ser um parâmetro, mas o número de colunas não, tem que ser constante, pois é usado pelo compilador para a conta de endereçamento.

6.6 Aplicação: 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 preestabelecido, nossos códigos ficam particularizados para determinada dimensão de matriz. Para ilustrar, vamos considerar a implementação de algumas funções algébricas que trabalham com matrizes quadradas. Nessa implementação, vamos optar por usar o tipo real de dupla precisão. Estas funções podem ser agrupadas num arquivo e servir como uma biblioteca de funções algébricas.

Nos códigos, vamos assumir que a dimensão das matrizes é $N \times N$, onde N é uma constante simbólica. Por exemplo, se quisermos que nosso código seja usado para matrizes 4×4 , fazemos:

```
#define N 4
```

6.6.1 Matriz simétrica

Como primeiro exemplo, vamos considerar uma função para verificar se 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 se concluirmos que a matriz não é simétrica.

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

Como na prática esta função também preestabelece 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).

6.6.2 Matriz transposta

Uma função que também pode ser útil calcula a transposta de uma matriz. Se M é uma 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. A primeira representa a matriz de entrada e a segunda representa 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-se o elemento simétrico da matriz de saída.

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

Como estamos trabalhando com matrizes quadradas, podemos também pensar numa 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 transpõe (double A[][][N])
{
    for (int i=0; i<N; i++) {
        for (int j=0; j<i; j++) {
            double t = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = t;
        }
    }
}
```

6.6.3 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 num 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[])
{
    for (int i=0; i<N; i++) {
        w[i] = 0.0;
        for (int 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 numa 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])
{
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            C[i][j] = 0.0;
            for (int k=0; k<N; k++) {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
}
```

Exercícios

1. Escreva uma função que receba como parâmetro um vetor de números inteiros de tamanho n e retorne quantos números pares estão armazenados nesse vetor. Essa função deve obedecer ao protótipo a seguir. Escreva um programa para testar sua função.

```
int pares (int n, int* vet);
```

2. Implemente uma função que receba como parâmetro um vetor de números inteiros de tamanho n e inverta a ordem dos elementos armazenados nesse vetor. Essa função deve obedecer ao protótipo a seguir. Escreva um programa para testar sua função.

```
void inverte (int n, int* vet);
```

3. Escreva uma função que retorne o valor mínimo armazenado em um vetor. Essa função deve obedecer ao protótipo a seguir. Escreva um programa para testar sua função.

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

4. Escreva versões recursivas das funções dos exercícios anteriores e teste-as.
5. A média harmônica, H_n , de um conjunto de valores é dada por:

$$\frac{1}{H_n} = \sum_{i=0}^{n-1} \frac{1}{v_i}$$

Escreva uma função para calcular e retornar a média harmônica de um conjunto de valores. Essa função deve obedecer ao protótipo a seguir. Escreva um programa para testar sua função.

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

6. A média geométrica, G_n , de um conjunto de valores é dada por:

$$G_n = \sqrt[n]{\prod_{i=0}^{n-1} v_i}$$

Escreva uma função para calcular e retornar a média geométrica de um conjunto de valores. Essa função deve obedecer ao protótipo a seguir. Escreva um programa para testar sua função.

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

7. A média ponderada de um conjunto de valores é expressa por:

$$m = \frac{\sum v_i w_i}{\sum w_i}$$

onde w_i representa os pesos associados aos valores.

Escreva uma função para calcular a média ponderada de um conjunto de valores. A função deve receber como parâmetros os vetores dos valores e dos pesos, e deve retornar a média calculada. Essa função deve obedecer ao protótipo a seguir. Escreva um programa para testar sua função.

```
float ponderada (int n, float *v, float *w);
```

8. Implemente uma função que receba como parâmetros um vetor de números inteiros v e seu número de elementos n , e verifique se os elementos do vetor correspondem aos termos de uma PA (progressão aritmética), isto é, se existe um número inteiro k tal que $v[i]$ seja igual a $v[0] + i*k$. Caso os elementos do vetor correspondam aos termos de uma PA, a função deve retornar o valor calculado para k . Por exemplo, se for passado para essa função o vetor $\{2, 10, 18, 26, 34\}$, ela deve retornar o valor 8. Caso o vetor não corresponda a uma PA, a função deve retornar 0. Assuma que o vetor sempre terá pelo menos três elementos. Tal função deve ter o seguinte protótipo:

```
int testa_PA (int n, int* v);
```

9. Reescreva as funções para calcular a soma e o produto de polinômios representados por vetores para considerar que os dois polinômios de entrada possam ter graus diferentes. Os coeficientes não existentes no polinômio de menor grau têm valores nulos, mas não são representados no vetor.

10. Considere *histogramas* como sendo o número de ocorrências de valores em diferentes intervalos. Considere ainda um experimento laboratorial em que foram colhidos n medições, todas elas maiores ou iguais a 0 e menores que 1. Escreva uma função para preencher um vetor com 10 elementos que represente o *histograma* destas medidas. O primeiro elemento do vetor deve armazenar o número de medidas maiores ou iguais a 0 e menores que 0.1, o segundo elemento deve armazenar o número de medidas maiores ou iguais a 0.1 e menores que 0.2 e assim por diante. A função deve receber o vetor, v , com as n medidas do experimento e deve preencher o vetor h que, sabe-se, tem dimensão igual a 10. Por exemplo, se for passado como entrada o vetor:

$$v = \{0.11, 0.2, 0.03, 0.56, 0.323, 0.345, 0.234, 0.56, 0.6546, 0.123, 0.123, 0.999\}$$

a função deve preencher o vetor h como:

$$h = \{1, 3, 2, 2, 0, 2, 1, 0, 0, 1\}$$

A função deve seguir o protótipo:

```
void histograma (int n, float *v, int *h);
```

11. Escreva uma função que verifique se uma matriz quadrada de dimensão $N \times N$, onde N representa uma constante simbólica, é uma *matriz triangular inferior*. Numa 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 protótipo:

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

12. Escreva uma função que verifique se uma matriz quadrada de dimensão $N \times N$, onde N representa uma constante simbólica, é uma *matriz identidade*. Numa 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 protótipo:

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

13. Escreva uma função que, dadas duas matrizes quadradas, 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 numa matriz identidade. A função deve retornar 1 se B é a inversa de A e 0 caso contrário, e deve seguir o seguinte protótipo:

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