7. Vetores

Neste capítulo estudaremos o conceito de vetor, sua declaração e uso. Como são usados vetores em argumentos de funções. E, ainda, como trabalhar com vetores de mais de uma dimensão.

7.1 Introdução

Em muitas aplicações queremos trabalhar com conjuntos de dados que são **semelhantes em tipo**. Por exemplo o conjunto das alturas dos alunos de uma turma, ou um conjunto de seus nomes. Nestes casos, seria conveniente poder colocar estas informações sob um mesmo conjunto, e poder referenciar cada dado individual deste conjunto por um número índice. Em programação, este tipo de estrutura de dados é chamada de **vetor** (ou *array*, em inglês) ou, de maneira mais formal **estruturas de dados homogêneas**.

Exemplo: A maneira mais simples de entender um vetor é através da visualização de um **lista**, de elementos com um nome coletivo e um índice de referência aos valores da lista.

```
n   nota
0   8.4
1   6.9
2   4.5
3   4.6
4   7.2
```

Nesta lista, `n` representa um número de referência e `nota` é o nome do conjunto. Assim podemos dizer que a 2ª nota é `6.9` ou representar `nota[1] = 6.9`

Esta não é a única maneira de estruturar conjunto de dados. Também podemos organizar dados sob forma de tabelas. Neste caso, cada dado é referenciado por dois índices e dizemos que se trata de um **vetor bidimensional** (ou **matriz**)[1]. Vetores de mais de uma dimensão serão vistos na seção 7.5.

7.2 Declaração e inicialização de vetores

[1] Alguns autores preferem chamar todos os tipos de estruturas homogêneas, não importando o numero de índices de referência (ou dimensões) de *vetores*. Outros preferem chamar de *matrizes*. Outros ainda distinguem *vetores* (uma dimensão) de *matrizes* (mais de uma dimensão), etc. Não vamos entrar no mérito da questão (existem boas justificativas para todas as interpretações) e, nesta apostila, vamos usar a primeira nomenclatura: toda estrutura homogênea de dados será chamada de vetor.

7.2.1 Declaração de vetores

Em C, um vetor é um conjunto de variáveis de um **mesmo tipo** que possuem um nome identificador e um índice de referência.

Sintaxe: A sintaxe para a declaração de um vetor é a seguinte:

tipo nome`[`*tam*`];`

onde:

tipo é o **tipo** dos elementos do vetor: `int, float, double` ...

nome é o **nome** identificador do vetor. As regras de nomenclatura de vetores são as mesmas usadas em variáveis (seção 2.2.1).

tam é o tamanho do vetor, isto é, o número de elementos que o vetor pode armazenar.

Exemplo: Veja as declarações seguintes:

```
int idade[100]; // declara um vetor chamado 'idade' do tipo
                // 'int' que recebe 100 elementos.
float nota[25]; // declara um vetor chamado 'nota' do tipo
                // 'float' que pode armazenar 25 números.
char nome[80];  // declara um vetor chamado 'nome' do tipo
                // 'char' que pode armazenar 80 caracteres.
```

Na declaração de um vetor estamos reservando espaço de memória para os elementos de um vetor. A quantidade de memória (em *bytes*) usada para armazenar um vetor pode ser calculada como:

*quantidade de memória = tamanho do tipo * tamanho do vetor*

Assim, no exemplo anterior, a quantidade de memória utilizada pelos vetores é, respectivamente, `200(2x100)`, `100(4x25)` e `80(80x1)` *bytes*.

7.2.2 Referência a elementos de vetor

Cada elemento do vetor é referenciado pelo **nome** do vetor seguido de um **índice** inteiro. O **primeiro** elemento do vetor tem índice `0` e o **último** tem índice `tam-1`. O índice de um vetor deve ser **inteiro**.

Exemplo: Algumas referências a vetores:

```
#define MAX 5
int i = 7;
float valor[10];                // declaração de vetor
valor[1] = 6.645;
valor[MAX] = 3.867;
valor[i] = 7.645;
valor[random(MAX)] = 2.768;
valor[sqrt(MAX)] = 2.705;     // NÃO é válido!
```

7.2.2 Inicialização de vetores

Assim como podemos inicializar variáveis (por exemplo: `int j = 3;`), podemos inicializar vetores.

Sintaxe: A sintaxe para a inicialização dos elementos de um vetor é:

tipo nome`[`*tam*`] = {`*lista de valores*`};`

onde:

lista de valores é uma lista, separada por vírgulas, dos valores de cada elemento do vetor.

Exemplo: Veja as inicializações seguintes. Observe que a inicialização de `nota` gera o vetor do exemplo do início desta seção.

```
int dia[7] = {12,30,14,7,13,15,6};
float nota[5] = {8.4,6.9,4.5,4.6,7.2};
char vogal[5] = {'a', 'e', 'i', 'o', 'u'};
```

Opcionalmente, podemos inicializar os elementos do vetor enumerando-os um a um.

Exemplo: Observe que estas duas inicializações são possíveis:

```
int cor_menu[4] = {BLUE,YELLOW,GREEN,GRAY};
```

ou

```
int cor_menu[4];
```

```
cor_menu[0] = BLUE;
cor_menu[1] = YELLOW;
cor_menu[2] = GREEN;
cor_menu[3] = GRAY;
```

Programa Exemplo: O arquivo `e0701.cpp` contém um programa que mostra o uso de vetores: declaração, inicialização, leitura e escrita de elementos...

7.3 Tamanho de um vetor e segmentação de memória

7.3.1 Limites

Na linguagem C, devemos ter cuidado com os limites de um vetor. Embora na sua declaração, tenhamos definido o tamanho de um vetor, o C não faz nenhum teste de verificação de acesso a um elemento dentro do vetor ou não.

Por exemplo se declaramos um vetor como `int valor[5]`, teoricamente só tem sentido usarmos os elementos `valor[0], ..., valor[4]`. Porém, o C não acusa **erro** se usarmos `valor[12]` em algum lugar do programa. Estes testes de limite **devem** ser feitos **logicamente** dentro do programa.

Este fato se deve a maneira como o C trata vetores. A memória do microcomputador é um espaço (físico) particionado em porções de 1 *byte*. Se **declaramos** um vetor como `int vet[3]`, estamos **reservando** 6 *bytes* (3 segmentos de 2 *bytes*) de memória para armazenar os seus elementos. O primeiro segmento será reservado para `vet[0],` o segundo segmento para `vet[1]` e o terceiro segmento para `vet[2]`. O segmento inicial é chamado de segmento **base**, de modo que `vet[0]` será localizado no segmento base. Quando acessamos o elemento `vet[i]`, o processador acessa o segmento localizado em **base+i**. Se `i` for igual a `2`, estamos acessando o segmento **base+2** ou `vet[2]`(o ultimo segmento reservado para o vetor). Porém, se `i` for igual a `7`, estamos a acessando segmento **base+7** que **não foi reservado** para os elementos do vetor e que provavelmente está sendo usado por uma outra variável ou contém informação espúria (lixo).

Observe que acessar um segmento fora do espaço destinado a um vetor pode **destruir informações** reservadas de outras variáveis. Estes erros são difíceis de detectar pois o compilador não

gera nenhuma mensagem de erro... A solução mais adequada é sempre avaliar os limites de um vetor antes de manipulá-lo.

A princípio este fato poderia parecer um defeito da linguagem, mas na verdade trata-se de um recurso muito poderoso do C. Poder manipular sem restrições todos os segmentos de memória é uma flexibilidade apreciada pelos programadores.

Programa Exemplo: O arquivo `e0702.cpp` contém um programa que mostra o acesso de elementos dentro e fora de um vetor. Note que o compilador não acusa nenhum erro de sintaxe!

7.3.2 Tamanho parametrizado

Na linguagem C não é possível, usando a sintaxe descrita acima, declarar um vetor com tamanho **variável**.

Exemplo: O trecho de código seguinte faz uma declaração **errada** de vetor.

```
...
int num;
puts("Quantos números?");
scanf("%d", &num);
float valor[num];  // declaração de vetor (errado!)
...
```

Mas é possível declarar um vetor com tamanho **parametrizado**: usando uma **constante simbólica**. Declaramos uma constante simbólica (parâmetro) com a diretiva `#define` no cabeçalho do programa e depois declaramos o vetor com esta constante simbólica como tamanho do vetor. Deste modo podemos alterar o número de elementos do vetor **antes** de qualquer **compilação** do programa. Esta é uma maneira simples de administrar o espaço de memória usado pelo programa, e também testar os limites de um vetor.

Programa Exemplo: O arquivo `e0703.cpp` contém um programa que mostra a declaração de um vetor com tamanho parametrizado. Mostra também o uso deste parâmetro como teste de limite do vetor. Compile este programa com outros valores para o parâmetro `MAX` e verifique que a execução do programa é alterada automaticamente.

No capitulo seguinte, seção ?, será vista uma maneira de declararmos um vetor com um número variável de elementos, usando ponteiros. Este tipo de declaração é chamada de *alocação dinâmica de memória.*

7.4 Passando Vetores para Funções

Vetores, assim como variáveis, podem ser usados como argumentos de funções. Vejamos como se declara uma função que recebe um vetor e como se passa um vetor para uma função.

Sintaxe: Na *passagem* de vetores para funções usamos a seguinte sintaxe:

```
nome_da_função(nome_do_vetor)
```

onde:

nome_da_função é o nome da função que se está chamando.

nome_do_vetor é o nome do vetor que queremos passar. Indicamos **apenas** o nome do vetor, **sem** índices.

Sintaxe: Na *declaração* de funções que recebem vetores:

```
tipo_função nome_função(tipo_vetor nome_vetor[]){
...
}
```

onde:

tipo_função é o tipo de retorno da função.

nome_função é o nome da função.

tipo_vetor é o tipo de elementos do vetor.

nome_vetor é o nome do vetor. Observe que depois do nome do vetor temos um índice vazio `[]` para indicar que estamos recebendo um vetor.

Exemplo: Observe o exemplo abaixo:

Na declaração da função:

```
float media(float vetor[],float N){  // declaração da função
  ...
}
```

Na chamada da função:

```
void main(){
  float valor[MAX]; // declaração do vetor
  ...
  med = media(valor, n); // passagem do vetor para a função
  ...
}
```

Programa Exemplo: O arquivo `e0704.cpp` contém um programa que mostra a passagem de vetores para funções.

Atenção: Ao contrário das variáveis comuns, o conteúdo de um vetor **pode ser modificado** pela função chamada. Isto significa que podemos passar um vetor para uma função e alterar os valores de seus elementos. Isto ocorre porque a passagem de **vetores** para funções é feita de modo especial dito *Passagem por endereço*. Uma abordagem mais detalhada deste procedimento será feita no capítulo ? sobre ponteiros.

Portanto devemos ter cuidado ao manipularmos os elementos de um vetor dentro de uma função para não modifica-los por descuido.

Programa Exemplo: O arquivo `e0705.cpp` contém um programa que mostra a modificação de um vetor por uma função. Neste caso a modificação é **desejada** pois queremos ordenar os elementos do vetor.

7.5 Vetores Multidimensionais

Vetores podem ter mais de uma dimensão, isto é, mais de um índice de referência. Podemos ter vetores de duas, três, ou mais dimensões. Podemos entender um vetor de duas dimensões (por exemplo) associando-o aos dados de um tabela.

Exemplo: Um vetor bidimensional pode ser visualizado através de uma **tabela**.

```
nota     0     1     2
```

```
0       8.4   7.4   5.7
1       6.9   2.7   4.9
2       4.5   6.4   8.6
3       4.6   8.9   6.3
4       7.2   3.6   7.7
```

Nesta tabela representamos as notas de 5 alunos em 3 provas diferentes (matemática, física e química, por exemplo). O nome `nota` é o nome do conjunto, assim podemos dizer que a nota do 3º aluno na 2ª prova é `6.4` ou representar `nota[2,1] = 6.4`

7.5.1 Declaração e inicialização

A declaração e inicialização de vetores de mais de uma dimensão é feita de modo semelhante aos vetores unidimensionais.

Sintaxe: A sintaxe para declaração de vetores multidimensionais é:

```
tipo nome[tam_1][tam_2]...[tam_N]={{lista},{lista},...{lista}};
```

onde:

`tipo` é o tipo dos elementos do vetor.

`nome` é o nome do vetor.

`[tam_1][tam_2]...[tam_N]` é o tamanho de cada dimensão do vetor.

`{{lista},{lista},...{lista}}` são as listas de elementos.

Exemplo: veja algumas declarações e inicializações de vetores de mais de uma dimensão. Observe que a inicialização de `nota` gera a tabela do exemplo do início desta seção.

```
float nota[5][3] = {{8.4,7.4,5.7},
                    {6.9,2.7,4.9},
                    {4.5,6.4,8.6},
                    {4.6,8.9,6.3},
                    {7.2,3.6,7.7}};
int tabela[2][3][2] = {{{10,15}, {20,25}, {30,35}},
                       {{40,45}, {50,55}, {60,65}};
```

Neste exemplo, `nota` é um vetor **duas** dimensões (`[][]`). Este vetor é composto de 5 vetores de 3 elementos cada. `tabela` é vetor de três dimensões (`[][][]`). Este vetor é composto de 2 vetores de 3 sub-vetores de 2 elementos cada.

7.5.2 Passagem de vetores multidimensionais para funções

A sintaxe para *passagem* de vetores multidimensionais para funções é semelhante a passagem de vetores unidimensionais: chamamos a função e passamos o **nome** do vetor, **sem** índices. A única mudança ocorre na declaração de funções que recebem vetores:

Sintaxe: Na *declaração* de funções que recebem vetores:

```
tipo_f função(tipo_v vetor[tam_1][tam_2]...[tam_n]){
...
}
```

Observe que depois do nome do vetor temos os índices com contendo os tamanhos de cada dimensão do vetor.

Exemplo: Observe o exemplo abaixo:

Na declaração da função:

```
int max(int vetor[5][7],int N, int M){  // declaração da função
...
}
```

Na chamada da função:

```
void main(){
  int valor[5][7]; // declaração do vetor
  ...
  med = media(valor, n); // passagem do vetor para a função
  ...
}
```

Programa Exemplo: O arquivo `e0706.cpp` contém um programa que mostra a manipulação de vetores bidimensionais: leitura de elementos, escrita, passagem para funções, etc.

Observações: Algumas observações a respeito de vetores multidimensionais podem ser feitas:

- Do mesmo modo que vetores unidimensionais, os vetores multidimensionais podem ter seus elementos modificados pela função chamada.
- Os índices dos vetores multidimensionais, também começam em `0`. Por exemplo: `vet[0][0]`, é o primeiro elemento do vetor.
- Embora uma tabela não seja a única maneira de visualizar um vetor bidimensional, podemos entender o **primeiro** índice do vetor como o índice de **linhas** da tabela e o **segundo** índice do vetor como índice das **colunas**.