Aula 02 - VARIÁVEIS UNIDIMENSIONAIS - VETORES

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**).

DECLARAÇÃO E INICIALIZAÇÃO 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. Os vetores precisam ser declarados, como quaisquer outras variáveis, para que o compilador conheça o tipo de matriz e reserve espaço de memória suficiente para armazená-la. Os elementos da matriz são guardados numa seqüência contínua de memória, isto é, um seguido ao outro.

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.
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.
```

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;
```

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'};
```

TAMANHO DE UM VETOR E SEGMENTAÇÃO DE MEMÓRIA

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.

Exemplo 1 – Leia um vetor de 5 elementos inteiros e:
a) exiba o vetor lido;
b) exiba a soma dos valores.

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

#define MAX 5         // definicao do parametro MAX
void main(){
      int i=3;
      int valor[MAX], soma=0;   // declaracao do vetor usando MAX

      clrscr();
      printf("Entre com %d numeros\n",MAX);
      for(i = 0; i < MAX; i++)
      { printf ("Vetor [%d] = ", i);
            scanf ("%d", &valor[i]);
            soma = soma + valor[i];
      }

      printf ("\n Elementos digitados foram: ");
      for (i=0; i<MAX; i++)
             printf ("%d  ", valor [i]);
      printf ("\n A soma e: %d", soma);
      getch();
}
```

Exercícios:

1. Faça um programa que leia 6 números e exiba:
a) o maior número lido;
b) o menor número lido;
c) quantos números são iguais ao 1º número lido.

2) Faça um programa que leia 8 números inteiros e exiba quantos são maiores que a média desses valores.

3) Faça um programa que leia n notas (máximo de 20) de alunos de uma colégio e exiba:
a) quantos alunos foram aprovados;
b) quantos alunos foram reprovados;
obs: faça a consistência das notas.

4) Leia um vetor A, com no máximo 10 elementos. Crie um vetor B da seguinte forma:
a) coloque os números pares primeiro ;
b) coloque os número impares logo após os pares;
c) exiba o vetor B.

Este exercício deve ser feito em casa

- Em uma cidade do interior, sabe-se que, de janeiro a abril de 1976 (121 dias), não ocorreu temperatura inferior a 15ºC nem superior a 40ºC. As temperaturas verificadas em cada dia estão disponíveis em uma unidade de entrada de dados.

Faça um programa que calcule e escreva:
a) a menor temperatura ocorrida
b) a maior temperatura ocorrida
c) a temperatura média
d) o número de dias nos quais a temperatura foi inferior a média das temperaturas.

ORDENAÇÃO DE VETORES

Algumas tarefas de programação são tão comuns, que, com o passar dos anos, foram desenvolvidos algoritmos altamente eficientes e padronizados para se encarregar dessas tarefas.

Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação de informações em um conjunto ordenado. Os métodos de ordenação classificam-se em dois grupos: se o arquivo a ser ordenado cabe todo na memória principal, então o método de ordenação é denominado **ordenação interna**, caso contrário, é denominado de **ordenação externa.**

PRINCÍPIOS GERAIS DE ORDENAÇÃO

Os algoritmos de ordenação apresentados comparam um elemento de um array com outro, e, se os dois elementos estiverem desordenados, os algoritmos trocam a ordem deles dentro do array. Esse processo é o seguinte:

```
if (a[i] > a[i+1])
{
        aux := a[i];
        a[i]  := a[i+1];
        a[i+1] := aux;
}
```

A primeira linha do código verifica se os dois elementos do array estão desordenados. Em geral, os arrays encontram-se em ordem quando o elemento atual é menor que o elemento seguinte. Se os elementos não estiverem adequadamente ordenados, ou seja, se o elemento atual for maior que o elemento seguinte, sua ordem será trocada. A troca exige uma variável de armazenamento temporário do mesmo tipo que a dos elementos do array que está sendo ordenado.

Exemplo 1: Leia um vetor de n elementos reais e exiba-o ordenado.

```
#include <conio.h>
#include <stdio.h>
#define MAX 10

void main()
{
   int i,n, bandeira;
   float vetor[MAX], aux;

   clrscr();
   // Leitura da quantidade de elementos
```

```
   n = 0;
   while ((n <= 0) || (n > MAX))
      { printf ("Entre com N: ");
            scanf ("%d", &n);
      }

   printf("Leitura dos elementos do Vetor\n");
   for(i = 0; i < n; i++)
      { printf ("vetor[%d] = ", i);
              scanf ("%f", &vetor[i]);
      }

   // ordenacao
   bandeira = 1;
   while (bandeira)
   { bandeira = 0;
     for (i=0; i<n-1; i++)
            if (vetor [i] > vetor [i+1])
                   { aux = vetor[i];
                     vetor[i] = vetor [i+1];
                     vetor[i+1] = aux;
                     bandeira = 1;
                   }
  }

// exibicao dos elementos ordenados
 printf (" \nElementos ordenados \n");
 i = 0;
 for (i=0; i < n; i++)
       printf ("%2.2f  ",vetor[i]);
 getch();
}
```

MÉTODO DE ORDENAÇÃO – BUBLE-SORT (BOLHA)

O algoritmo da ordenação BUBLE-SORT é simples de ser entendido - ele começa no final do array a ser ordenado e executa a ordenação na direção do início do array. A rotina compara cada elemento com seu precedente. Se os elementos estiverem desordenados, serão trocados. A rotina continua até alcançar o início do vetor.

Como a ordenação funciona de trás para frente através do array, comparando cada par de elementos adjacentes (próximos, vizinhos), o elemento de menor valor sempre “flutuará” no topo depois da primeira passagem. Após a segunda passagem, o segundo elemento de menor valor flutuará para a segunda posição do array, e assim por diante, até que o algoritmo tenha verificado uma vez cada elemento do array.

Exemplo 2: Altere o método de ordenação acima para o método bolha.

```
 #include <conio.h>
 #include <stdio.h>
 #define MAX 10

 void main()
 {
```

```
    int i,j,n;
    float vetor[MAX], aux;

    . . . .
    . . . .

// ordenacao pelo método bolha.
    for (j=1; j<n; j++)
          for (i=n-1; i>=j; i--)
                if (vetor[i] < vetor[i-1])
                      { aux = vetor[i];
                        vetor[i] = vetor[i-1];
                        vetor [i-1] =aux;
                      }
   . . . .
   . . . .
   getch();
}
```

MÉTODO DE ORDENAÇÃO POR INSERÇÃO

A idéia básica deste método é inserir um determinado registro em uma seqüência ordenada, de tal maneira que a seqüência resultante também permaneça em ordem. Desta forma, em cada passo, a partir de I=2, o i-ésimo item da seqüência fonte é apanhado e transferido para a seqüência destino, sendo inserido no seu devido lugar.

Exemplo 3: Altere o método de ordenação acima para o método inserção.

```
// ordenacao
     i = 0;
     while (i < n-1)
     { j = i+1;
       while (j < n)
          { if (vetor[i] > vetor[j])
                { aux = vetor[i];
                  vetor[i] = vetor[j];
                  vetor [j] =aux;
                }
            j++;
          }
       i++;
     }
```

PESQUISA EM VETORES (MÉTODOS DE LOCALIZAÇÃO)

Na programação, localizar significa procurar um determinado item dentro de um grupo de itens como, por exemplo, procurar um determinado número inteiro em um array de inteiros, encontrar o nome de uma

pessoa em um array de cadeias, e assim por diante. Os dois métodos de localização aqui apresentados – o seqüencial e o binário – chegam ao mesmo objetivo por caminhos diferentes.

LOCALIZAÇÃO SEQUENCIAL

O programa simplesmente começa do início do array a ser ordenado e compara cada elemento com o valor que está sendo procurado.

```
#include <conio.h>
#include <stdio.h>
#define MAX 10
void main()
{
    int i,n, pos, achou;
    float vetor[MAX],x;
    clrscr();
    // Leitura da quantidade de elementos
    n = 0;
    while ((n <= 0) || (n > MAX))
     { printf ("Entre com N: ");
       scanf ("%d", &n);
     }

    printf("Leitura dos elementos do Vetor\n");
    for(i = 0; i < n; i++)
     { printf ("vetor[%d] = ", i);
      scanf ("%f", &vetor[i]);
     }

    // elemento a ser pesquisado
    printf ("Elemento a ser pesquisado: ");
    scanf ("%f", &x);

   // pesquisa sequencial
    i = 0;
    achou = 0;
    while ((i<n) && (!achou))
     { if ( vetor[i] == x)
          {
             achou = 1;
             pos = i;
          }
       i++;
     }
    if (achou)
          printf ("O elemento %2.2f foi encontrado na posicao %d ",x,pos);
    else
          printf ("Elemento %2.2f nao encontrado",x);
    getch();
}
```

LOCALIZAÇÃO BINÁRIA

A localização binária representa um dos mais eficientes métodos de localização conhecidos e um grande aperfeiçoamento da localização seqüencial. Com um array de 100 elementos, por exemplo, uma localização seqüencial exige uma média de 50 comparações para encontrar um valor coincidente. A localização binária exige no máximo sete comparações e no mínimo quatro para alcançar a mesma meta. Quanto maior a lista, maior a eficiência relativa da localização binária.

Para efetuar uma localização binária, a lista deve estar ordenada. A localização começa comparando o elemento alvo com o elemento mediano do array. Se o elemento-alvo for maior que o elemento mediano, a localização prossegue na metade superior da lista. Se o valor-alvo for menor que o elemento mediano, o elemento-alvo estará na metade inferior.

```
#include <conio.h>
#include <stdio.h>
#define MAX 10

void main()
{
     int i,j, n, pos, achou, meio ,inicio, fim;
     float vetor[MAX],x, aux;

     clrscr();
     // Leitura da quantidade de elementos
     n = 0;
     while ((n <= 0) || (n>MAX))
     { printf ("Entre com N: ");
       scanf ("%d", &n);
     }

     printf("Leitura dos elementos do Vetor\n");
     for(i = 0; i < n; i++)
     { printf ("vetor[%d] = ", i);
       scanf ("%f", &vetor[i]);
     }

     // ordenacao do vetor - metodo insercao
     i = 0;
     while (i < n-1)
        while (j < n)   { j = i+1;

           { if (vetor[i] > vetor[j])
                  { aux = vetor[i];
                    vetor[i] = vetor[j];
                    vetor [j] =aux;
                  }
             j++;
           }
        i++;
      }

     // elemento a ser pesquisado

     printf ("Elemento a ser pesquisado: ");
     scanf ("%f", &x);
```

```
// pesquisa binaria
     achou = 0;
     inicio = 0;
     fim = n-1;
     while ((inicio <= fim) && (!achou))
           {
                 meio = (inicio + fim)/2;
                 if (vetor[meio] == x)
                       { achou = 1;
                         pos   = meio;
                       }
                 else if (x > vetor [meio])
                             inicio = meio + 1;
                      else
                             fim = meio - 1;
           }
     clrscr();
     printf ("Vetor ordenado\n");
     for (i=0; i<n; i++)
           printf ("vetor [%d] = %2.2f\n",i, vetor[i]);

     if (achou)
           printf ("\nO elemento %2.2f foi encontrado na posicao %d ",x,pos);
     else
           printf ("\nElemento %2.2f nao encontrado",x);
     getch();
}
```

Este exercício deve ser feito em casa

- **Intercalação** é o processo utilizado para construir uma tabela ordenada, de tamanho n + m, a partir de duas tabelas já ordenadas de tamanho n e m. Por exemplo, a partir das tabelas:

A =	1	3	6	7

e

B =	2	4	5

construímos a tabela

C =	1	2	3	4	5	6	7