Aula 8 – Vetores e Ponteiros

1. Introdução

Como já vimos, variáveis são abstrações que nos permitem acessar uma posição da memória do computador por meio de um nome simbólico que escolhemos. Às vezes, precisamos armazenar um grupo de variáveis de mesmo tipo sob o mesmo nome. Para isso foram criadas as variáveis indexadas.

- "Indexada" vem do fato das diversas variáveis serem individualizadas por índices;
- Variável com um índice = vetor; com dois índices = matriz. Os nomes vêm da matemática;
- Número de índices = dimensão. Não há limites, mas é raro uma dimensão maior que 2.

Primeiro veremos como utilizar variáveis indexadas de forma estática (número pré-definido de elementos). Em seguida, aprenderemos a definir o número de elementos de forma dinâmica, utilizando ponteiros.

- Ponteiros são variáveis que apontam para uma posição de memória qualquer;
- A partir desta posição podem haver um ou mais elementos do tipo do ponteiro;
- Vetores e ponteiros podem ser tratados de forma uniforme em C.

2. Variáveis indexadas (vetores e matrizes)

2.1. Declaração

```
// Use: tipo variavel[tamanho]
int notas[31];          // Um vetor com 31 valores inteiros.
float tabela[10][5];    // Uma matriz com 10 linhas e 5 colunas.
char nome[50];          // Um vetor de caracteres, ou seja, string!
```

2.2. Operações

- Não há operações sobre todo o conjunto, somente sobre seus indivíduos;
- Algumas linguagens de programação podem ter funções específicas (ex.: ordenar um vetor, comparar um vetor com outro, etc.);
- Não veremos nenhuma função deste tipo nesta aula.

2.3. Acesso aos indivíduos

- Usa-se o operador de indexação: `[ ]`:

```
int primeiraNota = notas[0];     // O 1o elemento tem índice zero.
int segundaNota = notas[1];      // O 2o tem índice 1, etc.
```

- Atenção: o primeiro elemento tem índice 0 (zero), o último tem índice (tamanho – 1);
- Quando individualizados, funcionam como variáveis normais:

```
tabela[1][4] = 1.4;          // Atribui valor à 2a linha, 5a coluna.
scanf("%d", &notas[30]);     // Lê a última nota.
printf("%f", tabela[0][0]); // Imprime o primeiro número da matriz.
```

- O programador deve tomar cuidado para não acessar índices inválidos: o compilador não emitirá um erro, mas muito provavelmente haverá um erro na hora da execução do programa:

```
tabela[1][5] = 1.5;          // 2a linha, 6a coluna??
scanf("%d", &notas[31]);     // Índice 31 = 32o elemento. Não existe!
```

2.4. Exemplo: ler 10 números e imprimi-los em ordem inversa, multiplicados por 2

```
#include <stdio.h>

main() {
     int i, numeros[10];

     printf("Informe 10 números inteiros:\n");
     for (i = 0; i < 10; i++)
          scanf("%d", &numeros[i]);

     printf("Dobro dos números informados em ordem inversa:\n");
     for (i = 9; i >= 0; i--) {
          numeros[i] = numeros[i] * 2;
          printf("%d\n", numeros[i]);
     }
}
```

2.5. Relembrando: *strings* são vetores de char

- Mencionado na aula sobre a linguagem C: char representa 1 caractere. Para formar *strings*, utilizamos vetores de char:

```
char nome[50];
```

- Relembrando as funções:
  - Para ler uma *string*, usa-se `gets(str);`
  - Para atribuir uma *string* a outra, usa-se `strcpy(strDestino, strOrigem);`
  - Para concatenar uma *string* a outra, usa-se `strcat(strDestino, strOrigem);`
  - Para saber o tamanho de uma *string*, usa-se `strlen(str);`
  - Para comparar uma *string* com outra, usa-se `strcmp(str1, str2)`: retorna 0 se iguais, < 0 se `str1 < str2`, > 0 se `str1 > str2`;
- Todas as funções acima requerem `#include <string.h>`.
- Exemplo:

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

main() {
     char umNome[30], outroNome[50];
     int tam;

     // Lê um nome.
     printf("Digite um nome: ");
     gets(umNome);

     // Copia-o para o outro vetor.
     strcpy(outroNome, umNome);

     // Informa o tamanho do nome digitado.
```

```
        tam = (int)strlen(outroNome);
        printf("O nome digitado possui %d caracteres.\n", tam);

        // Lê outro nome e adiciona ";" ao final de ambos.
        printf("Digite outro nome: ");
        gets(umNome);
        strcat(umNome, ";");
        strcat(outroNome, ";");

        // Compara os nomes e imprime em ordem alfabética.
        printf("Em ordem alfabética:\n");
        if (strcmp(umNome, outroNome) < 0)
            printf("%s\n%s\n", umNome, outroNome);
        else
            printf("%s\n%s\n", outroNome, umNome);
}
```

2.6. Inicialização de vetores e *strings*

- Quando declaramos variáveis, elas podem ser inicializadas para um valor fixo:

```
int valor = 10;
```

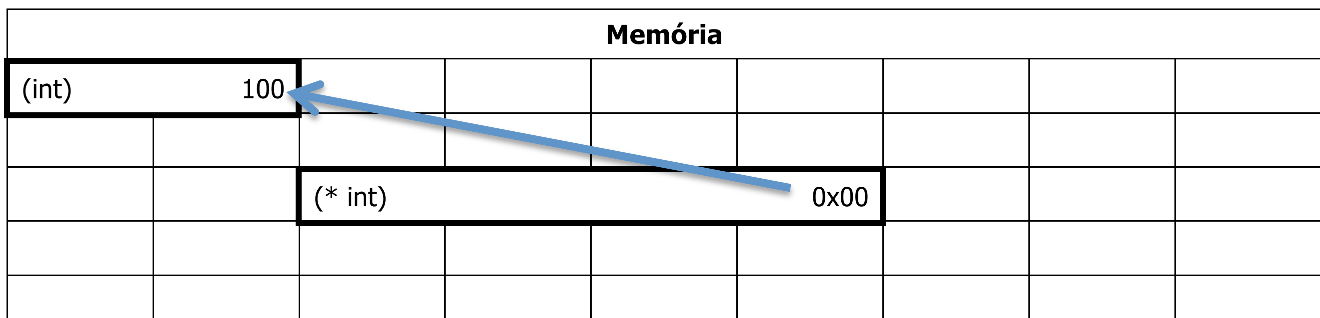
- O mesmo pode ser feito com vetores e *strings*:

```
float vetor[6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matriz[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char umNome[10] = { 'J', 'o', 'a', 'o', '\0' };
char outroNome[10] = "Joao";
char vetorStrings[3][10] = { "Joao", "Maria", "Jose" };

// O tamanho pode ser determinado automaticamente pelo C:
char mensagem[] = "Bom dia turma!";          // 14 chars
int matriz[][2] = { 1,2,2,4,3,6,4,8,5,10 };  // 5 x 2 int
```

3. Ponteiros

- Ponteiros são tipos especiais de variáveis que armazenam não um dado diretamente (um inteiro, um real, um caractere) mas sim o endereço de memória onde um dado se encontra;



- Ponteiros são declarados como se fossem variáveis normais do tipo desejado, porém usando um * para indicar que é um ponteiro:

```
int *ponteiroParaInt;
```

- Assim como variáveis normais, um ponteiro não inicializado (como o exemplo acima) possui um valor qualquer (lixo de memória) e, portanto, aponta para um local desconhecido. Usar um ponteiro assim pode causar graves erros no programa;
- Para inicializar um ponteiro, precisamos atribuir um endereço de memória a ele. Como não sabemos o endereço das variáveis, utilizamos o operador & (que já vimos no scanf!):

```
int umInteiro = 100;
int *ponteiroParaInt = &umInteiro;
```

- Com o ponteiro apontando para um endereço válido, podemos usá-lo. Para acessar o valor apontado pelo ponteiro, utilizamos novamente o *, que é o operador de resolução de referências:

```
int umInteiro = 100;
int *ponteiroParaInt = &umInteiro;
printf("%d\n", *ponteiroParaInt);           // Imprime 100.
*ponteiroParaInt = *ponteiroParaInt * 2;    // Note os dois usos de *
printf("%d\n", umInteiro);                  // Imprime 200!
printf("%p\n", ponteiroParaInt);            // Imprime o endereço.
```

- Como visto no exemplo acima, modificar o valor apontado pelo ponteiro obviamente modifica também o valor da variável original, cujo endereço colocamos no ponteiro;

3.1. Ponteiros como vetores

- Nos exemplos anteriores, o ponteiro apontava para um único valor de um determinado tipo;
- Podemos, no entanto, usar ponteiros como vetores, apontando para o primeiro de uma sequência de valores de um determinado tipo e, em seguida, navegar pelos demais valores;
- Veja no exemplo:

```
int i;
float vetor[6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
float *ponteiro = vetor;
for (i = 0; i < 6; i++) {
     printf("%.1f\n", *ponteiro);
     ponteiro++;
}
```

- No código acima, repare o seguinte:
  - O ponteiro recebe o valor de vetor, sem usar o & como anteriormente. Isso se dá porque vetores e ponteiros são tratados de forma uniforme (vetores são ponteiros);
  - O `printf()` imprime o valor apontado pelo ponteiro. Como sabemos que é ponteiro para `float`, ele imprime da primeira vez apenas o primeiro elemento;
  - Em seguida, `ponteiro++` faz com que o ponteiro aponte para o próximo elemento na memória. Sabendo que é um ponteiro pra `float`, sabemos quantos bytes pular para chegar ao próximo elemento;
- Para demonstrar a uniformidade de tratamento de vetores e ponteiros, veja no próximo código que o operador de indexação `[]` é usado no ponteiro. Repare também que `&vetor[0] == vetor`:

```
int i;
float vetor[6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
float *ponteiro = &vetor[0];
for (i = 0; i < 6; i++) printf("%.1f\n", ponteiro[i]);
```

3.2. Ponteiros para ponteiros e a função main()

- Ponteiros são variáveis especiais, mas são também variáveis. Portanto é possível criarmos ponteiros que apontam para outros ponteiros;
- Como vimos, *strings* são vetores de `char` e ponteiros e vetores são tratados de forma igual. Portanto, para termos um vetor de *strings* usando ponteiros será necessário um ponteiro para ponteiro para `char` (equivalente a uma matriz de `char`):

```
// A conversão de char[][] para char** deve ser feita em 2 etapas:
char vetorStrings[3][10] = { "Joao", "Maria", "Jose" };
char *vetorPonteiros[3];
char **ponteiro = vetorPonteiros;
int i;
for (i = 0; i < 3; i++) {
     vetorPonteiros[i] = vetorStrings[i];
     printf("%s\n", *ponteiro);
     ponteiro++;
}
```

- Ponteiro para ponteiro para char é usado pelo método `main()` para ler argumentos passados pela linha de comando:

```
#include <stdio.h>

main(int argc, char **argv) {
     int i;
     for (i = 0; i < argc; i++)
          printf("Argumento %d: %s\n", i, argv[i]);
}
```

- Assumindo que o programa acima foi salvo no arquivo `eco.c`:

```
$ gcc -o eco eco.c
$ ./eco Um Dois Três
Argumento 0: ./eco
Argumento 1: Um
Argumento 2: Dois
Argumento 3: Três
```

3.3. Alocação dinâmica de vetores

- Às vezes precisamos armazenar informações em vetores mas não sabemos a quantidade de informações a armazenar;
- Para isso, a biblioteca `<stdlib.h>` oferece uma função para alocação dinâmica de memória. Veja no exemplo abaixo:

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

```
main() {
     int i, qtd, *numeros;

     printf("Quantos números deseja informar? ");
     scanf("%d", &qtd);
     numeros = (int *)malloc(qtd * sizeof(int));

     printf("Informe %d números inteiros:\n", qtd);
     for (i = 0; i < qtd; i++)
          scanf("%d", &numeros[i]);

     printf("Dobro dos números informados em ordem inversa:\n");
     for (i = qtd - 1; i >= 0; i--) {
          numeros[i] = numeros[i] * 2;
          printf("%d\n", numeros[i]);
     }
}
```

- A função `malloc()` recebe como parâmetro o número de bytes a alocar;
- Como não temos certeza de quantos bytes um inteiro ocupa, usamos `sizeof(int)` para obter esta informação;
- Multiplicamos isso pela quantidade de inteiros que queremos no vetor (`qtd`);
- A função `malloc()` retorna um ponteiro genérico (pode ser usada para alocar dinamicamente vetores de qualquer tipo). Por isso precisamos converter o resultado para `(int *)`;
- O restante do programa é igual ao exemplo do início da aula, trocando o valor fixo `10` pela variável `qtd`, informada pelo usuário.

Exercícios – Vetores e Ponteiros

1) Elabore um algoritmo para realizar a soma de dois vetores, A e B, de números reais e de tamanho 5.

2) Estenda o algoritmo anterior para realizar a soma de duas matrizes, A e B, de números reais e de dimensão 3 x 5.

3) Reescreva o exercício 1 utilizando alocação dinâmica de vetores, ou seja: pergunte ao usuário o tamanho dos vetores, em seguida leia A e B e imprima sua soma.