AULA 19

VETORES

Até o momento, estudamos os tipos de dados básicos ou elementares da linguagem C de programação: `char`, `int`, `float` e `double`. Os tipos de dados básicos se caracterizam pelo fato que seus valores não podem ser decompostos. Por outro lado, se os valores de um tipo de dados podem ser decompostos ou subdivididos em valores mais simples, então o tipo de dados é chamado de **complexo**, **composto** ou **estruturado**. A organização desses valores e as relações estabelecidas entre eles determinam o que conhecemos como **estrutura de dados**. Estudaremos algumas dessas estruturas de dados em nossas aulas futuras, como por exemplo as listas lineares, pilhas e filas.

Nesta aula iniciaremos o estudo sobre variáveis compostas, partindo de uma variável conhecida como **variável composta homogênea unidimensional** ou simplesmente **vetor**. Características específicas da linguagem C no tratamento de vetores também serão abordadas.

19.1 Motivação

Como um exemplo da necessidade do uso da estrutura de dados conhecida como vetor, considere o seguinte problema:

> Dadas cinco notas de uma prova dos(as) estudantes de uma disciplina, calcular a média das notas da prova e a quantidade de estudantes que obtiveram nota maior que a média e a quantidade de estudantes que obtiveram nota menor que a média.

Uma tentativa natural e uma idéia inicial para solucionar esse problema consiste no uso de uma estrutura de repetição para acumular o valor das cinco notas informadas e o cálculo posterior da média destas cinco notas. Esses passos resolvem o problema inicial do cálculo da média das provas dos estudantes. Mas e a computação das quantidades de alunos que obtiveram nota maior e menor que a média já computada? Observe que após lidas as cinco notas e processadas para o cálculo da média em uma estrutura de repetição, a tarefa de encontrar as quantidades de estudantes que obtiveram nota superior e inferior à média é impossível, já que as notas dos estudantes não estão mais disponíveis na memória. Isto é, a menos que o(a) programador(a) peça ao usuário para informar as notas dos(as) estudantes novamente, não há como computar essas quantidades.

Apesar disso, ainda podemos resolver o problema usando uma estrutura seqüencial em que todas as cinco notas informadas pelo usuário ficam armazenadas na memória e assim podemos posteriormente computar as quantidades solicitadas consultando estes valores. Veja o programa 19.1.

Programa 19.1: Um programa para resolver o problema da seção 19.1.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int menor, maior;
5    float nota1, nota2, nota3, nota4, nota5, media;
6    printf("Informe as notas dos alunos: ");
7    scanf("%f%f%f%f%f%f", &nota1, &nota2, &nota3, &nota4, &nota5);
8    media = (nota1 + nota2 + nota3 + nota4 + nota5) / 5;
9    printf("\nMedia das provas: %2.2f\n", media);
10   menor = 0;
11   if (nota1 < media)
12     menor++;
13   if (nota2 < media)
14     menor++;
15   if (nota3 < media)
16     menor++;
17   if (nota4 < media)
18     menor++;
19   if (nota5 < media)
20     menor++;
21   maior = 0;
22   if (nota1 > media)
23     maior++;
24   if (nota2 > media)
25     maior++;
26   if (nota3 > media)
27     maior++;
28   if (nota4 > media)
29     maior++;
30   if (nota5 > media)
31     maior++;
32   printf("\nQuantidade de estudantes com nota inferior à média: %d\n", menor);
33   printf("\nQuantidade de estudantes com nota superior à média: %d\n", maior);
34   return 0;
35 }
```

A solução que apresentamos no programa 19.1 é de fato uma solução para o problema. Isto é, dadas como entrada as cinco notas dos(as) estudantes em uma prova, o programa computa de forma correta as saídas que esperamos, ou seja, as quantidades de estudantes com nota inferior e superior à média da prova. No entanto, o que aconteceria se a sala de aula tivesse mais estudantes, como por exemplo 100? Ou 1.000 estudantes? Certamente, a estrutura seqüencial não seria apropriada para resolver esse problema, já que o programador teria de digitar centenas ou milhares de linhas repetitivas, incorrendo inclusive na possibilidade de propagação de erros e na dificuldade de encontrá-los. E assim como esse, outros problemas não podem ser resolvidos sem uma extensão na maneira de armazenar e manipular as entradas de dados.

19.2 Definição

Uma **variável composta homogênea unidimensional**, ou simplesmente um **vetor**, é uma estrutura de armazenamento de dados que se dispõe de forma linear na memória e é usada para armazenar valores de um mesmo tipo. Um vetor é então uma lista de células na memória de tamanho fixo cujos conteúdos são do mesmo tipo básico. Cada uma dessas células armazena um, e apenas um, valor. Cada célula do vetor tem um **endereço** ou **índice** através do qual podemos referenciá-la. O termo 'variável composta homogênea unideminsional' é bem explícito e significa que temos uma: (*i*) variável, cujos valores podem ser modificados durante a execução de um programa; (*ii*) composta, já que há um conjunto de valores armazenado na variável; (*iii*) homogênea, pois os valores armazenados na variável composta são todos de um mesmo tipo básico; e (*iv*) unidimensional, porque a estrutura de armazenamento na variável composta homogênea é linear. No entanto, pela facilidade, usamos o termo 'vetor' com o mesmo significado.

A forma geral de declaração de um vetor é dada a seguir:

```
tipo identificador[tamanho];
```

onde:

- `tipo` é um tipo de dados conhecido ou definido pelo(a) programador(a);
- `identificador` é o nome do vetor, fornecido pelo(a) programador(a); e
- `tamanho` é a quantidade de células a serem disponibilizadas para uso no vetor.

Por exemplo, a declaração a seguir

```
float nota[100];
```

faz com que 100 células contíguas de memória sejam reservadas, cada uma delas podendo armazenar números de ponto flutuante do tipo `float`. A referência a cada uma dessas células é realizada pelo identificador do vetor `nota` e por um índice. Na figura 19.1 mostramos o efeito da declaração do vetor `nota` na memória de um computador.

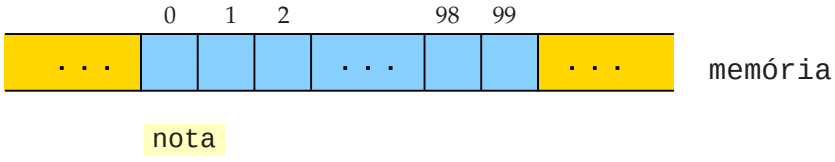

Figura 19.1: Vetor com 100 posições.

É importante observar que, na linguagem C, a primeira célula de um vetor tem índice 0, a segunda célula tem índice 1, a terceira tem índice 2 e assim por diante. Para referenciar o conteúdo da célula 0 do vetor `nota`, devemos usar o identificador do vetor e o índice 0, envolvido por colchetes, isto é, `nota[0]`. Assim, o uso de `nota[35]` em uma expressão qualquer referencia o trigésimo sexto elemento do vetor `nota`. Podemos, também, usar uma variável do tipo inteiro como índice de um vetor ou ainda uma expressão aritmética do tipo inteiro. Por exemplo, `nota[i]` acessa a $(i+1)$-ésima célula do vetor `nota`, supondo que a variável `i` é do tipo inteiro. Ou ainda, podemos fazer `nota[2*i+j]` para acessar a posição de índice $2i+j$ do vetor `nota`, supondo que `i` e `j` são variáveis do tipo inteiro é contêm valores compatíveis de tal forma que o resultado da expressão aritmética $2i+j$ seja um valor no intervalo de índices válido para o vetor `nota`. O compilador da linguagem C não verifica de antemão se os limites dos índices de um vetor estão corretos, trabalho que deve ser realizado pelo(a) programador(a).

Um erro comum, aparentemente inocente, mas que pode ter causas desastrosas, é mostrado no trecho de código abaixo:

```
int A[10], i;
for (i = 1; i <= 10; i++)
  A[i] = 0;
```

Alguns compiladores podem fazer com que a estrutura de repetição `for` acima seja executada infinitamente. Isso porque quando a variável `i` atinge o valor `10`, o programa armazena o valor `0` em `A[10]`. Observe, no entanto, que `A[10]` não existe e assim `0` é armazenado no compartimento de memória que sucede `A[9]`. Se a variável `i` ocorre na memória logo após `A[9]`, como é bem provável, então `i` receberá o valor `0` fazendo com que o laço inicie novamente.

19.3 Inicialização

Podemos atribuir valores iniciais a quaisquer variáveis de qualquer tipo básico no momento de suas respectivas declarações. Até o momento, não havíamos usado declarações e inicializações em conjunto. O trecho de código abaixo mostra declarações e inicializações simultâneas de variáveis de tipos básicos:

```
char c = 'a';
int num, soma = 0;
float produto = 1.0, resultado;
```

No exemplo acima, as variáveis `c`, `soma` e `produto` são inicializadas no momento da declaração, enquanto que `num` e `resultado` são apenas declaradas.

Apesar de, por alguns motivos, não termos usado declarações e inicializações simultâneas com variáveis de tipos básicos, essa característica da linguagem C é muito favorável quando tratamos de variáveis compostas. Do mesmo modo, podemos atribuir um valor inicial a um vetor no momento de sua declaração. As regras para declaração e atribuição simultâneas para vetores são um pouco mais complicadas, mas veremos as mais simples no momento. A forma mais comum de se fazer a inicialização de um vetor é através de uma lista de expressões constantes envolvidas por chaves e separadas por vírgulas, como no exemplo abaixo:

```
int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Se o inicializador tem menos elementos que a capacidade do vetor, os elementos restantes são inicializados com o valor 0 (zero), como abaixo:

```
int A[10] = {1, 2, 3, 4};
```

O resultado na memória é equivalente a termos realizado a inicialização abaixo:

```
int A[10] = {1, 2, 3, 4, 0, 0, 0, 0, 0, 0};
```

No entanto, não é permitido que o inicializador tenha mais elementos que a quantidade de compartimentos do vetor.

Podemos então facilmente inicializar um vetor todo com zeros da seguinte maneira:

```
int A[10] = {0};
```

Se um inicializador está presente em conjunto com a declaração de um vetor, então o seu tamanho pode ser omitido, como mostrado a seguir:

```
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

O compilador então interpreta que o tamanho do vetor `A` é determinado pela quantidade de elementos do seu inicializador. Isso significa que, após a execução da linha do exemplo de código acima, o vetor `A` tem 10 compartimentos de memória, do mesmo modo como se tivéssemos especificado isso explicitamente, como no primeiro exemplo.

19.4 Exemplo com vetores

Vamos agora resolver o problema da seção 19.1 do cálculo das médias individuais de estudantes e da verificação da quantidade de estudantes com média inferior e também superior à média da classe. Solucionaremos esse problema construindo um programa que usa vetores e veremos como nossa solução se mostra muito mais simples, mais eficiente e facilmente extensível para qualquer quantidade de estudantes que se queira. Vejamos então o programa 19.2 a seguir.

Programa 19.2: Solução do problema proposto na seção 19.1 usando um vetor.

```
#include <stdio.h>
int main(void)
{
  int i, menor, maior;
  float nota[5], soma, media;
  for (i = 0; i < 5; i++) {
    printf("Informe a nota do(a) estudante %d: ", i+1);
    scanf("%f", &nota[i]);
  }
  soma = 0.0;
  for (i = 0; i < 5; i++)
    soma = soma + nota[i];
  media = soma / 5;
  menor = 0;
  maior = 0;
  for (i = 0; i < 5; i++) {
    if (nota[i] < media)
      menor++;
    if (nota[i] > media)
      maior++;
  }
  printf("\nMedia das provas: %2.2f\n", media);
  printf("Quantidade de alunos com nota inferior à média: %d\n", menor);
  printf("Quantidade de alunos com nota superior à média: %d\n", maior);
  return 0;
}
```

Observe que o programa 19.2 é mais conciso e de mais fácil compreensão que o programa 19.1. O programa é também mais facilmente extensível, no sentido que muito poucas modificações são necessárias para que esse programa possa solucionar problemas semelhantes com outras quantidades de estudantes. Isto é, se a turma tem 5, 10 ou 100 estudantes, ou ainda um número desconhecido n que será informado pelo usuário durante a execução do programa, uma pequena quantidade de esforço será necessária para alteração de poucas linhas do programa.

19.5 Macros para constantes

Em geral, quando um programa contém constantes, uma boa idéia é dar nomes a essas constantes. Podemos atribuir um nome ou identificador a uma constante usando a definição de uma **macro**. Uma macro é definida através da diretiva de pré-processador `#define` e tem o seguinte formato geral:

```
#define identificador constante
```

onde `#define` é uma diretiva do pré-processador da linguagem C e `identificador` é um nome associado à `constante` que vem logo a seguir. Observe que, por ser uma diretiva do pré-processador, a linha contendo uma definição de uma macro não é finalizada por `;`. Em geral, a definição de uma macro ocorre logo no início do programa, após as diretivas `#include` para inclusão de cabeçalhos de bibliotecas de funções. Além disso, o identificador de uma macro é, preferencial mas não obrigatoriamente, descrito em letras maiúsculas.

Exemplos de macros são apresentados a seguir:

```
#define CARAC 'a'
#define NUMERADOR 4
#define MIN -10000
#define TAXA 0.01567
```

Quando um programa é compilado, o pré-processador troca cada macro definida no código pelo valor que ela representa. Depois disso, um segundo passo de compilação é executado. O programa 14.5 pode ser reescrito com o uso de uma macro, como podemos ver no programa 19.3, onde uma macro com identificador `PI` é definida e usada no código.

Programa 19.3: Cálculo da área do círculo.

```
1  #include <stdio.h>
2  #define PI 3.141592f
3  int main(void)
4  {
5    double raio, area;
6    printf("Digite o valor do raio: ");
7    scanf("%lf", &raio);
8    area = PI * raio * raio;
9    printf("A área do círculo de raio %f é %f\n", raio, area);
10   return 0;
11 }
```

O uso de macros com vetores é bastante útil porque, como já vimos, um vetor faz alocação estática da memória, o que significa que em sua declaração ocorre uma reserva prévia de um número fixo de compartimentos de memória. Por ser uma alocação estática, não há possibilidade de aumento ou diminuição dessa quantidade após a execução da linha de código contendo a declaração do vetor. O programa 19.2 pode ser ainda modificado como no programa 19.4 com a inclusão de uma macro que indica a quantidade de notas a serem processadas.

Programa 19.4: Solução do problema proposto na seção 19.1 usando uma macro e um vetor.

```
1  #include <stdio.h>
2  #define MAX 5
3  int main(void)
4  {
5    int i, menor, maior;
6    float nota[MAX], soma, media;
7    for (i = 0; i < MAX; i++) {
8      printf("Informe a nota do(a) estudante %d: ", i+1);
9      scanf("%f", &nota[i]);
10   }
11   soma = 0.0;
12   for (i = 0; i < MAX; i++)
13     soma = soma + nota[i];
14   media = soma / MAX;
15   menor = 0;
16   maior = 0;
17   for (i = 0; i < MAX; i++) {
18     if (nota[i] < media)
19       menor++;
20     if (nota[i] > media)
21       maior++;
22   }
23   printf("\nMedia das provas: %2.2f\n", media);
24   printf("Quantidade de estudantes com nota inferior à média: %d\n", menor);
25   printf("Quantidade de estudantes com nota superior à média: %d\n", maior);
26   return 0;
27 }
```

A macro `MAX` é usada quatro vezes no programa 19.4: na declaração do vetor `nota`, nas expressões relacionais das duas estruturas de repetição `for` e no cálculo da média. A vantagem de se usar uma macro é que, caso seja necessário modificar a quantidade de notas do programa por novas exigências do usuário, isso pode ser feito rápida e facilmente em uma única linha do código, onde ocorre a definição da macro.

Voltaremos a discutir mais sobre macros, sobre a diretiva `#define` e também sobre outras diretivas do pré-processador na aula 39.

Exercícios

19.1 Dada uma seqüência de n números inteiros, com $1 \leqslant n \leqslant 100$, imprimi-la em ordem inversa à de leitura.

Programa 19.5: Programa para solucionar o exercício 19.1.

```
1  #include <stdio.h>
2  #define MAX 100
3  int main(void)
4  {
5    int i, n, A[MAX];
6    printf("Informe n: ");
7    scanf("%d", &n);
8    for (i = 0; i < n; i++) {
9      printf("Informe o número %d: ", i+1);
10     scanf("%d", &A[i]);
11   }
12   printf("Números na ordem inversa da leitura:\n");
13   for (i = n-1; i >= 0; i--)
14     printf("%d  ", A[i]);
15   printf("\n");
16   return 0;
17 }
```

19.2 Uma prova consta de 30 questões, cada uma com cinco alternativas identificadas pelas letras A, B, C, D e E. Dado o cartão gabarito da prova e o cartão de respostas de n estudantes, com $1 \leqslant n \leqslant 100$, computar o número de acertos de cada um dos estudantes.

19.3 Tentando descobrir se um dado era viciado, um dono de cassino o lançou n vezes. Dados os n resultados dos lançamentos, determinar o número de ocorrências de cada face.

19.4 Um jogador viciado de cassino deseja fazer um levantamento estatístico simples sobre uma roleta. Para isso, ele fez n lançamentos nesta roleta. Sabendo que uma roleta contém 37 números (de 0 a 36), calcular a freqüência de cada número desta roleta nos n lançamentos realizados.