5. Vetores e alocação dinâmica

W. Celes e J. L. Rangel

5.1. Vetores

A forma mais simples de estruturarmos um conjunto de dados é por meio de vetores. Como a maioria das linguagens de programação, C permite a definição de vetores. Definimos um vetor em C da seguinte forma:

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

A declaração acima diz que `v` é um vetor de inteiros dimensionado com 10 elementos, isto é, reservamos um espaço de memória **contínuo** para armazenar 10 valores inteiros. Assim, se cada `int` ocupa 4 bytes, a declaração acima reserva um espaço de memória de 40 bytes, como ilustra a figura abaixo.

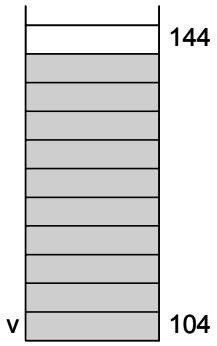

Figura 5.1: Espaço de memória de um vetor de 10 elementos inteiros.

O acesso a cada elemento do vetor é feito através de uma indexação da variável `v`. Observamos que, em C, a indexação de um vetor varia de zero a *n-1*, onde *n* representa a dimensão do vetor. Assim:

```
v[0]   →   acessa o primeiro elemento de v
v[1]   →   acessa o segundo elemento de v
...
v[9]   →   acessa o último elemento de v
```

Mas:

```
v[10]   →   está ERRADO (invasão de memória)
```

Para exemplificar o uso de vetores, vamos considerar um programa que lê 10 números reais, fornecidos via teclado, e calcula a média e a variância destes números. A média e a variância são dadas por:

$$m = \frac{\sum x}{N}, \quad v = \frac{\sum (x-m)^2}{N}$$

Uma possível implementação é apresentada a seguir.

```
/* Cálculo da media e da variância de 10 números reais */

#include <stdio.h>

int main ( void )
{
   float v[10];      /* declara vetor com 10 elementos */
   float med, var;   /* variáveis para armazenar a média e a variância */
   int i;            /* variável usada como índice do vetor */

   /* leitura dos valores */
   for (i = 0; i < 10; i++)       /* faz índice variar de 0 a 9 */
      scanf("%f", &v[i]);         /* lê cada elemento do vetor */

   /* cálculo da média */
   med = 0.0;                        /* inicializa média com zero */
   for (i = 0; i < 10; i++)
      med = med + v[i];              /* acumula soma dos elementos */
   med = med / 10;                   /* calcula a média */

   /* cálculo da variância */
   var = 0.0;                         /* inicializa variância com zero */
   for ( i = 0; i < 10; i++ )
     var = var+(v[i]-med)*(v[i]-med); /* acumula quadrado da diferença */
   var = var / 10;                    /* calcula a variância */

   printf ( "Media = %f   Variancia = %f  \n", med, var );
   return 0;
}
```

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

Na verdade, existe uma associação forte entre vetores e ponteiros, pois se existe a declaração:

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

a variável `v`, que representa o vetor, é uma constante que armazena o endereço inicial do vetor, isto é, `v`, sem indexação, aponta para o primeiro elemento do vetor.

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 isto, num vetor 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 último 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)` (é lógico que a forma indexada é mais clara e adequada). 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.

Os vetores também podem ser inicializados na declaração:

```
int v[5] = { 5, 10, 15, 20, 25 };
```

ou simplesmente:

```
int v[] = { 5, 10, 15, 20, 25 };
```

Neste último caso, a linguagem dimensiona o vetor pelo número de elementos inicializados.

Passagem de vetores para funções

Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor. Se passarmos 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 acima, usando funções separadas para o cálculo da média e da variância. (Aqui, usamos ainda os operadores de atribuição += para acumular as somas.)

```
/* Cálculo da media e da variância de 10 reais (segunda versão) */

#include <stdio.h>

/* Função para cálculo da média */
float media (int n, float* v)
{
   int i;
```

```
   float s = 0.0;
   for (i = 0; i < n; i++)
      s += v[i];
   return s/n;
}

/* Função para cálculo da variância */
float variancia (int n, float* v, float m)
{
   int i;
   float s = 0.0;
   for (i = 0; i < n; i++)
      s += (v[i] - m) * (v[i] - m);
   return s/n;
}

int main ( void )
{
   float v[10];
   float med, var;
   int i;

   /* leitura dos valores */
   for ( i = 0; i < 10; i++ )
      scanf("%f", &v[i]);

   med = media(10,v);
   var = variancia(10,v,med);

   printf ( "Media = %f   Variancia = %f  \n", med, var);
   return 0;
}
```

Observamos ainda 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 exemplo abaixo ilustra:

```
/* Incrementa elementos de um vetor */

#include <stdio.h>

void incr_vetor ( int n, int *v )
{
   int i;
   for (i = 0; i < n; i++)
      v[i]++;
}

int main ( void )
{
   int a[ ] = {1, 3, 5};
   incr_vetor(3, a);
   printf("%d %d %d \n", a[0], a[1], a[2]);
   return 0;
}
```

A saída do programa é `2 4 6`, pois os elementos do vetor serão incrementados dentro da função.

5.2. Alocação dinâmica

Até aqui, na declaração de um vetor, foi preciso dimensioná-lo. Isto nos obrigava a saber, de antemão, quanto espaço seria necessário, isto é, tínhamos que prever o número máximo de elementos no vetor durante a codificação. Este pré-dimensionamento do vetor é um fator limitante. Por exemplo, se desenvolvermos um programa para calcular a média e a variância das notas de uma prova, teremos que prever o número máximo de alunos. Uma solução é dimensionar o vetor com um número absurdamente alto para não termos limitações quando da utilização do programa. No entanto, isto levaria a um desperdício de memória que é inaceitável em diversas aplicações. Se, por outro lado, formos modestos no pré-dimensionamento do vetor, o uso do programa fica muito limitado, pois não conseguiríamos tratar turmas com o número de alunos maior que o previsto.

Felizmente, a linguagem C oferece meios de requisitarmos espaços de memória em tempo de execução. Dizemos que podemos alocar memória dinamicamente. Com este recurso, nosso programa para o cálculo da média e variância discutido acima pode, em tempo de execução, consultar o número de alunos da turma e então fazer a alocação do vetor dinamicamente, sem desperdício de memória.

Uso da memória

Informalmente, podemos dizer que existem três maneiras de reservarmos espaço de memória para o armazenamento de informações. A primeira delas é através do uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado. A segunda maneira é através do uso de variáveis locais. Neste caso, como já discutimos, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Por este motivo, a função que chama não pode fazer referência ao espaço local da função chamada. As variáveis globais ou locais podem ser simples ou vetores. Para os vetores, precisamos informar o número máximo de elementos, caso contrário o compilador não saberia o tamanho do espaço a ser reservado.

A terceira maneira de reservarmos memória é requisitando ao sistema, em tempo de execução, um espaço de um determinado tamanho. Este espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa. Por isso, podemos alocar dinamicamente um espaço de memória numa função e acessá-lo em outra. A partir do momento que liberarmos o espaço, ele estará disponibilizado para outros usos e não podemos mais acessá-lo. Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar.

Apresentamos abaixo um *esquema didático* que ilustra de maneira fictícia a distribuição do uso da memória pelo sistema operacional[1].

[1] A rigor, a alocação dos recursos é bem mais complexa e varia para cada sistema operacional.

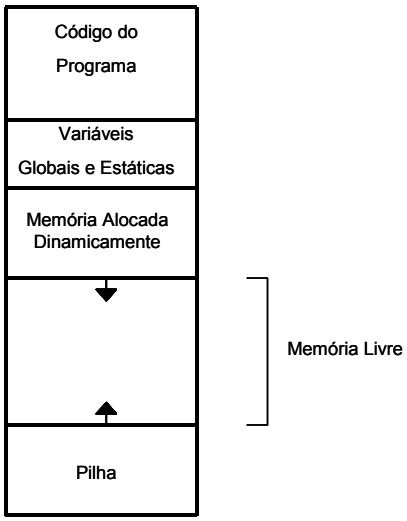

Figura 5.2: Alocação esquemática de memória.

Quando requisitamos ao sistema operacional para executar um determinado programa, o código em linguagem de máquina do programa deve ser carregado na memória, conforme discutido no primeiro capítulo. O sistema operacional reserva também os espaços necessários para armazenarmos as variáveis globais (e estáticas) existentes no programa. O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Cada vez que uma determinada função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Este espaço pertence à pilha de execução e, quando a função termina, é desempilhado. A parte da memória não ocupada pela pilha de execução pode ser requisitada dinamicamente. Se a pilha tentar crescer mais do que o espaço disponível existente, dizemos que ela "estourou" e o programa é abortado com erro. Similarmente, se o espaço de memória livre for menor que o espaço requisitado dinamicamente, a alocação não é feita e o programa pode prever um tratamento de erro adequado (por exemplo, podemos imprimir a mensagem "Memória insuficiente" e interromper a execução do programa).

Funções da biblioteca padrão

Existem funções, presentes na biblioteca padrão *stdlib*, que permitem alocar e liberar memória dinamicamente. A função básica para alocar memória é `malloc`. Ela recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Para exemplificar, vamos considerar a alocação dinâmica de um vetor de inteiros com 10 elementos. Como a função `malloc` retorna o endereço da área alocada e, neste exemplo, desejamos armazenar valores inteiros nessa área, devemos declarar um ponteiro de inteiro para receber o endereço inicial do espaço alocado. O trecho de código então seria:

```
int *v;

v = malloc(10*4);
```

Após este comando, se a alocação for bem sucedida, `v` armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros. Podemos,

então, tratar `v` como tratamos um vetor declarado estaticamente, pois, se `v` aponta para o inicio da área alocada, podemos dizer que `v[0]` acessa o espaço para o primeiro elemento que armazenaremos, `v[1]` acessa o segundo, e assim por diante (até `v[9]`).

No exemplo acima, consideramos que um inteiro ocupa 4 bytes. Para ficarmos independentes de compiladores e máquinas, usamos o operador `sizeof( )`.

```
v = malloc(10*sizeof(int));
```

Além disso, devemos lembrar que a função `malloc` é usada para alocar espaço para armazenarmos valores de qualquer tipo. Por este motivo, `malloc` retorna um ponteiro genérico, para um tipo qualquer, representado por `void*`, que pode ser convertido automaticamente pela linguagem para o tipo apropriado na atribuição. No entanto, é comum fazermos a conversão explicitamente, utilizando o operador de molde de tipo (*cast*). O comando para a alocação do vetor de inteiros fica então:

```
v = (int *) malloc(10*sizeof(int));
```

A figura abaixo ilustra de maneira esquemática o que ocorre na memória:

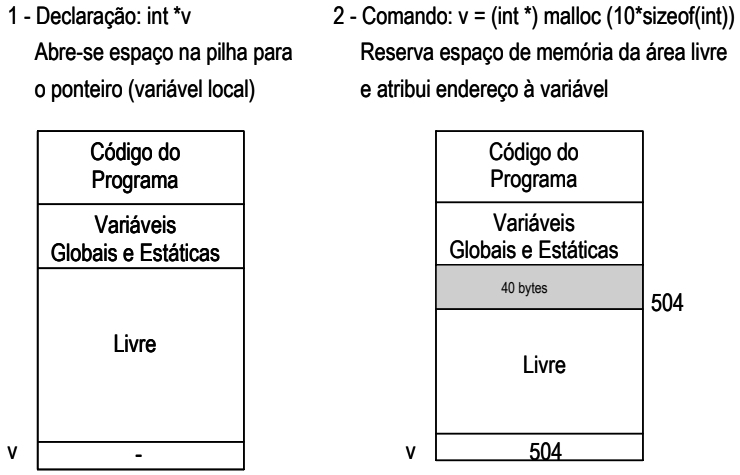

Figura 5.3: Alocação dinâmica de memória.

Se, porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo `NULL`, definido em *stdlib.h*). Podemos cercar o erro na alocação do programa verificando o valor de retorno da função `malloc`. Por exemplo, podemos imprimir uma mensagem e abortar o programa com a função `exit`, também definida na *stdlib*.

```
…
v = (int*) malloc(10*sizeof(int));
if (v==NULL)
{
   printf("Memoria insuficiente.\n");
   exit(1);  /* aborta o programa e retorna 1 para o sist. operacional */
}
…
```

Para liberar um espaço de memória alocado dinamicamente, usamos a função `free`. Esta função recebe como parâmetro o ponteiro da memória a ser liberada. Assim, para liberar o vetor `v`, fazemos:

```
free (v);
```

Só podemos passar para a função `free` um endereço de memória que tenha sido alocado dinamicamente. Devemos lembrar ainda que não podemos acessar o espaço na memória depois que o liberamos.

Para exemplificar o uso da alocação dinâmica, alteraremos o programa para o cálculo da média e da variância mostrado anteriormente. Agora, o programa lê o número de valores que serão fornecidos, aloca um vetor dinamicamente e faz os cálculos. Somente a função principal precisa ser alterada, pois as funções para calcular a média e a variância anteriormente apresentadas independem do fato de o vetor ter sido alocado estática ou dinamicamente.

```
/* Cálculo da média e da variância de n reais */

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

...

int main ( void )
{
   int i, n;
   float *v;
   float med, var;

   /* leitura do número de valores */
   scanf("%d", &n);
   /* alocação dinâmica */
   v = (float*) malloc(n*sizeof(float));
   if (v==NULL)    {
      printf("Memoria insuficiente.\n”);
      return 1;
   }
   /* leitura dos valores */
   for (i = 0; i < n; i++)
      scanf("%f", &v[i]);
   med = media(n,v);
   var = variancia(n,v,med);
   printf("Media = %f    Variancia = %f  \n", med, var);
   /* libera memória */
   free(v);
   return 0;
}
```