CAPÍTULO 6

PONTEIROS

6.1 Definição

Uma variável declarada como ponteiro armazena um endereço de memória. A declaração

```
int *a;
```

indica que a variável a é um ponteiro que aponta para um inteiro, ou seja, armazena o endereço de uma variável inteira.

Dois operadores são utilizados para se trabalhar com ponteiros. Um é o operador de endereço (**&**), que retorna o endereço de memória da variável. Este operador já foi utilizado em capítulos anteriores. O outro é o operador indireto (*****) que é o complemento de (*****) e retorna o conteúdo da variável existente no endereço (ponteiro). O exemplo a seguir ilustra a utilização desses operadores.

```
#include <stdio.h>

void main(void)
{
  int* a;
  int b = 2;
  int c;

  a = &b;
  c = b;
  printf("%d %d %d\n", *a, b, c);
  b = 3;
  printf("%d %d %d\n", *a, b, c);
}

SAIDA
2 2 2
3 3 2
```

No exemplo acima, as variáveis *b* e *c* são declaradas como inteiras, ou seja, cada uma delas ocupa dois bytes, em endereços de memória diferentes. A variável *a* é declarada como um ponteiro que aponta para um inteiro. O comando `a = &b` indica que a variável *a* armazenará o endereço da variável *b*. Já o comando `c = b` indica que a variável *c*, que fica armazenada em outro endereço de memória, guardará o valor da variável *b*.

A primeira linha de impressão nos fornece o conteúdo do ponteiro *a*, obtido através do operador indireto (*****), e o valor das variáveis *b* e *c*. Note que o conteúdo do ponteiro *a* é o valor da variável *b*, já que ele aponta para o endereço da variável.

Ao alterarmos o valor da variável *b*, o conteúdo de seu endereço é alterado. A variável *c*, por ocupar outro endereço de memória, permanece inalterada. Já o conteúdo do ponteiro *a* foi alterado, já que aponta para o endereço da variável *b*.

6.2 Passagem de Argumentos por Endereço

Até o momento, utilizamos funções e seus argumentos em todos os casos foram passados por valor, isto é, é criada uma cópia da variável dentro da função. Nesta seção veremos a passagem de argumentos por endereço, onde a própria variável é utilizada na função, e não uma cópia dela.

Suponha o seguinte programa:

```
#include <stdio.h>

void troca(int a, int b)
{
  int aux;

  aux = a;
  a = b;
  b = aux;
}

void main(void)
{
  int a = 2;
  int b = 3;

  printf("%d %d\n", a, b);
  troca(a, b);
  printf("%d %d\n", a, b);
}
```

A função troca() tem como objetivo receber duas variáveis inteiras e trocar o seu valor. Escrita da maneira proposta no exemplo acima, ao chamarmos a função, são criadas cópias das variáveis *a* e *b* dentro da função troca(). Estas cópias, que só existem dentro da função, tem o seu valor trocado, mas isso não implica nenhuma alteração nas variáveis *a* e *b* da função main().

Para que possamos alterar o valor de argumentos dentro de uma função, é necessário que estes argumentos sejam passados por endereço. Desta forma, a função trabalhará com a própria variável fornecida como argumento. O programa é reescrito como:

```
#include <stdio.h>

void troca(int* pa, int* pb)
{
  int aux;

  aux = *pa;
  *pa = *pb;
  *pb = aux;
}

void main(void)
{
  int a = 2;
  int b = 3;

  printf("%d %d\n", a, b);
  troca(&a, &b);
  printf("%d %d\n", a, b);
}
```

Percebem-se algumas alterações, descritas a seguir.

- a função troca() passa a receber argumentos do tipo **int***, ao invés de **int**. Isto é necessário para que a função receba o endereço das variáveis fornecidas, e possa alterar-lhes o valor;
- dentro da função, as variáveis *a* e *b* são tratadas com o operador indireto (*****), para que possamos trabalhar com seu conteúdo;
- Na chamada da função troca(), são passados como argumentos os endereços das variáveis, através do operador endereço (**&**).

Neste curso nós já passamos argumentos para funções através de seu endereço. Isto foi utilizado na função scanf(). O motivo para utilizarmos o operador endereço (**&**) na função scanf() e não o utilizarmos na função printf() é que somente a primeira altera o valor das variáveis que foram passadas como argumento.

6.3 Operações com ponteiros

A linguagem C oferece cinco operações básicas que podem ser executadas em ponteiros. O próximo programa mostra estas possibilidades. Para mostrar o resultado de cada operação, o programa imprimirá o valor do ponteiro (que é um endereço), o valor armazenado na variável apontada e o endereço do próprio ponteiro.

```
#include <stdio.h>

void main(void)
{
  int x = 5;
  int y = 6;
  int* px;
  int* py;

  /* Atribuicao de ponteiros */
  px = &x;
  py = &y;
  /* Comparacao de ponteiros */
  if(px < py)
    printf("py-px = %u\n", py - px);
  else
    printf("px-py = %u\n", px - py);
  printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
  printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);
  px++;
  printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
  py = px + 3;
  printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);
  printf("py-px = %u\n", py - px);
}

SAIDA

py-px = 1
px = 65492, *px = 5, &px = 65496
py = 65494, *py = 6, &py = 65498
px = 65494, *px = 6, &px = 65496
py = 65500, *py = -24, &py = 65498
py-px = 3
```

6.3.1 Atribuição

Um endereço pode ser atribuído a um ponteiro através do operador igual (=). No exemplo, atribuímos a *px* o endereço de *x* e a *py* o endereço de *y*.

6.3.2 Conteúdo

O operador indireto (*****) fornece o valor armazenado no endereço apontado.

6.3.3 Endereço

Como todas as variáveis, os ponteiros têm um endereço e um valor. Seu valor é o endereço de memória apontado. O operador (**&**) retorna o endereço de memória utilizado para armazenar o ponteiro. Em resumo:

- o nome do ponteiro fornece o endereço para o qual ele aponta;
- o operador **&** junto ao nome do ponteiro retorna o endereço do ponteiro;
- o operador ***** junto ao nome do ponteiro fornece o conteúdo da variável apontada.

6.3.4 Soma e diferença

Ponteiros permitem os operadores soma (+), diferença (**-**), incremento (++) e decremento (--). Estas operações não são tratadas como soma e diferença comuns entre inteiros. Elas levam em conta o espaço de memória utilizado pela variável apontada.

No exemplo, a variável *px* antes do incremento valia. 65492. Depois do incremento ela passou a valer 65494 e não 65493. O incremento de um ponteiro faz com que ele seja deslocado levando-se em conta o tamanho da variável apontada. No caso de um inteiro, o ponteiro será deslocado 2 bytes, que é o que ocorre no exemplo.

Assim como no incremento, o decremento, a soma e a diferença levam em conta o tamanho da variável apontada. Assim, com *py* = *px* + 3, a variável *py* passou a apontar para um endereço de memória localizado 6 bytes depois de *px*.

6.3.5 Comparações

Os operadores ==, !=, >, <, <=, >= são aceitos entre ponteiros e comparam os endereços de memória.

6.3.6 Ponteiros para void

Como dito anteriormente, ponteiros podem apontar para qualquer tipo de variável. Podemos ter inclusive ponteiros que apontam para o tipo **void**. Estes ponteiros armazenam um endereço de memória, sem que seja definido qual o tipo de variável que ocupa aquele espaço de memória.

Em ponteiros para **void** (**void***), não é possível a utilização do operador indireto (*****), uma vez que o compilador não sabe para qual o tipo de variável ele deva converter o conteúdo do endereço apontado.

Também não são permitidos os operadores de soma e diferença, já que o compilador não sabe quanto espaço ocupa a variável armazenada naquele endereço.

6.4 Ponteiros e vetores

Vetores em C são tratados pelo compilador como ponteiros. Isso significa que o nome do vetor representa o endereço ocupado pelo seu primeiro elemento. O programa a seguir exemplifica a utilização de ponteiros representando vetores:

```
#include <stdio.h>

void main(void)
{
  int  i, *px, x[] = {8, 5, 3, 1, 7};

  px = x;
  for(i = 0; i < 5; i++)
  {
    printf("%d\n", *px);
    px++;
  }
}
```

Nota-se no exemplo que fazemos com que o ponteiro *px* aponte para o vetor *x*. Repare que não é utilizado o operador endereço. Isto ocorre porque o nome do vetor já é tratado pelo compilador como um endereço. No caso, seria possível ao invés de

```
px = x;
```

utilizarmos

```
px = &x[0];
```

O mesmo resultado seria obtido, já que o nome do vetor é tratado como um ponteiro que aponta para o endereço do primeiro elemento do vetor.

A linha que incrementa o valor de *px* faz com que o ponteiro aponte para o endereço de memória do próximo inteiro, ou seja, o próximo elemento do vetor.

Escreveremos agora o mesmo programa de outra maneira:

```
#include <stdio.h>

void main(void)
{
  int  x[] = {8, 5, 3, 1, 7};
  int  i;

  for(i = 0; i < 5; i++)
    printf("%d\n", *(x + i));
}
```

Note que neste exemplo chamamos cada elemento do vetor através de **(x + i)*. Como pode-se notar,

```
*(x + 0) = *x = x[0];
*(x + 1) = x[1];
*(x + 2) = x[2];
...
*(x + indice) = x[indice].
```

Como exemplo, vamos fazer uma função equivalente à função *strcpy*, da biblioteca *string.h*.

```
int strcpy(char* s1, char* s2)
{
  /* Copia a string s2 em s1 */
  while(*s2 != '\0')
  {
    *s1 = *s2;
    s1++;
    s2++;
  }
  *s1 = '\0';
}
```

Nesta função, todos os caracteres de *s2* serão copiados em *s1*, até que seja encontrado o caractere nulo. Para passar para o caractere seguinte, foi utilizado o operador ++.

6.5 Alocação dinâmica de memória

Quando um vetor é declarado, o endereço de memória para o qual ele aponta não teve necessariamente seu espaço reservado (alocado) para o programa. Nos exemplos anteriores, os ponteiros utilizados sempre apontavam para outras variáveis existentes.

Nesta seção veremos o caso de alocação dinâmica, isto é, memória alocada em tempo de execução do programa. Para isto, usaremos as função *malloc* e *free*, presentes nas bibliotecas *stdlib.h* e *alloc.h*.

6.5.1 Função malloc

Sintaxe:

```
void* malloc(unsigned long size);
```

A função malloc aloca um espaço de memória (*size* bytes) e retorna o endereço do primeiro byte alocado. Como pode-se alocar espaço para qualquer tipo de variável, o tipo de retorno é **void***. Torna-se necessário converte-lo para um ponteiro do tipo desejado.

No caso de não haver espaço suficiente para alocar a quantidade de memória desejada, a função retorna um ponteiro para o endereço 0 (ou NULL). Caso a variável *size* fornecida seja 0, a função também retorna NULL.

Ao contrário das variáveis estaticamente alocadas apresentadas até agora, o a memória dinamicamente alocada não é automaticamente liberada. É necessário que o programador libere a memória dinamicamente alocada através da função *free*.

6.5.2 Função free

Sintaxe:

```
void free(void* block);
```

A função free desaloca a memória alocada..

6.5.3 Exemplo das funções malloc e free

A seguir será fornecido um exemplo das funções *malloc* e *free*.

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

void main(void)
{
  char* str = (char*) malloc(10);

  strcpy(str, "Alo mundo");
  printf("Variavel str: %s\n", str);
  free(str);
}
```

O programa apresentado aloca dinamicamente 10 bytes e associa o endereço do bloco alocado ao ponteiro *str*. Como a função *malloc* retorna **void***, é necessário converter o resultado para o tipo desejado, no caso **char***.

A partir daí trabalha-se com o ponteiro normalmente. Ao terminar a utilização do espaço alocado, é necessário liberar a memória, através da função *free*.

6.6 Ponteiros para ponteiros

Como apresentado anteriormente, ponteiros podem apontar para qualquer tipo de variável, inclusive para outros ponteiros. A seguir será fornecido um exemplo de uma matriz alocada dinamicamente.

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

void main(void)
{
  float** Matriz;
  int nLinhas, nColunas, i, j;

  printf("Quantas linhas?\n");
  scanf("%d", &nLinhas);
  printf("Quantas colunas?\n");
  scanf("%d", &nColunas);
  /* Aloca a matriz */
  Matriz = (float**) malloc(nLinhas * sizeof(float*));
  for(i = 0; i < nLinhas; i++)
    Matriz[i] = (float*) malloc(nColunas * sizeof(float));
  /* Define os elementos da matriz */
  for(i = 0; i < nLinhas; i++)
    for(j = 0; j < nColunas; j++)
      Matriz[i][j] = i * j;
  /* Imprime a matriz */
  for(i = 0; i < nLinhas; i++)
  {
    for(j = 0; j < nColunas; j++)
      printf("%f\t", Matriz[i][j]);
    printf("\n");
  }
  /* Desaloca a matriz */
  for(i = 0; i < nLinhas; i++)
    free(Matriz[i]);
  free(Matriz);
}
```

Note que neste exemplo não se sabe inicialmente qual será o tamanho da matriz. Caso desejássemos alocar o espaço estaticamente, seria necessário definir a matriz com número de linhas e colunas grande, de modo que a escolha do usuário não ultrapassasse entes limites. Por exemplo,

```
float Matriz[1000][1000];
```

Há dois inconvenientes imediatos de se alocar vetores e matrizes estaticamente quando não sabemos de antemão o seu tamanho: um espaço de memória é provavelmente desperdiçado; e ficamos limitados ao tamanho definido inicialmente. Alocando memória dinamicamente, somente a quantidade necessária de memória é utilizada, ao passo que nosso limite de tamanho é o limite físico de memória do computador.

Para alocar a matriz, inicialmente alocamos espaço para um vetor de ponteiros. O tamanho deste vetor será o número de linhas da matriz. Cada um destes ponteiros apontará para um vetor de **float** dinamicamente alocado.

O acesso a cada elemento da matriz é feito de forma semelhante à matriz alocada estaticamente. O primeiro índice entre colchetes retorna o vetor que contém a linha da matriz equivalente àquele índice. O segundo índice entre colchetes fornece o elemento contido na coluna.

Note que as linhas não se localizam necessariamente juntas na memória, podendo estar em qualquer lugar. O vetor de ponteiros indica onde cada linha se encontra na memória.

Para desalocar a matriz, é necessário desalocar primeiro cada linha para só então desalocar o vetor de ponteiros.

6.6.1 Passando matrizes alocadas dinamicamente como argumento de funções

No capítulo 5, vimos como passar uma matriz estaticamente alocada como argumento de uma função. Era necessário que a função conhecesse o número de colunas da matriz, para que soubesse onde acabava uma linha e começava outra. Este problema não existe com matrizes dinamicamente alocadas, podendo ser passada uma matriz de qualquer tamanho para a função.

Para que a função saiba o tamanho da matriz, é necessário que se passe como argumento o seu tamanho. O exemplo abaixo ilustra uma função que recebe três matrizes A, B e C. Esta função soma A com B e armazena o resultado em C.

```
void SomaMatrizes(int nLinhas, int nColunas, float** C,
                  const float** A, const float** B)
{
  int i, j;

  for(i = 0; i < nLinhas; i++)
  {
    for(j = 0; j < nColunas; j++)
    {
      C[i][j] = A[i][j] + B[i][j];
    }
  }
}
```

Para chamar esta função, é necessário que se tenha três matrizes de **float** alocadas dinamicamente. Estas matrizes devem ter o mesmo tamanho, já que não é possível somar matrizes de tamanhos

diferentes. Note que, desenvolvendo a função para matrizes alocadas dinamicamente, é possível utilizar matrizes de qualquer tamanho.

Note a utilização do modificador **const** nos argumentos A e B da função. Isto indica que são argumentos constantes, isto é, seus elementos não serão alterados dentro da função. A utilização do modificador **const**, neste caso, apesar de opcional, fornece segurança ao programador.

Sabe-se, antes de escrever a função, que ela não deve mexer nos argumentos A e B. Como matrizes são passadas por endereço, a princípio a função poderia alterar estes argumentos. Como não se deseja que a função altere os argumentos A e B, coloca-se o modificador const à frente dos argumentos constantes.

Caso tentemos modificar dentro da função algum elemento da matriz A ou da matriz B, o compilador acusará erro de compilação. Se não utilizássemos o modificador **const**, só perceberíamos o erro muito depois, com saídas incorretas do programa. O tempo de depuração de erros de compilação é muito menor que o de erros de execução.

6.7 Ponteiros para funções

Um ponteiro para uma função é um caso especial de tipo apontado. Se você definiu um ponteiro para função e inicializou-o para apontar para uma função particular ele terá o valor do endereço onde a função está localizada na memória.

O exemplo seguinte mostra este uso.

```
#include <stdio.h>

void ImprimeOla(void)
{
  printf("Ola");
}

void Imprime(const char* str)
{
  printf(str);
}

void main(void)
{
  void (*f1)(void);
  void (*f2)(const char*);

  f1 = ImprimeOla;
  f2 = Imprime;
  (*f1)();
  (*f2)(" mundo!\n");
}
```

6.7.1 A função qsort()

No capítulo 5 utilizamos o método da bolha para a ordenação de um vetor. Este método, apesar de bastante intuitivo, é lento quando comparado a outros métodos de ordenação. A biblioteca C contém uma função de ordenação que utiliza o algoritmo *quick sort*, consideravelmente mais rápido que o

método da bolha: a função *qsort*. Esta função está presente na biblioteca *stdlib.h*. Sua sintaxe é a seguinte.

```
void qsort(void* base, unsigned long nelem, unsigned long width,
           int(*fcmp)(const void*, const void*));
```

Esta função ordena *nelem* elementos de tamanho *width* localizados em *base*. O usuário deve fornecer uma função de comparação, *fcomp*, que compara dois elementos, *elem1* e *elem2*. Esta função deve retornar

```
< 0, se elem1 < elem2
= 0, se elem1 == elem2
> 0, se elem1 > elem2
```

Note que tanto *base* quanto os argumentos da função, *elem1* e *elem2* são do tipo **void***. Isto é necessário já que deseja-se que a função ordene qualquer tipo de variável.

Como exemplo, vamos criar um programa que ordena um vetor em ordem crescente e decrescente.

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

int OrdenaCrescente(const void* pa, const void* pb)
{
  int* px = (int*) pa;
  int* py = (int*) pb;

  return (*px) - (*py);
}

int OrdenaDecrescente(const void* pa, const void* pb)
{
  int* px = (int*) pa;
  int* py = (int*) pb;

  return (*py) - (*px);
}

void Imprime(int* piValor, int iTamanho)
{
  int i;

  for(i = 0; i < iTamanho; i++)
  {
    printf("%d ", piValor[i]);
  }
  printf("\n");
}

void main(void)
{
  int piValor[] = {1, 5, 3, 7, 4, 5, 9};

  qsort(piValor, 7, sizeof(int), OrdenaCrescente);
  Imprime(piValor, 7);
  qsort(piValor, 7, sizeof(int), OrdenaDecrescente);
  Imprime(piValor, 7);
}
```

Na função main(), o programa declara um vetor *piValor* e o inicializa com números inteiros desordenados.

Em seguida, o programa chama a função qsort() para ordená-lo em ordem crescente. Na chamada, é fornecido o vetor, *piValor* (lembrando: vetores são ponteiros), o número de elementos, 7, o tamanho de cada elemento, obtido através do operador **sizeof**, e a função de comparação, OrdenaCrescente(). Completada a ordenação, o programa imprime o vetor, através da função Imprime().

O programa chama novamente a função qsort() para ordená-lo em ordem decrescente. A única diferença é que agora a função de ordenação fornecida é a função OrdenaDecrescente().

As funções OrdenaCrescente() e OrdenaDecrescente() tem que ter a sintaxe exigida pela função qsort(), ou seja, recebem dois argumentos **void*** e retornam um **int** que indica qual dos dois argumentos vem na frente na ordenação. Como recebem argumentos **void***, é necessário converter para o tipo **int** antes de comparar.

A função OrdenaCrescente() retorna a diferença entre o primeiro e o segundo elementos. Isso faz com que o valor retorno satisfaça às condições impostas pela função qsort(). A função OrdenaDecrescente() retorna o contrário da função OrdenaCrescente(), de modo que a função qsort() interprete um valor maior como vindo à frente na ordenação.

A função qsort() pode ordenar um vetor de qualquer tipo de variável, desde que definidos os critérios de comparação entre dois elementos do vetor.