8. PONTEIROS

O correto entendimento e uso de ponteiros é muito importante para um programador C, pois: eles fornecem os meios pelos quais as funções podem modificar seus argumentos; eles são usados para suportar as rotinas de alocação dinâmica de C; e eles podem aumentar a eficiência de certas rotinas. Ponteiros são um dos aspectos mais fortes e mais perigosos de C, porque quando são utilizados de maneira incorreta podem ocasionar erros que são muito difíceis de encontrar.

Um ponteiro é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de uma outra variável de memória. Se uma variável contém o endereço de outra, então a primeira variável é dita para **apontar** para a segunda, como mostra a figura 8.1.

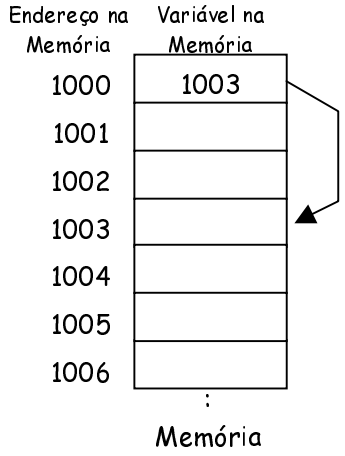

Figura 8.1 – Uma variável que aponta para outra [SCH 96]

8.1. Declaração e Manipulação

Se uma variável irá conter um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste no tipo de base, um * e o nome da variável. A forma geral para declarar uma variável ponteiro é:
*<tipo> *<nome>;*
onde <tipo> é qualquer tipo válido em C e <nome> é o nome da variável ponteiro.

O tipo base do ponteiro define que tipo de variáveis o ponteiro pode apontar. Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória. No entanto, toda a aritmética de ponteiros é feita por meio do tipo base, assim é importante declarar o ponteiro corretamente.

Existem dois operadores especiais para ponteiros: ***** e **&**. O & é um operador unário que devolve o endereço na memória do operando. Por exemplo:
*int *m, cont=100, q;*
m = &cont;
É colocado em "m" o endereço da memória que contém a variável "cont". O endereço não tem relação alguma com o valor de "cont". O operador & pode ser imaginado como retornando "o endereço de". Assim, o comando de atribuição anterior significa "m recebe o endereço de cont".

O segundo operador de ponteiro, *, é o complemento de &. É um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se "m" contém o endereço da variável "cont",

*q = *m;*

coloca o valor de "cont" em "q". Portanto, "q" terá o valor 100. O operador * pode ser imaginado como "no endereço". Nesse caso, o comando anterior significa "q recebe o valor que está no endereço m".

Alguns iniciantes em C podem confundir o sinal de multiplicação e o símbolo "no endereço de" (*), porém esses operadores não têm nenhuma relação um com o outro. Tanto & como * têm uma precedência maior do que todos os operadores aritméticos, exceto o menos unário, com o qual eles se parecem.

As variáveis ponteiros sempre devem apontar para o tipo de dado correto. Por exemplo, quando um ponteiro é declarado como sendo do tipo *int*, o ponteiro assume que qualquer endereço que ele contenha aponta para uma variável inteira. Como C permite a atribuição de qualquer endereço a uma variável ponteiro, o fragmento de código a seguir compila sem nenhuma mensagem de erro (ou apenas uma advertência – *warning*), mas não produz o resultado desejado.

```
main () {
        float x, y;
        int *p;
        p = &x; // p aponta para um float
        y = *p; // o valor atribuído para y não é o esperado
}
```

Neste caso, como "p" é declarado como um ponteiro para inteiros, apenas dois *bytes* de informação são transferidos para "y", não os 8 *bytes* que normalmente formam um número em ponto flutuante.

Em geral, expressões envolvendo ponteiros concordam com as mesmas regras de qualquer outra expressão de C. Algumas diferenças:

- **Atribuição**: um ponteiro pode ser usado no lado direito de uma comando de atribuição para passar seu valor para um outro ponteiro. Por exemplo:

```
main() {
    int x;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;  // agora p1 e p2 apontam para x
    printf("%p", p2);  // escreve o endereço de x e não o seu valor
}
```

- **Aritmética**: existem apenas duas operações aritméticas que podem ser usadas com ponteiros, que são adição e subtração. Para entender o que ocorre na aritmética de ponteiros, considere "p1" um ponteiro para um inteiro, que ocupa 2 *bytes*, com o valor atual 2000. Após a expressão "*p1++;*", p1 conterá 2002, e não 2001. Cada vez que "p1" é incrementado, ele aponta para o próximo inteiro. O mesmo ocorre nos decrementos. Generalizando a partir do exemplo, cada vez que um ponteiro é incrementado, ele aponta para a posição de memória do próximo elemento do seu tipo base. E cada vez que é decrementado, ele aponta para a posição do elemento anterior. Em outras palavras, os ponteiros são incrementados e decrementados relativamente ao tamanho do tipo base, de forma que ele sempre aponta para o próximo elemento ou para o elemento anterior, respectivamente. Também é possível somar e subtrair inteiros de ponteiros. Assim, a expressão "*p1=p1+12;*" faz "p1" apontar para o décimo segundo elemento do tipo "p1" adiante do elemento que ele está atualmente apontando. Além de adição e subtração entre um ponteiro e um inteiro, nenhuma outra operação aritmética pode ser efetuada com ponteiros.
- **Comparação**: É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros "p" e "q", o comando a seguir é perfeitamente válido:

```
if (p < q) printf("p aponta para uma memória mais baixa que q \n");
```

Geralmente, comparações de ponteiros são usadas quando dois ou mais ponteiros apontam para um objeto comum. Como exemplo, um par de rotinas de pilha são desenvolvidas de forma a guardar valores inteiros. Uma pilha é uma lista em que o primeiro acesso a entrar é o último a sair. É freqüentemente comparada a uma pilha de pratos em uma mesa – o primeiro prato colocado é o último a ser usado. Pilhas são muito usadas em compiladores, interpretadores, planilhas e outros software *relacionados com o sistema. Para criar uma pilha são necessárias duas funções:* push() *e* pop()*. A primeira coloca os valores na pilha e a segunda retira-os. Essas rotinas são mostradas no próximo exemplo com uma função* main() *bem simples para utilizá-las. Se for digitado 0, um valor será retirado da pilha, e se for digitado –1 o programa é encerrado.*

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 50

void push(int i);
int pop(void);

int *tos, *p1, stack[SIZE]; /* stack fornece memória para a pilha */

main() {
  int value;
  tos = stack; /* faz tos conter o topo da pilha, evitando que se retirem elementos da pilha vazia */
  p1 = stack; /* inicializa p1 – aponta para o primeiro byte em stack */
  do {
    printf("Entre com o valor: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("valor do topo , %d\n", pop());
  } while(value!=-1);
}

void push(int i) {
  p1++;
  if(p1==(tos+SIZE)) {
    printf("estouro da pilha");
    exit(1);
  }
  *p1 = i;
}

pop(void) {
  if(p1==tos) {
    printf("pilha vazia");
    exit(1);
  }
  p1--;
  return *(p1+1); /* retorna o conteúdo da posição p1+1 */
}
```

Torna-se importante atentar para o fato de que após um ponteiro ser declarado, mas antes que lhe seja atribuído um valor, ele contém um valor desconhecido (lixo). Se um ponteiro for usado antes de receber um valor, ou seja, antes de ser inicializado, provavelmente "quebrará" não apenas o programa como também o sistema operacional do computador.

Há uma importante convenção que a maioria dos programadores de C segue quando trabalha com ponteiros: um ponteiro que atualmente não aponta para um local de memória válido recebe o valor nulo (que é zero). Por convenção, qualquer ponteiro que é nulo implica que ele não aponta para nada e não deve ser usado. Porém, apenas o fato de um ponteiro ter um valor nulo não o torna "seguro". Se for usado um ponteiro nulo no lado esquerdo de um comando de atribuição, ainda se correrá o risco de "quebrar" o programa ou o sistema operacional.

Como um ponteiro nulo é assumido como sendo não usado, é possível utilizar o ponteiro nulo para tornar fáceis de codificar e mais eficientes muitas rotinas. Por exemplo, um ponteiro nulo pode ser usado para marcar o final de uma matriz de ponteiros, e assim, uma rotina que acessa essa matriz sabe que chegará ao final ao encontrar o valor nulo. Alguns exemplos de inicialização de ponteiros são:

```
int *p = NULL; // Macro que define um ponteiro nulo
char *p = "alo mundo";
int x = 10;
int *p = &x;
```

8.2. Ponteiros e Matrizes

Há uma estreita relação entre ponteiros e matrizes. Considerando o seguinte fragmento de programa:

```
char str[80], *p1;
p1 = str;
```

Aqui "p1" foi inicializado com o endereço do primeiro elemento da matriz "str". Para acessar o quinto elemento em "str", teria que ser escrito

```
str[4]
```

ou

```
*(p1+4)
```

Os dois comandos devolvem o quinto elemento. Observe que como as matrizes começam em zero, deve-se usar 4 para acessar o quinto elemento. Da mesma maneira adiciona-se 4 ao ponteiro "p1" para acessar o quinto elemento, pois "p1" aponta atualmente para o primeiro elemento de "str". Também é importante lembra que o nome de uma matriz sem um índice retorna o endereço inicial da matriz, que é o primeiro elemento.

C fornece dois métodos para acessar elementos de matrizes: aritmética de ponteiros e indexação de matrizes (como apresentado no capítulo 7). Como aritmética de ponteiros pode ser mais rápida que indexação de matrizes, e velocidade é geralmente uma consideração em programação, programadores em C normalmente usam ponteiros para acessar elementos de matrizes. Para exemplificar a diferença, duas versões da função "*putstr()*", uma com indexação de matrizes e uma com ponteiros, são apresentadas a seguir. Esta função escreve uma *string* no dispositivo de saída padrão.

```
// Indexa s como uma matriz
void puts (char *s) {
        register int t;
        for (t=0; s[t]; ++t)
                putchar(s[t]);
}

// Acessa s como um ponteiro
void putstr (char *s) {
        while (*s)
                putchar(*s++); // " *s++; " é equivalente a " *s; s++; "
}
```

Ponteiros também podem ser organizados em matrizes como qualquer outro tipo de dado. A declaração de uma matriz de ponteiros "int" de tamanho 10, é:

```
int *x[10];
```

Para atribuir o endereço de uma variável inteira, chamada "var", ao terceiro elemento da matriz de ponteiros, deve-se escrever:

```
x[2] = &var;
```

Para encontrar o valor de "var", escreve-se:

```
*x[2];
```

Se for necessário passar uma matriz de ponteiros para uma função, pode ser usado o mesmo método que é utilizado para passar outras matrizes – simplesmente chame a função com o nome da matriz sem qualquer índice. Por exemplo, a seguinte função recebe a matriz "x" como parâmetro:

```
void display_array (int *q[]) {
        int t;
        for (t=0; t<10; t++)
                printf ("%d ", *q[t]);
}
```

Neste caso, é importante lembrar que "q" não é um ponteiro para inteiros; "q" é um ponteiro para uma matriz de ponteiros para inteiros. Portanto, é necessário declarar o parâmetro "q" como uma matriz de ponteiros para inteiros, como apresentado no código anterior. Ela não pode ser simplesmente declarada como um ponteiro para inteiros.

Matrizes de ponteiros são usadas normalmente como ponteiros para *strings*. Assim, é possível criar uma função que exiba uma mensagem de erro quando é dado seu número de código, como mostrado a seguir:

```
void syntax_error (int num) {
        static char *err[] = {
                "Arquivo não pode ser aberto \n",
                "Erro de leitura \n",
                "Erro de escrita \n",
                "Falha da mídia \n"
        };
        printf("%s", err[num]);
}
```

No exemplo anterior, a matriz "err" contém ponteiros para cada *string*. Como pode-se observar, "printf()" dentro de "syntax_error" é chamada com um ponteiro de caracteres que aponta para uma das várias mensagens de erro indexadas pelo número de erro passado para a função. Por exemplo, se for passado o valor 1 a mensagem "Erro de leitura" é apresentada, se for passado o valor 2, a mensagem "Erro de escrita" é apresentada, e assim por diante. Outro exemplo de matriz de ponteiros a caracteres é o argumento da linha de comandos "argv" (seção 4.4) [SCH 96].

8.3. Alocação Dinâmica

Ponteiros fornecem o suporte necessário para o poderoso sistema de alocação dinâmica de C. **Alocação dinâmica** é o meio pelo qual um programa pode obter memória enquanto está em execução. Variáveis globais têm o armazenamento alocado em tempo de compilação e variáveis locais usam a pilha. No entanto, nem variáveis globais, nem locais podem ser acrescentadas durante o tempo de execução. Porém, haverá momentos em que um programa precisará usar quantidades de armazenamento variáveis. Por exemplo, um processador de texto ou um banco de dados aproveita toda a RAM de um sistema. Porém, como a quantidade de RAM varia entre computadores tais programas não poderão usar variáveis normais, por isso alocam memória conforme o necessário.

A memória alocada pelas funções de alocação dinâmica de C é obtida do *heap* (região de memória livre). Embora o seu tamanho seja desconhecido, o *heap* geralmente contém uma quantidade razoavelmente grande de memória livre.

A alocação dinâmica em C baseia-se nas funções ***malloc()***, para alocar memória, e ***free()***, para liberar a memória alocada. Apesar de existirem outras funções de alocação dinâmica, estas são as mais importantes. Elas operam em conjunto, usando a região de memória livre para estabelecer e manter uma lista de armazenamento disponível. Os protótipos destas funções, que estão descritos na *stdlib.h*, são:

```
void *malloc(size_t <número_de_bytes>);
```

*void free(void *p);*

Aqui, "<número_de_bytes>" é o número de *bytes* de memória que deve ser alocado, e o tipo "size_t" é definido em *stdlib.h* como (mais ou menos) um inteiro sem sinal. A função *malloc()* devolve um ponteiro do tipo *void*, o que significa que este pode ser atribuído a qualquer tipo de ponteiro. Após uma chamada bem-sucedida, *malloc()* devolve um ponteiro para o primeiro *byte* da região de memória alocada do heap. Se não há memória disponível para satisfazer a requisição de *malloc()*, ele devolve um nulo. Exemplos de utilização destas funções:

```
char *p1;
int *p2;
:
if ( !(p1=malloc(1000)) ) { // Se a alocação dos 1000 bytes retornar nulo
        printf("Sem memória disponível. \n");
        exit(1);            // Aborta a execução do programa; Poderia ter um tratamento de erro.
}
:
p2 = malloc(50*sizeof(int));       // usando a função "sizeof" há uma garantia de portabilidade
:
free(p1);
free(p2);
```

É importante salientar que o ponteiro enviado para a função *free()*, deve ser um ponteiro para memória alocada anteriormente por *malloc()*. Nunca deve-se usar *free()* com um argumento inválido, pois isso destruiria a lista de memória livre.

Algumas vezes pode ser necessário alocar memória usando *malloc()*, mas operar na memória como se ela fosse uma matriz, usando indexação de matrizes. Em outras palavras, é possível criar uma matriz alocada dinamicamente. Como qualquer ponteiro pode ser indexado como se fosse uma matriz unidimensional, isso não representa nenhum problema, como mostra o próximo exemplo [SCH 96].

```
// Aloca espaço para uma string dinamicamente, solicita a entrada do usuário e, em seguida,
// imprime a string de trás para frente.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void main(void) {
  char *s;
  register int t;
  s = malloc(80);
  if(!s) {
    printf("Falha na solicitação de memória \n");
    exit(1);
  }
  gets(s);
  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
  free(s);
}
```

Acessar memória alocada como se fosse uma matriz unidimensional é simples. No entanto, matrizes dinâmicas multidimensionais levantam alguns problemas. Como as dimensões da matriz não foram definidas no programa, não é possível indexar diretamente um ponteiro como se ele fosse uma matriz multidimensional. Para conseguir uma matriz alocada dinamicamente, é necessário passar o ponteiro como um parâmetro a uma função. Dessa forma, a função pode definir as dimensões do parâmetro que recebe o ponteiro, permitindo, assim, a indexação normal de matriz. O exemplo a seguir, que constrói uma tabela dos números de 1 a 10 elevados a primeira, à segunda, à terceira e à quarta potência, mostra como isso funciona [SCH 96].

```
/* Apresenta as potências dos números de 1 a 10. Nota: muito embora esse programa esteja correto,
   alguns compiladores C apresentarão uma mensagem de advertência, ou até erro, com relação aos
   argumentos para as funções table() e show(). Se forem apenas advertências, ignore. */
```

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

int pwr(int a, int b);
void table(int p[4][10]);
void show(int p[4][10]);

void main(void) {
  int *p;
  p = (int *) malloc(40*sizeof(int));
  if(!p) {
    printf("Falha na solicitação de memória. \n");
    exit(1);
  }
  // aqui, p , é simplesmente um ponteiro
  table(p);
  show(p);
}
// Constrói a tabela de potências
void table(int p[4][10]) {  // agora o compilador tem uma matriz para trabalhar
  register int i, j;
  for(j=1; j<11; j++)
    for(i=1; i<5; i++) p[i-1][j-1] = pwr(j, i);
}
// Exibe a tabela de potências inteiras
void show(int p[4][10]) {  // agora o compilador tem uma matriz para trabalhar
  register int i, j;
  printf("%10s %10s %10s %10s\n", "N", "N^2", "N^3", "N^4");
  for(j=1; j<11; j++) {
    for(i=1; i<5; i++) printf("%10d ", p[i-1][j-1]);
    printf("\n");
  }
}
// Eleva um inteiro a uma potência especificada
pwr(int a, int b) {
  register int t=1;
  for(; b; b--) t = t*a;
  return t;
}
```