5

Ponteiros

O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C. Há três razões para isso: primeiro, ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos; segundo, eles são usados para suportar as rotinas de alocação dinâmica de C, e terceiro, o uso de ponteiros pode aumentar a eficiência de certas rotinas.

Ponteiros são um dos aspectos mais fortes e mais perigosos de C. Por exemplo, ponteiros não-inicializados, ou *ponteiros selvagens*, podem provocar uma quebra do sistema. Talvez pior, é fácil usar ponteiros incorretamente, ocasionando erros que são muito difíceis de encontrar.

O Que São Ponteiros?

Um *ponteiro* é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de uma outra variável na memória. Se uma variável contém o endereço de uma outra, então a primeira variável é dita para *apontar* para a segunda. A Figura 5.1 ilustra essa situação.

Variáveis Ponteiros

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;*

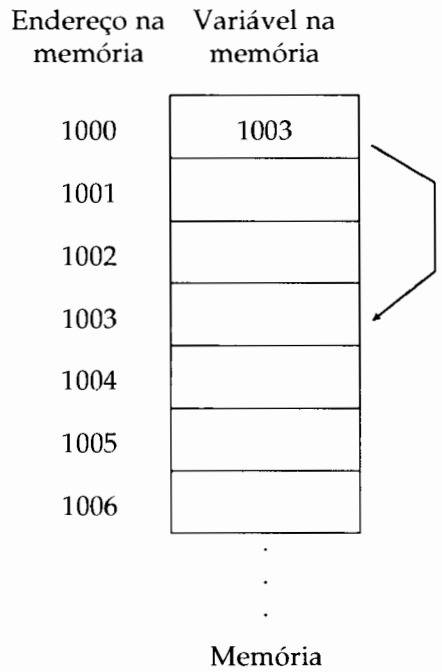

Figura 5.1 Uma variável aponta para outra.

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. (A aritmética de ponteiros é discutida mais adiante neste capítulo.)

Os Operadores de Ponteiros

Existem dois operadores especiais para ponteiros: * e &. O & é um operador unário que devolve o endereço na memória do seu operando. (Um operador unário requer apenas um operando.) Por exemplo,

```
m = &count;
```

coloca em **m** o endereço da memória que contém a variável **count**. Esse endereço é a posição interna ao computador da variável. O endereço não tem relação alguma com o valor de **count**. O operador **&** pode ser imaginado como retornando

"o endereço de". Assim, o comando de atribuição anterior significa "**m** recebe o endereço de **count**".

Para entender melhor a atribuição anterior, assuma que a variável **count** usa a posição de memória 2000 para armazenar seu valor. Assuma também que **count** tem o valor 100. Então, após a atribuição anterior, **m** terá o valor 2000.

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 **count**,

```
q = *m;
```

coloca o valor de **count** em **q**. Portanto, **q** terá o valor 100 porque 100 estava armazenado na posição 2000, que é o endereço que estava armazenado em **m**. 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-se porque o sinal de multiplicação e o símbolo de "no endereço" são idênticos, como também o AND bit a bit é igual ao símbolo de "endereço de". 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 seguinte compila sem nenhuma mensagem de erro (ou apenas uma advertência, dependendo do seu compilador), mas não produz o resultado desejado:

```
void main(void)
{
  float x, y;
  int  *p;

  /* O próximo comando faz com que p (que é ponteiro
     para inteiro) aponte para um float. */
  p = &x;

   /* O próximo comando não funciona como esperado. */
  y = *p;
}
```

Isso não irá atribuir o valor de **x** a **y**. Já que **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.

Expressões com Ponteiros

Em geral, expressões envolvendo ponteiros concordam com as mesmas regras de qualquer outra expressão de C. Nesta seção uns poucos aspectos especiais de expressões com ponteiros serão examinados.

Atribuição de Ponteiros

Como é o caso com qualquer variável, um ponteiro pode ser usado no lado direito de um comando de atribuição para passar seu valor para um outro ponteiro. Por exemplo,

```
#include <stdio.h>

void main(void)
{
  int x;
  int *p1, *p2;

  p1 = &x;
  p2 = p1;

  printf("%p", p2); /* escreve o endereço de x, não seu
                       valor! */
}
```

Agora, tanto **p1** quanto **p2** apontam para **x**. O endereço de **x** é mostrado, usando o modificador de formato de **printf()** **%p**, que faz com que **printf()** apresente um endereço no formato usado pelo computador host.

Aritmética de Ponteiros

Existem apenas duas operações aritméticas que podem ser usadas com ponteiros: adição e subtração. Para entender o que ocorre na aritmética de ponteiros, consideremos **p1** um ponteiro para um inteiro com o valor atual 2000. Assuma, também, que os inteiros são de 2 bytes. Após a expressão

```
p1++;
```

p1 contém 2002, não 2001. Cada vez que **p1** é incrementado, ele aponta para o próximo inteiro. O mesmo é verdade nos decrementos. Por exemplo, assumindo que **p1** tem o valor 2000, a expressão

```
p1--;
```

faz com que **p1** receba o valor 1998.

Generalizando a partir do exemplo anterior, as regras a seguir governam a aritmética de ponteiros. Cada vez que um ponteiro é incrementado, ele aponta para a posição de memória do próximo elemento do seu tipo base. Cada vez que é decrementado, ele aponta para a posição do elemento anterior. Com ponteiros para caracteres, isso freqüentemente se parece com a aritmética "normal". Contudo, todos os outros ponteiros incrementam ou decrementam pelo tamanho do tipo de dado que eles apontam. Por exemplo, assumindo caracteres de 1 byte e inteiros de 2 bytes, quando um ponteiro para caracteres é incrementado, seu valor aumenta em um. Porém, quando um ponteiro inteiro é incrementado, seu valor aumenta em dois. Isso ocorre porque os ponteiros são incrementados e decrementados relativamente ao tamanho do tipo base de forma que ele sempre aponta para o próximo elemento. De maneira mais geral, toda a aritmética de ponteiros é feita relativamente ao tipo base do ponteiro, para que ele sempre aponte para o elemento do tipo base apropriado. A Figura 5.2 ilustra esse conceito.

Você não está limitado a apenas incremento e decremento. Você pode somar ou subtrair inteiros de ponteiros. 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. Especificamente, você não pode multiplicar ou dividir ponteiros; não pode aplicar os operadores de deslocamento e de mascaramento bit a bit com ponteiros; e não pode adicionar ou subtrair o tipo **float** ou o tipo **double** a ponteiros.

```
char *ch=3000;
int *i=3000;
```

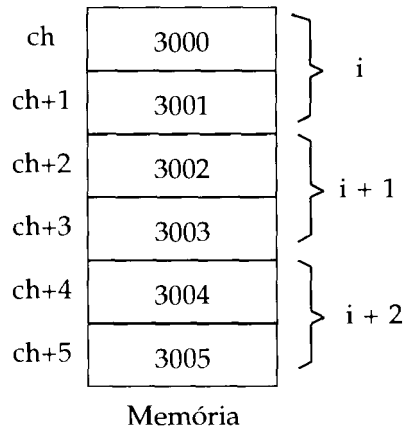

Figura 5.2 Toda aritmética de ponteiros é relativa a seu tipo base.

Comparação de Ponteiros

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros **p** e **q**, o fragmento de código seguinte 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 freqüentemente usadas em compiladores, interpretadores, planilhas e outros softwares relacionados com o sistema. Para criar uma pilha, são necessárias duas funções: **push()** e **pop()**. A função **push()** coloca os valores na pilha e **pop()** retira-os. Essas rotinas são mostradas aqui com uma função **main()** bem simples para utilizá-las. Se você digitar **0**, um valor será retirado da pilha. Para encerrar o programa, digite **-1**.

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

# define SIZE 50
```

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

int *tos, *p1, stack[SIZE];

void main(void)
{
  int value;

  tos = stack; /* faz tos conter o topo da pilha */
  p1 = stack; /* inicializa p1 */

  do {
    printf("Digite o valor: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("valor do topo é %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("Estouro da pilha");
    exit(1);
  }
  p1--;
  return *(p1+1);
}
```

Você pode ver que a memória para a pilha é fornecida pela matriz **stack**. O ponteiro **p1** é ajustado para apontar para o primeiro byte em **stack**. A pilha é realmente acessada pela variável **p1**. A variável **tos** contém o endereço do topo da pilha. O valor de **tos** evita que se retirem elementos da pilha vazia. Uma vez

que a pilha tenha sido inicializada, **push()** e **pop()** podem ser usadas como uma pilha de inteiros. Tanto **push()** como **pop()** realizam um teste relacional com o ponteiro **p1** para detectar erros de limite. Em **push()**, **p1** é testado junto ao final da pilha adicionando-se **SIZE** (o tamanho da pilha) a **tos**. Em **pop()**, **p1** é verificado junto a **tos** para assegurar que não se retirem elementos da pilha vazia.

Em **pop()**, os parênteses são necessários no comando **return**. Sem eles, o comando seria

```
return *p1 +1;
```

que retornaria o valor da posição **p1** mais um, não o valor da posição **p1+1**. Você deve usar parênteses para garantir a ordem correta de avaliação quando usar ponteiros.

Ponteiros e Matrizes

Há uma estreita relação entre ponteiros e matrizes. Considere este 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 de ser escrito

```
str[4]
```

ou

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

Os dois comandos devolvem o quinto elemento. Lembre-se de que matrizes começam em 0, assim, deve-se usar 4 para indexar **str**. Também deve-se adicionar 4 ao ponteiro **p1** para acessar o quinto elemento porque **p1** aponta atualmente para o primeiro elemento de **str**. (Recorde-se que um 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. Aritmética de ponteiros pode ser mais rápida que indexação de matrizes. Já que velocidade é geralmente uma consideração em programação, programadores em C normalmente usam ponteiros para acessar elementos de matrizes.

Essas duas versões de **putstr()** — uma com indexação de matrizes e uma com ponteiros — ilustram como você pode usar ponteiros em lugar de indexação de matrizes. A função **putstr()** 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++);
}
```

A maioria dos programadores profissionais em C acharia a segunda versão mais fácil de ler e entender. Na realidade, a versão com ponteiros é a forma pela qual rotinas desse tipo são normalmente escritas em C.

Matrizes de Ponteiros

Ponteiros 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, uma função que recebe a matriz **x** se parece com isto:

```
void display_array(int *q[])
{
  int t;

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

Lembre-se de 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 mostrado no código anterior. Ela não pode ser simplesmente declarada como um ponteiro para inteiros, porque não é isso o que ela é.

Matrizes de ponteiros são usadas normalmente como ponteiros para strings. Você pode criar uma função que exiba uma mensagem de erro, quando é dado seu número de código, como mostrado aqui:

```
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]);
}
```

A matriz **err** contém ponteiros para cada string. Como você pode ver, **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 2, a mensagem **Erro de escrita** é apresentada.

Observe que o argumento da linha de comandos **argv** é uma matriz de ponteiros a caracteres. (Veja o Capítulo 6.)

Indireção Múltipla

Você pode ter um ponteiro apontando para outro ponteiro que aponta para o valor final. Essa situação é chamada *indireção múltipla*, ou *ponteiros para ponteiros*.

Ponteiros para ponteiros podem causar confusão. A Figura 5.3 ajuda a esclarecer o conceito de indireção múltipla. Como você pode ver, o valor de um ponteiro normal é o endereço de uma variável que contém o valor desejado. No caso de um ponteiro para um ponteiro, o primeiro ponteiro contém o endereço do segundo, que aponta para a variável que contém o valor desejado.

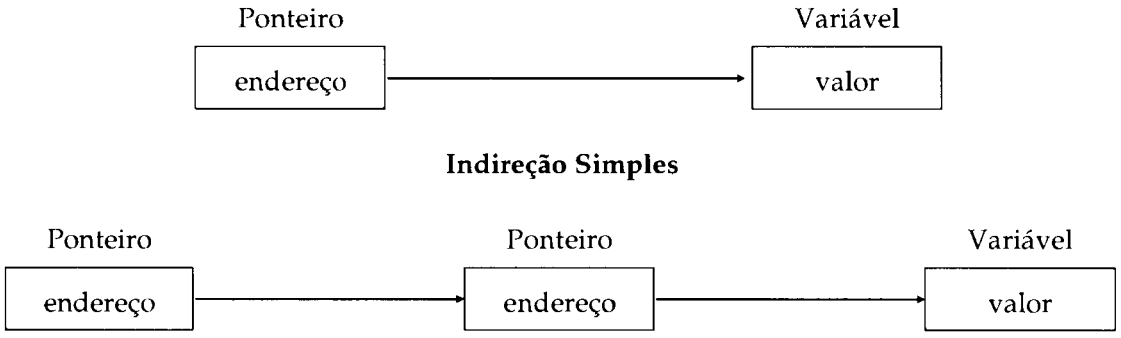

Figura 5.3 Indireção simples e múltipla.

A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. De fato, indireção excessiva é difícil de seguir e propensa a erros conceituais. (Não confunda indireção múltipla com listas encadeadas.)

NOTA: Não confunda a múltipla indireção com estruturas de dados de alto nível, tais como listas ligadas, que utilizam ponteiros. Estes são dois conceitos fundamentalmente diferentes.

Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. Isso é feito colocando-se um * adicional na frente do nome da variável. Por exemplo, a seguinte declaração informa ao compilador que **newbalance** é um ponteiro para um ponteiro do tipo **float**:

```
float **newbalance;
```

É importante entender que **newbalance** não é um ponteiro para um número em ponto flutuante, mas um ponteiro para um ponteiro **float**.

Para acessar o valor final apontado indiretamente por um ponteiro a um ponteiro, você deve utilizar o operador asterisco duas vezes, como neste exemplo:

```
#include <stdio.h>

void main(void)
{
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* imprime o valor de x */
}
```

Aqui, **p** é declarado como um ponteiro para um inteiro e **q**, como um ponteiro para um ponteiro para um inteiro. A chamada a **printf()** imprime **10** na tela.

Inicialização de Ponteiros

Após um ponteiro ser declarado, mas antes que lhe seja atribuído um valor, ele contém um valor desconhecido. Se você tentar usar um ponteiro antes de lhe dar um valor, provavelmente quebrará não apenas seu programa como também o sistema operacional de seu computador — um tipo de erro muito desagradável!

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 você usar um ponteiro nulo no lado esquerdo de um comando de atribuição, ainda correrá o risco de quebrar seu programa ou o sistema operacional.

Como um ponteiro nulo é assumido como sendo não usado, você pode utilizar o ponteiro nulo para tornar fáceis de codificar e mais eficientes muitas rotinas. Por exemplo, você poderia usar um ponteiro nulo para marcar o final de uma matriz de ponteiros. Uma rotina que acessa essa matriz sabe que chegará ao final ao encontrar o valor nulo. A função **search()**, mostrada aqui, ilustra esse tipo de abordagem.

```
/* procura um nome */
search(char *p[], char *name)
{
  register int t;

  for(t=0; p[t]; ++t)
    if(!strcmp(p[t], name)) return t;

    return -1; /* não encontrado */
}
```

O laço **for** dentro de **search()** é executado até que seja encontrada uma coincidência ou um ponteiro nulo. Como o final da matriz é marcado com um ponteiro nulo, a condição de controle do laço falha quando ele é atingido.

É uma prática comum entre programadores em C inicializar strings. Você viu um exemplo disso na função **syntax_error()**, na seção "Matrizes de Ponteiros". Uma outra variação no tema de inicialização é o seguinte tipo de declaração de string:

```
char *p = "alo mundo";
```

Como você pode observar, o ponteiro **p** não é uma matriz. A razão pela qual esse tipo de inicialização funciona deve-se à maneira como o compilador opera. Todo compilador C cria o que é chamada de *tabela de string*, que é usada internamente pelo compilador para armazenar as constantes string usadas pelo programa. Assim, o comando de declaração anterior coloca o endereço de **"alo mundo"**, armazenado na tabela de strings no ponteiro **p**. **p** pode ser usado por todo o programa como qualquer outra string. Por exemplo, o programa que segue é perfeitamente válido:

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

char *p = "alo mundo";

void main(void)
{
  register int t;

  /* imprime o conteúdo da string de trás para frente */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);
}
```

Ponteiros para Funções

Um recurso confuso, mas poderoso de C, é o *ponteiro para função*. Muito embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. O endereço de uma função é o ponto de entrada da função. Portanto, um ponteiro de função pode ser usado para chamar uma função.

Para entender como funcionam os ponteiros de funções, você deve conhecer um pouco como uma função é compilada e chamada em C. Primeiro, quando cada função é compilada, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido. Quando é feita uma chamada à função, enquanto seu programa está sendo executado, é efetuada uma chamada em linguagem de máquina para esse ponto de entrada. Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, ele pode ser usado para chamar essa função.

O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos. (Isso é semelhante à maneira como o endereço de uma matriz é obtido quando apenas o nome da matriz, sem índices, é usado.) Para ver como isso é feito, estude o programa seguinte, prestando bastante atenção às declarações:

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

void check(char *a, char *b,
           int (*cmp) (const char *, const char *));

void main(void)
{
  char s1[80], s2[80];
  int (*p)();

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);
}

void check(char *a, char *b,
           int (*cmp) (const char *, const char *))
{
```

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "string.h"

void check(char *a, char *b,
           int (*cmp)  (const char *, const char*));
int numcmp(const char *a, const char *b);

void main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  if(isalpha(*s1))
         check(s1, s2, strcmp);
  else
         check(s1, s2, numcmp);
}

void check(char *a, char *b,
           int (*cmp)  (const char *, const char*))
{
  printf("testando igualdade\n");
  if(!(*cmp)(a, b)) printf("igual");
  else printf("diferente");
}

numcmp(const char *a, const char *b)
{
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

As Funções de Alocação Dinâmica em C

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. Como você sabe, variáveis globais têm o armazenamento alocado em tempo de compilação. 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 esses programas não poderão usar variáveis normais. Em vez disso, esses e outros programas alocam memória, conforme necessário, usando as funções do sistema de alocação dinâmica de C.

A memória alocada pelas funções de alocação dinâmica de C é obtida do *heap* — a região de memória livre que está entre seu programa e a área de armazenamento permanente e a pilha. Embora o tamanho do heap seja desconhecido, ele geralmente contém uma quantidade razoavelmente grande de memória livre.

O coração do sistema de alocação dinâmica de C consiste nas funções **malloc()** e **free()**. (Na verdade, C tem diversas outras funções de alocação dinâmica, mas essas duas são as mais importantes.) Essas funções operam em conjunto, usando a região de memória livre para estabelecer e manter uma lista de armazenamento disponível. A função **malloc()** aloca memória e **free()** a libera. Isto é, cada vez que é feita uma solicitação de memória por **malloc()**, uma porção da memória livre restante é alocada. Cada vez que é efetuada uma chamada a **free()** para liberação de memória, a memória é devolvida ao sistema. Qualquer programa que use essas funções deve incluir o cabeçalho **STDLIB.H**.

A função **malloc()** tem este protótipo:

void *malloc(size_t *número_de_bytes*);

Aqui, *número_de_bytes* é o número de bytes de memória que você quer alocar. (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 você pode atribuí-lo 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()**, ocorre uma falha de alocação e **malloc()** devolve um nulo.

O fragmento de código mostrado aqui aloca 1000 bytes de memória.

```
char *p;
p = malloc(1000); /* obtém 1000 bytes */
```

Após a atribuição, **p** aponta para o primeiro dos 1000 bytes de memória livre.

O próximo exemplo aloca espaço para 50 inteiros. Observe o uso de **sizeof** para assegurar portabilidade.

```
int *p;
p = malloc(50*sizeof(int));
```

Como o heap não é infinito, sempre que alocar memória, você deve testar o valor devolvido por **malloc()**, antes de usar o ponteiro, para estar certo de que não é nulo. Usar um ponteiro nulo quase certamente travará o computador. A maneira adequada de alocar memória é ilustrada neste fragmento de código:

```
if(!(p=malloc(100)) {
  printf("sem memória.\n");
  exit(1);
}
```

Obviamente, você pode substituir algum outro tipo de manipulador de erro em lugar do **exit()**. Apenas tenha certeza de não usar o ponteiro **p** se ele for nulo.

A função **free()** é o oposto de **malloc()**, visto que ela devolve memória previamente alocada ao sistema. Uma vez que a memória tenha sido liberada, ela pode ser reutilizada por uma chamada subseqüente a **malloc()**. A função **free()** tem este protótipo:

void free(void *p);

Aqui, *p* é um ponteiro para memória alocada anteriormente por **malloc()**. É muito importante que você *nunca* use **free()** com um argumento inválido; isso destruiria a lista de memória livre.

O subsistema de alocação dinâmica de C é usado em conjunção com ponteiros para suportar uma variedade de construções de programação importantes, como listas encadeadas e árvores binárias. Você verá diversos exemplos disso na Parte 3. Um outro uso importante de alocação dinâmica é a matriz dinâmica, discutida a seguir.

Matrizes Dinamicamente Alocadas

Algumas vezes você terá de alocar memória, usando **malloc()**, mas operar na memória como se ela fosse uma matriz, usando indexação de matrizes. Em essência, você pode querer criar uma *matriz dinamicamente alocada*. Como qualquer ponteiro pode ser indexado como se fosse uma matriz unidimensional, isso não representa nenhum problema. Por exemplo, o programa seguinte mostra como você pode usar uma matriz alocada dinamicamente:

```
/* 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);
}
```

Como o programa mostra, antes de seu primeiro uso, **s** é testado para assegurar que a solicitação de alocação foi bem-sucedida e um ponteiro válido foi devolvido por **malloc()**. Isso é absolutamente necessário para evitar o uso acidental de um ponteiro nulo, que, como exposto anteriormente, quase certamente provocaria um problema. Observe como o ponteiro **s** é usado na chamada a **gets()** e, em seguida, indexado como uma matriz para imprimir a string de trás para frente.

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, você não pode indexar diretamente um ponteiro como se ele fosse uma matriz multidimensional. Para conseguir uma matriz alocada dinamicamente, você deve usar este truque: 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. Para ver como isso funciona, estude o exemplo seguinte, que constrói uma tabela dos números de 1 a 10 elevados a primeira, à segunda, à terceira e à quarta potências:

```
/* Apresenta as potências dos números de 1 a 10.
   Nota: muito embora esse programa esteja correto,
   alguns compiladores apresentarão uma mensagem de
   advertência com relação aos argumentos para as funções
   table() e show(). Se isso acontecer, 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 = 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;
}
```

A saída produzida por esse programa é mostrada na Tabela 5.1.

Tabela 5.1 A saída do programa de potências.

N	N^2	N^3	N^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000

Como o programa anterior ilustra, ao definir um parâmetro de função com as dimensões da matriz desejada, você pode enganar o compilador C para manipular matrizes dinâmicas multidimensionais. De fato, no que diz respeito ao compilador C, você tem uma matriz 4,10 dentro das funções **show()** e **table()**. A diferença é que o armazenamento para a matriz é alocado manualmente usando o comando **malloc()**, em lugar de usar o comando normal de declaração de matriz. Além disso, note o uso de **sizeof** para calcular o número de bytes necessários a uma matriz inteira de 4,10. Isso garante a portabilidade desse programa para computadores com inteiros de tamanhos diferentes.

Problemas com Ponteiros

Nada pode trazer mais problemas do que um ponteiro selvagem! Ponteiros são uma bênção que pode trazer problemas. Eles lhe dão uma capacidade formidável

e são necessários em muitos programas. Ao mesmo tempo, quando um ponteiro acidentalmente contém um valor errado, ele pode ser o erro mais difícil de descobrir.

Um erro com um ponteiro irregular é difícil de encontrar, porque o ponteiro não é realmente o problema. O problema é que, toda vez que você realiza uma operação usando o ponteiro, está lendo ou escrevendo em alguma parte desconhecida da memória. Se você ler essa porção, o pior que pode acontecer será obter lixo. Contudo, se você escrever nela, estará escrevendo sobre outras partes do seu código ou dados. Isso pode não aparecer até mais adiante na execução do seu programa e levá-lo a procurar o erro no lugar errado. Pode haver pouca ou nenhuma evidência a sugerir que o ponteiro seja o problema. Esse tipo de erro leva programadores a perder horas de sono. Como os erros de ponteiros são verdadeiros pesadelos, faça o possível para nunca gerar um. Para ajudar você a evitá-los, alguns dos erros mais comuns são discutidos aqui. O exemplo clássico de um erro com ponteiro é o *ponteiro não inicializado*. Considere este programa.

```
/*Este programa está errado. */
void main(void)
{
  int x, *p;

  x = 10;
  *p = x;
}
```

Ele atribui o valor 10 a alguma posição de memória desconhecida. O ponteiro **p** nunca recebeu um valor; portanto, ele contém lixo. Esse tipo de problema sempre passa despercebido, quando seu programa é pequeno, porque as probabilidades estão a favor de que **p** contenha um endereço "seguro" — um endereço que não esteja em seu código, área de dados ou sistema operacional. Contudo, quando seu programa cresce, a probabilidade de **p** apontar para algo vital aumenta. Por fim, seu programa pára de funcionar. A solução é sempre ter certeza de que um ponteiro está apontando para algo válido antes de usá-lo.

Um segundo erro comum é provocado por um simples equívoco sobre como usar um ponteiro. Considere o seguinte:

```
/* Este programa está errado. */
#include <stdio.h>

void main(void) /*
{
  int x, *p;
```

Essa não é uma boa forma de inicializar as matrizes **first** e **second** com os números de 0 a 19. Embora possa funcionar em alguns compiladores, sob certas circunstâncias, está-se assumindo que as duas matrizes serão colocadas uma após a outra com **first** primeiro. Isso pode não ser sempre o caso.

O próximo programa ilustra um tipo de erro muito perigoso. Veja se você é capaz de encontrá-lo.

```
/* Esse programa tem um erro. */
#include <string.h>
#include <stdio.h>

void main(void)
{
  char *p1;
  char s[80];

  p1 = s;
  do {
    gets(s); /* lê uma string */

    /* imprime o equivalente decimal de cada caractere */
    while(*p1) printf(" %d", *p1++);

  } while (strcmp(s, "done"));
}
```

Esse programa usa **p1** para imprimir os valores ASCII associados a cada caractere contido em **s**. O problema é que o endereço de **s** é atribuído a **p1** apenas uma vez. Na primeira iteração do laço, **p1** aponta para o primeiro caractere em **s**. Porém, na segunda iteração, ele continua de onde foi deixado, porque ele não é reinicializado para o começo de **s**. O próximo caractere pode ser parte de uma outra string, uma outra variável ou um pedaço do programa. A maneira apropriada de escrever esse programa é

```
/* Esse programa está correto. */
#include <string.h>
#include <stdio.h>

void main(void)
{
  char *p1;
  char s[80];
```

```
  do {
    p1 = s;
    gets(s); /* lê uma string */
    /* imprime o equivalente decimal de cada caractere */
    while(*p1) printf(" %d", *p1++);

  } while (strcmp(s, "done"));
}
```

Aqui, cada vez que o laço repete, **p1** é ajustado para o início da string. Em geral, você deve lembrar-se de reinicializar um ponteiro se ele for reutilizado.

O fato de manipular ponteiros incorretamente e poder provocar erros traiçoeiros não é razão para não usá-los. Apenas seja cuidadoso e assegure-se de que você sabe para onde cada ponteiro está apontando antes de usá-lo.