第5章 指　针

正确地理解和使用指针是成功地进行C/C++程序设计的关键。其理由有三个：第一，指针提供了函数修改它们的调用变元的方法；第二，指针支持动态分配；第三，指针可以改善某些例程的效率。此外，像在第二部分中看到的那样，指针在C++中还有别的作用。

指针是C/C++中最强也是最危险的特征之一。例如，未初始化的指针（或包含无效值的指针）可以使系统崩溃。或许更糟的是，指针极易用错，引起难以发现的程序故障。

由于指针非常重要，并且它们很容易误用，本章将详细讨论指针的问题。

5.1 什么是指针

指针是包含内存地址的变量，这个地址是内存中另一个对象（通常是另一个变量）的位置。例如，如果一个变量包含另一个变量的地址，我们说第一个变量指向第二个变量。图5.1显示了这种情况。

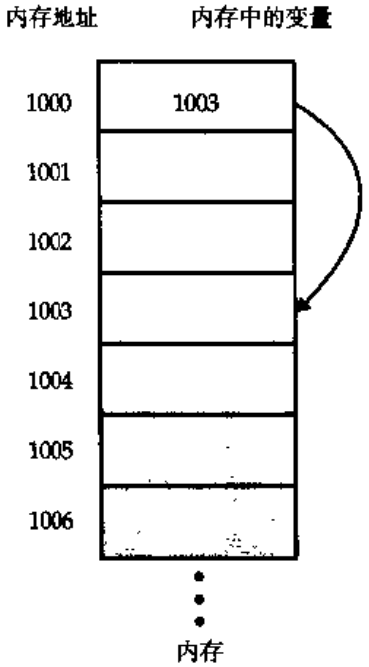

图5.1　一个变量指向另一个变量

5.2 指针变量

如果一个变量要包含一个指针，必须进行声明。指针声明由一个基类型、一个*和变量名组成。声明一个指针变量的一般形式是：

*type *name;*

其中，type 是指针的基类型并且可以是任何有效的类型。指针变量的名字由 name 指定。

指针的基类型定义了指针可以指向的变量的类型。从技术上讲，任何指针可以指向内存中的任何地方。然而，所有指针运算的完成都与它的基类型有关，所以重要的是正确声明指针（本章后面将讨论指针运算）。

5.3 指针运算符

我们在第2章中讨论了指针运算符，这里将详细看看它们，首先从回顾它们的基本运算开始。有两种专用的指针运算符 * 和 &。& 是一个一元运算符，返回其操作数的内存地址（记住，一元运算符仅要求一个操作数）。例如，

```
m = &count;
```

把变量 count 的内存地址放到 m 中。这个地址是这个变量在计算机中的内部位置，它与 count 的值没有任何关系。可以把 & 理解为返回“什么内容的地址”。因此，前面的赋值语句意味着“m 接受 count 的地址”。

为了更好地理解上面的赋值语句，假定变量 count 使用内存地址 2000 来存储它的值，同时假定 count 的值是 100。那么，在经过了前面的赋值后，m 的值将变为 2000。

第二个指针运算符 * 是 & 的补充。它是一个一元运算符，该运算符返回其后地址的值。例如，如果 m 包含变量 count 的内存地址，那么

```
q = *m;
```

把 count 的值放到 q 中。因此，q 的值变为 100，因为 100 存储在地址 2000 处，它是存储在 m 中的内存地址。可以把 * 理解为“从地址中取值”。此时，前面的语句意味着“q 收到地址 m 处的值”。

& 和 * 比所有其他的算术运算符有更高的优先级，但是一元减除外，一元减和它们有同样的优先级。

必须确定指针变量总是指向正确的数据类型。例如，当声明一个指针为 int 型时，编译器假定它包含的任何地址指向一个整数变量——不管它实际上是不是。因为可以把任何地址赋给一个指针变量，所以虽然下面的程序可以编译而没有错误，但是不会生成需要的结果：

```
#include <stdio.h>

int main(void)
{
  double x = 100.1, y;
  int *p;

  /* The next statement causes p (which is an
     integer pointer) to point to a double. */
  p = (int *)&x;

  /* The next statement does not operate as
     expected. */
  y = *p;

  printf("%f", y); /* won't output 100.1 */
  return 0;
}
```

这不会把x的值赋给y。因为p被声明为一个整型指针，仅仅4字节的信息（假定整数为4字节）被传给y，不是正常时组成double型数的8字节。

注意：在C++中，不使用显式的类型转换就把一种类型的指针转换为另一种类型是不合法的。在C中，强制转换用在大多数指针转换中。

5.4 指针表达式

通常，涉及指针的表达式与其他表达式所用的原则一样。本节讨论指针表达式的几个特殊方面。

5.4.1 指针赋值

像使用任何变量一样，可以在赋值语句的右边使用一个指针来把它的值赋给另一个指针。例如，

```
#include <stdio.h>

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

  p1 = &x;
  p2 = p1;

  printf(" %p", p2); /* print the address of x, not x's value! */

  return 0;
}
```

p1和p2现在都指向x。使用%p printf( )格式限定符显示x的地址，它使printf( )以宿主计算机使用的格式显示一个地址。

5.4.2 指针运算

对于指针，只能使用两种算术运算：加和减。为了理解指针运算的过程，假设p1是一个整型指针，其当前值为2000，同时假定整数是2字节长。那么，经过

```
p1++;
```

运算后，p1的值变为2002，而不是2001，因为p1每递增一次，它就将指向下一个整数，递减也是一样。例如，假定p1的值为2000，那么，在经过下面的表达式运算后，

```
p1--;
```

p1的值将变为1998。

从前面的例子中，可以得出下面的指针运算规则。每次指针递增时，它指向其基类型的下一个元素的内存位置。每次指针递减时，它指向前一个元素的位置。在使用字符指针时，这看起来很像“正常的”算术运算，因为字符总是1字节长的。所有其他指针将递增或递减它们所指向的数据的长度这种方法保证了指针总是指向它的基类型的适宜的元素。图5.2演示了这个

概念。

指针运算并不限于递增和递减运算符。例如，可以向指针或从指针中加或减去整数。下面的表达式

```
p1 = p1 + 12;
```

让p1指向p1类型的、在其现在指向的元素后面的第12个元素。

除了加、减指针和整数外，只允许另外一种算术运算：可以从一个指针中减去另一个指针，结果是两个指针之间的基类型的对象数。所有其他算术运算都是非法的。具体来讲，不能对指针进行乘或除操作，不能把两个指针相加，不能对指针应用位运算符，不能向指针或从指针中加或减float或double类型的数据。

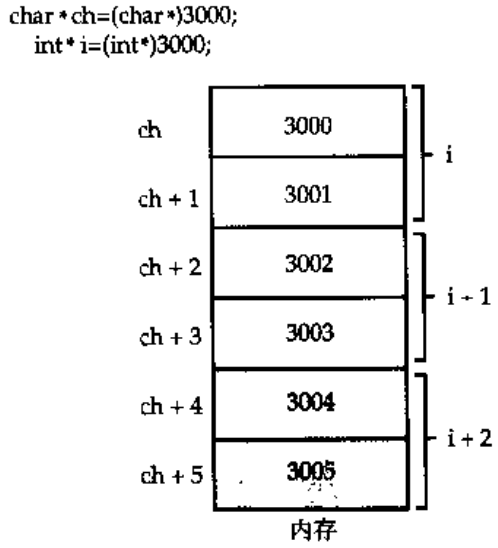

图5.2 所有的指针运算都与它的基类型相关（假定整数为2字节长）

5.4.3 指针的比较

可以在一个关系表达式中比较两个指针。例如，如果有两个指针p和q，下面的表达式是完全合法的：

```
if(p<q) printf("p points to lower memory than q\n");
```

一般来讲，当两个或更多的指针指向同一个对象，如数组时，可以使用指针进行比较。例如，开发一对存储和检索整数值的堆栈例程。堆栈是一个使用“先进后出”存取的线性表，常被用于和桌子上的盘子相比较——放在底下的盘子最后使用。堆栈常用在编译器、解释器、电子表格和其他系统相关的软件中。要创建一个堆栈，需要两个函数：push( )和pop( )。push( )函数把数据压入堆栈，而pop( )函数将数据弹出。这里显示的例程用一个简单的main( )函数调用它们，程序把用户键入的数值压入堆栈。如果键入了0，则从堆栈中弹出一个值；键入-1时，程序停止运行。

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

#define SIZE 50
```

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

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

int main(void)
{
  int value;

  tos = stack; /* tos points to the top of stack */
  p1 = stack; /* initialize p1 */

  do {
    printf("Enter value: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("value on top is %d\n", pop());
  } while(value!=-1);

  return 0;
}

void push(int i)
{
  p1++;
  if(p1==(tos+SIZE)) {
    printf("Stack Overflow.\n");
    exit(1);
  }
  *p1 = i;
}

int pop(void)
{
  if(p1==tos) {
    printf("Stack Underflow.\n");
    exit(1);
  }
  p1--;
  return *(p1+1);
}
```

可以看到，堆栈的内存是由数组 stack 提供的。指针 p1 被设置为指向 stack 中的第一个元素，并用于访问堆栈。变量 tos 存放堆栈顶部的内存地址，用于防止堆栈满或下溢。一旦对堆栈进行了初始化，就可以使用 push( )和 pop( )函数了。函数 push( )和 pop( )对指针 p1 进行关系测试以检测是否出现了溢出错误。在 push( )中，通过增加 SIZE（堆栈的大小）到 tos 中，可以把 p1 和堆栈顶部进行比较，这会防止堆栈溢出。在 pop( )中，必须在返回语句中加上括号。如果没有使用括号，语句将是：

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

这会返回 p1 地址中的值加 1，而不是地址 p1+1 的值。

5.5 指针和数组

在指针和数组之间有非常密切的关系。考虑下面的程序段：

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

其中，p1被设置为str中第一个元素的地址。如果要访问str中的第5个元素，可以写成：

```
str[4]
```

或

```
*(p1-4)
```

这两条语句都将返回第5个元素。记住，数组下标从0开始。要访问第5个元素，必须使用4来表示str。也可以给指针p1加4来表示访问第5个元素，因为p1现在指向str数组的第1个元素（回忆一下不带下标的数组名返回数组的起始地址，即第1个元素的地址）。

前面的例子适用于普通的场合。本质上，C/C++提供了两种访问数组元素的方法：指针运算和数组下标检索。尽管标准数组下标符号有时更容易理解，指针运算更快些。因为在编程过程中速度经常是要考虑的因素，所以C/C++程序员一般使用指针来访问数组元素。

下面两个版本的putstr()（其中一个用下标检索，另一个使用指针），演示了在使用数组下标的地方是如何使用指针的。putstr()函数一次一个字符地把一个字符串写到标准输出设备上。

```
/* Index s as an array. */
void putstr(char *s)
{
  register int t;

  for(t=0; s[t]; ++t) putchar(s[t]);
}

/* Access s as a pointer. */
void putstr(char *s)
{
  while(*s) putchar(*s++);
}
```

大多数专业C/C++程序员会发现第二种形式更易读、易理解。事实上，在C/C++中，这种程序通常都是用指针形式编写的。

5.5.1 指针数组

指针可以像任何其他数据类型那样数组化。大小为10的int指针数组的声明是：

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

要把一个整数变量var的地址赋给指针数组的第3个元素，可写为：

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

若求var的值，可写为：

```
*x[2]
```

如果要将一个指针数组传递给一个函数，可以使用传递其他数组时所用的同样的方法，即简单地用数组名调用函数，无需任何下标。例如，可以接收数组 x 的函数如下：

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

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

记住，q 不是一个指向整数的指针，而是一个指向整型指针数组的指针。因此，需要把参数 q 声明为指向整型指针数组的指针。不能把 q 简单地声明为整型指针，因为它本身就不是整型指针。

指针数组经常用于存放指向字符串的指针。可以创建一个函数，这个函数根据错误代码编号来输出错误信息，如下所示：

```
void syntax_error(int num)
{
  static char *err[] = {
    "Cannot Open File\n",
    "Read Error\n",
    "Write Error\n",
    "Media Failure\n"
  };

  printf("%s", err[num]);
}
```

数组 err 存放指向每个字符串的指针。如你所见，调用 syntax_error( )中的 printf( )时用的是字符指针，这个字符指针指向由传递给函数的错误编号检索的各种错误信息之一。例如，如果 num 被传递了一个 2，将显示 Write Error 信息。

有趣的是，命令行参数 argv 是一个字符指针数组（参见第 6 章）。

5.6 多级间址

可以构造一个指向另一个指针的指针，而后者指向目标值。这种情况称为多级间址（multiple indirection）或指针的指针。指针的指针可能令人感到困惑，图 5.3 帮助我们澄清了多级间址的概念。可以看到，正常指针的值是包含所要值的对象的地址。而对于指针的指针，第一个指针是第二个指针的地址，而第二个指针指向包含所要值的对象。多级间址可在所需要的层次上执行，但是很少有需要一个以上指针的指针的情况。事实上，过度的间址很难跟踪并且易于产生概念上的错误。

注意：不要把多级间址和高级数据结构，例如使用指针的链表相混淆。这是两个完全不同的概念。

将变量声明为指针的指针时，必须在变量名前加一个额外的星号。例如，下面的声明告诉编译器 newbalance 是一个指向 float 型指针的指针：

```
float **newbalance;
```

注意 newbalance 不是一个指向浮点数的指针，而是一个指向 float 指针的指针。

要访问被一个指针的指针间接指向的目标值，必须两次使用星号运算符，如下例所示：

```
#include <stdio.h>

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

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

  printf("%d", **q); /* print the value of x */

  return 0;
}
```

其中，p 声明为指向整数的指针，q 声明为指向另一指向整数的指针的指针。调用 printf( ) 时将在屏幕上显示数字 10。

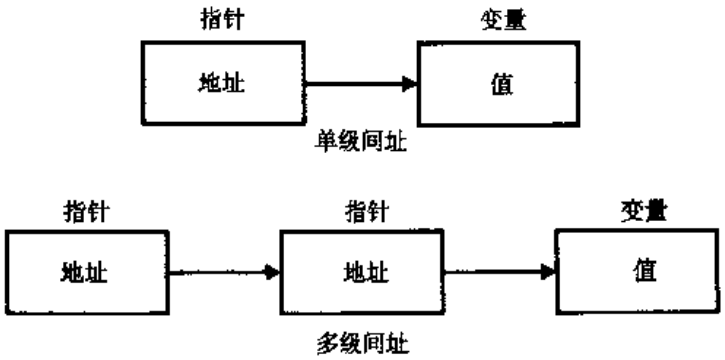

图 5.3 单级间址和多级间址

5.7 初始化指针

一个非静态本地指针在声明之后、但未赋值时其所包含的值是未知的（全局和静态本地指针被自动初始化为null）。如果在尝试使用指针时没有赋予它一个有效值，可能会导致程序崩溃，甚至会破坏计算机的操作系统，这是一个非常严重的错误。

大多数C/C++程序员在操作指针时都遵循一个重要的惯例：当前没有指向一个有效的内存位置的指针被赋予空值（即0）。按照惯例，空指针意味着它不指向任何地方且不应该使用。然而，值为 null 的指针并不安全，null 的使用仅仅是程序员自己的习惯而已，并不是C或C++施加的原则。例如，如果在赋值语句的左边使用空指针，仍然有使程序或操作系统崩溃的危险。

因为我们假定空指针是未用的，所以为了使带有指针的程序更容易编写、更有效率，可以使用空指针。例如，可以使用空指针来标志指针数组的末尾。当遇到空值时，访问那个数组的例程知道它到达了数组末尾。下面显示的 search( )函数演示了这种方法。

```
/* look up a name */
int search(char *p[], char *name)
{
  register int t;
```

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

    return -1; /* not found */
}
```

直到找到了一个匹配或遇到了一个空指针，在search( )内的for循环才终止运行。假定数组的结尾用一个空值标志，当到达它时，控制循环的条件失败。

C/C++程序员经常要初始化字符串，在“指针数组”一节中的syntax_error( )函数里我们见过这样一个例子。下面的字符串声明是关于初始化问题的另一个变种。

```
char *p = "hello world";
```

可以看出，指针p不是一个数组。这种初始化方法可行的原因与编译器的操作方式有关。所有的C/C++编译器都会创建称为字符串表的东西，用于存储程序所用的字符串常量。因此，前面的声明语句把hello world的地址（存储在字符串表中）放到指针p中。在整个程序中，可以像使用任何其他字符串一样使用p(除了它不应该改变外)。例如，下面的程序是完全有效的：

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

char *p = "hello world";

int main(void)
{
  register int t;

  /* print the string forward and backwards */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);

  return 0;
}
```

在标准C++中，从技术上讲，字符串文字的类型是const char *，但是C++提供了一种到char *的自动转换。因此，前面的程序仍然是有效的。然而，我们并不赞成使用这种自动转换特征，它意味着在编写新代码时不要使用它。对于新程序，应该假定字符串文字实际上是常量，前面程序中的p的声明应该像这样写：

```
const char *p = "hello world";
```

5.8 指向函数的指针

函数指针是C++中一个特别令人困惑但却非常有用的特征。即使函数不是变量，但它却具有内存物理地址，而且该地址可以赋给一个指针。这个地址是函数的入口点，且是函数调用时使用的地址。一旦一个指针指向函数，就可以通过那个指针调用这个函数。函数指针也允许函数作为参数传递给其他函数。

仅用函数名就可以获得函数地址，无需任何括号或参数（这类似于数组地址获得的方式，只用数组名，不用下标就可以获得数组地址）。要明白这是怎样工作的，研究一下下面的程序，特别要注意声明语句：

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

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

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

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
```

在调用check( )函数时，两个字符指针和一个函数指针被作为参数传递。在check( )函数内，变元被作为字符指针和函数指针声明。注意函数指针是怎样声明的。当声明其他函数指针时，尽管函数的返回类型和参数可以不同，必须使用类似的形式。为了让编译器正确地翻译这条语句，*cmp周围的括号是必需的。在check( )内，表达式

```
(*cmp)(a, b)
```

调用strcmp( )，带参数a和b的函数cmp指向该函数。在*cmp两边的括号是必需的，这是通过指针调用函数的一种方法。也可以使用第二种更简单的语法，如下所示：

```
cmp(a, b);
```

你会经常见到第一种形式的原因是：它告诉读代码的人正在通过一个指针调用函数（即，cmp是一个函数指针，不是函数名），除此之外，两个表达式是等价的。

注意，可以直接使用strcmp( )来调用check( )，如下所示：

```
check(s1, s2, strcmp);
```

这消除了对附加的指针变量的需要。

你也许要问：为什么人们要用这种方式写程序呢？很明显，在前面的例子中，并没有得到任何好处，反而引入了很大的混乱。然而，有时把函数作为参数传递或创建函数数组也有很大的优点。例如，当编写编译器或解释器程序时，词法分析器（即用来分析表达式的部分）经常调用各种支持函数，如计算算术操作（正弦、余弦、正切等）、执行I/O操作或访问系统资源的那些函数。代替用大的switch语句把所有函数列入其中，可以创建一个函数指针数组。用这种

方法，可以借助其下标选择相应的函数。通过研究前面例子的扩展版本，可以掌握这种用法。在这个程序中，可以用 check( )来检查字符是否相等，或数字是否相等，只要简单地用不同的比较函数调用它就行了。

```
#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);

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

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

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}

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

在这个程序中，如果键入一个字母，就把 strcmp( )传递给 check( )，否则，使用 numcmp( )。因为 check( )调用它被传递的函数，所以在不同的情况中，它可以使用不同的比较函数。

5.9 C语言的动态分配函数

指针对C/C++的动态分配系统提供了必需的支持。动态分配（Dynamic allocation）是一种方法，借助于这种方法，程序可在运行时获得内存。如你所知，全局变量是在编译时分配内存的，局部变量使用堆栈。在程序执行时，全局变量和局部变量都不能加入。然而，有时事先不可能知道程序所需要的存储空间，例如，程序可能使用动态数据结构，如链表或二叉树。这种结构本质上是动态的，会随着需要而增加或减小。为了实现这种数据结构，就要求程序能够分配和释放内存空间。

C++实际上支持两种动态分配系统：由C定义的那种和C++特有的那种。C++特有的系统比起由C使用的系统来有几个方面的改进，我们将在本书第二部分讨论这种方法。这里，只描述C的动态分配函数。

由C语言的动态分配函数分配的内存是从堆中获得的——堆是位于你的程序、程序的永久存储区和堆栈之间的空闲内存区域。尽管堆的大小是未知的，通常它包含一个较大的空闲内存。

C语言的动态分配系统的核心由函数malloc( )和free( )组成（大多数编译器支持其他几个动态分配函数，但是这两个是最重要的）。这些函数协同工作，使用空闲内存区来建立和维护可用存储列表。函数malloc( )分配内存，而函数free( )释放它。即，每调用一次malloc( )，就有一部分剩余的内存被分配。每调用一次free( )，就会把内存返回给系统。使用这两个函数的任何程序都应该包括头文件stdlib.h (C++程序也可以使用C++风格的头文件<cstdlib>)。

函数malloc( )的原型是：

```
void *malloc(size_t number_of_bytes);
```

其中，number_of_bytes是希望分配的内存字节数（在stdlib.h中size_t的类型被定义为无符号整型）。函数malloc( )返回一个void *类型的指针，这意味着可以把它赋给任何类型的指针。在成功的调用之后，malloc( )返回指向从堆中分配的内存区的第一个字节的指针。如果没有足够的内存来满足malloc( )的要求，就会出现分配失败的情况，malloc( )返回一个空值。

下面的代码段分配了1000字节的连续内存空间：

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

赋值后，p指向1000字节的空闲内存的起始处。

在前面的例子中，注意没有使用强制类型转换来把malloc( )的返回值赋给p。在C语言中，void *指针被自动转换为赋值语句左边的指针类型。然而，重要的是要理解：这种自动转换在C++中并不出现。在C++中，当void *指针被赋值给另一种指针类型时，需要一种明确的类型转换。因此，在C++中，前面的赋值语句必须写为：

```
p = (char *) malloc(1000);
```

一般的原则是：在C++中，当把一种类型的指针赋给（或转换为）另一种类型的指针时，必须使用强制类型转换，这是C和C++语言最根本的区别之一。

下面的例子为50个整数分配空间，注意使用了sizeof来保证可移植性。

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

因为堆不是无限大的，所以在分配内存时，必须检查由malloc( )返回的值，以确保在使用这个指针前它不为空。使用空指针几乎肯定会崩溃程序。下面的代码段演示了分配内存并测试指针的有效性的合适方法。

```
p = (int *) malloc(100);
if(!p) {
  printf("Out of memory.\n");
  exit(1);
}
```

当然，也可以用其他错误处理程序代替exit( )，但要确保不要使用空指针p。

函数free( )与malloc( )相反，它将以前分配的内存返回给系统。一旦释放了内存，就可以在后续的malloc( )调用中重新使用。函数free( )的原型是：

```
void free(void *p);
```

其中，p是指向以前用malloc( )分配的内存的指针。关键是永远不要使用无效的参数调用free( )，否则，将破坏自由链。

5.10 指针应用中的问题

没有什么比混乱的指针带来的麻烦更多了。指针是一个好坏兼有的东西，它的功能强大，很多程序都需要它。同时，指针有时会包含错误的值，它可以是最难发现的bug。

错误指针很难发现，因为指针本身不是问题。问题是：每次使用坏指针执行操作时，都会对某些未知的存储单元进行读或写操作。如果是读操作，最坏的可能是得到无效数据。然而，如果是写操作，可能会重写其他代码或数据。这种错误可能要到程序执行的后期才会显示出来，可能引导你在错误的地方寻找故障，可能没有任何证据来表明指针是问题的根源。这种故障会使程序员一次一次地陷入失眠状态。

因为指针错误如同恶梦一般，所以这里讨论一些常见的指针错误。指针错误的一个经典例子是未初始化的指针。考虑下面这个程序：

```
/* This program is wrong. */
int main(void)
{
  int x, *p;

  x = 10;
  *p = x;

  return 0;
}
```

这个程序把10这个值赋给某个未知的内存单元。下面是原因：因为指针p没有赋予任何值，当执行赋值语句*p = x时，它包含一个未知数值。这使得x的值被写到某个未知的内存单元中。当程序很小时，通常不会注意到这个问题，因为p存放“安全”地址（不是在代码、数据区、或操作系统中）的可能性较大。然而，随着程序的增长，p指向关键地址的可能性增大，最终，导致程序无法工作。解决方案是总是确保指针在使用前指向某种有效的东西。

第二个常见的错误是由对怎样使用指针产生误解而引起的。考虑下面的程序：

```
/* This program is wrong. */
#include <stdio.h>

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

  x = 10;
  p = x;

  printf("%d", *p);
```

```
  return 0;
}
```

对printf( )的调用不会在屏幕上显示x的值10。它显示出某个未知的值，因为赋值语句：

```
p = x;
```

是错误的。那条语句把10这个值赋给指针p。然而，p是用来存放一个地址而不是一个值的。要更正这个程序，可以写为：

```
p = &x;
```

另一个有时发生的错误是由对变量在内存中的位置所作的不正确的假定引起的。你永远不会知道数据放在内存中的哪个位置，或者它是否是以同样的方式放置的，或者是否每种编译器按同样的方式处理它。由于这些原因，在不指向共同对象的指针之间做任何比较都会产生意想不到的结果。例如，下面的程序是错误的（在极特殊的情况下，你可能使用像这样的东西来确定变量的相对位置）。

```
char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;
if(p1 < p2) . . .
```

当假定两个邻接的数组可作为一个数组通过简单的指针递增进行越界存取时，也会出现相关的错误。例如，

```
int first[10], second[10];
int *p, t;
p = first;
for(t=0; t<20; ++t) *p++ = t;
```

这种用数字0到19将数组first和second初始化不是一种好方法。即使在某些环境的编译器下它可能工作，但是它假定两个数组会被放回到内存中，且first在前。实际情况并不总是如此。

下面的程序演示了一种非常危险的故障。看一下你是否能够找到它。

```
/* This program has a bug. */
#include <string.h>
#include <stdio.h>

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

  p1 = s;
  do {
    gets(s); /* read a string */

    /* print the decimal equivalent of each
       character */
    while(*p1) printf(" %d", *p1++);
```

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

  return 0;
}
```

这个程序使用p1来打印包含于s中的字符的ASCII值。问题是p1仅被赋予s的地址一次。第一次循环时，p1指向s中的第一个字符。然而，第二次循环时，它将从它离开的那里继续，因为它没有被重置到s的起始位置。而这下一个字符可能是第二个字符串、另一个变量或程序段的一部分。这个程序的正确写法是：

```
/* This program is now correct. */
#include <string.h>
#include <stdio.h>

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

  do {
    p1 = s;
    gets(s); /* read a string */

  /* print the decimal equivalent of each
     character */
  while(*p1) printf(" %d", *p1++);

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

  return 0;
}
```

其中，循环每进行一次，p1被设置到字符串的起始处。一般来讲，如果要重用指针，应该记住对它进行重新初始化。

实际上，指针的不正确使用，会导致严重的故障，但这并不是避免使用指针的理由。在使用指针前，只要小心一点，确信指针指向哪里即可。