第 11 章

动态内存分配

数组的元素存储于内存中连续的位置上。当一个数组被声明时，它所需要的内存在编译时就被分配。但是，你也可以使用动态内存分配在运行时为它分配内存。在本章中，我们将研究这两种技巧的区别，看看什么时候我们应该使用动态内存分配以及怎样进行动态内存分配。

11.1 为什么使用动态内存分配

当你声明数组时，你必须用一个编译时常量指定数组的长度。但是，数组的长度常常在运行时才知道，这是由于它所需要的内存空间取决于输入数据。例如，一个用于计算学生等级和平均分的程序可能需要存储一个班级所有学生的数据，但不同班级的学生数量可能不同。在这些情况下，我们通常采取的方法是声明一个较大的数组，它可以容纳可能出现的最多元素。

提示：

这种方法的优点是简单，但它有好几个缺点。首先，这种声明在程序中引入了人为的限制，如果程序需要使用的元素数量超过了声明的长度，它就无法处理这种情况。要避免这种情况，显而易见的方法是把数组声明得更大一些，但这种做法使它的第 2 个缺点进一步恶化。如果程序实际需要的元素数量比较少时，巨型数组的绝大部分内存空间都被浪费了。这种方法的第 3 个缺点是如果输入的数据超过了数组的容纳范围时，程序必须以一种合理的方式作出响应。它不应该由于一个异常而失败，但也不应该打印出看上去正确实际上却是错误的结果。实现这一点所需要的逻辑其实很简单，但人们在头脑中很容易形成“数组永远不会溢出”这个概念，这就诱使他们不去实现这种方法。

11.2 malloc 和 free

C 函数库提供了两个函数，malloc 和 free，分别用于执行动态内存分配和释放。这些函数维护一个可用内存池。当一个程序另外需要一些内存时，它就调用 malloc 函数，malloc 从内存池中提取一块合适的内存，并向该程序返回一个指向这块内存的指针。这块内存此时并没有以任何方式进行初始化。如果对这块内存进行初始化非常重要，你要么自己动手对它进行初始化，要么使用 calloc

函数（在下一节描述）。当一块以前分配的内存不再使用时，程序调用 free 函数把它归还给内存池供以后之需。

这两个函数的原型如下所示，它们都在头文件 stdlib.h 中声明。

```
void    *malloc( size_t size );
void    free( void *pointer );
```

malloc 的参数就是需要分配的内存字节（字符）数[1]。如果内存池中的可用内存可以满足这个需求，malloc 就返回一个指向被分配的内存块起始位置的指针。

malloc 所分配的是一块连续的内存。例如，如果请求它分配 100 个字节的内存，那么它实际分配的内存就是 100 个连续的字节，并不会分开位于两块或多块不同的内存。同时，malloc 实际分配的内存有可能比你请求的稍微多一点。但是，这个行为是由编译器定义的，所以你不能指望它肯定会分配比你的请求更多的内存。

如果内存池是空的，或者它的可用内存无法满足你的请求，会发生什么情况呢？在这种情况下，malloc 函数向操作系统请求，要求得到更多的内存，并在这块新内存上执行分配任务。如果操作系统无法向 malloc 提供更多的内存，malloc 就返回一个 NULL 指针。因此，对每个从 malloc 返回的指针都进行检查，确保它并非 NULL 是非常重要的。

free 的参数必须要么是 NULL，要么是一个先前从 malloc、calloc 或 realloc（稍后描述）返回的值。向 free 传递一个 NULL 参数不会产生任何效果。

malloc 又是如何知道你所请求的内存需要存储的是整数、浮点值、结构还是数组呢？它并不知情——malloc 返回一个类型为 void *的指针，正是缘于这个原因。标准表示一个 void *类型的指针可以转换为其他任何类型的指针。但是，有些编译器，尤其是那些老式的编译器，可能要求你在转换时使用强制类型转换。

对于要求边界对齐的机器，malloc 所返回的内存的起始位置将始终能够满足对边界对齐要求最严格的类型的要求。

11.3 calloc 和 realloc

另外还有两个内存分配函数，calloc 和 realloc。它们的原型如下所示：

```
void    *calloc( size_t num_elements,
                 size_t element_size );
void    realloc( void *ptr, size_t new_size );
```

calloc 也用于分配内存。malloc 和 calloc 之间的主要区别是后者在返回指向内存的指针之前把它初始化为 0。这个初始化常常能带来方便，但如果你的程序只是想把一些值存储到数组中，那么这个初始化过程纯属浪费时间。calloc 和 malloc 之间另一个较小的区别是它们请求内存数量的方式不同。calloc 的参数包括所需元素的数量和每个元素的字节数。根据这些值，它能够计算出总共需要分配的内存。

realloc 函数用于修改一个原先已经分配的内存块的大小。使用这个函数，你可以使一块内存扩大或缩小。如果它用于扩大一个内存块，那么这块内存原先的内容依然保留，新增加的内存添加到原先内存块的后面，新内存并未以任何方法进行初始化。如果它用于缩小一个内存块，该内存块尾

[1] 注意这个参数的类型是 size_t，它是一个无符号类型，定义于 stdlib.h。

部的部分内存便被拿掉，剩余部分内存的原先内容依然保留。

如果原先的内存块无法改变大小，realloc 将分配另一块正确大小的内存，并把原先那块内存的内容复制到新的块上。因此，在使用 realloc 之后，你就不能再使用指向旧内存的指针，而是应该改用 realloc 所返回的新指针。

最后，如果 realloc 函数的第 1 个参数是 NULL，那么它的行为就和 malloc 一模一样。

11.4　使用动态分配的内存

这里有一个例子，它用 malloc 分配一块内存。

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

符号 NULL 定义于 stdio.h，它实际上是字面值常量 0。它在这里起着视觉提醒器的作用，提醒我们进行测试的值是一个指针而不是整数。

如果内存分配成功，那么我们就拥有了一个指向 100 个字节的指针。在整型为 4 个字节的机器上，这块内存将被当作 25 个整型元素的数组，因为 pi 是一个指向整型的指针。

提示：

但是，如果你的目标就是获得足够存储 25 个整数的内存，这里有一个更好的技巧来实现这个目的。

```
pi = malloc( 25 * sizeof( int ) );
```

这个方法更好一些，因为它是可移植的。即使是在整数长度不同的机器上，它也能获得正确的结果。

既然你已经有了一个指针，那么你该如何使用这块内存呢？当然，你可以使用间接访问和指针运算来访问数组的不同整数位置，下面这个循环就是这样做的，它把这个新分配的数组的每个元素都初始化为 0：

```
int     *pi2, i;
...
pi2 = pi;
for( i = 0; i < 25; i += 1 )
        *pi2++ = 0;
```

正如你所见，你不仅可以使用指针，也可以使用下标。下面的第 2 个循环所执行的任务和前面一个相同。

```
int     i;
...
for( i = 0; i < 25; i += 1 )
        pi[i] = 0;
```

11.5　常见的动态内存错误

在使用动态内存分配的程序中，常常会出现许多错误。这些错误包括对 NULL 指针进行解引用

操作、对分配的内存进行操作时越过边界、释放并非动态分配的内存、试图释放一块动态分配的内存的一部分以及一块动态内存被释放之后被继续使用。

警告：

动态内存分配最常见的错误就是忘记检查所请求的内存是否成功分配。程序 11.1 展现了一种技巧，可以很可靠地进行这个错误检查。MALLOC 宏接受元素的数目能及每种元素的类型，计算总共需要的内存字节数，并调用 alloc 获得内存[1]。alloc 调用 malloc 并进行检查，确保返回的指针不是 NULL。

这个方法最后一个难解之处在于第 1 个非比寻常的#define 指令。它用于防止由于其他代码块直接塞入程序而导致的偶尔直接调用 malloc 的行为。增加这个指令以后，如果程序偶尔调用了 malloc，程序将由于语法错误而无法编译。在 alloc 中必须加入#undef 指令，这样它才能调用 malloc 而不至于出错。

警告：

动态内存分配的第二大错误来源是操作内存时超出了分配内存的边界。例如，如果你得到一个 25 个整型的数组，进行下标引用操作时如果下标值小于 0 或大于 24 将引起两种类型的问题。

第 1 种问题显而易见：被访问的内存可能保存了其他变量的值。对它进行修改将破坏那个变量，修改那个变量将破坏你存储在那里的值。这种类型的 bug 非常难以发现。

第 2 种问题不是那么明显。在 malloc 和 free 的有些实现中，它们以链表的形式维护可用的内存池。对分配的内存之外的区域进行访问可能破坏这个链表，这有可能产生异常，从而终止程序。

```
/*
** 定义一个不易发生错误的内存分配器。
*/
#include <stdlib.h>

#define   malloc                    不要直接调用 malloc!
#define   MALLOC(num,type) (type *)alloc( (num) * sizeof(type) )
extern    void *alloc( size_t size );
```

程序 11.1a　错误检查分配器:接口　　alloc.h

```
/*
** 不易发生错误的内存分配器的实现
*/
#include <stdio.h>
#include "alloc.h"
#undef    malloc

    void *
    alloc( size_t size )
    {
    void  *new_mem;
    /*
    ** 请求所需的内存，并检查确实分配成功
    */
    new_mem = malloc( size );
```

[1] #define 宏在第 14 章详细描述。

```
    if( new_mem == NULL ){
    printf( "Out of memory!\n" );
    exit( 1 );
    }
    return new_mem;
}
```

程序 11.1b　错误检查分配器：实现　　alloc.c

```
    /*
    ** 一个使用很少引起错误的内存分配器的程序
    */
    #include "alloc.h"

    void
    function()
    {
        int  *new_memory;

        /*
        ** 获得一串整型数的空间
        */
        new_memory = MALLOC( 25, int );
        /* ... */
    }
```

程序 11.1c　使用错误检查分配器　　a_client.c

当一个使用动态内存分配的程序失败时，人们很容易把问题的责任推给 malloc 和 free 函数。但它们实际上很少是罪魁祸首。事实上，问题几乎总是出在你自己的程序中，而且常常是由于访问了分配内存以外的区域而引起的。

警告：

当你使用 free 时，可能出现各种不同的错误。传递给 free 的指针必须是一个从 malloc、calloc 或 realloc 函数返回的指针。传给 free 函数一个指针，让它释放一块并非动态分配的内存可能导致程序立即终止或在晚些时候终止。试图释放一块动态分配内存的一部分也有可能引起类似的问题，像下面这样：

```
/*
** Get 10 integers
*/
pi = malloc( 10 * sizeof( int ) );
...
/*
** Free only the last 5 integers; keep the first 5
*/
free( pi + 5 );
```

释放一块内存的一部分是不允许的。动态分配的内存必须整块一起释放。但是，realloc 函数可以缩小一块动态分配的内存，有效地释放它尾部的部分内存。

警告：

最后，你必须小心在意，不要访问已经被 free 函数释放了的内存。这个警告看上去很显然，但

这里仍然存在一个很微妙的问题。假定你对一个指向动态分配的内存的指针进行了复制，而且这个指针的几份拷贝散布于程序各处。你无法保证当你使用其中一个指针时它所指向的内存是不是已被另一个指针释放。另一方面，你必须确保程序中所有使用这块内存的地方在这块内存被释放之前停止对它的使用。

内存泄漏

当动态分配的内存不再需要使用时，它应该被释放，这样它以后可以被重新分配使用。分配内存但在使用完毕后不释放将引起内存泄漏(memory leak)。在那些所有执行程序共享一个通用内存池的操作系统中，内存泄漏将一点点地榨干可用内存，最终使其一无所有。要摆脱这个困境，只有重启系统。

其他操作系统能够记住每个程序当前拥有的内存段，这样当一个程序终止时，所有分配给它但未被释放的内存都归还给内存池。但即使在这类系统中，内存泄漏仍然是一个严重的问题，因为一个持续分配却一点不释放内存的程序最终将耗尽可用的内存。此时，这个有缺陷的程序将无法继续执行下去，它的失败有可能导致当前已经完成的工作统统丢失。

11.6 内存分配实例

动态内存分配一个常见的用途就是为那些长度在运行时才知的数组分配内存空间。程序 11.2 读取一列整数，并按升序排列它们，最后打印这个列表。

```
/*
** 读取、排序和打印一列整型值。
*/
#include <stdlib.h>
#include <stdio.h>

/*
** 该函数由'qsort'调用，用于比较整型值。
*/
int
compare_integers( void const *a, void const *b )
{
     register int   const *pa = a;
     register int   const *pb = b;

     return *pa > *pb ? 1 : *pa < *pb ? -1 : 0;
}

int
main()
{
     int   *array;
     int   n_values;
     int   i;

     /*
     ** 观察共有多少个值。
```

```
*/
printf( "How many values are there? " );
if( scanf( "%d", &n_values ) != 1 || n_values <= 0 ){
    printf( "Illegal number of values.\n" );
    exit( EXIT_FAILURE );
}

/*
** 分配内存，用于存储这些值。
*/
array = malloc( n_values * sizeof( int ) );
if( array == NULL ){
    printf( "Can't get memory for that many values.\n" );
    exit( EXIT_FAILURE );
}

/*
** 读取这些数值。
*/
for( i = 0; i < n_values; i += 1 ){
     printf( "? " );
     if( scanf( "%d", array + i ) != 1 ){
          printf( "Error reading value #%d\n", i );
          free( array );
          exit( EXIT_FAILURE );
     }
}

/*
** 对这些值排序。
*/
qsort( array, n_values, sizeof( int ), compare_integers );

/*
** 打印这些值。
*/
for( i = 0; i < n_values; i += 1 )
     printf( "%d\n", array[i] );

     /*
     ** 释放内存并退出。
     */
      free( array );
      return EXIT_SUCCESS;
}
```

程序 11.2　排序一列整型值　　sort.c

用于保存这个列表的内存是动态分配的，这样当你编写程序时就不必猜测用户可能希望对多少个值进行排序。可以排序的值的数量仅受分配给这个程序的动态内存数量的限制。但是，当程序对一个小型的列表进行排序时，它实际分配的内存就是实际需要的内存，因此不会造成浪费。

现在让我们考虑一个读取字符串的程序。如果你预先不知道最长的那个字符串的长度，你就无法使用普通数组作为缓冲区。反之，你可以使用动态分配内存。当你发现一个长度超过缓冲区的输

入行时，你可以重新分配一个更大的缓冲区，把该行的剩余部分也装到它里面。这个技巧的实现留作编程练习。

```
/*
** 用动态分配内存制作一个字符串的一份拷贝。注意：调用程序应该负责检查这块内
** 存是否成功分配！这样做允许调用程序以任何它所希望的方式对错误作出反应。
*/
#include <stdlib.h>
#include <string.h>

char *
strdup( char const *string )
{
      char*new_string;

      /*
      ** 请求足够长度的内存，用于存储字符串和它的结尾 NUL 字节。
      */
      new_string = malloc( strlen( string ) + 1 );

      /*
      ** 如果我们得到内存，就复制字符串。
      */
      if( new_string != NULL )
          strcpy( new_string, string );

      return new_string;
}
```

程序 11.3　复制字符串　　**strdup.c**

输入被读入到缓冲区，每次读取一行。此时可以确定字符串的长度，然后就分配内存用于存储字符串。最后，字符串被复制到新内存。这样缓冲区又可以用于读取下一个输入行。

程序 11.3 中名叫 strdup 的函数返回一个输入字符串的拷贝，该拷贝存储于一块动态分配的内存中。函数首先试图获得足够的内存来存储这个拷贝。内存的容量应该比字符串的长度多一个字节，以便存储字符串结尾的 NUL 字节。如果内存成功分配，字符串就被复制到这块新内存。最后，函数返回一个指向这块内存的指针。注意，如果由于某些原因导致内存分配失败，new_string 的值将为 NULL。在这种情况下，函数将返回一个 NULL 指针。

这个函数是非常方便的，也非常有用。事实上，尽管标准没有提及，但许多环境都把它作为函数库的一部分。

我们的最后一个例子说明了你可以怎样使用动态内存分配来消除使用变体记录造成的内存空间浪费。程序 11.4 是第 10 章存货系统例子的修改版本。程序 11.4a 包含了存货记录的声明。

和以前一样，存货系统必须处理两种类型的记录，分别用于零件和装配件。第 1 个结构保存零件的专用信息（这里只显示这个结构的一部分），第 2 个结构保存装配件的专用信息。最后一个声明用于存货记录，它包含了零件和装配件的一些共有信息以及一个变体部分。

由于变体部分的不同字段具有不同的长度（事实上，装配件记录的长度是可变的），所以联合包含了指向结构的指针而不是结构本身。动态分配允许程序创建一条存货记录，它所使用的内存的大

小就是进行存储的项目的长度，这样就不会浪费内存。

程序 11.4b 是一个函数，它为每个装配件创建一条存货记录。这个任务取决于装配件所包含的不同零件的数目，所以这个值是作为参数传递给函数的。

这个函数为三样东西分配内存：存货记录、装配件结构和装配件结构中的零件数组。如果这些分配中的任何一个失败，所有已经分配的内存将被释放，函数返回一个 NULL 指针。否则，type 和 info.subassy->n_parts 字段被初始化，函数返回一个指向该记录的指针。

为零件存货记录分配内存较之装配件存货记录容易一些，因为它只需要进行两项内存分配。因此，这个函数在此不予解释。

```
/*
** 存货记录的声明。
*/

/*
** 包含零件专用信息的结构。
*/
typedef struct {
        int    cost;
        int    supplier;
        /* 其他信息。 */
} Partinfo;

/*
** 存储装配件专用信息的结构。
*/
typedef    struct    {
           int  n_parts;
struct     SUBASSYPART {
     char partno[10];
     short     quan;
} *part;
} Subassyinfo;

/*
** 存货记录结构，它是一个变体记录。
*/
typedef    struct    {
char partno[10];
int   quan;
enum { PART, SUBASSY }  type;
union     {
      Partinfo *part;
      Subassyinfo   *subassy;
} info;
} Invrec;
```

程序 11.4a　存货系统声明　　inventor.h

```
/*
** 用于创建 SUBASSEMBLY(装配件)存货记录的函数。
*/
```

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

Invrec *
create_subassy_record( int n_parts )
{
Invrec    *new_rec;

    /*
    ** 试图为 Invrec 部分分配内存。
    */
    new_rec = malloc( sizeof( Invrec ) );
    if( new_rec != NULL ){
    /*
    ** 内存分配成功，现在存储 SUBASSYINFO 部分。
    */
      new_rec->info.subassy =
          malloc( sizeof( Subassyinfo ) );
     if( new_rec->info.subassy != NULL ){
 /*
** 为零件获取一个足够大的数组。
*/
        new_rec->info.subassy->part = malloc(
            n_parts * sizeof( struct SUBASSYPART ) );
        if( new_rec->info.subassy->part != NULL ){
 /*
 ** 获取内存，填充我们已知道值的字段，然后返回。
 */
             new_rec->type = SUBASSY;
             new_rec->info.subassy->n_parts =
                 n_parts;
             return new_rec;
        }

   /*
   ** 内存已用完，释放我们原先分配的内存。
   */
            free( new_rec->info.subassy );
        }
        free( new_rec );
    }
    return NULL;
}
```

程序 11.4b　动态创建变体记录　　invcreat.c

程序 11.4c 包含了这个例子的最后部分：一个用于销毁存货记录的函数。这个函数对两种类型的存货记录都适用。它使用一条 switch 语句判断传递给它的记录的类型并释放所有动态分配给这个记录的所有字段的内存。最后，这个记录便被删除。

在这种情况下，一个常见的错误是在释放记录中的字段所指向的内存前便释放记录。在记录被释放之后，你就可能无法安全地访问它所包含的任何字段。

```
/*
** 释放存货记录的函数。
*/

#include <stdlib.h>
#include "inventor.h"

void
discard_inventory_record( Invrec *record )
{
     /*
     ** 删除记录中的变体部分
     */
     switch( record->type ){
     case SUBASSY:
          free( record->info.subassy->part );
          free( record->info.subassy );
          break;

     case PART:
          free( record->info.part );
          break;
     }

     /*
     ** 删除记录的主体部分
     */
     free( record );
}
```

程序 11.4c　变体记录的销毁　　invdelet.c

下面的代码段尽管看上去不是非常的一目了然，但它的效率比程序 11.4c 稍有提高。

```
if( record->type == SUBASSY )
     free( record->info.subassy->part );

free( record->info.part );
free( record );
```

这段代码在释放记录的变体部分时并不区分零件和装配件。联合的任一成员都可以传递给 free 函数，因为后者并不理会指针所指向内容的类型。

11.7　总结

当数组被声明时，必须在编译时知道它的长度。动态内存分配允许程序为一个长度在运行时才知道的数组分配内存空间。

malloc 和 calloc 函数都用于动态分配一块内存，并返回一个指向该块内存的指针。malloc 的参数就是需要分配的内存的字节数。和它不同的是，calloc 的参数是你需要分配的元素个数和每个元素的长度。calloc 函数在返回前把内存初始化为零，而 malloc 函数返回时内存并未以任何方式进行初始化。调用 realloc 函数可以改变一块已经动态分配的内存的大小。增加内存块大小时有可能采取

的方法是把原来内存块上的所有数据复制到一个新的、更大的内存块上。当一个动态分配的内存块不再使用时，应该调用 free 函数把它归还给可用内存池。内存被释放之后便不能再被访问。

如果请求的内存分配失败，malloc、calloc 和 realloc 函数返回的将是一个 NULL 指针。错误地访问分配内存之外的区域所引起的后果类似越界访问一个数组，但这个错误还可能破坏可用内存池，导致程序失败。如果一个指针不是从早先的 malloc、calloc 或 realloc 函数返回的，它是不能作为参数传递给 free 函数的。你也不能只释放一块内存的一部分。

内存泄漏是指内存被动态分配以后，当它不再使用时未被释放。内存泄漏会增加程序的体积，有可能导致程序或系统的崩溃。

11.8 警告的总结

1. 不检查从 malloc 函数返回的指针是否为 NULL。
2. 访问动态分配的内存之外的区域。
3. 向 free 函数传递一个并非由 malloc 函数返回的指针。
4. 在动态内存被释放之后再访问它。

11.9 编程提示的总结

1. 动态内存分配有助于消除程序内部存在的限制。
2. 使用 sizeof 计算数据类型的长度，提高程序的可移植性。

11.10 问题

1. 在你的系统中，你能够声明的静态数组最大长度能达到多少？使用动态内存分配，你最大能够获取的内存块有多大？
2. 当你一次请求分配 500 个字节的内存时，你实际获得的动态分配的内存数量总共有多大？当你一次请求分配 5000 个字节时又如何？它们存在区别吗？如果有，你如何解释？

3. 在一个从文件读取字符串的程序中，有没有什么值可以合乎逻辑地作为输入缓冲区的长度？

4. 有些 C 编译器提供了一个称为 alloca 的函数，它与 malloc 函数的不同之处在于它在堆栈上分配内存。这种类型的分配有什么优点和缺点？

5. 下面的程序用于读取整数，整数的范围在 1 和从标准输入读取的 size 之间，它返回每个值出现的次数。这个程序包含了几个错误，你能找出它们吗？

```
#include <stdlib.h>

int *
frequency( int size )
{
        int     *array;
        int     i;
```

```
/*
**获得足够的内存来容纳计数。
*/
arry = (int *)malloc ( size * 2)

/*
**调整指针，让它后退一个整型位置，这样我们就可以使用范围 1-size 的下标。
*/
arry - = 1;

/*
**把各个元素值清零。
for ( i =0; i <=size; i +=1 )
        array[i]= 0 ;

/*
**计数每个值出现的次数，然后还回结果。
*/
while(scanf( *%d*, &i ) = = )  )
       arry[ i ]  +=1;


       free(arry);
       return arry;
}
```

6. 假定你需要编写一个程序，并希望最大限度地减少堆栈的使用量。动态内存分配能不能对你有所帮助？使用标量数据又该如何？
7. 在程序 11.4b 中，删除两个 free 函数的调用会导致什么后果？

11.11　编程练习

★ 1. 请你自己尝试编写 calloc 函数，函数内部使用 malloc 函数来获取内存。

★★ 2. 编写一个函数，从标准输入读取一列整数，把这些值存储于一个动态分配的数组中并返回这个数组。函数通过观察 EOF 判断输入列表是否结束。数组的第 1 个数是数组包含的值的个数，它的后面就是这些整数值。

★★★ 3. 编写一个函数，从标准输入读取一个字符串，把字符串复制到动态分配的内存中，并返回该字符串的拷贝。这个函数不应该对读入字符串的长度作任何限制！

★★★ 4. 编写一个程序，按照下图的样子创建数据结构。最后三个对象都是动态分配的结构。第 1 个对象则可能是一个静态的指向结构的指针。你不必使这个程序过于全面——我们将在下一章讨论这个数据结构。