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

bbs.theithome.com

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。

bbs.theithome.com

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

bbs.theithome.com

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。

bbs.theithome.com

```
    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

bbs.theithome.com

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。
```


bbs.theithome.com

```
    */
    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

bbs.theithome.com

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

bbs.theithome.com

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。
*/
```


bbs.theithome.com

```
#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。

bbs.theithome.com

```
/*
** 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

bbs.theithome.com

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;
```


bbs.theithome.com

```
/*
**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？

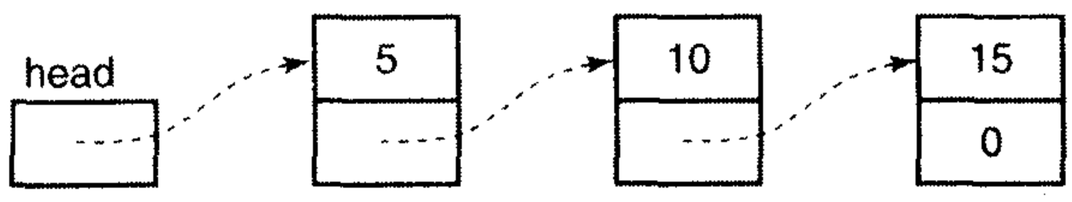
11.11 编程练习

★ 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。



bbs.theithome.com

bbs.theithome.com

第12章 使用结构和指针

你可以通过组合使用结构和指针创建强大的数据结构。本章我们将深入讨论一些使用结构和指针的技巧。我们将花许多时间讨论一种称为链表的数据结构，这不仅因为它非常有用，而且许多用于操纵链表的技巧也适用于其他数据结构。

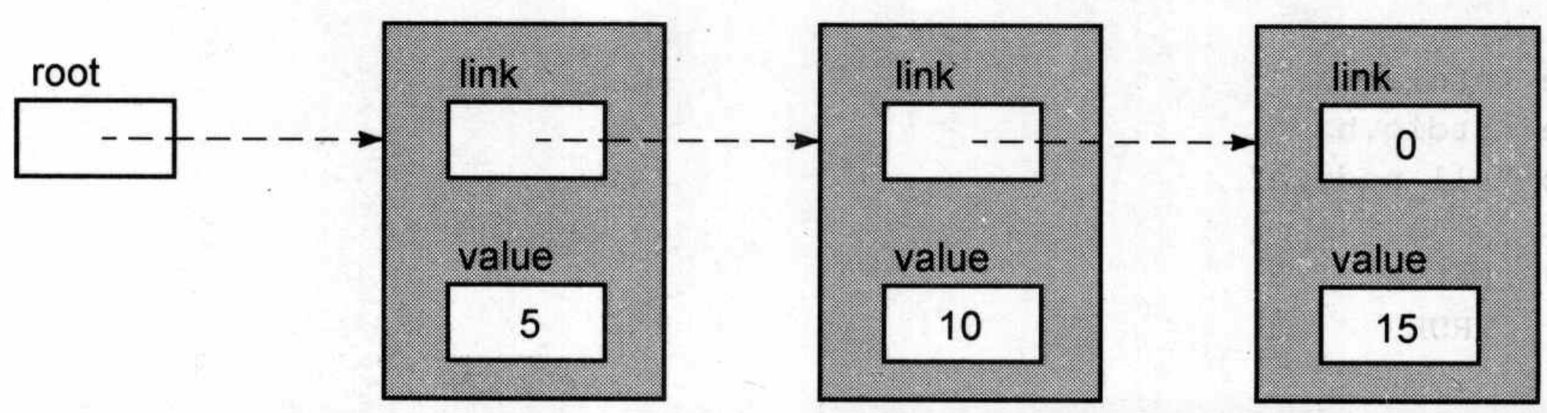
12.1 链表

有些读者可能还不熟悉链表，这里对它作一简单介绍。链表(linked list)就一些包含数据的独立数据结构（通常称为节点）的集合。链表中的每个节点通过链或指针连接在一起。程序通过指针访问链表中的节点。通常节点是动态分配的，但有时你也能看到由节点数组构建的链表。即使在这种情况下，程序也是通过指针来遍历链表的。

12.2 单链表

在单链表中，每个节点包含一个指向链表下一节点的指针。链表最后一个节点的指针字段的值为 NULL，提示链表后面不再有其他节点。在你找到链表的第 1 个节点后，指针就可以带你访问剩余的所有节点。为了记住链表的起始位置，可以使用一个根指针(root pointer)。根指针指向链表的第 1 个节点。注意根指针只是一个指针，它不包含任何数据。

下面是一张单链表的图。



本例中的节点是用下面的声明创建的结构。

bbs.theithome.com

```
typedef struct  NODE   {
        struct  NODE   *link;
        int             value;
} Node;
```

存储于每个节点的数据是一个整型值。这个链表包含三个节点。如果你从根指针开始，随着指针到达第 1 个节点，你可以访问存储于那个节点的数据。随着第 1 个节点的指针可以到达第 2 个节点，你可以访问存储在那里的数据。最后，第 2 个节点的指针带你来到最后一个节点。零值提示它是一个 NULL 指针，在这里它表示链表中不再有更多的节点。

在上面的图中，这些节点相邻在一起，这是为了显示链表所提供的逻辑顺序。事实上，链表中的节点可能分布于内存中的各个地方。对于一个处理链表的程序而言，各节点在物理上是否相邻并没有什么区别，因为程序始终用链（指针）从一个节点移动到另一个节点。

单链表可以通过链从开始位置遍历链表直到结束位置，但链表无法从相反的方向进行遍历。换句话说，当你的程序到达链表的最后一个节点时，如果你想回到其他任何节点，你只能从根指针从头开始。当然，程序在移动到下一个节点前可以保存一个指向当前位置的指针，甚至可以保存指向前面几个位置的指针。但是，链表是动态分配的，可能增长到几百或几千个节点，所以要保存所有指向前面位置的节点的指针是不可行的。

在这个特定的链表中，节点根据数据的值按升序链接在一起。对于有些应用程序而言，这种顺序非常重要，比如根据一天的时间安排约会。对于那些不要求排序的应用程序，当然也可以创建无序的链表。

12.2.1 在单链表中插入

我们怎么才能把一个新节点插入到一个有序的单链表中呢？假定我们有一个新值，比如 12，想把它插入到前面那个链表中。从概念上说，这个任务非常简单：从链表的起始位置开始，跟随指针直到找到第 1 个值大于 12 的节点，然后把这个新值插入到那个节点之前的位置。

实际的算法则比较有趣。我们按顺序访问链表，当到达内容为 15 的节点（第 1 个值大于 12 的节点）时就停下来。我们知道这个新值应该添加到这个节点之前，但**前一个**节点的指针字段必须进行修改以实现这个插入。但是，我们已经越过了这个节点，无法返回去。解决这个问题的方法就是始终保存一个指向链表当前节点之前的那个节点的指针。

我们现在将开发一个函数，把一个节点插入到一个有序的单链表中。程序 12.1 是我们的第 1 次尝试。

```
/*
** 插入到一个有序的单链表。函数的参数是一个指向链表第 1 个节点的指针以及需要插入的值。
*/
#include <stdlib.h>
#include <stdio.h>
#include "sll_node.h"

#define   FALSE    0
#define   TRUE     1

int
sll_insert( Node *current, int new_value )
{
```


bbs.theithome.com

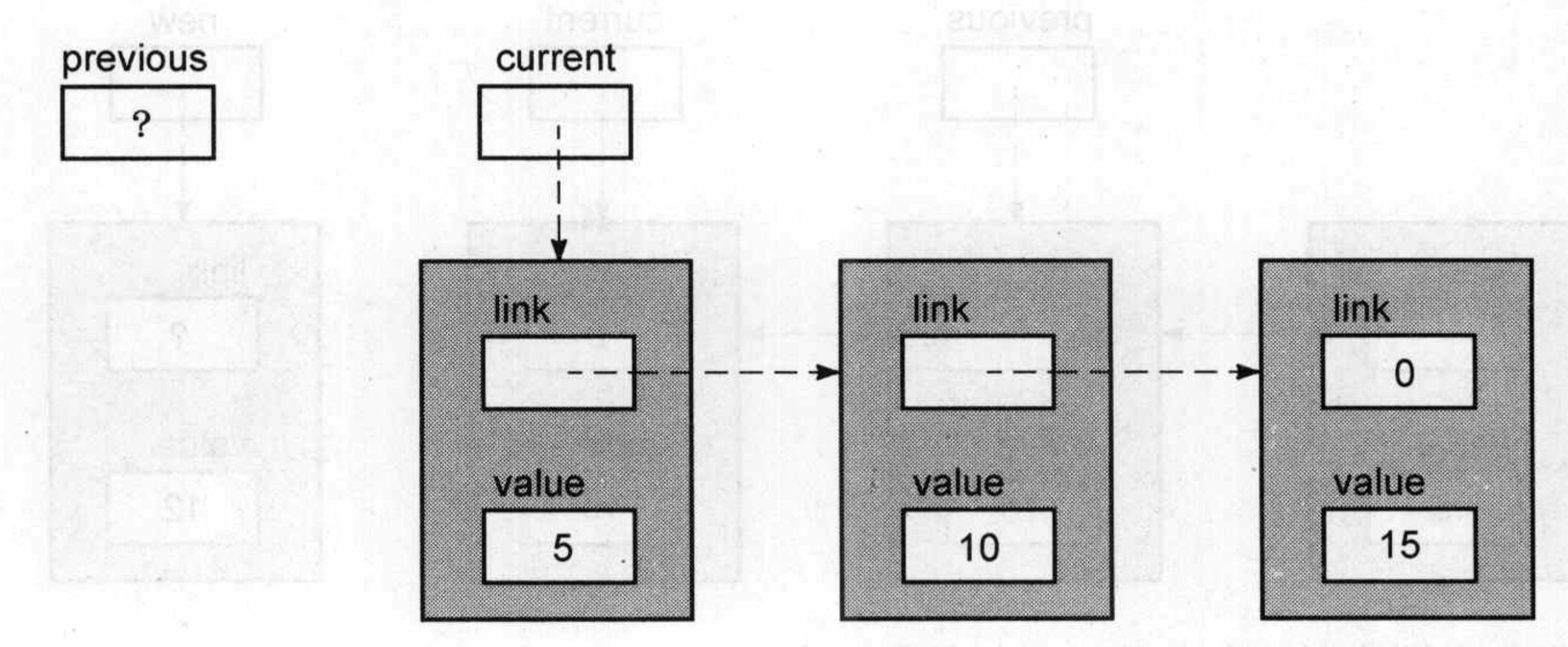
```
    Node   *previous;
    Node   *new;

    /*
    ** 寻找正确的插入位置，方法是按顺序访问链表，直到到达其值大于或等于
    ** 新插入值的节点。
    */
    while( current->value < new_value ){
         previous = current;
         current = current->link;
    }

    /*
    ** 为新节点分配内存，并把新值存储到新节点中，如果内存分配失败，
    **  函数返回 FALSE。
    */
    new = (Node *)malloc( sizeof( Node ) );
    if( new == NULL )
         return FALSE;
    new->value = new_value;

    /*
    ** 把新节点插入到链表中，并返回 TRUE。
    */
    new->link = current;
    previous->link = new;
    return TRUE;
}
```

程序 12.1 插入到一个有序的单链表：第 1 次尝试　　insert1.c

我们用下面这种方法调用这个函数：

```
result = sll_insert( root, 12 );
```

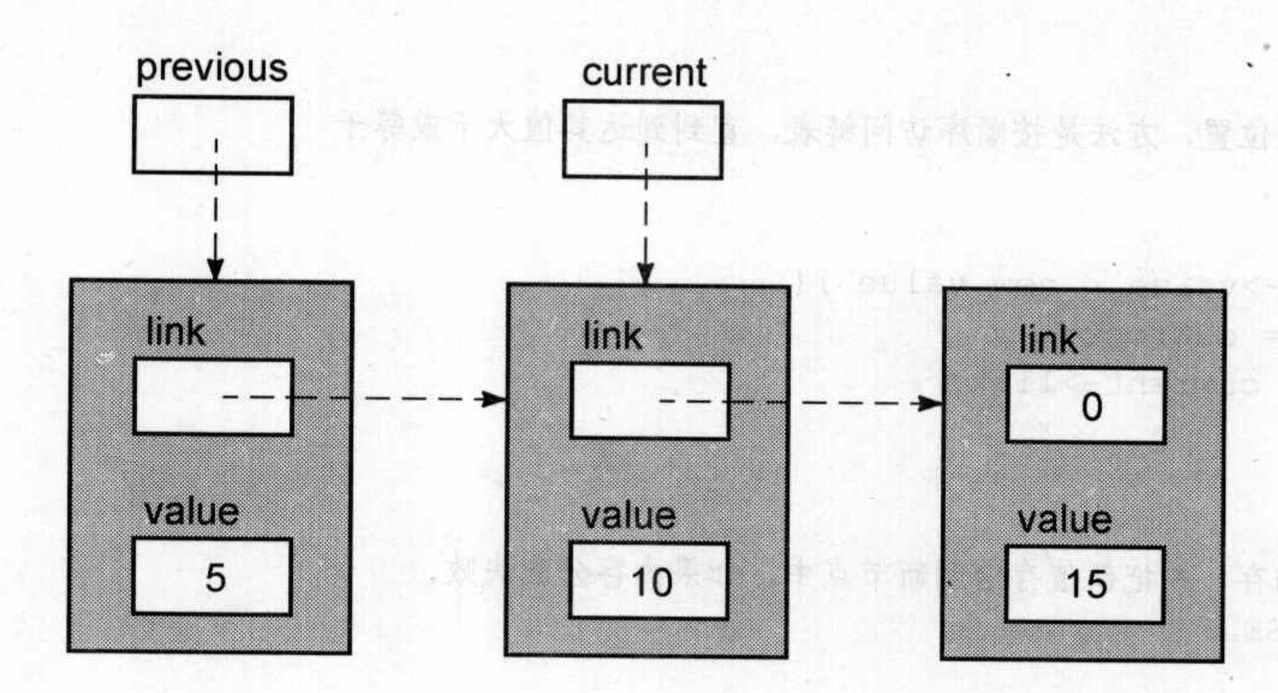
让我们仔细跟踪代码的执行过程，看看它是否把新值 12 正确地插入到链表中。首先，传递给函数的参数是 root 变量的值，它是指向链表第 1 个节点的指针。当函数刚开始执行时，链表的状态如下：



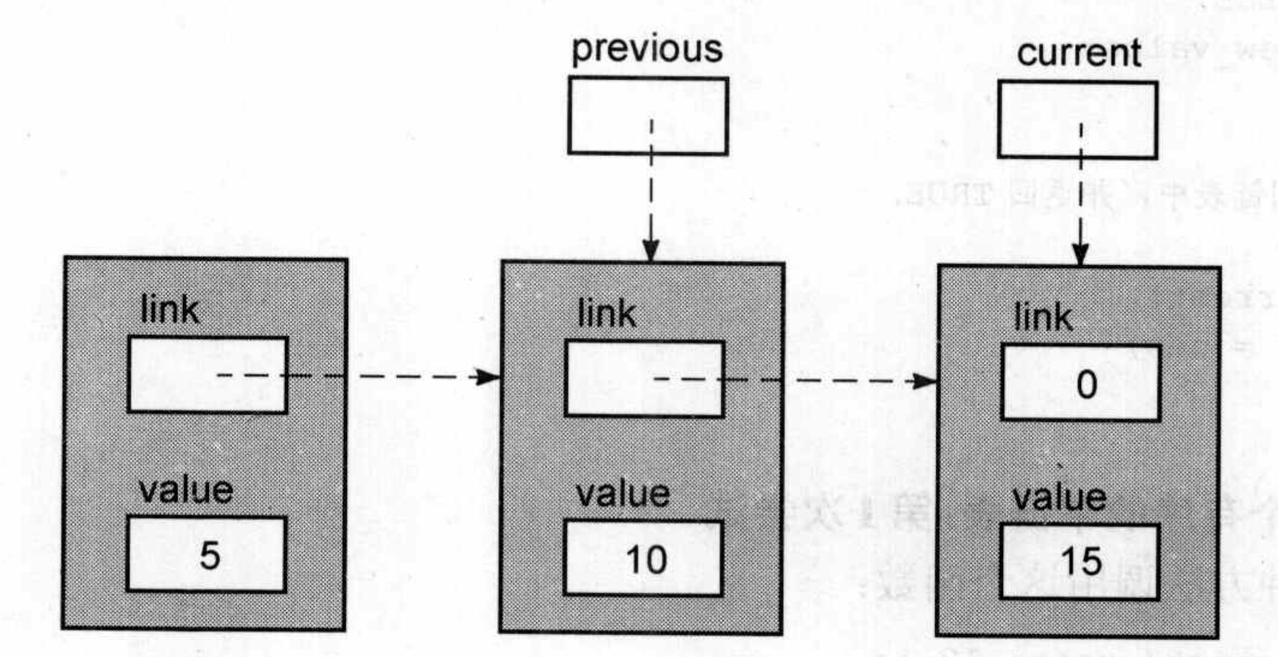
这张图并没有显示 root 变量，因为函数不能访问它。它的值的一份拷贝作为形参 current 传递给函数，但函数不能访问 root。现在 current->value 是 5，它小于 12，所以循环体再次执行。当我们回

bbs.theithome.com

到循环的顶部时，current 和 previous 指针都向前移动了一个节点。

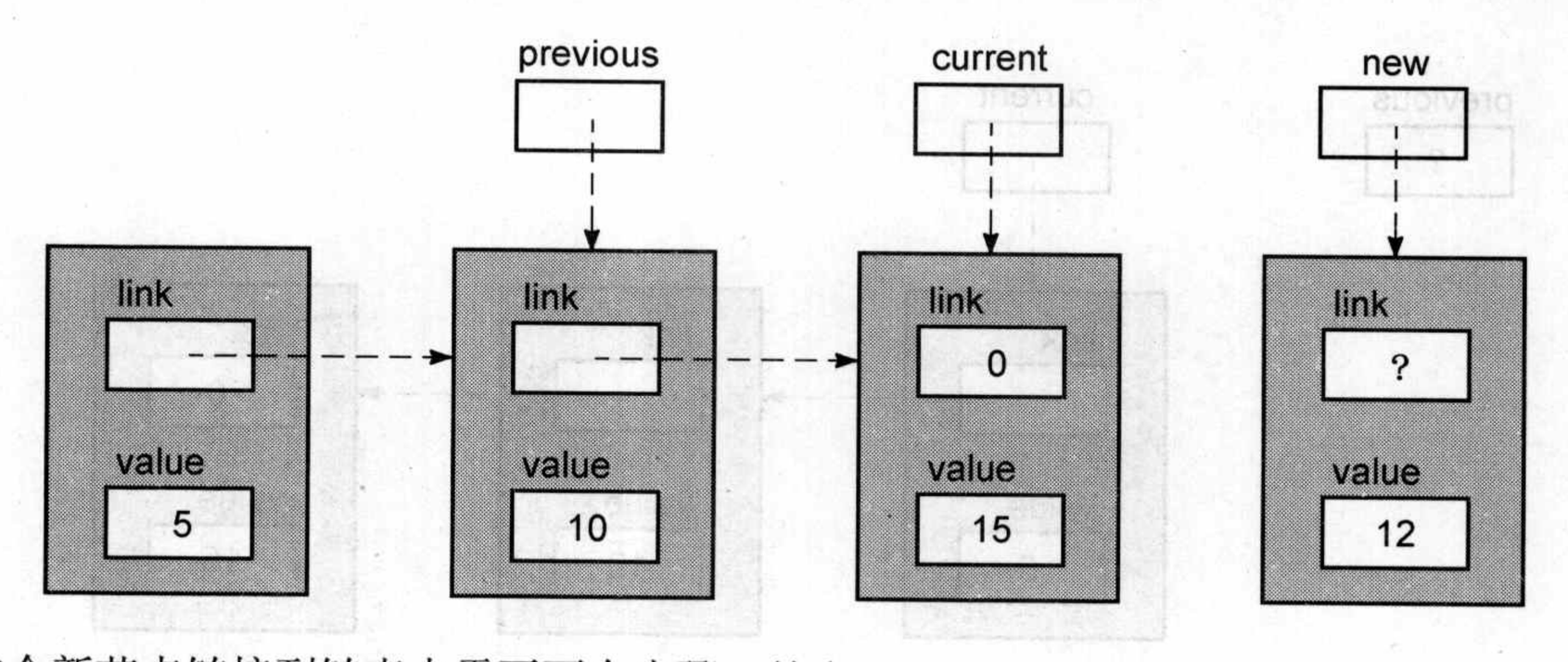


现在，current->value 的值为 10，因此循环体还将继续执行，结果如下：



现在，current->value 的值大于 12，所以退出循环。

此时，重要的是 previous 指针，因为它指向我们必须加以修改以插入新值的那个节点。但首先，我们必须得到一个新节点，用于容纳新值。下面这张图显示了新值被复制到新节点之后链表的状态。



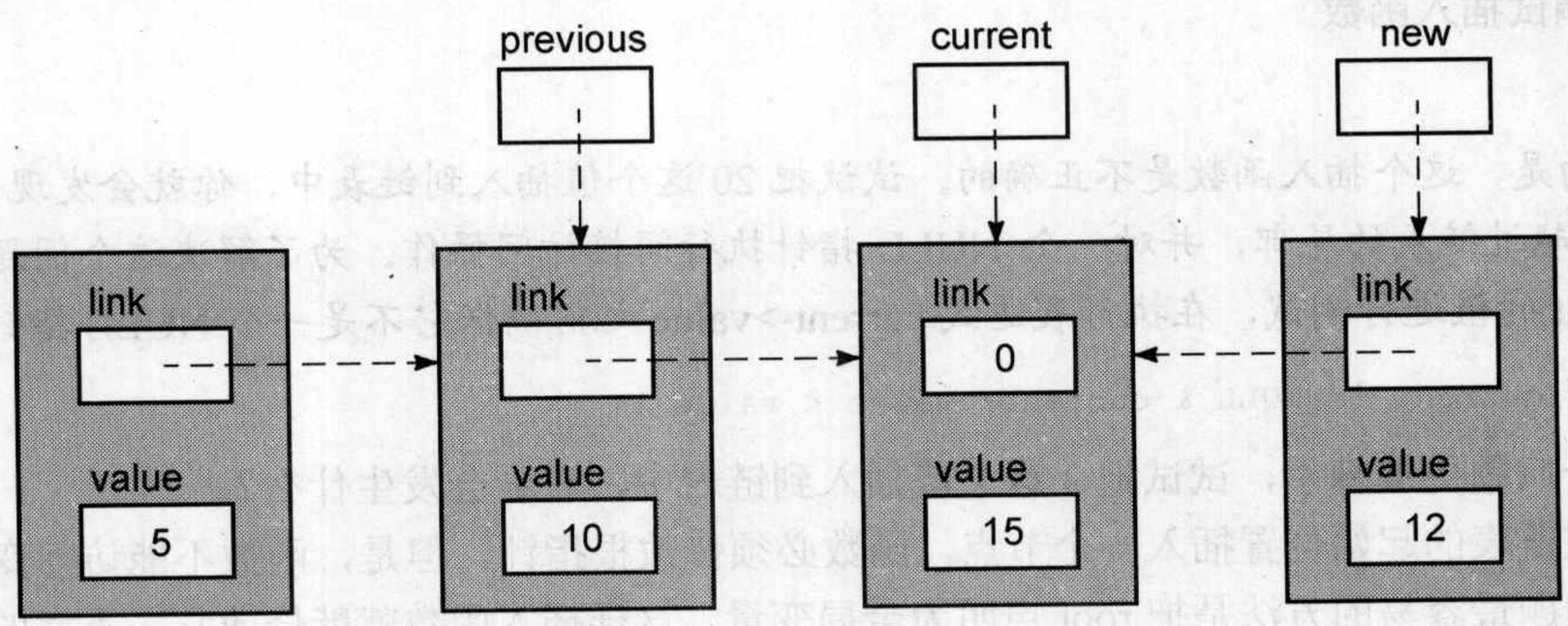
把这个新节点链接到链表中需要两个步骤。首先，

```
new->link = current;
```

使新节点指向将成为链表下一个节点的节点，也就是我们所找到的第 1 个值大于 12 的那个节点。在

bbs.theithome.com

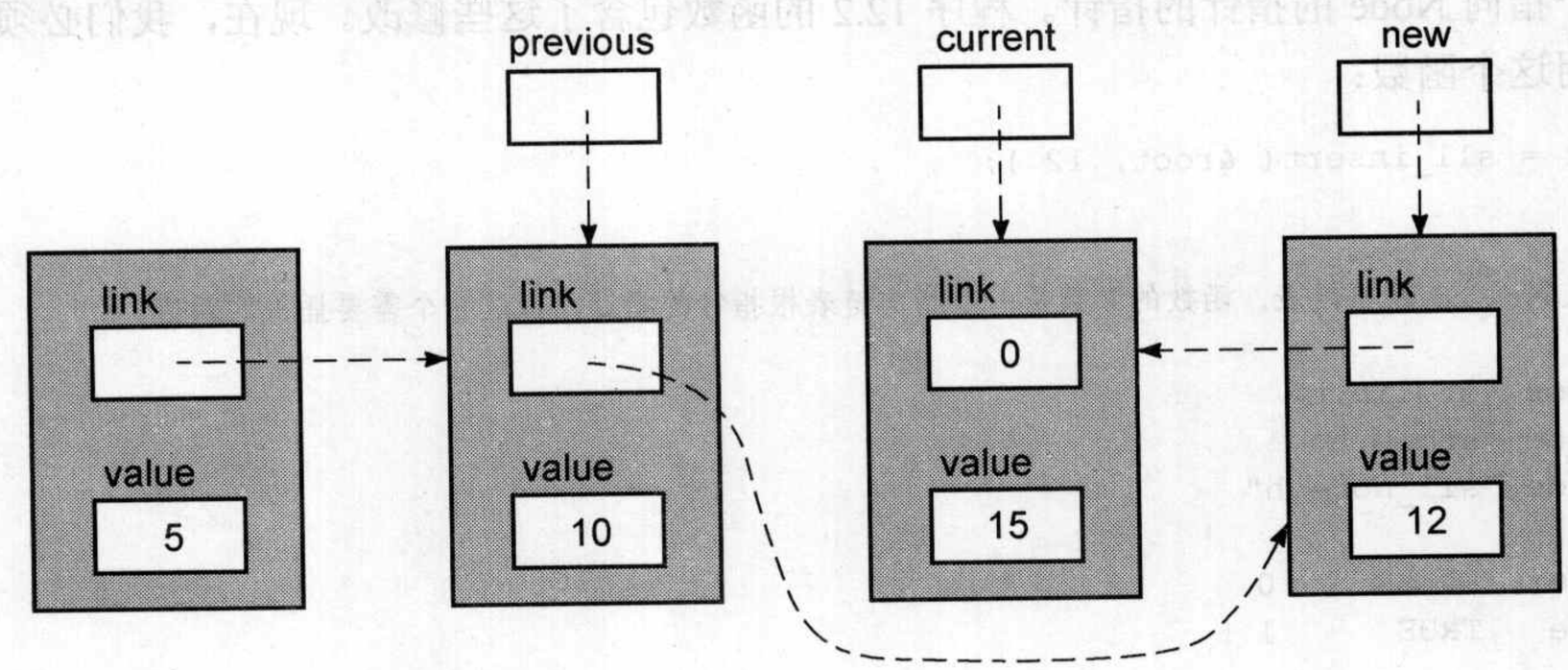
这个步骤之后，链表的内容如下所示：



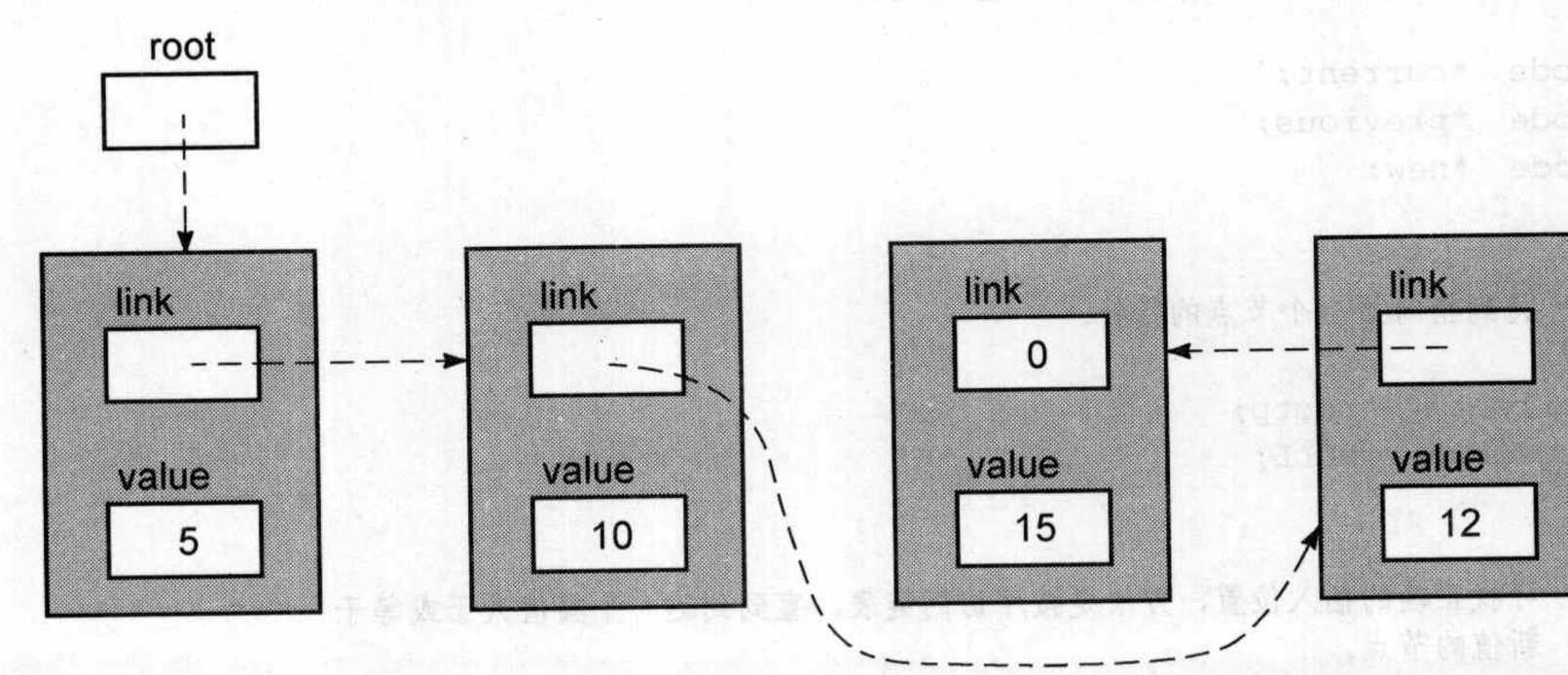
第二个步骤是让 previous 指针所指向的节点（也就是最后一个值小于 12 的那个节点）指向这个新节点。下面这条语句用于执行这项任务。

```
previous->link = new;
```

这个步骤之后，链表的状态如下：



然后函数返回，链表的最终样子如下：



从根指针开始，随各个节点的 link 字段逐个访问链表，我们可以发现这个新节点已被正确地插入到链表中。

bbs.theithome.com

一、调试插入函数

警告：

不幸的是，这个插入函数是不正确的。试试把 20 这个值插入到链表中，你就会发现一个问题：while 循环越过链表的尾部，并对一个 NULL 指针执行间接访问操作。为了解决这个问题，我们必须对 current 的值进行测试，在执行表达式 current->value 之前确保它不是一个 NULL 指针：

```
while( current != NULL & current->value < value ){
```

下一个问题更加棘手，试试把 3 这个值插入到链表中，看看会发生什么？

为了在链表的起始位置插入一个节点，函数必须修改根指针。但是，函数不能访问变量 root。修正这个问题最容易的方法是把 root 声明为全局变量，这样插入函数就能修改它。不幸的是，这是**最坏**的一种问题解决方法。因为这样一来，函数只对这个链表起作用。

稍好的解决方法是把一个指向 root 的指针作为参数传递给函数。然后，使用间接访问，函数不仅可以获得 root（指向链表第 1 个节点的指针，也就是根指针）的值，也可以向它存储一个新的指针值。这个参数的类型是什么呢？root 是一个指向 Node 的指针，所以参数的类型应该是 Node **，也就是一个指向 Node 的指针的指针。程序 12.2 的函数包含了这些修改。现在，我们必须以下面这种方式调用这个函数：

```
result = sll_insert( &root, 12 );

/*
** 插入到一个有序单链表。函数的参数是一个指向链表根指针的指针，以及一个需要插入的新值。
*/
#include <stdlib.h>
#include <stdio.h>
#include "sll_node.h"

#define   FALSE     0
#define   TRUE      1

int
sll_insert( Node **rootp, int new_value )
{
        Node  *current;
        Node  *previous;
        Node  *new;

        /*
        ** 得到指向第 1 个节点的指针。
        */
        current = *rootp;
        previous = NULL;

        /*
        ** 寻找正确的插入位置，方法是按序访问链表，直到到达一个其值大于或等于
        ** 新值的节点。
        */
        while( current != NULL && current->value < new_value ){
             previous = current;
```


bbs.theithome.com

```
    current = current->link;
}

/*
** 为新节点分配内存，并把新值存储到新节点中，如果内存分配失败，
**  函数返回 FALSE。
*/
new = (Node *)malloc( sizeof( Node ) );
if( new == NULL )
    return FALSE;
new->value = new_value;

/*
** 把新节点插入到链表中，并返回 TRUE。
*/
new->link = current;
if( previous == NULL )
    *rootp = new;
else
    previous->link = new;
return TRUE;
}
```

程序 12.2 插入到一个有序单链表：第 2 次尝试　　insert2.c

这第 2 个版本包含了另外一些语句。

```
previous = NULL;
```

我们需要这条语句，这样我们在以后就可以检查新值是否应为链表的第 1 个节点。

```
current = *rootp;
```

这条语句对根指针参数执行间接访问操作，得到的结果是 root 的值，也就是指向链表第 1 个节点的指针。

```
If (previous == NULL)
        *rootp = new;
else
        previous->link = new;
```

这条语句被添加到函数的最后。它用于检查新值是否应该被添加到链表的起始位置。如果是，我们使用间接访问修改根指针，使它指向新节点。

这个函数可以正确完成任务，而且在许多语言中，这是你能够获得的最佳方案。但是，我们还可以做得更好一些，因为 C 允许我们获得现存对象的地址（即指向该对象的指针）。

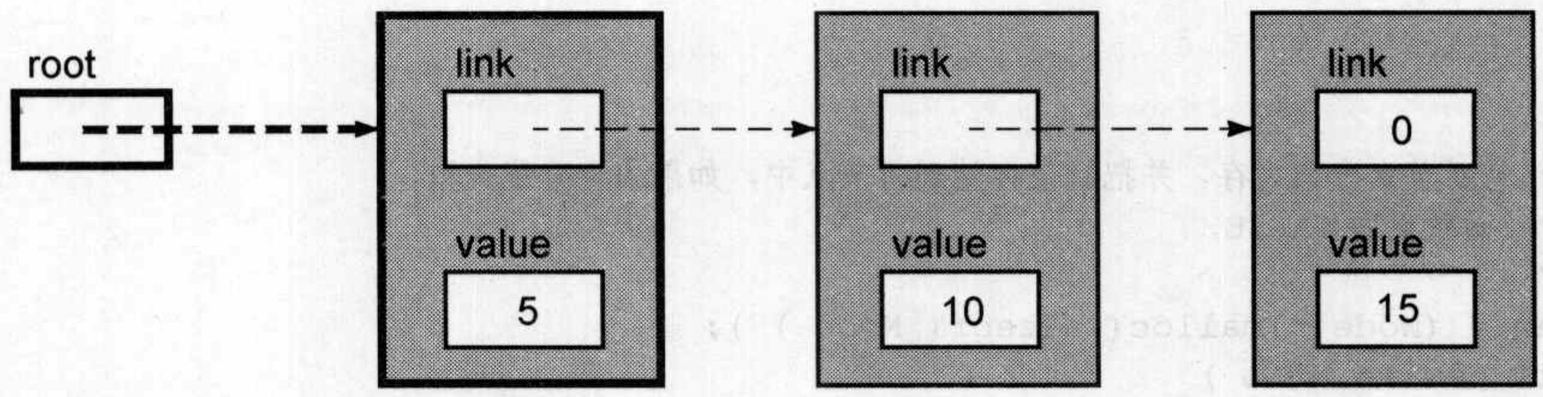
二、优化插入函数

看上去，把一个节点插入到链表的起始位置**必须**作为一种特殊情况进行处理。毕竟，我们此时插入新节点需要修改的指针是根指针。对于任何其他节点，对指针进行修改时实际修改的是前一个节点的 link 字段。这两个看上去不同的操作实际上是一样的。

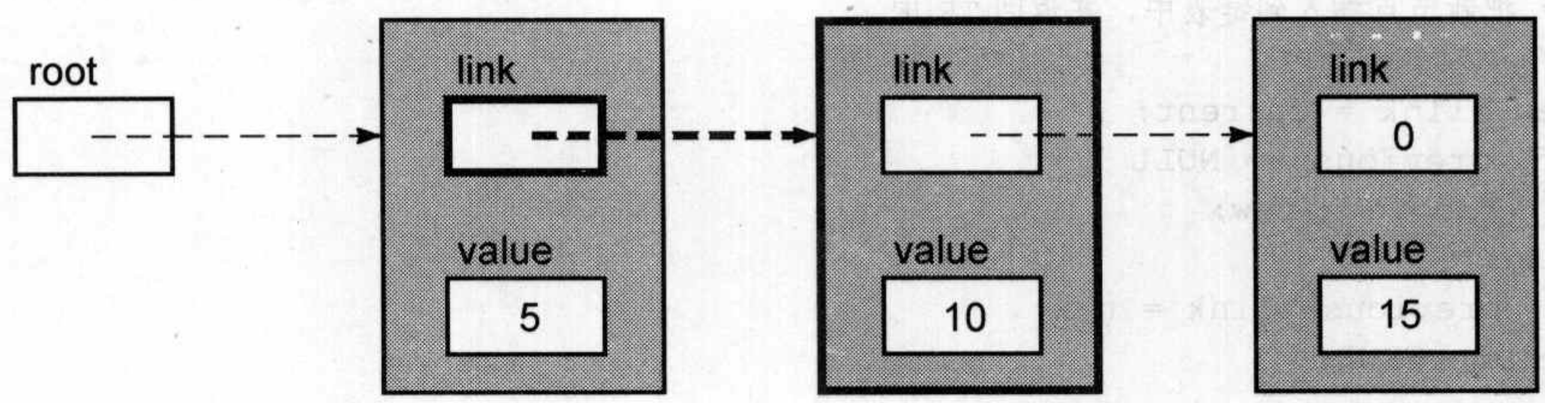
消除特殊情况的关键在于：我们必须认识到，链表中的每个节点都有一个指向它的指针。对于第 1 个节点，这个指针是根指针；对于其他节点，这个指针是前一个节点的 link 字段。重点在于每个节点都有一个指针指向它。至于该指针是不是位于一个节点的内部则无关紧要。

让我们再次观察这个链表，弄清这个概念。这是第 1 个节点和指向它的指针。

bbs.theithome.com

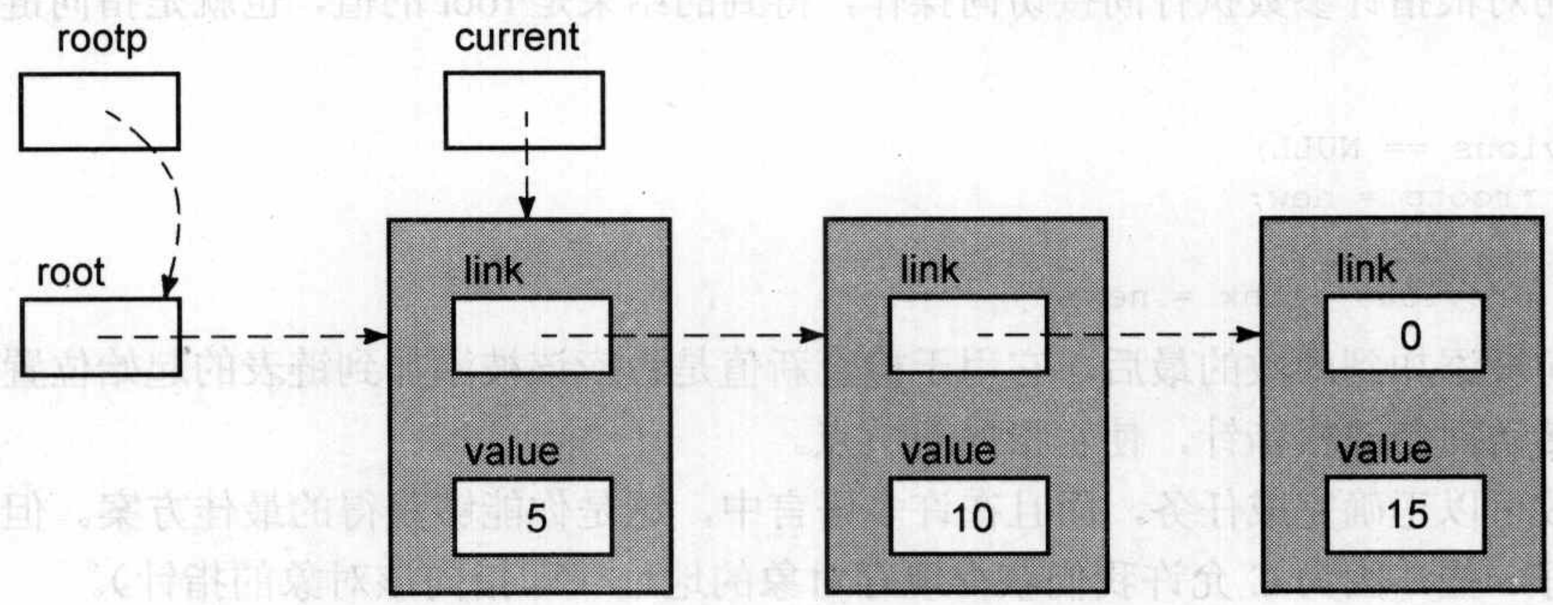


如果新值插入到第 1 个节点之前，这个指针就必须进行修改。下面是第 2 个节点和指向它的指针。



如果新值需要插入到第 2 个节点之前，那么**这个**指针必须进行修改。注意我们只考虑指向这个节点的指针，至于哪个节点包含这个指针则无关紧要。对于链表中的其他节点，都可以应用这个模式。

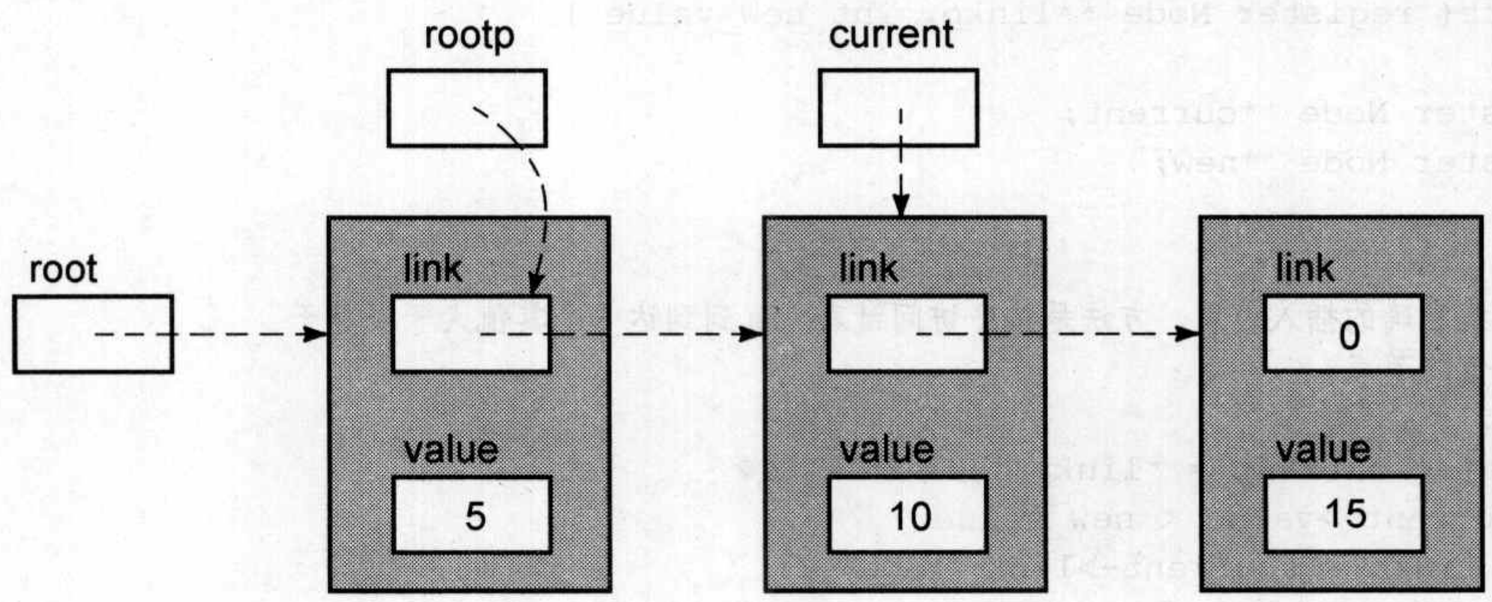
现在让我们看一下修改后的函数（当它开始执行时）。下面显示了第 1 条赋值语句之后各个变量的情况。



我们拥有一个指向当前节点的指针，以及一个“指向当前节点的 link 字段的”指针。除此之外，我们就不需要别的了！如果当前节点的值大于新值，那么 rootp 指针就会告诉我们哪个 link 字段必须进行修改，以便让新节点链接到链表中。如果在链表其他位置的插入也可以用同样的方式进行表示，就不存在前面提到的特殊情况了。其关键在于我们前面看到的指针/节点关系。

当移动到下一个节点时，我们保存一个“指向下一个节点的 **link 字段**的”指针，而不是保存一个指向前一个**节点**的指针。我们很容易画出一张描述这种情况的图。

bbs.theithome.com



注意，这里 rootp 并不指向节点本身，而是指向节点内部的 link 字段。这是简化插入函数的关键所在，但我们必须能够取得当前节点的 link 字段的地址。在 C 中，这种操作是非常容易的。表达式&current->link 就可以达到这个目的。程序 12.3 是我们的插入函数的最终版本。rootp 参数现在称为 linkp，因为它现在指向的是不同的 link 字段，而不仅仅是根指针。我们不再需要 previous 指针，因为我们的 link 指针可以负责寻找需要修改的 link 字段。前面那个函数最后部分用于处理特殊情况的代码也不见了，因为我们始终拥有一个指向需要修改的 link 字段的指针——我们用一种和修改节点的 link 字段完全一样的方式修改 root 变量。最后，我们在函数的指针变量中增加了 register 声明，用于提高结果代码的效率。

我们在最终版本中的 while 循环中增加了一个窍门，它嵌入了对 current 的赋值。下面是一个功能相同，但长度稍长的循环。

```
/*
** Look for the right place.
*/
current = *linkp;
while( current !=NULL && current->value < value ){
        linkp = &current->link;

current = * linkp;
}
```

一开始，current 被设置为指向链表的第 1 个节点。while 循环测试我们是否到达了链表的尾部。如果没有，它接着检查我们是否到达了正确的插入位置。如果不是，循环体继续执行，并把 linkp 设置为指向当前节点的 link 字段，并使 current 指向下一个节点。

循环的最后一条语句和循环之前的那条语句相同，这就促使我们对它进行“简化”，方法是把 current 的赋值嵌入到 while 表达式中。其结果是一个稍为复杂但更加紧凑的循环，因为我们消除了 current 的冗余赋值。

```
/*
** 插入到一个有序单链表。函数的参数是一个指向链表第一个节点的指针，以及一个需要插入的新值
*/
#include <stdlib.h>
#include <stdio.h>
#include "sll_node.h"

#define   FALSE     0
#define   TRUE      1

int
```


bbs.theithome.com

```
sll_insert( register Node **linkp, int new_value )
{
    register Node  *current;
    register Node  *new;

    /*
    ** 寻找正确的插入位置，方法是按序访问链表，直到到达一个其值大于或等于
    ** 新值的节点。
    */
    while( ( current = *linkp ) != NULL &&
        current->value < new_value )
          linkp = &current->link;

    /*
    ** 为新节点分配内存，并把新值存储到新节点中，如果内存分配失败，
    ** 函数返回 FALSE。
    */
    new = (Node *)malloc( sizeof( Node ) );
    if( new == NULL )
        return FALSE;
    new->value = new_value;

    /*
    ** 在链表中插入新节点，并返回 TRUE。
    */
    new->link = current;
    *linkp = new;
    return TRUE;
}
```

程序 12.3 插入到一个有序的单链表：最终版本 insert3.c

提示：

消除特殊情况使这个函数更为简单。这个改进之所以可行是由于两方面的因素。第 1 个因素是我们正确解释问题的能力。除非你可以在看上去不同的操作中总结出共性，不然你只能编写额外的代码来处理特殊情况。通常，这种知识只有在你学习了一阵数据结构并对其有进一步的理解之后才能获得。第 2 个因素是 C 语言提供了正确的工具帮助你归纳问题的共性。

这个改进的函数依赖于 C 能够取得现存对象的地址这一能力。和许多 C 语言特性一样，这个能力既威力巨大，又暗伏凶险。例如，在 Modula 和 Pascal 中并不存在“取地址”操作符，所以指针唯一的来源就是动态内存分配。我们没有办法获得一个指向普通变量的指针或甚至是指向一个动态分配的结构的字段的指针。对指针不允许进行算术运算，也没有办法把一种类型的指针通过强制类型转换为另一种类型的指针。这些限制的优点在于它们可以防止诸如“越界引用数组元素”或“产生一种类型的指针但实际上指向另一种类型的对象”这类错误。

警告：

C 的指针限制要少得多，这也是我们能改进插入函数的原因所在。另一方面，C 程序员在使用指针时必须加倍小心，以避免产生错误。Pascal 语言的指针哲学有点类似下面这样的说法：“使用锤子可能会伤着你自己，所以我们不给你锤子。”C 语言的指针哲学则是：“给你锤子，实际上你可以使用好几种锤子。祝你好运！”有了这个能力之后，C 程序员较之 Pascal 程序员更容易

bbs.theithome.com

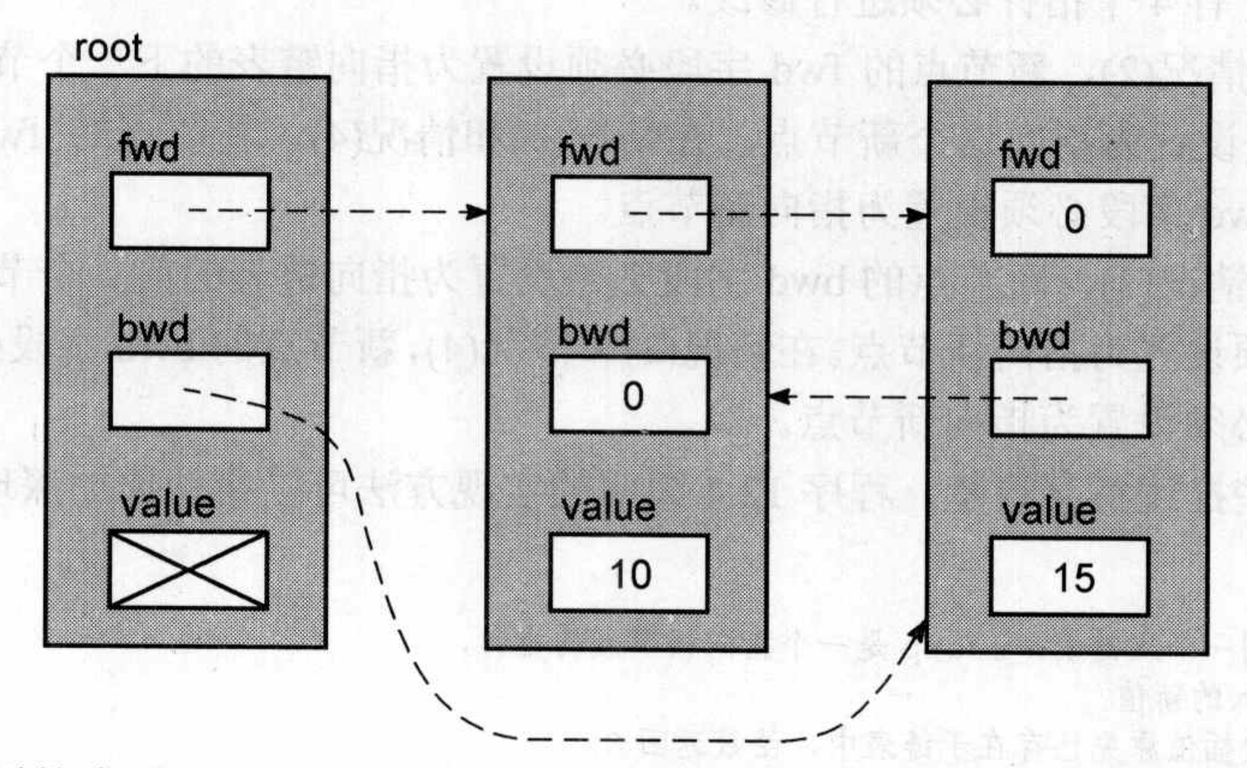
陷入麻烦，但优秀的 C 程序员可以比他们的 Pascal 和 Modula 同行产生体积更小、效率更高、可维护性更佳的代码。这也是 C 语言在业界为何如此流行以及经验丰富的 C 程序员为何如此受青睐的原因之一。

12.2.2　其他链表操作

为了让单链表更加有用，我们需要增加更多的操作，如查找和删除。但是，用于这些操作的算法非常直截了当，很容易用插入函数所说明的技巧来实现。因此，我把这些函数留作练习。

12.3　双链表

单链表的替代方案就是双链表。在一个双链表中，每个节点都包含两个指针——指向前一个节点的指针和指向后一个节点的指针。这可以使我们以任何方向遍历双链表，甚至可以忽前忽后地在双链表中访问。下面的图展示了一个双链表。



下面是节点类型的声明。

```
typedf struct  NODE   {
       struct  NODE   *fwd;
       struct  NODE   *bwd;
       int            value;
} Node;
```

现在，存在两个根指针：一个指向链表的第 1 个节点，另一个指向最后一个节点。这两个指针允许我们从链表的任何一端开始遍历链表。

我们可能想把两个根指针分开声明为两个变量。但这样一样，我们必须把两个指针都传递给插入函数。为根指针声明一个完整的节点更为方便，只是它的值字段绝不会被使用。在我们的例子中，这个技巧只是浪费了一个整型值的内存空间。对于值字段非常大的链表，分开声明两个指针可能更好一些。另外，我们也可以在根节点的值字段中保存其他一些关于链表的信息，例如链表当前包含的节点数量。

根节点的 fwd 字段指向链表的第 1 个节点，根节点的 bwd 字段指向链表的最后一个节点。如果链表为空，这两个字段都为 NULL。链表第 1 个节点的 bwd 字段和最后一个节点的 rwd 字段都为 NULL。在一个有序的链表中，各个节点将根据 value 字段的值以升序排列。

bbs.theithome.com

12.3.1 在双链表中插入

这一次，我们要编写一个函数，把一个值插入到一个有序的双链表中。dll_insert 函数接受两个参数：一个指向根节点的指针和一个整型值。

我们先前所编写的单链表插入函数把重复的值也添加到链表中。在有些应用程序中，不插入重复的值可能更为合适。dll_insert 函数只有当欲插入的值原先不存在于链表中时才将其插入。

让我们用一种更为规范的方法来编写这个函数。当我们把一个节点插入到一个链表时，可能出现 4 种情况：

1. 新值可能必须插入到链表的中间位置。
2. 新值可能必须插入到链表的起始位置。
3. 新值可能必须插入到链表的结束位置。
4. 新值可能必须既插入到链表的起始位置，又插入到链表的结束位置（即原链表为空）。

在每种情况下，有 4 个指针必须进行修改。

- 在情况(1)和情况(2)，新节点的 fwd 字段必须设置为指向链表的下一个节点，链表下一个节点的 bwd 字段必须设置为指向这个新节点。在情况(3)和情况(4)，新节点的 fwd 字段必须设置为 NULL，根节点的 bwd 字段必须设置为指向新节点。
- 在情况(1)和情况(3)，新节点的 bwd 字段必须设置为指向链表的前一个节点，而链表前一个节点的 fwd 字段必须设置为指向新节点。在情况(2)和情况(4)，新节点的 bwd 字段必须设置为 NULL，根节点的 fwd 字段必须设置为指向新节点。

如果你觉得这些描述不甚清楚，程序 12.4 简明的实现方法可以帮助你加深理解。

```
/*
** 把一个值插入到一个双链表，rootp 是一个指向根节点的指针，
** value 是欲插入的新值。
** 返回值：如果欲插值原先已存在于链表中，函数返回 0；
** 如果内存不足导致无法插入，函数返回-1；如果插入成功，函数返回 1。
*/
#include <stdlib.h>
#include <stdio.h>
#include "doubly_linked_list_node.h"

int
dll_insert( Node *rootp, int value )
{
    Node  *this;
    Node  *next;
    Node  *newnode;

    /*
    ** 查看 value 是否已经存在于链表中，如果是就返回。
    ** 否则，为新值创建一个新节点（"newnode"将指向它）。
    ** "this"将指向应该在新节点之前的那个节点，
    ** "next"将指向应该在新节点之后的那个节点。
    */
    for( this = rootp; (next = this->fwd) != NULL; this = next ){
        if( next->value == value )
```


bbs.theithome.com

```
            return 0;
        if( next->value > value )
            break;
    }
    newnode = (Node *)malloc( sizeof( Node ) );
    if( newnode == NULL )
        return -1;
    newnode->value = value;

    /*
    ** 把新值添加到链表中。
    */
    if( next != NULL ){
    /*
    ** 情况 1 或 2: 并非位于链表尾部。
    */
            if( this != rootp ){        /* 情况 1: 并非位于链表起始位置 */
                newnode->fwd = next;
                this->fwd = newnode;
                newnode->bwd = this;
                next->bwd = newnode;
            }
            else {                      /* 情况 2: 位于链表起始位置 */
                newnode->fwd = next;
                rootp->fwd = newnode;
                newnode->bwd = NULL;
                next->bwd = newnode;
            }
    }
    else {
    /*
    ** 情况 3 或 4: 位于链表的尾部。
    */
            if( this != rootp ){        /* 情况 3: 并非位于链表的起始位置 */
                newnode->fwd = NULL;
                this->fwd = newnode;
                newnode->bwd = this;
                rootp->bwd = newnode;
            }
            else {                      /* 情况 4: 位于链表的起始位置 */
                newnode->fwd = NULL;
                rootp->fwd = newnode;
                newnode->bwd = NULL;
                rootp->bwd = newnode;
            }
    }
    return 1;
}
```

程序 12.4 简明的双链表插入函数　　dll_ins1.c

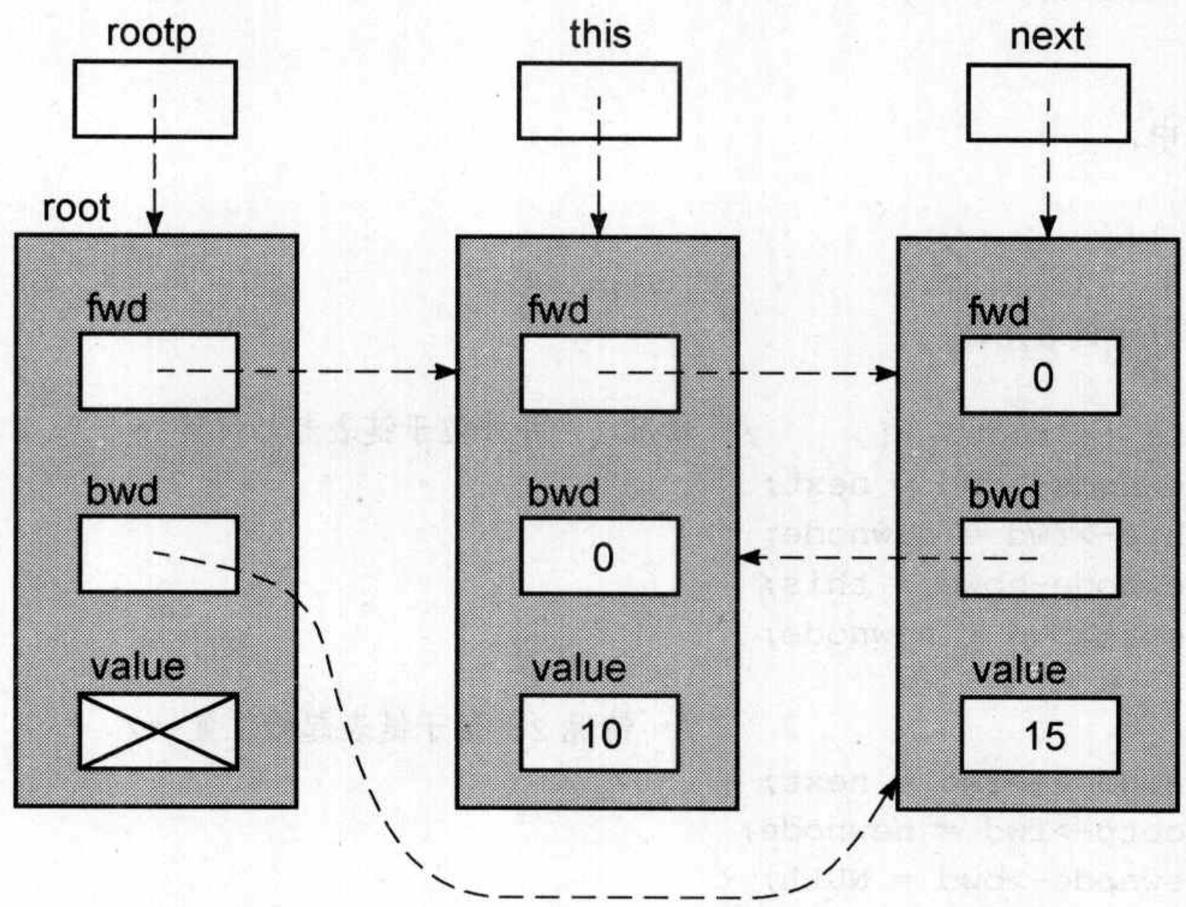
一开始，函数使 this 指向根节点。next 指针始终指向 this 之后的那个节点。它的思路是这两个指针同步前进，直到新节点应该插入到这两者之间。for 循环检查 next 所指节点的值，判断是否到达需要插入的位置。

如果在链表中找到新值，函数就简单地返回。否则，当到达链表尾部或找到适当的插入位置时

bbs.theithome.com

循环终止。在任何一种情况下，新节点都应该插入到 this 所指的节点后面。注意，在我们决定新值是否应该实际插入到链表之前，并不为它分配内存。如果事先分配内存，如果发现新值原先已经存在于链表中，就有可能发生内存泄漏。

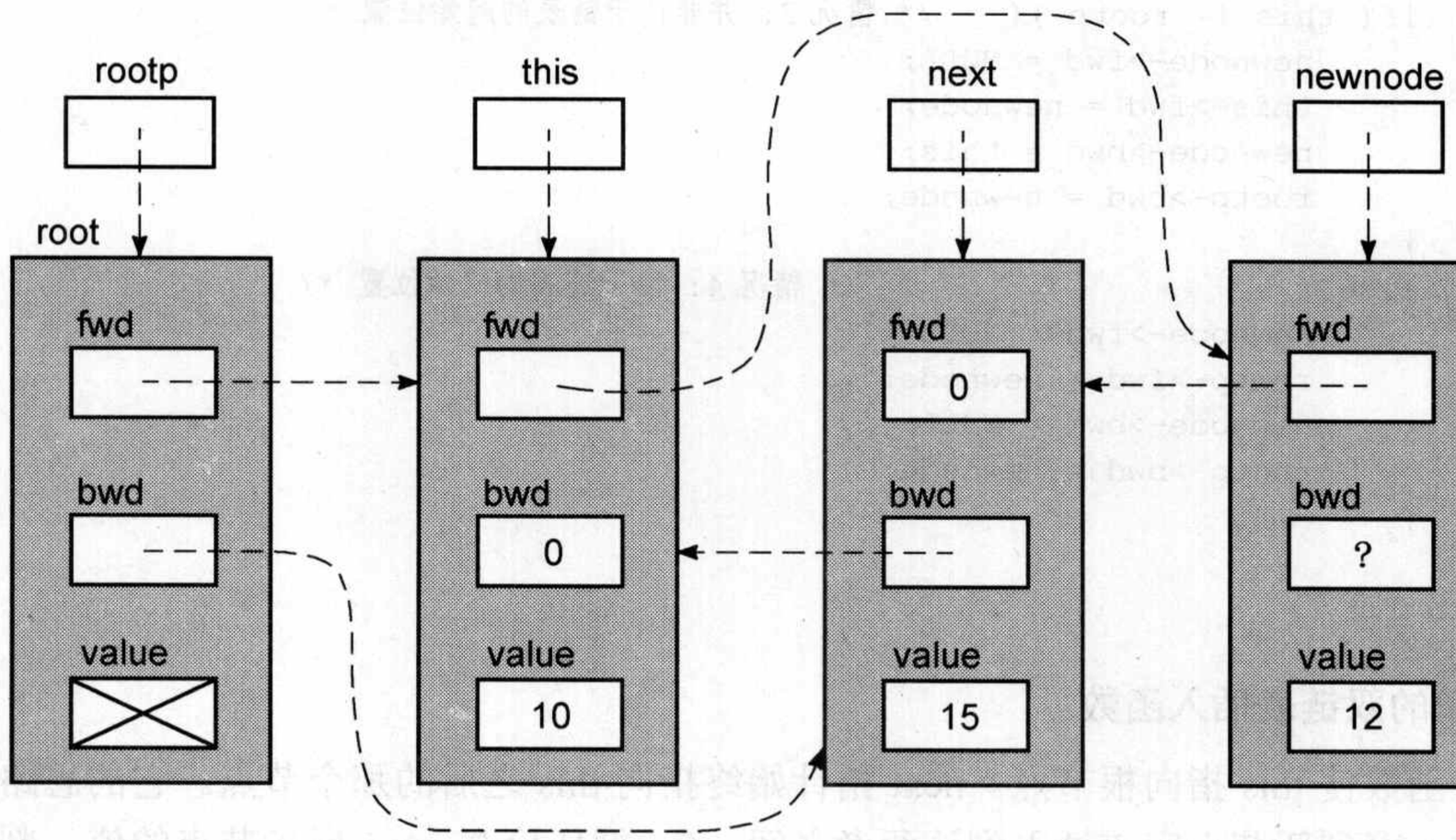
4 种情况是分开实现的。让我们通过把 12 插入到链表中来观察情况 1。下面这张图显示了 for 循环终止之后几个变量的状态。



然后，函数为新节点分配内存，下面几条语句执行之后，

```
newnode->fwd = next;
this->fwd = newnode;
```

链表的样子如下：

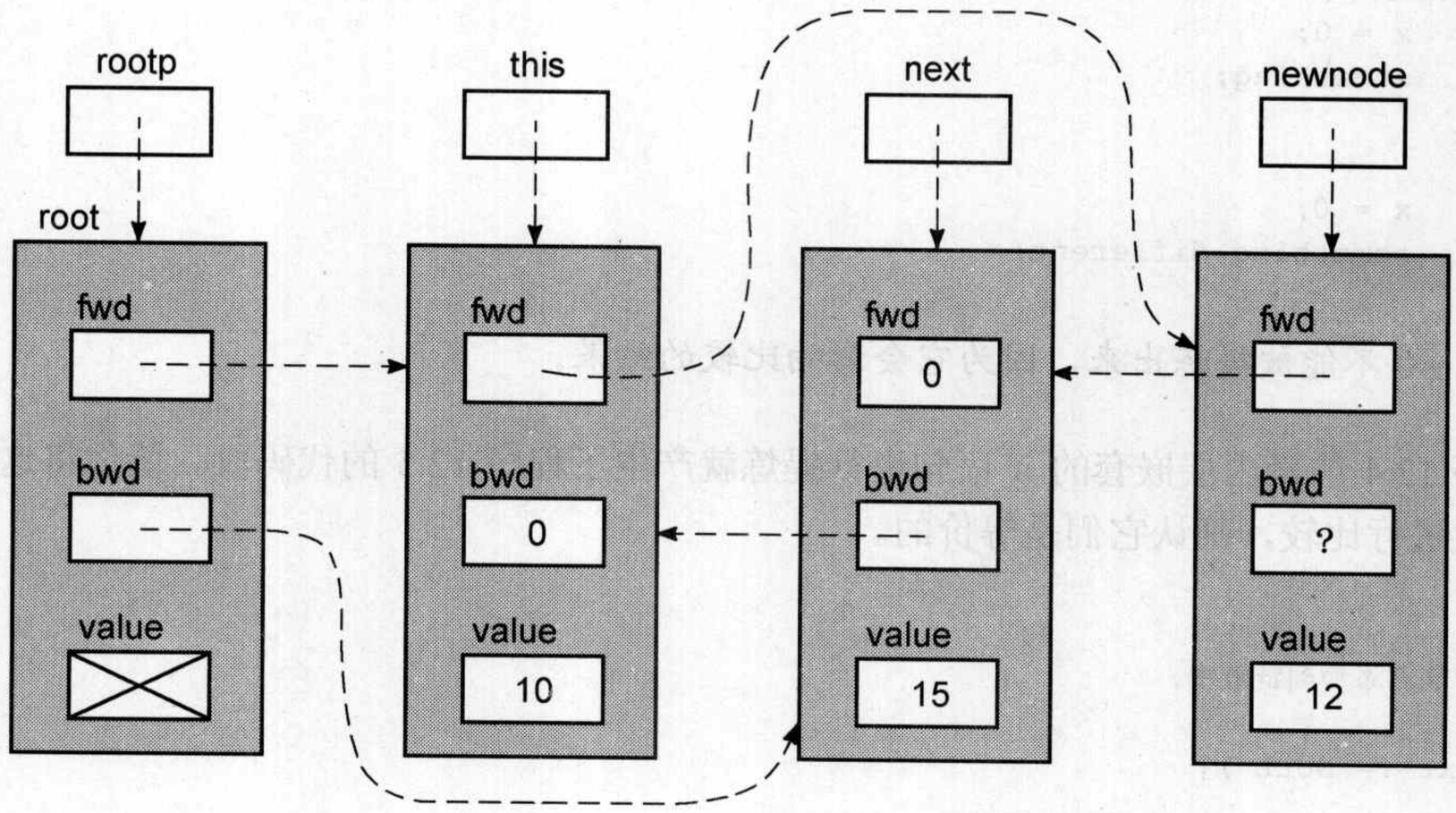


然后，执行下列语句：

bbs.theithome.com

```
newnode->bwd = this;
    next->bwd = newnode;
```

这就完成了把新值插入到链表的过程：



请研究一下代码，确定应该如何处理剩余的几种情况，确保它们都能正确工作。

简化插入函数

提示：

细心的程序员会注意到在函数中各个嵌套的 if 语句群存在大量的相似之处，而优秀的程序员将会对程序中出现这么多的重复代码感到厌烦。所以，我们现在将使用两个技巧消除这些重复的代码。第 1 个技巧是语句提炼(statement factoring)，如下面的例子所示：

```
if( x == 3) {
        i = 1;
        something;
        j = 2;
}
else {
        i = 1;
        something different;
        j = 2;
}
```

注意不管表达式 x = = 3 的值是真还是假，语句 i = 1 和 j = 2 都将执行。在 if 之前执行 i = 1 将不会影响 x == 3 的测试结果，所以这两条语句都可以被提炼出来，这样就产生了更为简单但同样完整的语句：

```
i = 1;
if( x == 3 )
        something;
else
        something different;
j = 2;
```


bbs.theithome.com

警告：

如果 if 之前的语句会对测试的结果产生影响，千万不要把它提炼出来。例如，在下面的例子中：

```
if( x == 3 ){
        x = 0;
        something;
}
else {
        x = 0;
        something different;
}
```

语句 x = 0 不能被提炼出来，因为它会影响比较的结果。

把程序 12.4 的最内层嵌套的 if 语句进行提炼就产生了程序 12.5 的代码段。请你将这段代码和前面的函数进行比较，确认它们是等价的。

```
/*
** 把新节点添加到链表中。
*/
if( next != NULL ){
    /*
    ** 情况 1 或 2：并非位于链表的尾部。
    */
          newnode->fwd = next;
          if( this != rootp ){     /* 情况 1：并非位于链表起始位置 */
               this->fwd = newnode;
               newnode->bwd = this;
          }
          else {                        /* 情况 2：位于链表起始位置 */
               rootp->fwd = newnode;
               newnode->bwd = NULL;
          }
          next->bwd = newnode;
}
else {
     /*
     ** 情况 3 或 4：位于链表尾部。
     */
          newnode->fwd = NULL;
          if( this != rootp ){  /* 情况 3：并不位于链表起始位置 */
               this->fwd = newnode;
               newnode->bwd = this;
          }
          else {                        /* 情况 4：位于链表起始位置 */
               rootp->fwd = newnode;
               newnode->bwd = NULL;
          }
          rootp->bwd = newnode;
          }
```

程序 12.5 双链表插入逻辑的提炼 dll_ins2.c

第 2 个简化技巧很容易用下面这个例子进行说明：

bbs.theithome.com

```
if( pointer !=NULL )
        field = pointer;
else
        fileld = NULL;
```

这段代码的意图是设置一个和 pointer 相等的变量，如果 pointer 未指向任何东西，这个变量就设置为 NULL。但是，请看下面这条语句：

```
field = pointer;
```

如果 pointer 的值不是 NULL，field 就像前面一样获得它的值的一份拷贝。但是，如果 pointer 的值是 NULL，那么 field 将从 pointer 获得一份 NULL 的拷贝，这和把它赋值为常量 NULL 的效果是一样的。这条语句所执行的任务和前面那条 if 语句相同，但它明显简单多了。

在程序 12.5 中运用这个技巧的关键是找出那些虽然看上去不一样但实际上执行相同任务的语句，然后对它们进行改写，写成同一种形式。我们可以把情况 3 和情况 4 的第 1 条语句改写为：

```
newnode->fwd = next;
```

由于 if 语句刚刚判断出 next == NULL。这个改动使 if 语句两边的第 1 条语句相等，所以我们可以把它提炼出来。请做好这个修改，然后对剩余的代码进行研究。

你发现了吗？现在两个嵌套的 if 语句是相等的，所以它们也可以被提炼出来。这些改动的结果显示在程序 12.6 中。

我们还可以对代码作进一步的完善。第 1 条 if 语句的 else 子句的第 1 条语句可以改写为：

```
this->fwd = newnode;
```

这是因为 if 语句已经判断出 this = = rootp。现在，这条改写后的语句以及它的同类也可以被提炼出来。

程序 12.7 是实现了所有修改的完整版本。它所执行的任务和最初的函数相同，但体积要小得多。局部指针被声明为寄存器变量，进一步改善了代码的体积和速度。

```
/*
** 把新节点添加到链表中。
*/
newnode->fwd = next;

if( this != rootp ){
     this->fwd = newnode;
     newnode->bwd = this;
}
else {
     rootp->fwd = newnode;
     newnode->bwd = NULL;
}
if( next != NULL )
     next->bwd = newnode;
else
     rootp->bwd = newnode;
```

程序 12.6 双链表插入逻辑的进一步提炼　　dll_ins3.c

```
/*
** 把一个新值插入到一个双链表中。rootp是一个指向根节点的指针，
** value是需要插入的新值。
```


bbs.theithome.com

```
** 返回值：如果链表原先已经存在这个值，函数返回 0。
** 如果为新值分配内存失败，函数返回-1。
** 如果新值成功地插入到链表中，函数返回 1。
*/
#include <stdlib.h>
#include <stdio.h>
#include "doubly_liked_list_node.h"

int
dll_insert( register Node *rootp, int value )
{
    register Node  *this;
    register Node  *next;
    register Node  *newnode;

    /*
    ** 查看 value 是否已经存在于链表中，如果是就返回。
    ** 否则，为新值创建一个新节点（"newnode"将指向它）。
    ** "this"将指向应该在新节点之前的那个节点，
    ** "next"将指向应该在新节点之后的那个节点。
    */
    for( this = rootp; (next = this->fwd) != NULL; this = next ){
        if( next->value == value )
            return 0;
        if( next->value > value )
            break;
    }
    newnode = (Node *)malloc( sizeof( Node ) );
    if( newnode == NULL )
        return -1;
    newnode->value = value;

    /*
    ** 把新节点添加到链表中。
    */
    newnode->fwd = next;
    this->fwd = newnode;

    if( this != rootp )
        newnode->bwd = this;
    else
        newnode->bwd = NULL;

    if( next != NULL )
        next->bwd = newnode;
    else
        rootp->bwd = newnode;

    return 1;
}
```

程序 12.7 双链表插入函数的最终简化版本 dll_ins4.c

这个函数无法再大幅度改善了，但我们可以让源代码更小一些。第 1 条 if 语句的目的是判断赋值语句右边一侧的值。我们可以用一个条件表达式取代 if 语句。我们也可以用条件表达式取代第 2

bbs.theithome.com

条 if 语句，但这个修改的意义并不是很大。

提示：

程序 12.8 的代码确实更小一些，但它是不是真的更好？尽管它的语句数量减少了，但必须执行的比较和赋值操作还是和前面的一样多，所以这段代码的运行速度并不比前面的更快。这里存在两个微小的差别：newnode->bwd 和->bwd = newnode 都只编写了一次而不是两次。这些差别能不能产生更小的目标代码呢？也许会，这取决于你的编译器优化措施是否出色。但是，即使会产生更小的代码，其差别也是很小的，但这段代码的可读性较之前面的代码有所下降，尤其是对于那些缺乏经验的 C 程序员而言。因此，程序 12.8 维护起来或许更困难一些。

如果程序的大小或者执行速度确实至关重要，我们可能只好考虑用汇编语言来编写函数。但即便在编码方式上采取如此巨大的变化，也不能保证肯定会有任何重大的改进。另外还要考虑到汇编代码难于编写、难于阅读和难于维护。所以，只有当迫不得已的时候，我们才能求诸于汇编语言。

```
/*
** 把新节点添加到链表中。
*/
newnode->fwd = next;
this->fwd = newnode;
newnode->bwd = this != rootp ? this : NULL;
( next != NULL ? next : rootp )->bwd = newnode;
```

程序 12.8 使用条件表达式实现插入函数　　dll_ins5.c

12.3.2　其他链表操作

和单链表一样，双链表也需要更多的操作。本章的编程练习将给你更多的实践机会来编写它们。

12.4　总结

单链表是一种使用指针来存储值的数据结构。链表中的每个节点包含一个字段，用于指向链表的下一个节点。另外有一个独立的根指针指向链表的第 1 个节点。由于节点在创建时是采用动态分配内存的方式，所以它们可能分布于内存之中。但是，遍历链表是根据指针进行的，所以节点的物理排列无关紧要。单链表只能以一个方向进行遍历。

为了把一个新值插入到一个有序的单链表中，你首先必须找到链表中合适的插入位置。对于无序单链表，新值可以插入到任何位置。把一个新节点链接到链表中需要两个步骤。首先，新节点的 link 字段必须设置为指向它的目标后续节点。其次，前一个节点的 link 字段必须设置为指向这个新节点。在许多其他语言中，插入函数保存一个指向前一个节点的指针来完成第 2 个步骤。但是，这个技巧使插入到链表的起始位置成为一种特殊情况，需要单独处理。在 C 语言中，你可以通过保存一个指向必须进行修改的 link 字段的指针，而不是保存一个指向前一个节点的指针，从而消除了这个特殊情况。

双链表中的每个节点包含两个 link 字段：其中一个指向链表的下一个节点，另一个指向链表的前一个节点。双链表有两个根指针，分别指向第 1 个节点和最后一个节点。因此，遍历双链表可以从任何一端开始，而且在遍历过程中可以改变方向。为了把一个新节点插入到双链表中，我们必须

bbs.theithome.com

修改 4 个指针。新节点的前向和后向 link 字段必须被设置，前一个节点的后向 link 字段和后一个节点的前向 link 字段也必须进行修改，使它们指向这个新节点。

语句提炼是一种简化程序的技巧，其方法是消除程序中冗余的语句。如果一条 if 语句的“then”和“else”子句以相同序列的语句结尾，它们可以被一份单独的出现于 if 语句之后的拷贝代替。相同序列的语句也可以从 if 语句的起始位置提炼出来，但这种提炼不能改变 if 的测试结果。如果不同的语句事实上执行相同的功能，你可以把它们写成相同的样子，然后再使用语句提炼简化程序。

12.5 警告的总结

1. 落到链表尾部的后面。
2. 使用指针时应格外小心，因为 C 并没有对它们的使用提供安全网。
3. 从 if 语句中提炼语句可能会改变测试结果。

12.6 编程提示的总结

1. 消除特殊情况使代码更易于维护。
2. 通过提炼语句消除 if 语句中的重复语句。
3. 不要仅仅根据代码的大小评估它的质量。

12.7 问题

1. 程序 12.3 能否进行改写，不使用 current 变量？如果可以，把你的答案和原先的函数作一比较。
2. 有些数据结构课本建议在单链表中使用“头节点”。这个哑节点始终是链表的第 1 个元素，这就消除了插入到链表起始位置这个特殊情况。讨论这个技巧的利与弊。
3. 在程序 12.3 中，插入函数会把重复的值插入到什么位置？如果把比较操作符由<改为<=会有什么效果？
4. 讨论一些技巧，怎样省略双链表中根节点的值字段。
5. 如果程序 12.7 中对 malloc 的调用在函数的起始部分执行会有什么结果？
6. 能不能对一个无序的单链表进行排序？
7. 索引表(concordance list)是一种字母链表，表中的节点是出现于一本书或一篇文章中的单词。你可以使用一个有序的字符串单链表实现索引表，使用插入函数时不插入重复的单词。和这种实现方法有关的问题是搜索链表的时间将随着链表规模的扩大而急剧增长。

   图 12.1 说明了另一种存储索引表的数据结构。它的思路是把一个大型的链表分解为 26 个小型的链表——每个链表中的所有单词都以同一个字母开头。最初链表中的每个节点包含一个字母和一个指向一个有序的以该字母开头的单词的单链表（以字符串形式存储）的指针。

   使用这种数据结构，搜索一个特定的单词所花费的时间与使用一个存储所有单词的单链表相比，有没有什么变化？

bbs.theithome.com

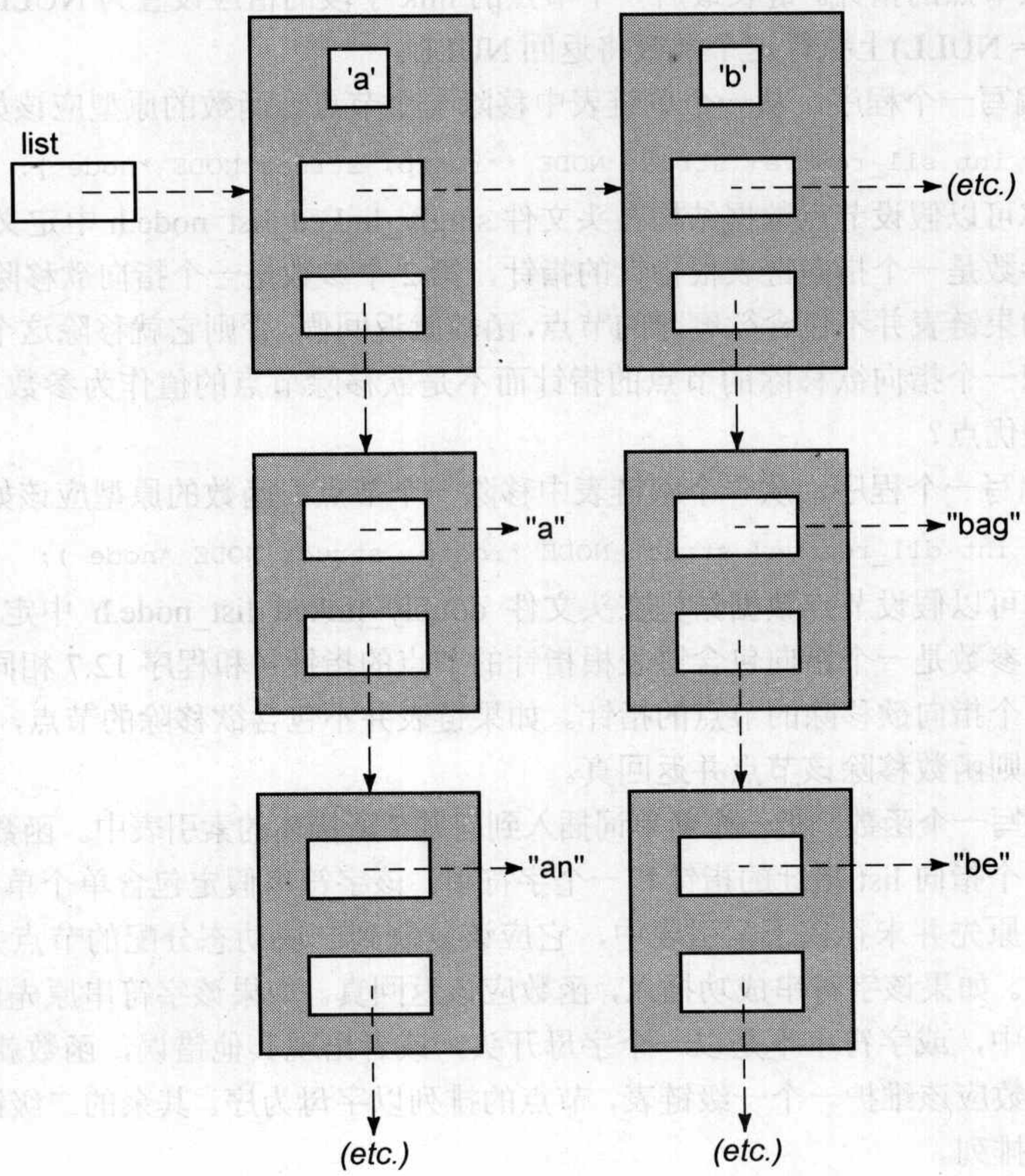


图 12.1　一个索引表

12.8　编程练习

★ 1．编写一个函数，用于计数一个单链表的节点个数。它的唯一参数就是一个指向链表第 1 个节点的指针。编写这个函数时，你必须知道哪些信息？这个函数还能用于执行其他任务吗？

★ 2．编写一个函数，在一个无序的单链表中寻找一个特定的值，并返回一个指向该节点的指针。你可以假设节点数据结构在头文件 singly_linked_list_node.h 中定义。如果想让这个函数适用于有序的单链表，需不需要对它作些修改？

★★★ 3．重新编写程序 12.7 的 dll_insert 函数，使头和尾指针分别以一个单独的指针传递给函数，而不是作为一个节点的一部分。从函数的逻辑而言，这个改动有何效果？

★★★★ 4．编写一个函数，反序排列一个单链表的所有节点。函数应该具有下面的原型：

```
struct NODE * sll_reverse( struct NODE *first);
```

在头文件 singly_linked_list_node.h 中声明节点数据结构。

函数的参数指向链表的第 1 个节点。当链表被重排之后，函数返回一个指向链表新

bbs.theithome.com

头节点的指针。链表最后一个节点的 link 字段的值应设置为 NULL，在空链表(first == NULL)上执行这个函数将返回 NULL。

★★★ 5. 编写一个程序，从一个单链表中移除一个节点。函数的原型应该如下：

```
int sll_remove( struct NODE **rootp, struct NODE *node );
```

你可以假设节点数据结构在头文件 singly_linked_list_node.h 中定义。函数的第 1 个参数是一个指向链表根指针的指针，第 2 个参数是一个指向欲移除的节点的指针。如果链表并不包含欲删除的节点，函数就返回假，否则它就移除这个节点并返回真。把一个指向欲移除的节点的指针而不是欲移除节点的值作为参数传递给函数有哪些优点？

★★★ 6. 编写一个程序，从一个双链表中移除一个节点。函数的原型应该如下：

```
int dll_remove( struct NODE *rootp, struct NODE *node );
```

你可以假设节点数据结构在头文件 doubly_linked_list_node.h 中定义。函数的第 1 个参数是一个指向包含链表根指针的节点的指针（和程序 12.7 相同），第 2 个参数是个指向欲移除的节点的指针。如果链表并不包含欲移除的节点，函数就返回假，否则函数移除该节点并返回真。

★★★★★ 7. 编写一个函数，把一个新单词插入到问题 7 所描述的索引表中。函数接受两个参数，一个指向 list 指针的指针和一个字符串。该字符串假定包含单个单词。如果这个单词原先并未存在于索引表中，它应该复制到一块动态分配的节点并插入到索引表中。如果该字符串成功插入，函数应该返回真。如果该字符串原先已经存在于索引表中，或字符串不是以一个字母开头，或者出现其他错误，函数就返回假。

函数应该维护一个一级链表，节点的排列以字母为序。其余的二级链表则以单词为序排列。

bbs.theithome.com

第13章

高级指针话题

本章收集了各种各样的涉及指针的技巧。有些技巧非常实用，另外一些技巧则学术味更浓一些，还有一些则纯属找乐。但是，这些技巧都很好地说明了这门语言的各种原则。

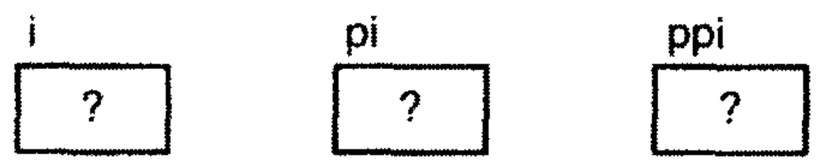
13.1 进一步探讨指向指针的指针

在上一章，我们使用了指向指针的指针，用于简化向单链表插入新值的函数。另外还存在许多领域，指向指针的指针能够发挥重要的作用。

这里有一个通用的例子。

```
int     i;
int     *pi;
int     **ppi;
```

这些声明在内存中创建了下列变量。如果它们是自动变量，我们无法猜测它们的初始值。



有了上面这些信息之后，请问下面各条语句的效果是什么呢？

```
① printf( "%d\n", ppi );
② printf( "%d\n", &ppi ).
③ *ppi = 5;
```

① 如果 ppi 是个自动变量，它就未被初始化，这条语句将打印一个随机值。如果它是个静态变量，这条语句将打印 0。

② 这条语句将把存储 ppi 的地址作为十进制整数打印出来。这个值并不是很有用。

③ 这条语句的结果是不可预测的。对 ppi 不应该执行间接访问操作，因为它尚未被初始化。

接下来的两条语句用处比较大。

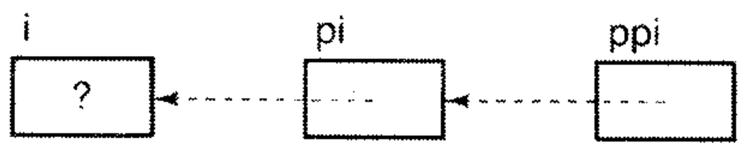
```
ppi = &pi;
```

这条语句把 ppi 初始化为指向变量 pi。以后我们就可以安全地对 ppi 执行间接访问操作了。

bbs.theithome.com

```
*ppi = &i;
```

这条语句把 pi（通过 ppi 间接访问）初始化为指向变量 i。经过上面最后两条语句之后，这些变量变成了下面这个样子：



现在，下面各条语句具有相同的效果：

```
i='a';
*pi='a';
**ppi='a';
```

在一条简单的对 i 赋值的语句就可以完成任务的情况下，为什么还要使用更为复杂的涉及间接访问的方法呢？这是因为简单赋值并不总是可行，例如链表的插入。在那些函数中，我们无法使用简单赋值，因为变量名在函数的作用域内部是未知的。函数所拥有的只是一个指向需要修改的内存位置的指针，所以要对该指针进行间接访问操作以访问需要修改的变量。

在前一个例子中，变量 i 是一个整数，pi 是一个指向整型的指针。但 ppi 是一个指向 pi 的指针，所以它是一个指向整型的指针的指针。假定我们需要另一个变量，它需要指向 ppi。那么，它的类型当然是“指向整型的指针的指针的指针”，而且它应该像下面这样声明：

```
int ***pppi;
```

间接访问的层次越多，你需要用到它的次数就越少。但是，一旦你真正理解了间接访问，无论出现多少层间接访问，你应该都能十分轻松地应付。

提示：

只有当确实需要时，你才应该使用多层间接访问。不然的话，你的程序将会变得更庞大、更缓慢并且更难于维护。

13.2 高级声明

在使用更高级的指针类型之前，我们必须观察它们是如何声明的。前面的章节介绍了表达式声明的思路以及 C 语言的变量如何通过推论进行声明。我们在第 8 章声明指向数组的指针时已经看到过一些推论声明的例子。让我们通过观察一系列越来越复杂的声明进一步探索这个话题。

首先让我们来看几个简单的例子。

```
int      f;   /* 一个整型变量 */
int      *f;  /* 一个指向整型的指针 */
```

不过，请回忆一下第 2 个声明是如何工作的：它把**表达式***f 声明为一个整数。根据这个事实，你肯定能推断出 f 是个指向整型的指针。C 声明的这种解释方法可以通过下面的声明得到验证。

```
int* f, g;
```

它并没有声明两个指针。尽管它们之间存在空白，但星号是作用于 f 的，只有 f 才是一个指针。g 只是一个普通的整型变量。

下面是另外一个例子，你以前曾见过：

```
int      f();
```


bbs.theithome.com

它把 f 声明为一个函数，它的返回值是一个整数。旧式风格的声明对函数的参数并未提供任何信息。它只声明 f 的返回值类型。现在我将使用这种旧式风格，这样例子看上去简单一些，后面我再回到完整的原型形式。

下面是一个新例子：

```
int        *f();
```

要想推断出它的含义，你必须确定表达式*f( )是如何进行求值的。首先执行的是函数调用操作符（），因为它的优先级高于间接访问操作符。因此，f 是一个函数，它的返回值类型是一个指向整型的指针。

如果“推论声明”看上去令你觉得有点讨厌，你只要这样考虑就可以了：用于声明变量的表达式和普通的表达式在求值时所使用的规则相同。你不需要为这类声明学习一套单独的语法。如果你能够对一个复杂表达式求值，你同样可以推断出一个复杂声明的含义，因为它们的原理是相同的。

接下来的一个声明更为有趣：

```
int   (*f)();
```

确定括号的含义是分析这个声明的一个重要步骤。这个声明有两对括号，每对的含义各不相同。第 2 对括号是函数调用操作符，但第 1 对括号只起到聚组的作用。它迫使间接访问在函数调用之前进行，使 f 成为一个函数指针，它所指向的函数返回一个整型值。

函数指针？是的，程序中的每个函数都位于内存中的某个位置，所以存在指向那个位置的指针是完全可能的。函数指针的初始化和使用将在本章后面详述。

现在，下面这个声明应该是比较容易弄懂了：

```
int        *(*f)();
```

它和前一个声明基本相同，f 也是一个函数指针，只是所指向的函数的返回值是一个整型指针，必须对其进行间接访问操作才能得到一个整型值。

现在，让我们把数组也考虑进去。

```
int        f[];
```

这个声明表示 f 是个整型数组。数组的长度暂时省略，因为我们现在关心的是它的类型，而不是它的长度[1]。

下面这个声明又如何呢？

```
int        *f[];
```

这个声明又出现了两个操作符。下标的优先级更高，所以 f 是一个数组，它的元素类型是指向整型的指针。

下面这个例子隐藏着一个圈套。不管怎样，让我们先推断出它的含义。

```
int        f()[];
```

f 是一个函数，它的返回值是一个整型数组。这里的圈套在于这个声明是非法的——函数只能返回标量值，不能返回数组。

这里还有一个例子，颇费思量。

```
int        f[]();
```

现在，f 似乎是一个数组，它的元素类型是返回值为整型的函数。这个声明也是非法的，因为

[1] 如果它们的链接属性是 external 或者是作用函数的参数，即使它们在声明时未注明长度，也仍然是合法的。

bbs.theithome.com

数组元素必须具有相同的长度，但不同的函数显然可能具有不同的长度。

但是，下面这个声明是合法的：

```
int        (*f[])();
```

首先，你必须找到所有的操作符，然后按照正确的次序执行它们。同样，这里有两对括号，它们分别具有不同的含义。括号内的表达式*f[]首先进行求值，所以 f 是一个元素为某种类型的指针的数组。表达式末尾的( )是函数调用操作符，所以 f 肯定是一个数组，数组元素的类型是函数指针，它所指向的函数的返回值是一个整型值。

如果你搞清楚了上面最后一个声明，下面这个应该是比较容易的了：

```
int        *(*f[])();
```

它和上面那个声明的唯一区别就是多了一个间接访问操作符，所以这个声明创建了一个指针数组，指针所指向的类型是返回值为整型指针的函数。

到现在为止，我使用的是旧式风格的声明，目的是为了让例子简单一些。但 ANSI C 要求我们使用完整的函数原型，使声明更为明确。例如：

```
int        (*f)( int, float );
int        *(*g[])( int, float );
```

前者把 f 声明为一个函数指针，它所指的函数接受两个参数，分别是一个整型值和浮点型值，并返回一个整型值。后者把 g 声明为一个数组，数组的元素类型是一个函数指针，它所指向的函数接受两个参数，分别是一个整型值和浮点型值，并返回一个整型指针。尽管原型增加了声明的复杂度，但我们还是应该大力提倡这种风格，因为它向编译器提供了一些额外的信息。

提示：

如果你使用的是 UNIX 系统，并能访问 Internet，你可以获得一个名叫 cdecl 的程序，它可以在 C 语言的声明和英语之间进行转换。它可以解释一个现存的 C 语言声明：

```
cdecl> explain int (*(*f)())[10];
declare f as pointer to function returning pointer to
    array 10 of int
```

或者给你一个声明的语法：

```
cdecl> declare x as pointer to array 10 of pointer to
    function returning int
int (*(*x)[10])()
```

cdecl 的源代码可以从 comp.sources.unix newsgroup 存档文件第 14 卷中获得。

13.3 函数指针

你不会每天都使用函数指针。但是，它们确有用武之地，最常见的两个用途是转换表(jump table)和作为参数传递给另一个函数。本节，我们将探索这两方面的一些技巧。但是，首先容我指出一个常见的错误，这是非常重要的。

警告：

简单声明一个函数指针并不意味着它马上就可以使用。和其他指针一样，对函数指针执行间接访问之前必须把它初始化为指向某个函数。下面的代码段说明了一种初始化函数指针的方法。

```
int        f( int );
```


bbs.theithome.com

```
int        (*pf)( int ) = &f;
```

第 2 个声明创建了函数指针 pf，并把它初始化为指向函数 f。函数指针的初始化也可以通过一条赋值语句来完成。在函数指针的初始化之前具有 f 的原型是很重要的，否则编译器就无法检查 f 的类型是否与 pf 所指向的类型一致。

初始化表达式中的&操作符是可选的，因为**函数名**被使用时总是由编译器把它转换为**函数指针**。&操作符只是显式地说明了编译器将隐式执行的任务。

在函数指针被声明并且初始化之后，我们就可以使用三种方式调用函数：

```
int     ans;

ans = f( 25 );
ans = (*pf)( 25 );
ans = pf( 25 );
```

第 1 条语句简单地使用名字调用函数 f，但它的执行过程可能和你想象的不太一样。函数名 f 首先被转换为一个函数指针，该指针指定函数在内存中的位置。然后，函数调用操作符调用该函数，执行开始于这个地址的代码。

第 2 条语句对 pf 执行间接访问操作，它把函数指针转换为一个函数名。这个转换并不是真正需要的，因为编译器在执行函数调用操作符之前又会把它转换回去。不过，这条语句的效果和第 1 条语句是完全一样的。

第 3 条语句和前两条语句的效果是一样的。间接访问操作并非必需，因为编译器需要的是一个函数指针。这个例子显示了函数指针通常是如何使用的。

什么时候我们应该使用函数指针呢？前面提到过，两个最常见的用途是把函数指针作为参数传递给函数以及用于转换表。让我们各看一个例子。

13.3.1　回调函数

这里有一个简单的函数，它用于在一个单链表中查找一个值。它的参数是一个指向链表第 1 个节点的指针以及那个需要查找的值。

```
Node *
search_list( Node *node, int const value )
{
        while( node != NULL ){
                if( node->value == value )
                        break;
                node = node->link;
        }
        return node;
}
```

这个函数看上去相当简单，但它只适用于值为整数的链表。如果你需要在一个字符串链表中查找，你不得不另外编写一个函数。这个函数和上面那个函数的绝大部分代码相同，只是第 2 个参数的类型以及节点值的比较方法不同。

一种更为通用的方法是使查找函数与类型无关，这样它就能用于任何类型的值的链表。我们必须对函数的两个方面进行修改，使它与类型无关。首先，我们必须改变比较的执行方式，这样函数就可以对任何类型的值进行比较。这个目标听上去好像不可能，如果你编写语句用于比较整型值，它怎么还可能用于其他类型如字符串的比较呢？解决方案就是使用函数指针。调用者编写一个函数，用于比较两个值，然后把一个指向这个函数的指针作为参数传递给查找函数。然后查找函数调用这

bbs.theithome.com

个函数来执行值的比较。使用这种方法，任何类型的值都可以进行比较。

我们必须修改的第 2 个方面是向函数传递一个指向值的**指针**而不是值本身。函数有一个 void * 形参，用于接收这个参数。然后指向这个值的指针便传递给比较函数。这个修改使字符串和数组对象也可以被使用。字符串和数组无法作为参数传递给函数，但指向它们的指针却可以。

使用这种技巧的函数被称为**回调函数**(callback function)，因为用户把一个函数指针作为参数传递给其他函数，后者将“回调”用户的函数。任何时候，如果你所编写的函数必须能够在不同的时刻执行不同类型的工作或者执行只能由函数调用者定义的工作，你都可以使用这个技巧。许多窗口系统使用回调函数连接多个动作，如拖拽鼠标和点击按钮来指定用户程序中的某个特定函数。

我们无法在这个上下文环境中为回调函数编写一个准确的原型，因为我们并不知道进行比较的值的类型。事实上，我们需要查找函数能作用于任何类型的值。解决这个难题的方法是把参数类型声明为 void *，表示“一个指向未知类型的指针”。

提示：

在使用比较函数中的指针之前，它们必须被强制转换为正确的类型。因为强制类型转换能够躲过一般的类型检查，所以你在使用时必须格外小心，确保函数的参数类型是正确的。

在这个例子里，回调函数比较两个值。查找函数向比较函数传递两个指向需要进行比较的值的指针，并检查比较函数的返回值。例如，零表示相等的值，非零值表示不相等的值。现在，查找函数就与类型无关，因为它本身并不执行实际的比较。确实，调用者必须编写必需的比较函数，但这样做是很容易的，因为调用者知道链表中所包含的值的类型。如果使用几个分别包含不同类型值的链表，为每种类型编写一个比较函数就允许单个查找函数作用于所有类型的链表。

程序 13.1 是类型无关查找函数的一种实现方法。注意函数的第 3 个参数是一个函数指针。这个参数用一个完整的原型进行声明。同时注意虽然函数绝不会修改参数 node 所指向的任何节点，但 node 并未被声明为 const。如果 node 被声明为 const，函数将不得不返回一个 const 结果，这将限制调用程序，它便无法修改查找函数所找到的节点。

```
/*
** 在一个单链表中查找一个指定值的函数。它的参数是一个指向链表第 1 个节点的
** 指针，一个指向我们需要查找的值的指针和一个函数指针，它所指向的函数用于比
** 较存储于链表中的类型的值。
*/
#include <stdio.h>
#include "node.h"

Node *
search_list( Node *node, void const *value,
    int (*compare)( void const *, void const * ) )
{
        while( node != NULL ){
             if( compare( &node->value, value ) == 0 )
                 break;
        node = node->link;
        }
        return node;
}
```

程序 13.1　类型无关的链表查找　　search.c

bbs.theithome.com

指向值参数的指针和&node->value 被传递给比较函数。后者是我们当前所检查的节点的值。在选择比较函数的返回值时，我选择了与直觉相反的约定，就是相等返回零值，不相等返回非零值。它的目的是为了与标准库的一些函数所使用的比较函数规范兼容。在这个规范中，不相等操作数的报告方式更为明确——负值表示第 1 个参数小于第 2 个参数，正值表示第 1 个参数大于第 2 个参数。

在一个特定的链表中进行查找时，用户需要编写一个适当的比较函数，并把指向该函数的指针和指向需要查找的值的指针传递给查找函数。例如，下面是一个比较函数，它用于在一个整数链表中进行查找。

```
int
compare_ints( void const *a, void const *b )
{
        if( *(int *)a == *(int *)b )
                return 0;
        else
                return 1;
}
```

这个函数将像下面这样使用：

```
desired_node = search_list( root, &desired_value,
    compare_ints );
```

注意强制类型转换：比较函数的参数必须声明为 void *以匹配查找函数的原型，然后它们再强制转换为 int *类型，用于比较整型值。

如果你希望在一个字符串链表中进行查找，下面的代码可以完成这项任务：

```
#include <string.h>
...
desired_node = search_list( root, "desired_value",
    strcmp );
```

碰巧，库函数 strcmp 所执行的比较和我们需要的完全一样，不过有些编译器会发出警告信息，因为它的参数被声明为 char *而不是 void *。

13.3.2　转移表

转移表最好用个例子来解释。下面的代码段取自一个程序，它用于实现一个袖珍式计算器。程序的其他部分已经读入两个数（op1 和 op2）和一个操作符（oper）。下面的代码对操作符进行测试，然后决定调用哪个函数。

```
switch( oper ){
case ADD:
        result = add( op1, op2 );
        break;

case SUB:
        result = sub( op1, op2 );
        break;

case MUL:
        result = mul( op1, op2 );
        break;

case DIV:
        result = div( op1, op2 );
        break;
...
```

对于一个新奇的具有上百个操作符的计算器，这条 switch 语句将会非常之长。

bbs.theithome.com

为什么要调用函数来执行这些操作呢？把具体操作和选择操作的代码分开是一种良好的设计方案。更为复杂的操作将肯定以独立的函数来实现，因为它们的长度可能很长。但即使是简单的操作也可能具有副作用，例如保存一个常量值用于以后的操作。

为了使用 switch 语句，表示操作符的代码必须是整数。如果它们是从零开始连续的整数，我们可以使用转换表来实现相同的任务。转换表就是一个函数指针数组。

创建一个转换表需要两个步骤。首先，声明并初始化一个函数指针数组。唯一需要留心之处就是确保这些函数的原型出现在这个数组的声明之前。

```
double  add( double, double );
double  sub( double, double );
double  mul( double, double );
double  div( double, double );
...
double  (*oper_func[])( double, double ) = {
        add, sub, mul, div, ...
};
```

初始化列表中各个函数名的正确顺序取决于程序中用于表示每个操作符的整型代码。这个例子假定 ADD 是 0，SUB 是 1，MUL 是 2，接下去以此类推。

第 2 个步骤是用下面这条语句替换前面整条 switch 语句！

```
result = oper_func[ oper ]( op1, op2 );
```

oper 从数组中选择正确的函数指针，而函数调用操作符将执行这个函数。

警告：

在转换表中，越界下标引用就像在其他任何数组中一样是不合法的。但一旦出现这种情况，把它诊断出来要困难得多。当这种错误发生时，程序有可能在三个地方终止。首先，如果下标值远远越过了数组的边界，它所标识的位置可能在分配给该程序的内存之外。有些操作系统能检测到这个错误并终止程序，但有些操作系统并不这样做。如果程序被终止，这个错误将在靠近转换表语句的地方被报告，问题相对而言较易诊断。

如果程序并未终止，非法下标所标识的值被提取，处理器跳到该位置。这个不可预测的值可能代表程序中一个有效的地址，但也可能不是这样。如果它不代表一个有效地址，程序此时也会终止，但错误所报告的地址从本质上说是一个随机数。此时，问题的调试就极为困难。

如果程序此时还未失败，机器将开始执行根据非法下标所获得的虚假地址的指令，此时要调试出问题根源就更为困难了。如果这个随机地址位于一块存储数据的内存中，程序通常会很快终止，这通常是由于非法指令或非法的操作数地址所致（尽管数据值有时也能代表有效的指令，但并不总是这样）。要想知道机器为什么会到达那个地方，唯一的线索是转移表调用函数时存储于堆栈中的返回地址。如果任何随机指令在执行时修改了堆栈或堆栈指针，那么连这个线索也消失了。

更糟的是，如果这个随机地址恰好位于一个函数的内部，那么该函数就会快乐地执行，修改谁也不知道的数据，直到它运行结束。但是，函数的返回地址并不是该函数所期望的保存于堆栈上的地址，而是另一个随机值。这个值就成为下一个指令的执行地址，计算机将在各个随机地址间跳转，执行位于那里的指令。

问题在于指令破坏了机器如何到达错误最后发生地点的线索。没有了这方面的信息，要查明问题的根源简直难如登天。如果你怀疑转移表有问题，可以在那个函数调用之前和之后各打印一条信息。如果被调用函数不再返回，用这种方法就可以看得很清楚。但困难在于人们很难认识到程序某个部分的失败可以是位于程序中相隔甚远的且不相关部分的一个转移表错误所引起的。

bbs.theithome.com

提示：

一开始，保证转移表所使用的下标位于合法的范围是很容易做到的。在这个计算器例子里，用于读取操作符并把它转换为对应整数的函数应该核实该操作符是有效的。

13.4　命令行参数

处理命令行参数是指向指针的指针的另一个用武之地。有些操作系统，包括 UNIX 和 MS-DOS，让用户在命令行中编写参数来启动一个程序的执行。这些参数被传递给程序，程序按照它认为合适的任何方式对它们进行处理。

13.4.1　传递命令行参数

这些参数如何传递给程序呢？C 程序的 main 函数具有两个形参[1]。第 1 个通常称为 argc，它表示命令行参数的数目。第 2 个通常称为 argv，它指向一组参数值。由于参数的数目并没有内在的限制，所以 argv 指向这组参数值（从本质上说是一个数组）的第 1 个元素。这些元素的每个都是指向一个参数文本的指针。如果程序需要访问命令行参数，main 函数在声明时就要加上这些参数：

```
int
main( int argc, char **argv )
```

注意这两个参数通常取名为 argc 和 argv，但它们并无神奇之处。如果你喜欢，也可以把它们称为“fred”和“ginger”，只不过程序的可读性会差一点。

图 13.1 显示了下面这条命令行是如何进行传递的：

```
$ cc -c -o main.c insert.c -o test
```

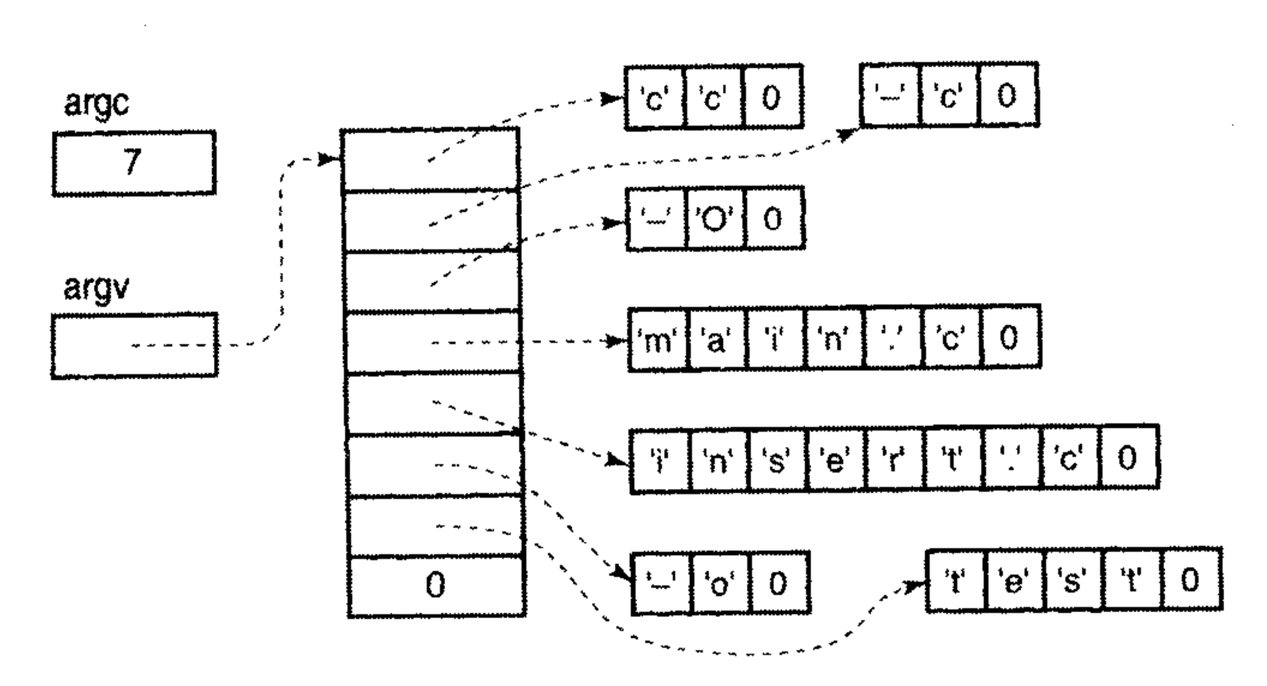


图 13.1　命令行参数

注意指针数组：这个数组的每个元素都是一个字符指针，数组的末尾是一个 NULL 指针。argc 的值和这个 NULL 值都用于确定实际传递了多少个参数。argv 指向数组的第 1 个元素，这就是它为什么被声明为一个指向字符的指针的指针的原因。

[1] 实际上，有些操作系统向 main 函数传递第 3 个参数，它是一个指向环境变量列表以及它们的值的指针。请参考你的编译器或操作系统文档，了解更多细节。

bbs.theithome.com

最后一个需要注意的地方是第 1 个参数就是程序的名称。把程序名作为参数传递有什么用意呢？程序显然知道自己的名字，通常这个参数是被忽略的。不过，如果程序通常采用几组不同的选项进行启动，此时这个参数就有用武之地了。UNIX 中用于列出一个目录的所有文件的 ls 命令就是一个这样的程序。在许多 UNIX 系统中，这个命令具有几个不同的名字。当它以名字 ls 启动时，它将产生一个文件的简单列表；当它以名字 l 启动，它就产生一个多列的简单列表；如果它以名字 ll 启动，它就产生一个文件的详细列表。程序对第 1 个参数进行检查，确定它是由哪个名字启动的，从而根据这个名字选择启动选项。

在有些系统中，参数字符串是挨个存储的。这样当你把指向第 1 个参数的指针向后移动，越过第 1 个参数的尾部时，就到达了第 2 个参数的起始位置。但是，这种排列方式是由编译器定义的，所以你不能依赖它。为了寻找一个参数的起始位置，你应该使用数组中合适的指针。

程序是如何访问这些参数的呢？程序13.2是一个非常简单的例子——它简单地打印出它的所有参数（除了程序名），非常像 UNIX 的 echo 命令。

```
/*
** 一个打印其命令行参数的程序
*/
#include <stdio.h>
#include <stdlib.h>

int
main( int argc, char **argv )
{
        /*
        ** 打印参数，直到遇到 NULL 指针（未使用 argc）。程序名被跳过。
        */
        while( *++argv != NULL )
              printf( "%s\n", *argv );
        return EXIT_SUCCESS;
}
```

程序 13.2　打印命令行参数　　echo.c

while 循环增加 argc 的值，然后检查*argv，看看是否到达了参数列表的尾部，方法是把每个参数都与表示列表末尾的 NULL 指针进行比较。如果还存在另外的参数，循环体就执行，打印出这个参数。在循环一开始就增加 argc 的值，程序名就被自动跳过了。

printf 函数的格式字符串中的%s 格式码要求参数是一个指向字符的指针。printf 假定该字符是一个以 NUL 字节结尾的字符串的第 1 个字符。对 argv 参数使用间接访问操作产生它所指向的值，也就是一个指向字符的指针——这正是格式所要求的。

13.4.2　处理命令行参数

让我们编写一个程序，用一种更加现实的方式处理命令行参数。这个程序将处理一种非常常见的形式——文件名参数前面的选项参数。在程序名的后面，可能有零个或多个选项，后面跟随零个或多个文件名，像下面这样：

```
prog -a -b -c name1 name2 name3
```

每个选项都以一条横杠开头，后面是一个字母，用于在几个可能的选项中标明程序所需的一个。

bbs.theithome.com

每个文件名以某种方式进行处理。如果命令行中没有文件名，就对标准输入进行处理。

为了让这些例子更为通用，我们的程序设置了一些变量，记录程序所找到的选项。一个现实程序的其他部分可能会测试这些变量，用于确定命令所请求的处理方式。在一个现实的程序中，如果程序发现它的命令行参数有一个选项，其对应的处理过程就可能也会执行。

下面的程序 13.3 和程序 13.2 颇为相似，因为它包含了一个循环，检查所有的参数。它们的主要区别在于我们现在必须区分选项参数和文件名参数。当循环到达并非以横杠开关的参数时就结束。第 2 个循环用于处理文件名。

```
/*
** 处理命令行参数
*/
#include <stdio.h>
#define   TRUE 1

/*
** 执行实际任务的函数的原型。
*/
void process_standard_input( void );
void process_file( char *file_name );

/*
** 选项标志，缺省初始化为 FALSE。
*/
int  option_a, option_b  /* etc. */ ;

void
main( int argc, char **argv )
{
        /*
        ** 处理选项参数：跳到下一个参数，并检查它是否以一个横杠开头。
        */
        while( *++argv != NULL && **argv == '-' ){
        /*
        ** 检查横杠后面的字母。
        */
              switch( *++*argv ){
              case 'a':
                    option_a = TRUE;
                    break;

        case 'b':
                    option_b = TRUE;
                    break;

        /* etc. */
        }
}

/*
** 处理文件名参数
*/
if( *argv == NULL )
    process_standard_input();
```


bbs.theithome.com

```
        else {
            do {
                process_file( *argv );
            } while( *++argv != NULL );
        }
}
```

程序 13.3　处理命令行参数　　cmd_line.c

注意，在程序 13.3 的 while 循环中，增加了下面这个测试：

```
**argv == '-'
```

双重间接访问操作访问参数的第 1 个字符，如图 13.2 所示。如果这个字符不是一个横杠，那就表示不再有其他的选项，循环终止。注意在测试**argv 之前先测试*argv 是非常重要的。如果*argv 为 NULL，那么**argv 中的第 2 个间接访问就是非法的。

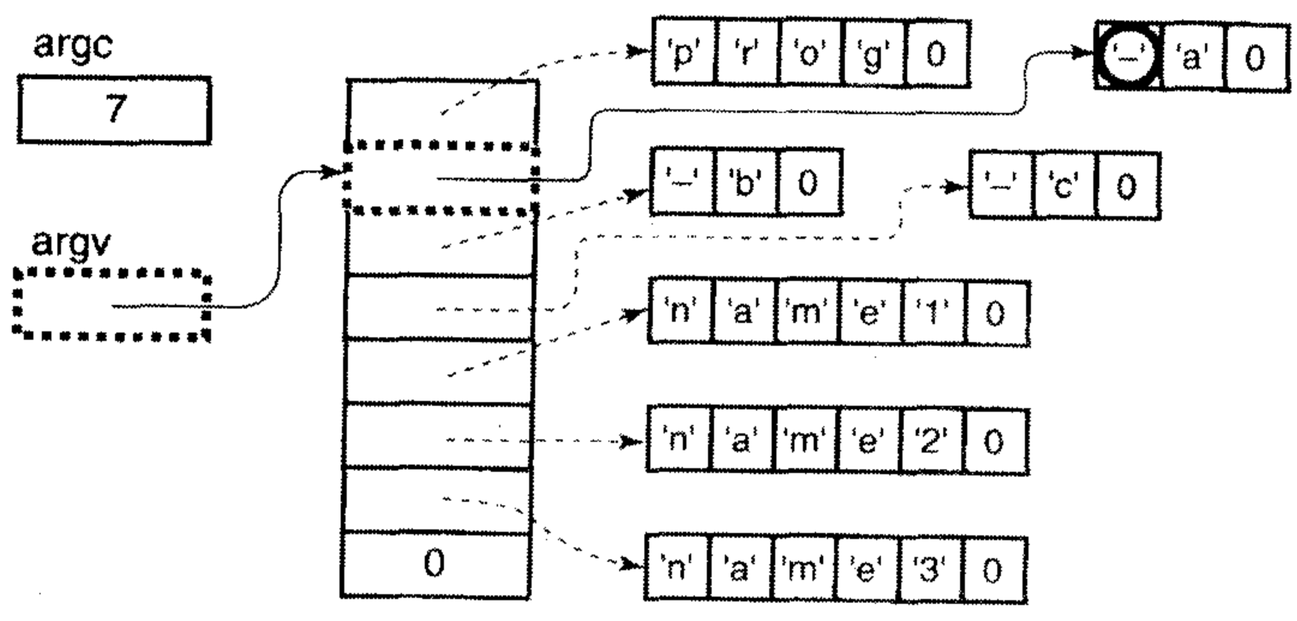


图 13.2　访问参数

switch 语句中的*++*argv 表达式你以前曾见到过。第 1 个间接访问操作访问 argv 所指的位置，然后这个位置执行自增操作。最后一个间接访问操作根据自增后的指针进行访问，如图 13.3 所示。switch 语句根据找到的选项字母设置一个变量，while 循环中的++操作符使 argv 指向下一个参数，用于循环的下一次迭代。

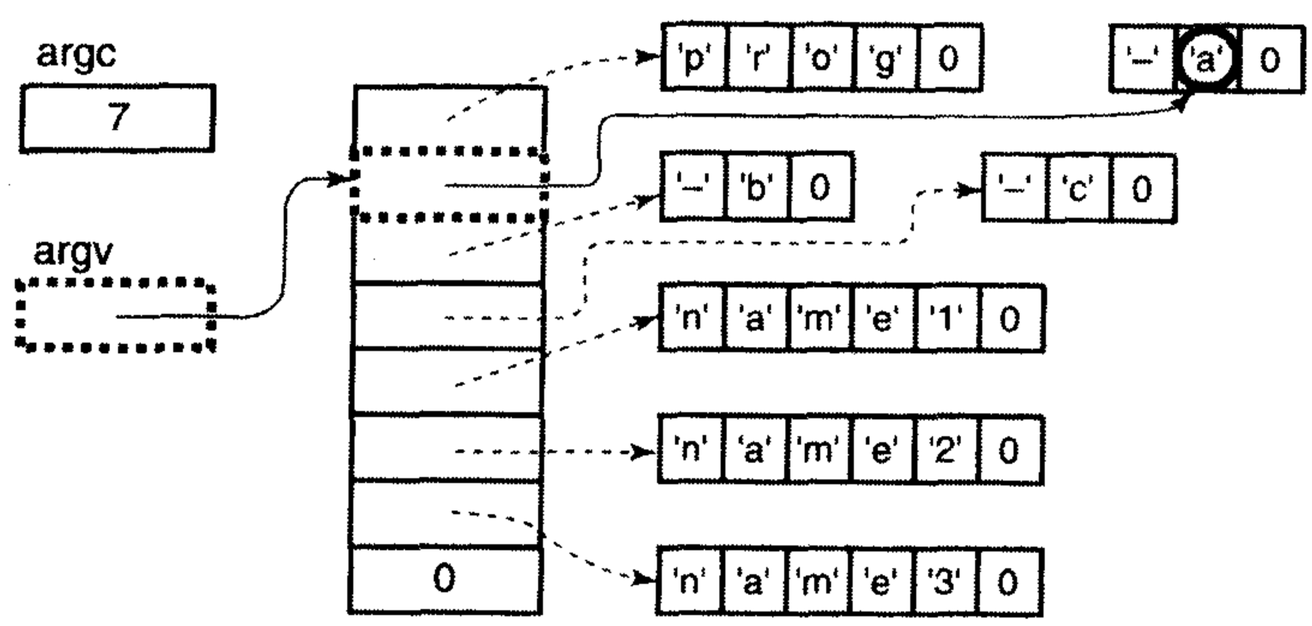


图 13.3　访问参数中的下一个字符

当不再存在其他选项时，程序就处理文件名。如果 argv 指向 NULL 指针，命令行参数里就没有别的东西了，程序就处理标准输入。否则，程序就逐个处理文件名。这个程序的函数调用较为通用，

bbs.theithome.com

它们并未显示一个现实程序可能执行的任何实际工作。然而，这个设计方式是非常好的。Main 程序处理参数，这样执行处理过程的函数就无需担心怎样对选项进行解析或者怎样挨个访问文件名。

有些程序允许用户在一个参数中放入多个选项字母，像下面这样：

```
prog -abc name1 name2 name3
```

一开始你可能会觉得这个改动会使我们的程序变得复杂，但实际上它很容易进行处理。每个参数都可能包含多个选项，所以我们使用另一个循环来处理它们。这个循环在遇到参数末尾的 NUL 字节时应该结束。

程序 13.3 中的 switch 语句由下面的代码段代替。

```
while( ( opt = *++*argv ) != '\0' ){
        switch( opt ){
        case 'a':
                option_a = TRUE;
                break;
        /* etc. */
        }
}
```

循环中的测试使参数指针移动到横杠后的那个位置，并复制一份位于那里的字符。如果这个字符并非 NUL 字节，那么就像前面一样使用 switch 语句来设置合适的变量。注意选项字符被保存到局部变量 opt 中，这可以避免在 switch 语句中对**argv 进行求值。

提示：

注意，使用这种方式，命令行参数可能只能处理一次，因为指向参数的指针在内层的循环中被破坏。如果必须多次处理参数，当你挨个访问列表时，对每个需要增值的指针都作一份拷贝。

在处理选项时还存在其他的可能性。例如，选项可能是一个单词而不是单个字母，或者可能有一些值与某些选项联系在一起，如下面的例子所示：

```
cc -o prog prog.c
```

本章的其中一个问题就是对这个思路的扩展。

13.5　字符串常量

现在是时候对以前曾提过的一个话题进行更深入的讨论了，这个话题就是字符串常量。当一个字符串常量出现于表达式中时，它的值是个指针常量。编译器把这些指定字符的一份拷贝存储在内存的某个位置，并存储一个指向第 1 个字符的指针。但是，当数组名用于表达式中时，它们的值也是指针常量。我们可以对它们进行下标引用、间接访问以及指针运算。这些操作对于字符串常量是不是也有意义呢？让我们来看一些例子。

下面这个表达式是什么意思呢？

```
"xyz" + 1
```

对于绝大多数程序员而言，它看上去像堆垃圾。它好像是试图在一个字符串上面执行某种类型的加法运算。但是，当你记得字符串常量实际上是个指针时，它的意义就变得清楚了。这个表达式计算“指针值加上 1”的值。它的结果是个指针，指向字符串中的第 2 个字符：y。

那么这个表达式又是什么呢？

```
*"xyz"
```


bbs.theithome.com

对一个指针执行间接访问操作时，其结果就是指针所指向的内容。字符串常量的类型是“指向字符的指针”，所以这个间接访问的结果就是它所指向的字符：x。注意表达式的结果并不是整个字符串，而只是它的第 1 个字符。

下一个例子看上去也是有点奇怪，不过现在你应该能够推断出这个表达式的值就是字符 z。

```
"xyz" [2]
```

最后这个例子包含了一个错误。偏移量 4 超出了这个字符串的范围，所以这个表达式的结果是一个不可预测的字符。

```
*( "xyz" + 4 )
```

什么时候人们可能想使用类似上面这些形式的表达式呢？程序 13.4 的函数是一个有用的例子。你能够推断出这个神秘的函数执行了什么任务吗？提示：用几个不同的输入值追踪函数的执行过程，并观察它的打印结果。答案将在本章结束时给出。

同时，让我们来看一个另外的例子。程序 13.5 包含了一个函数，它把二进制值转换为字符并把它们打印出来。你第 1 次看到这个函数是在程序 7.6 中。我们将修改这个例子，以十六进制的形式打印结果值。第 1 个修改很容易：只要把结果除以 16 而不是 10 就可以了。但是，现在余数可能是 0～15 的任何值，而 10～15 的值应该以字母 A～F 来表示。下面的代码是解决这个问题的一种典型方法。

```
remainder = value % 16;
if( remainder < 10 )
        putchar( remainder + '0' );
else
        putchar( remainder - 10 + 'A' );
```

我使用了一个局部变量来保存余数，而不是三次分别计算它。对于 0～9 的余数，就和以前一样打印一个十进制数字。但对于其他余数，就把它们以字母的形式打印出来。代码中的测试是必要的，因为在任何常见的字符集中，字母 A～F 并不是立即位于数字的后面。

```
/*
**  神秘函数
**
**    参数是一个 0～100 的值
*/
#include <stdio.h>

void
mystery( int n )
{
     n += 5;
     n /= 10;
     printf( "%s\n", "**********" + 10 - n );
}
```

程序 13.4　神秘函数　　mystery.c

```
/*
** 接受一个整型值（无符号），把它转换为字符，并打印出来。前导零被去除。
*/
#include <stdio.h>
```


bbs.theithome.com

```
void
binary_to_ascii( unsigned int value )
{
     unsigned int   quotient;

     quotient = value / 10;
     if( quotient != 0 )
          binary_to_ascii( quotient );
     putchar( value % 10 + '0' );
}
```

程序 13.5　把二进制值转换为字符　　btoa.c

下面的代码用一种不同的方法解决这个问题。

```
putchar( "0123456789ABCDEF" [value % 16 ] );
```

同样，余数将是一个 0～15 的值。但这次它使用下标从字符串常量中选择一个字符进行打印。前面的代码是比较复杂的，因为字母和数字在字符集中并不是相邻的。这个方法定义了一个字符串，使字母和数字相邻，从而避免了这种复杂性。余数将从字符串中选择一个正确的数字。

第 2 种方法比传统的方法要快，因为它所需要的操作更小。但是，它的代码并不一定比原来的方法更小。虽然指令减少了，但它付出的代价是多了一个 17 个字节的字符串常量。

提示：

但是，如果程序的可读性大幅度下降，对于因此获得的执行速度的略微提高是得不偿失的。当你使用一种不寻常的技巧或语句时，确保增加一条注释，描述它的工作原理。一旦解释清楚了这个例子，它实际上比传统的代码更容易理解，因为它更短一些。

现在让我们回到神秘函数。你是不是已经猜出它的意思？它根据参数值的一定比例打印相应数量的星号。如果参数为 0，它就打印 0 个星号；如果参数为 100，它就打印 10 个星号；位于 0～100 的参数值就打印出 0～10 个的星号。换句话说，这个函数打印一幅柱状图的一横，它比传统的循环方案要容易得多，效率也高得多。

13.6　总结

如果声明得当，一个指针变量可以指向另一个指针变量。和其他的指针变量一样，一个指向指针的指针在它使用之前必须进行初始化。为了取得目标对象，必须对指针的指针执行双重的间接访问操作。更多层的间接访问也是允许的（比如一个指向整型的指针的指针的指针），但它们与简单的指针相比用的较少。你也可以创建指向函数和数组的指针，还可以创建包含这类指针的数组。

在 C 语言中，声明是以推论的形式进行分析的。下面这个声明

```
int     *a;
```

把表达式*a 声明为一个整型。你必须随之推断出 a 是个指向整型的指针。通过推论声明，阅读声明的规则就和阅读表达式的规则一样了。

你可以使用函数指针来实现回调函数。一个指向回调函数的指针作为参数传递给另一个函数，后者使用这个指针调用回调函数。使用这种技巧，你可以创建通用型函数，用于执行普通的操作如在一个链表中查找。任何特定问题的某个实例的工作，如在链表中进行值的比较，由客户提供的回

bbs.theithome.com

调函数执行。

转移表也使用函数指针。转移表像 switch 语句一样执行选择。转移表由一个函数指针数组组成（这些函数必须具有相同的原型）。函数通过下标选择某个指针，再通过指针调用对应的函数。你必须始终保证下标值处于适当的范围之内，因为在转移表中调试错误是非常困难的。

如果某个执行环境实现了命令行参数，这些参数是通过两个形参传递给 main 函数的。这两个形参通常称为 argc 和 argv。argc 是一个整数，用于表示参数的数量。argv 是一个指针，它指向一个序列的字符型指针。该序列中的每个指针指向一个命令行参数。该序列以一个 NULL 指针作为结束标志。其中第 1 个参数就是程序的名字。程序可以通过对 argv 使用间接访问操作来访问命令行参数。

出现在表达式中的字符串常量的值是一个常量指针，它指向字符串的第 1 个字符。和数组名一样，你既可以用指针表达式也可以用下标来使用字符串常量。

13.7 警告的总结

1. 对一个未初始化的指针执行间接访问操作。
2. 在转移表中使用越界下标。

13.8 编程提示的总结

1. 如果并非必要，避免使用多层间接访问。
2. cdecl 程序可以帮助你分析复杂的声明。
3. 把 void *强制转换为其他类型的指针时必须小心。
4. 使用转移表时，应始终验证下标的有效性。
5. 破坏性的命令行参数处理方式使你以后无法再次进行处理。
6. 不寻常的代码始终应该加上一条注释，描述它的目的和原理。

13.9 问题

1. 下面显示了一列声明。

```
a.  int    abc();
b.  int    abc[3];
c.  int    **abc();
d.  int    (*abc)();
e.  int    (*abc)[6];
f.  int    *abc();
g.  int    **(*abc[6])();
h.  int    **abc[6];
i.  int    *(*abc)[6];
j.  int    *(*abc())();
k.  int    (**(*abc)())();
l.  int    (*(*abc)())[6];
m.  int    *(*(*(*abc)())[6])();
```


bbs.theithome.com

从下面的列表中挑出与上面各个声明匹配的最佳描述。

Ⅰ.　int 型指针（指向 int 的指针）。
Ⅱ.　int 型指针的指针。
Ⅲ.　int 型数组。
Ⅳ.　指向“int 型数组”的指针。
Ⅴ.　int 型指针数组。
Ⅵ.　指向“int 型指针数组”的指针。
Ⅶ.　int 型指针的指针数组。
Ⅷ.　返回值为 int 的函数。
Ⅸ.　返回值为“int 型指针”的函数。
Ⅹ.　返回值为“int 型指针的指针”的函数。
Ⅺ.　返回值为 int 的函数指针。
Ⅻ.　返回值为 int 型指针的函数指针。
XIII.　返回值为 int 型指针的指针的函数指针。
XIV.　返回值为 int 的函数指针的数组。
XV.　指向“返回值为 int 型指针的函数”的指针的数组。
XVI.　指向“返回值为 int 型指针的指针的函数”的指针的数组。
XVII.　返回值为“返回值为 int 的函数指针”的函数。
XVIII.　返回值为“返回值为 int 的函数的指针的指针”的函数。
XIX.　返回值为“返回值为 int 型指针的函数指针”的函数。
XX.　返回值为“返回值为 int 的函数指针”的函数指针。
XXI.　返回值为“返回值为 int 的函数指针的指针”的函数指针。
XXII.　返回值为“返回值为 int 型指针的函数指针”的函数指针。
XXIII.　返回值为“指向 int 型数组的指针”的函数指针。
XXIV.　返回值为“指向 int 型指针数组的指针”的函数指针。
XXV.　返回值为“指向‘返回值为 int 型指针的函数指针’的数组的指针”的函数指针。
XXVI.　非法。

2．给定下列声明：

```
char      *array[10];
char      **ptr = array;
```

如果变量 ptr 加上 1，它的效果是什么样的？

3．假定你将要编写一个函数，它的起始部分如下所示：

```
void func( int ***arg ){
```

参数的类型是什么？画一张图，显示这个变量的正确用法。如果想取得这个参数所指代的整数，你应该使用怎样的表达式？

4．下面的代码可以如何进行改进？

bbs.theithome.com

```
Transaction *trans;
trans->product->orders += 1;
trans->product->quantity_on_hand -= trans->quantity;
trans->product->supplier->reorder_quantity
    += trans->quantity;
if( trans->product->export_restricted ){
        ...
}
```

5. 给定下列声明：

```
typedef         struct  {
        int     x;
        int     y;
} Point;

Point   p;
Point   *a = &p;
Point   **b = &a;
```

判断下面各个表达式的值。

a. `a`

b. `*a`

c. `a->x`

d. `b`

e. `b->a`

f. `b->x`

g. `*b`

h. `*b->a`

i. `*b->x`

j. `b->a->x`

k. `(*b)->a`

l. `(*b)->x`

m. `**b`

6. 给定下列声明：

```
typedef         struct  {
        int     x;
        int     y;
} Point;

Point   x, y;
Point   *a = &x, *b = &y;
```

解释下列各语句的含义。

a. `x = y;`

b. `a = y;`

c. `a = b;`

d. `a = *b;`

e. `*a = *b;`

7. 许多 ANSI C 的实现都包含了一个函数，称为 getopt。这个函数用于帮助处理命令行参数。但是，getopt 在标准中并未提及。拥有这样一个函数，有什么优点？又有什么缺点？

8. 下面的代码段有什么错误（如果有的话）？你如何修正它？

bbs.theithome.com

```
char    * pathname = "/usr/temp/xxxxxxxxxxxxxxx"
…
/*
**Insert the filename in to the pathname.
*/
strcpy ( pathname+10 , "abcde");
```

9. 下面的代码段有什么错误（如果有的话）？你如何修正它？

```
char    pathname[] = "/usr/temp/";
...
/*
** Append the filename to the pathname.
*/
strcat( pathname, "abcde" );
```

10. 下面的代码段有什么错误（如果有的话）？你如何修正它？

```
char    *pathname [20] = "/usr/temp/ ";
…
/*
**  Append the filename to the pathname.
*/
stroat (pathrame,filename);
```

11. 标准表示如果对一个字符串常量进行修改，其效果是未定义的。如果你修改了字符串常量，有可能会出现什么问题呢？

13.10 编程练习

★★ 1. 编写一个程序，从标准输入读取一些字符，并根据下面的分类计算各类字符所占的百分比：

控制字符
空白字符
数字
小写字母
大写字母
标号符号
不可打印字符

这些字符的分类是根据 ctype.h 中的函数定义的。不能使用一系列的 If 语句。

★ 2. 编写一个通用目的的函数，遍历一个单链表。它应该接受两个参数：一个指向链表第 1 个节点的指针和一个指向一个回调函数的指针。回调函数应该接受单个参数，也就是指向一个链表节点的指针。对于链表中的每个节点，都应该调用一次这个回调函数。这个函数需要知道链表节点的什么信息？

★★ 3. 转换下面的代码段，使它改用转移表而不是 switch 语句。

```
Node *list;
Node *current;
Transaction  *transaction;
typedef enum  { NEW, DELETE, FORWARD, BACKWARD,
     SEARCH, EDIT } Trans_type;
...
switch( transaction->type ) {
case    NEW
```


bbs.theithome.com

```
            add_new_trans(list,transaction);
            break;
        case DELETE:
                current = delete_trans( list, current );
                break;

        case FORWARD:
                current = current->next;
                break;

        case BACKWARD:
                current = current->prev;
                break;

        case SEARCH:
                current = search( list, transaction );
                break;

        case EDIT:
                edit( current, transaction );
                break;

        default:
                printf( "Illegal transaction type!\n" );
                break;
        }
```

★★★★4. 编写一个名叫 sort 的函数，它用于对一个任何类型的数组进行排序。为了使函数更为通用，它的其中一个参数必须是一个指向比较回调函数的指针，该回调函数由调用程序提供。比较函数接受两个参数，也就是两个指向需要进行比较的值的指针。如果两个值相等，函数返回零；如果第 1 个值小于第 2 个，函数返回一个小于零的整数；如果第 1 个值大于第 2 个，函数返回一个大于零的整数。

sort 函数的参数将是：

1. 一个指向需要排序的数组的第 1 个值的指针。
2. 数组中值的个数。
3. 每个数组元素的长度。
4. 一个指向比较回调函数的指针。

sort 函数没有返回值。

你将不能根据实际类型声明数组参数，因为函数应该可以对不同类型的数组进行排序。如果你把数据当作一个字符数组使用，你可以用第 3 个参数寻找实际数组中每个元素的起始位置，也可以用它交换两个数组元素（每次一个字节）。

对于简单的交换排序，你可以使用下面的算法，当然也可以使用你认为更好的算法。

```
for i = 1 to 元素数-1 do
    for j = i + 1 to 元素数 do
        if 元素 i > 元素 j then
            交换元素 i 和元素 j
```

★★★★★5. 编写代码处理命令行参数是十分乏味的，所以最好有一个标准函数来完成这项工作。但是，不同的程序以不同的方式处理它们的参数。所以，这个函数必须非常灵活，以便使它能用于更多的程序。在本题中，你将编写这样一个函数。你的函数通过寻找和提取参数来提供灵活性。用户所提供的回调函数将执行实际的处理工作。

下面是函数的原型。注意它的第 4 个和第 5 个参数是回调函数的原型。

bbs.theithome.com

```
char **
do_args( int argc, char **argv, char *control,
    void (*do_arg)( int ch, char *value ),
    void (*illegal_arg)( int ch ) );
```

头两个参数就是 main 函数的参数，main 函数对它们不作修改，直接传递给 do_args 第 3 个参数是个字符串，用于标识程序期望接受的命令行参数。最后两个参数都是函数指针，它们是由用户提供的。

do_args 函数按照下面这样的方式处理命令行参数：

```
跳过程序名参数
while 下一次参数以一个横杠开头
    对于参数横杠后面的每个字符
        处理字符
返回一个指针，指向下一个参数指针。
```

为了“处理字符”，你首先必须观察该字符是否位于 control 字符串内。如果它并不位于那里，调用 illegal_arg 所指向函数，把这个字符作为参数传递过去。如果它位于 control 字符串内，但它的后面并不是跟一个+号，那么就调用 do_arg 所指向的函数，把这个字符和一个 NULL 指针作为参数传递过去。

如果该字符位于 control 字符串内并且后面跟一个+号，那么就应该有一个值与这个字符相联系。如果当前参数还有其他字符，它们就是我们需要的值。否则，下一个参数才是这个值。在任何一种情况下，你应该调用 do_arg 所指向的函数，把这个字符和指向这个值的指针传递过去。如果不存在这个值（当前参数没有其他字符，且后面不再有参数），那么你应该改而调用 illegal_arg 函数。**注意**：你必须保证这个值中的字符以后不会被处理。

当所有以一个横杠开头的参数被处理完毕后，你应该返回一个指向下一个命令行参数的指针的指针（也就是一个诸如&argv[4]或 argv+4 的值）。如果所有的命令行参数都以一个横杠开头，你就返回一个指向“命令行参数列表中结尾的 NULL 指针”的指针。

这个函数必须既不能修改命令行参数指针，也不能修改参数本身。为了说明这一点，假定程序 prog 调用这个函数：下面的例子显示了几个不同集合的参数的执行结果。

命令行：	$ prog –x –y z
control：	“x”
do_args 调用：	(*do_arg)(‘x’, 0)
	(*illegal_arg)(‘y’)
并且返回：	&argv[3]
命令行：	$ prog –x –y –z
control：	“x+y+z+”
do_args 调用：	(*do_arg)(‘x’, “-y”)
	(*illegal_arg)(‘z’)
并且返回：	&argv[4]

bbs.theithome.com

续表

命令行：	$ prog –abcd –ef ghi jkl
control：	“ab+cdef+g”
do_args 调用：	(*do_arg)(‘a’, 0)
	(*do_arg)(‘b’, “cd”)
	(*do_arg)(‘e’, 0)
	(*do_arg)(‘f’, “ghi”)
并且返回：	&argv[4]
命令行：	$ prog –a b –c –d –e –f
control：	“abcdef”
do_args 调用：	(*do_arg)(‘a’, 0)
并且返回：	&argv[2]

bbs.theithome.com

第14章

预处理器

编译一个 C 程序涉及很多步骤。其中第 1 个步骤被称为预处理(preprocessing)阶段。C 预处理器(preprocessor)在源代码编译之前对其进行一些文本性质的操作。它的主要任务包括删除注释、插入被#include 指令包含的文件的内容、定义和替换由#define 指令定义的符号以及确定代码的部分内容是否应该根据一些条件编译指令进行编译。

14.1 预定义符号

表 14.1 总结了由**预处理器**定义的符号。它们的值或者是字符串常量，或者是十进制数字常量。__FILE__和__LINE__在确认调试输出的来源方面很有用处。__DATE__和__TIME__常常用于在被编译的程序中加入版本信息。__STDC__用于那些在 ANSI 环境和非 ANSI 环境都必须进行编译的程序中结合条件编译（本章稍后描述）。

表 14.1　　预处理器符号

符　号	样例值	含义
__FILE__	“name.c”	进行编译的源文件名
__LINE__	25	文件当前行的行号
__DATE__	“Jan 31 1997”	文件被编译的日期
__TIME__	“18:04:30”	文件被编译的时间
__STDC__	1	如果编译器遵循 ANSI C，其值就为 1，否则未定义

14.2 #define

你已经见过#define 指令的一些简单用法，就是为数值命名一个符号。在本节，我将介绍#define 指令的更多用途。首先让我们观察一下它的更为正式的描述。

```
#define  name   stuff
```

有了这条指令以后，每当有符号 name 出现在这条指令后面时，预处理器就会把它替换成 stuff。

bbs.theithome.com

K&R C

早期的 C 编译器要求#出现在每行的起始位置，不过它的后面可以跟一些空白。在 ANSI C 中，这条限制被取消了。

替换文本并不仅限于数值字面值常量。使用#define 指令，你可以把**任何**文本替换到程序中。这里有几个例子：

```
#define reg             register
#define do_forever      for(;;)
#define CASE            break;case
```

第 1 个定义只是为关键字 register 创建了一个简短的别名。这个较短的名字使各个声明更容易通过制表符进行排列。第 2 条声明用一个更具描述性的符号来代替一种用于实现无限循环的 for 语句类型。最后一个#define 定义了一种简短记法，以便在 switch 语句中使用。它自动地把一个 break 放在每个 case 之前，这使得 switch 语句看上去更像其他语言的 case 语句。

如果定义中的 stuff 非常长，它可以分成几行，除了最后一行之外，每行的末尾都要加一个反斜杠，如下面的例子所示：

```
#define DEBUG_PRINT     printf( "File %s line %d:" \
                            " x=%d, y=%d, z=%d", \
                            __FILE__, __LINE__, \
                            x, y, z )
```

我利用了相邻的字符串常量被自动连接为一个字符串这个特性。当你调试一个存在许多涉及一组变量的不同计算过程的程序时，这种类型的声明非常有用。你可以很容易地插入一条调试语句打印出它们的当前值。

```
x *= 2;
y += x;
z = x * y;
DEBUG_PRINT;
```

警告：

这条语句在 DEBUG_PRINT 后面加了一个分号，所以你不应该在宏定义的尾部加上分号。如果你这样做了，结果就会产生两条语句——一条 printf 语句后面再加一条空语句。有些场合只允许出现一条语句，如果放入两条语句就会出现问题，例如：

```
if( ... )
        DEBUG_PRINT;
else
            ...
```

你也可以使用#define 指令把一序列语句插入到程序中。这里有一个完整循环的声明：

```
#define PROCESS_LOOP                    \
        for( i = 0; i < 10; i += 1 ){   \
                sum += i;               \
                if( i > 0 )             \
                        prod *= i;      \
        }
```

提示：

不要滥用这种技巧。如果相同的代码需要出现在程序的几个地方，通常更好的方法是把它实现为一个函数。本章后面我将详细讨论#define 宏和函数之间的优劣。

bbs.theithome.com

14.2.1　宏

#define 机制包括了一个规定，允许把参数替换到文本中，这种实现通常称为**宏**(macro)或定义宏(defined macro)。下面是宏的声明方式：

```
#define  name(parameter-list)   stuff
```

其中，parameter-list（参数列表）是一个由逗号分隔的符号列表，它们可能出现在 stuff 中。参数列表的左括号必须与 name 紧邻。如果两者之间有任何空白存在，参数列表就会被解释为 stuff 的一部分。

当宏被调用时，名字后面是一个由逗号分隔的值的列表，每个值都与宏定义中的一个参数相对应，整个列表用一对括号包围。当参数出现在程序中时，与每个参数对应的实际值都将被替换到 stuff 中。

这里有一个宏，它接受一个参数：

```
#define SQUARE(x)        x * x
```

如果在上述声明之后，你把

```
SQUARE( 5 )
```

置于程序中，预处理器就会用下面这个表达式替换上面的表达式：

```
5 * 5
```

警告：

但是，这个宏存在一个问题。观察下面的代码段：

```
a = 5;
printf("%d\n", SQUARE( a + 1 ) );
```

乍一看，你可能觉得这段代码将打印 36 这个值。事实上，它将打印 11。想知道为什么？请观察被替换的宏文本。参数 x 被文本 a + 1 替换，所以这条语句实际上变成了

```
printf("%d\n", a + 1 * a + 1 );
```

现在问题清楚了：由替换产生的表达式并没有按照预想的次序进行求值。

在宏定义中加上两个括号，这个问题便很轻松地解决了：

```
#define SQUARE(x)        (x) * ( x )
```

在前面那个例子里，预处理器现在将用下面这条语句执行替换，从而产生预期的结果。

```
printf("%d\n", ( a + 1 ) * ( a + 1 ) );
```

这里有另外一个宏定义。

```
#define  DOUBLE(x)        (x) + (x)
```

定义中使用了括号，用于避免前面出现的问题。但是，使用这个宏，可能会出现另外一个不同的错误。下面这段代码将打印出什么值？

```
a = 5;
printf("%d\n", 10 * DOUBLE( a ) );
```

警告：

看上去，它好像将打印 100，但事实上它打印的是 55。再一次，通过观察宏替换产生的文本，我们能够发现问题所在：

bbs.theithome.com

```
printf("%d\n", 10 * ( a ) + ( a ) );
```

乘法运算在宏所定义的加法运算之前执行。这个错误很容易修正：在定义宏时，你只要在整个表达式两边加上一对括号就可以了。

```
#define  DOUBLE(x)        ( (x) + (x) )
```

提示：

所有用于对数值表达式进行求值的宏定义都应该用这种方式加上括号，避免在使用宏时，由于参数中的操作符或邻近的操作符之间不可预料的相互作用。

下面是一对有趣的宏：

```
#define  repeat          do
#define   until(x)       while( ! (x) )
```

这两个宏创建了一种“新”的循环，其工作过程类似于其他语言中的 repeat/until 循环。它按照下面这样的方式使用：

```
repeat {
        statements
} until( i >= 10 );
```

预处理器将用下面的代码进行替换。

```
do {
        statements
} while( ! ( i >= 10 ) );
```

表达式 i>=10 两边的括号用于确保在!操作符执行之前先完成这个表达式的求值。

提示：

创建一套#define 宏，用一种看上去很像其他语言的方式编写 C 程序是完全可能的。在绝大多数情况下，你应该避免这种诱惑，因为这样编写出来的程序使其他 C 程序员很难理解。他们必须时常查阅这些宏的定义以便弄清实际的代码是什么意思。即使每个和这个项目生命期各个阶段相关的人都熟悉那种被模仿的语言，这个技巧仍然可能引起混淆，因为准确地模仿其他语言的各个方面是极为困难的。

14.2.2 #define 替换

在程序中扩展#define 定义符号和宏时，需要涉及几个步骤。

1．在调用宏时，首先对参数进行检查，看看是否包含了任何由#define 定义的符号。如果是，它们首先被替换。

2．替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被它们的值所替代。

3．最后，再次对结果文本进行扫描，看看它是否包含了任何由#define 定义的符号。如果是，就重复上述处理过程。

这样，宏参数和#define 定义可以包含其他#define 定义的符号。但是，宏不可以出现递归。

当预处理器搜索#define 定义的符号时，字符串常量的内容并不进行检查。你如果想把宏参数插入到字符串常量中，可以使用两种技巧。首先，邻近字符串自动连接的特性使我们很容易把一个字符串分成几段，每段实际上都是一个宏参数。这里有一个这种技巧的例子：

bbs.theithome.com

```
#define PRINT(FORMAT,VALUE)                \
        printf( "The value is " FORMAT "\n", VALUE )
...
PRINT( "%d", x + 3 );
```

这种技巧只有当字符串常量作为宏参数给出时才能使用。

第 2 个技巧使用预处理器把一个宏参数转换为一个字符串。#argument 这种结构被预处理器翻译为“argument”。这种翻译可以让你像下面这样编写代码：

```
#define PRINT(FORMAT,VALUE)                \
        printf( "The value of " #VALUE  \
        " is " FORMAT "\n", VALUE )
...
PRINT( "%d", x + 3 );
```

它将产生下面的输出：

```
The value of x + 3 is 25
```

##结构则执行一种不同的任务。它把位于它两边的符号连接成一个符号。作为用途之一，它允许宏定义从分离的文本片段创建标识符。下面这个例子使用这种连接把一个值添加到几个变量之一：

```
#define ADD_TO_SUM( sum_number, value ) \
        sum ## sum_number += value
...
ADD_TO_SUM( 5, 25 );
```

最后一条语句把值 25 加到变量 sum5。注意这种连接必须产生一个合法的标识符。否则，其结果就是未定义的。

14.2.3　宏与函数

宏非常频繁地用于执行简单的计算，比如在两个表达式中寻找其中较大（或较小）的一个：

```
#define MAX( a, b )      ( (a) > (b) ? (a) : (b) )
```

为什么不用函数来完成这个任务呢？有两个原因。首先，用于调用和从函数返回的代码很可能比实际执行这个小型计算工作的代码更大，所以使用宏比使用函数在程序的规模和速度方面都更胜一筹。

但是，更为重要的是，函数的参数必须声明为一种特定的类型，所以它只能在类型合适的表达式上使用。反之，上面这个宏可以用于整型、长整型、单浮点型、双浮点数以及其他任何可以用>操作符比较值大小的类型。换句话说，宏是与类型无关的。

和使用函数相比，使用宏的不利之处在于每次使用宏时，一份宏定义代码的拷贝都将插入到程序中。除非宏非常短，否则使用宏可能会大幅度增加程序的长度。

还有一些任务根本无法用函数实现。让我们仔细观察定义于程序 11.1a 的宏。这个宏的第 2 个参数是一种类型，它无法作为函数参数进行传递。

```
#define MALLOC(n, type) \
         ( (type *)malloc( (n) * sizeof( type ) ) )
```

你现在可以观察一下这个宏确切的工作过程。下面这个例子中的第 1 条语句被预处理器转换为第 2 条语句。

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


bbs.theithome.com

同样，请注意宏定义并没有用一个分号结尾。分号出现在调用这个宏的语句中。

14.2.4 带副作用的宏参数

当宏参数在宏定义中出现的次数超过一次时，如果这个参数具有副作用，那么当你使用这个宏时就可能出现危险，导致不可预料的结果。**副作用**就是在表达式求值时出现的永久性效果。例如，下面这个表达式

```
x + 1
```

可以重复执行几百次，它每次获得的结果都是一样的。这个表达式不具有副作用。但是

```
x++
```

就具有副作用：它增加 x 的值。当这个表达式下一次执行时，它将产生一个不同的结果。MAX 宏可以证明具有副作用的参数所引起的问题。观察下列代码，你认为它将打印出什么？

```
#define MAX( a, b )     ( (a) > (b) ? (a) : (b) )
...
x = 5;
y = 8;
z = MAX( x++, y++ );
printf( "x=%d, y=%d, z=%d\n", x, y, z );
```

这个问题并不轻松。记住第 1 个表达式是一个条件表达式，用于确定执行另两个表达式中的哪一个，剩余的那个表达式将不会执行。其结果是：x=6，y=10，z=9。

和往常一样，只要检查一下用宏替换后产生的代码，这个奇怪的结果就变得一目了然了。

```
z = ( ( x++ ) > ( y++ ) ? ( x++ ) : ( y++ ) );
```

虽然那个较小的值只增值了一次，但那个较大的值却增值了两次——第 1 次是在比较时，第 2 次在执行?符号后面的表达式时出现。

副作用并不仅限于修改变量的值。下面这个表达式

```
getchar()
```

也具有副作用。调用这个函数将“消耗”输入的一个字符，所以该函数的后续调用将得到不同的字符。如果用户的意图并不是想“消耗”输入字符，那么就不能重复调用这个函数。

考虑下面这个宏。

```
#define EVENPARITY( ch )                        \
        ( ( count_one_bits( ch ) & 1 ) ?        \
        ( ch ) | PARITYBIT : ( ch ) )
```

它使用了程序 5.1 的 count_one_bits 函数，该函数返回它的参数的二进制位模式中 1 的个数。这个宏的目的是产生一个具有偶校验[1]的字符。它首先计数字符中位 1 的个数，如果结果是一个奇数，PARITYBIT 值（一个值为 1 的位）与该字符执行 OR 操作，否则该字符就保留不变。但是，当这个宏以下面这种方式使用时，请想象一下会发生什么？

```
ch = EVENPARITY( getchar() );
```

[1] 奇偶校验(parity)是一种错误检测机制。在数据被存储或通过通信线路传送之前，为一个值计算（并添加）一个校验位，使数据的二进制模式中 1 的个数为一个偶数。以后，数据可以通过计算它的位 1 的个数来验证其有效性。如果结果是奇数，那么数据就出现了错误。这个技巧被称为偶校验(even parity)。奇校验(odd parity)的工作原理相同，只是计算并添加校验位之后，数据的二进制位模式中 1 的个数是奇数。

bbs.theithome.com

这条语句看上去很合理：读取一个字符并计算它的校验位。但是，它的结果是失败的，因为它实际上读入了两个字符！

14.2.5　命名约定

#define 宏的行为和真正的函数相比存在一些不同的地方，表 14.2 对此进行了总结。由于这些不同之处，所以让程序员知道一个标识符究竟是一个宏还是一个函数是非常重要的。不幸的是，使用宏的语法和使用函数的语法是完全一样的，所以语言本身并不能帮助你区分这两者。

提示：

为宏定义（对于绝大多数由#define 定义的符号也是如此）采纳一种命名约定是很重要的，上面这种混淆就是促使人们这样做的原因之一。一个常见的约定就是把宏名字全部大写。在下面这条语句中，

```
value = max( a, b );
```

max 究竟是一个宏还是一个函数并不明显。你很可能不得不仔细察看源文件以及它所包含的所有头文件来找出它的真实身份。另一方面，请看下面这条语句

```
value = MAX( a, b );
```

命名约定使 MAX 的身份一清二楚。如果宏使用可能具有副作用的参数时，这个约定尤为重要，因为它可以提醒程序员在使用宏之前先把参数存储到临时变量中。

表 14.2　　宏和函数的不同之处

属　性	#define 宏	函　数
代码长度	每次使用时，宏代码都被插入到程序中。除了非常小的宏之外，程序的长度将大幅度增长	函数代码只出现于一个地方；每次使用这个函数时，都调用那个地方的同一份代码
执行速度	更快	存在函数调用/返回的额外开销
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境里，除非它们加上括号，否则邻近操作符的优先级可能会产生不可预料的结果	函数参数只在函数调用时求值一次，它的结果值传递给函数。表达式的求值结果更容易预测
参数求值	参数每次用于宏定义时，它们都将重新求值。由于多次求值，具有副作用的参数可能会产生不可预料的结果	参数在函数被调用前只求值一次。在函数中多次使用参数并不会导致多种求值过程。参数的副作用并不会造成任何特殊的问题
参数类型	宏与类型无关。只要对参数的操作是合法的，它可以使用于任何参数类型	函数的参数是与类型有关的。如果参数的类型不同，就需要使用不同的函数；即使它们执行的任务是相同的

14.2.6　#undef

这条预处理指令用于移除一个宏定义。

```
#undef  name
```

如果一个现存的名字需要被重新定义，那么它的旧定义首先必须用#undef 移除。

14.2.7　命令行定义

许多 C 编译器提供了一种能力，允许你在命令行中定义符号，用于启动编译过程。当我们根据

bbs.theithome.com

同一个源文件编译一个程序的不同版本时，这个特性是很有用的。例如，假定某个程序声明了一个某种长度的数组。如果某个机器的内存很有限，这个数组必须很小，但在另一个内存充裕的机器上，你可能希望数组能够大一些。如果数组是用类似下面的形式进行声明的，

```
int        array[ARRAY_SIZE];
```

那么，在编译程序时，ARRAY_SIZE 的值可以在命令行中指定。

在 UNIX 编译器中，-D 选项可以完成这项任务。我们可以用两种方式使用这个选项。

```
-Dname
-Dname=stuff
```

第 1 种形式定义了符号 name，它的值为 1。第 2 种形式把该符号的值定义为等号后面的 stuff。用于 MS-DOS 的 Borland C 编译器使用相同的语法提供相同的功能。请查阅你的编译器文档，获取和你的系统有关的信息。

回到我们的例子，在 UNIX 系统中，编译这个程序的命令行可能是下面这个样子：

```
cc -DARRAY_SIZE=100 prog.c
```

这个例子说明了在程序中使用诸如数组长度这样的参数化量的另一个好处。如果在数组的声明中，它的长度以字面值常量的形式给出，或者如果需要在循环内部用一个字面值常量作为限量访问数组，这种技巧就无法使用。在你需要引用数组长度的地方，都必须使用符号常量。

提供符号命令行定义的编译器通常也提供在命令行中去除符号的定义。在 UNIX 编译器上，-U 选项用于执行这项任务。指定-Uname 将导致程序中符号 name 的初始定义被忽略。当它与条件编译结合使用时，这个特性是很有用的。

14.3 条件编译

在编译一个程序时，如果我们可以选择某条语句或某组语句进行翻译或者被忽略，常常会显得很方便。只用于调试程序的语句就是一个明显的例子。它们不应该出现在程序的产品版本中，但是你可能并不想把这些语句从源代码中物理删除，因为如果需要一些维护性修改时，你可能需要重新调试这个程序，还需要这些语句。

条件编译(conditional compilation)就是用于实现这个目的。使用条件编译，你可以选择代码的一部分是被正常编译还是完全忽略。用于支持条件编译的基本结构是#if 指令和与其匹配的#endif 指令。下面显示了它最简单的语法形式。

```
#if  constant-expression
          statements
#endif
```

其中，constant-expression（常量表达式）由预处理器进行求值。如果它的值是非零值（真），那么 statements 部分就被正常编译，否则预处理器就安静地删除它们。

所谓常量表达式，就是说它或者是字面值常量，或者是一个由#define 定义的符号。如果变量在执行期之前无法获得它们的值，那么它们如果出现在常量表达式中就是非法的，因为它们的值在编译时是不可预测的。

例如，将你所有的调试代码都以下面这种形式出现：

bbs.theithome.com

```
#if DEBUG
        printf( "x=%d, y=%d\n", x, y );
#endif
```

这样，不管我们是想编译还是忽略这个代码都很容易办到。如果想要编译它，只要使用

```
#define DEBUG  1
```

这个符号定义就可以了。如果想要忽略它，只要把这个符号定义为 0 就可以了。无论哪种情况，这段代码都可以保留在源文件中。

条件编译的另一个用途是在编译时选择不同的代码部分。为了支持这个功能，#if 指令还具有可选的#elif 和#else 子句。完整的语法如下所示：

```
#if constant-expression
        statements
#elif constant-expression
        other statements ...
#else
        other statements
#endif
```

#elif 子句出现的次数可以不限。每个 constant-expression（常量表达式）只有当前面所有常量表达式的值都为假时才会被编译。#else 子句中的语句只有当前面所有的常量表达式的值都为假时才会被编译，在其他情况下它都会被忽略。

K&R C

最初的 K&R C 并不具有#elif 指令。但是，在这类编译器中，可以使用嵌套的指令来获得相同的效果。

下面这个例子取自一个以几个不同版本进行销售的程序。每个版本都有一组不同的选项特性。编写这个代码的困难在于如何让它产生不同的版本。你必须避免为每个版本编写一组不同的源文件，这个代价太大了！因为各组源文件的绝大多数代码都是一样的，而且维护这个程序将成为一个恶梦。幸运的是，条件编译可以解决这个问题。

```
if( feature_selected == FEATURE1 )
#if     FEATURE1_ENABLED_FULLY
        feature1_function( arguments );
#elif   FEATURE1_ENABLED_PARTIALLY
        feature1_partial_function( arguments );
#else
        printf( "To use this feature, send $39.95;"
            " allow ten weeks for delivery.\n" );
#endif
```

这样，我们就只需要编写一组源文件。当它们被编译时，每个当前版本所需的特性（或特性层次）符号被定义为 1，其余的符号被定义为 0。

14.3.1　是否被定义

测试一个符号是否已被定义也是可能的。在条件编译中完成这个任务往往更为方便，因为程序如果并不需要控制编译的符号所控制的特性，它就不需要被定义。这个测试可以通过下列任何一种方式进行：

bbs.theithome.com

```
#if     defined(symbol)
#ifdef  symbol

#if     !defined(symbol)
#ifndef symbol
```

每对定义的两条语句是等价的，但#if 形式功能更强。因为常量表达式可能包含额外的条件，如下面所示：

```
#if X > 0 || defined( ABC ) && defined( BCD )
```

K&R C

有些 K&R C 编译器可能并未包含所有这些功能，这取决于它们的年代如何久远。

14.3.2 嵌套指令

前面提到的这些指令可以嵌套于另一个指令内部，如下面的代码段所示：

```
#if     defined( OS_UNIX )
        #ifdef  OPTION1
                unix_version_of_option1();
        #endif
        #ifdef  OPTION2
                unix_version_of_option2();
        #endif
#elif   defined( OS_MSDOS )
        #ifdef  OPTION2
                msdos_version_of_option2();
        #endif
#endif
```

在这个例子中，操作系统的选择将决定不同的选项可以使用哪些方案。这个例子同时说明了预处理器指令可以在它们前面添加空白，形成缩进，从而提高可读性。

为了帮助读者记住复杂的嵌套指令，为每个#endif 加上一个注释标签是很有帮助的，标签的内容就是#if（或#ifdef）后面的那个表达式。当#if（或#ifdef）和#endif 之间的代码块非常长时，这种做法尤为有用。例如：

```
#ifdef  OPTION1
        lengthy code for option1;
#else
        lengthy code for alternative;
#endif  /* OPTION1 */
```

有些编译器允许一个符号出现于#endif 指令中，它的作用和上面这种标签类似。不过这个符号对实际代码不会产生任何作用。标准并没有提及这种做法是否合法，所以更安全的做法还是使用注释。

14.4 文件包含

你已经看到过，#include 指令使另一个文件的内容被编译，就像它实际出现于#include 指令出现的位置一样。这种替换执行的方式很简单：预处理器删除这条指令，并用包含文件的内容取而代之。这样，一个头文件如果被包含到 10 个源文件中，它实际上被编译了 10 次。

bbs.theithome.com

提示：

这个事实意味着使用#include 文件涉及一些开销，但基于两个十分充分的理由，你不必担心这种开销。首先，这种额外开销实际上并不大。如果两个源文件都需要同一组声明，把这些声明复制到每个源文件中所花费的编译时间跟把这些声明放入一个头文件，然后再用#include 指令把它包含于每个源文件所花费的编译时间相差无几。同时，这个开销只是在程序被编译时才存在，所以对运行时效率并无影响。但是，更为重要的是，把这些声明放于一个头文件中具有重要的意义。如果其他源文件还需要这些声明，你就不必把这些拷贝逐一复制到这些源文件中，因此它们的维护任务也变得简单了。

提示：

当头文件被包含时，位于头文件内的所有内容都要被编译。这个事实意味着每个头文件只应该包含一组函数或数据的声明。和把一个程序需要的所有声明都放入一个巨大的头文件相比，使用几个头文件，每个头文件包含用于某个特定函数或模块的声明的做法更好一些。

提示：

程序设计和模块化的原则也支持这种方法。只把必要的声明包含于一个文件中这种做法更好一些，这样文件中的语句就不会意外地访问应该属于私有的函数或变量。同时，这种方法使你不需要在数百行不相关的代码中寻找你所需要的那组声明，因此它们的维护工作也更容易一些。

14.4.1　函数库文件包含

编译器支持两种不同类型的#include 文件包含：函数库文件和本地文件。事实上，它们之间的区别很小。

函数库头文件包含使用下面的语法。

```
#include <filename>
```

对于 filename，并不存在任何限制，不过根据约定，标准库文件以一个.h 后缀[1]结尾。

编译器通过观察由编译器定义的“一系列标准位置”查找函数库头文件。你所使用的编译器的文档应该说明这些标准位置是什么，以及你怎样修改它们或者在列表中添加其他位置。例如，在典型情况下，运行于 UNIX 系统上的 C 编译器在/user/include 目录查找函数库头文件。这种编译器有一个命令行选项，允许你把其他目录添加到这个列表中，这样你就可以创建你自己的头文件函数库。同样，请查阅你使用的编译器的文档，看看你的系统在这方面是怎样规定的。

14.4.2　本地文件包含

下面是#include 指令的另一种形式。

```
#include "filename"
```

标准允许编译器自行决定是否把本地形式的#include 和函数库形式的#include 区别对待。你可以对本地头文件先使用一种特殊的处理方式，如果失败，编译器再按照函数库头文件的处理方式对它们进行处理。处理本地头文件的一种常见策略就是在源文件所在的当前目录进行查找，如果该头文

[1] 从技术上说，函数库头文件并不需要以文件的形式存储，但对于程序员而言，这并非显而易见。

bbs.theithome.com

件并未找到，编译器就像查找函数库头文件一样在标准位置查找本地头文件。

你可以在所有的#include 语句中使用双引号而不是尖括号。但是，使用这种方法，有些编译器在查找函数库头文件时可能会浪费少许时间。对函数库头文件使用尖括号的另一个较好的理由是它能给读者提供一些信息。使用尖括号，下面这条语句

```
#include <errno.h>
```

显然引用的是一个函数库头文件。如果使用另一种形式，

```
#include "errno.h"
```

你就无法弄清楚这个和上面相同的文件到底是一个函数库头文件还是一个本地头文件。要想弄明白它究竟是哪种类型？唯一的方法是检查执行编译过程的目录。

UNIX 系统和 Borland C 编译器所支持的一种变体形式是使用绝对路径名(absolute pathname)，它不仅指定文件的名字，而且指定了文件的位置。UNIX 系统中的绝对路径名以一个斜杠开头，如下所示：

```
/home/fred/C/my_proj/declaration2.h
```

在 MS-DOS 系统中，它所使用的是反斜杠而不是斜杠。如果一个绝对路径名出现在任何一种形式的#include，那么正常的目录查找就被跳过，因为这个路径名指定了头文件的位置。

14.4.3 嵌套文件包含

在一个将被其他文件包含的文件中使用#include 指令是可能的。例如，考虑一组读取输入并且执行各种输入有效性验证任务的函数。函数返回的是被验证后的数据，如果到达文件尾时就返回常量 EOF。

这些函数的原型将被放入一个头文件中，并且用#include 指令包含到需要使用这些函数的源文件中。但是，每个使用 I/O 函数的文件必须同时包含 stdio.h 以获得 EOF 的声明。因此，包含这些函数原型的头文件也可能包含一条：

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

包含了这个头文件就自动引入了标准 I/O 声明。

标准要求编译器必须支持至少 8 层的头文件嵌套，但它并没有限定嵌套深度的最大值。事实上，我们并没有很好的理由让#include 指令的嵌套深度超过一层或两层。

提示：

嵌套#include 文件的一个不利之处在于它使得我们很难判断源文件之间的真正依赖关系。有些程序，如 UNIX 的 make 实用工具，必须知道这些依赖关系以便决定当某些文件被修改之后，哪些文件需要重新编译。

嵌套#include 文件的另一个不利之处在于一个头文件可能会被多次包含。为了说明这种错误，考虑下面的代码：

```
#include "x.h"
#include "x.h"
```

显然，这里文件 x.h 被包含了两次。没有人会故意编写这样的代码。但下面的代码

```
#include "a.h"
#include "b.h"
```


bbs.theithome.com

看上去没什么问题。如果 a.h 和 b.h 都包含一个嵌套的#include 文件 x.h，那么 x.h 在此处也同样出现了两次，只不过它的形式不是那么明显而已。

多重包含在绝大多数情况下出现于大型程序中，它往往需要使用很多头文件，因此要发现这种情况并不容易。要解决这个问题，我们可以使用条件编译。如果所有的头文件都像下面这样编写：

```
#ifndef _HEADERNAME_H
#define _HEADERNAME_H 1
/*
** All the stuff that you want in the header file
*/
#endif
```

那么，多重包含的危险就被消除了。当头文件第 1 次被包含时，它被正常处理，符号_HEADERNAME_H 被定义为 1。如果头文件被再次包含，通过条件编译，它的所有内容被忽略。符号_HEADERNAME_H 按照被包含文件的文件名进行取名，以避免由于其他头文件使用相同的符号而引起的冲突。

注意前一个例子中的定义也可以写作

```
#define _HEADERNAME_H
```

它的效果完全一样。尽管现在它的值是一个空字符串而不是“1”，但这个符号仍然被定义。

但是，你必须记住预处理器仍将读入整个头文件，即使这个文件的所有内容将被忽略。由于这种处理将拖慢编译速度，所以如果可能，应避免出现多重包含，不管它是否由于嵌套的#include 文件导致。

14.5 其他指令

预处理器还支持其他一些指令。首先，当程序编译之后，#error 指令允许你生成错误信息。下面是它的语法：

```
#error  text of error message
```

下面的代码段显示了你可以如何使用这个指令。

```
#if     defined( OPTION_A )
        stuff needed for option A
#elif   defined( OPTION_B )
        stuff needed for option B
#elif   defined( OPTION_C )
        stuff needed for option C
#else
        #error No option selected!
#endif
```

另外还用一种用途较小的#line 指令，它的形式如下：

```
#line     number  "string"
```

它通知预处理器 number 是下一行输入的行号。如果给出了可选部分“string”，预处理器就把它作为当前文件的名字。值得注意的是，这条指令将修改__LINE__符号的值，如果加上可选部分，它还将修改__FILE__符号的值。

这条指令最常用于把其他语言的代码转换为 C 代码的程序。C 编译器产生的错误信息可以引用源文件而不是翻译程序产生的 C 中间源文件的文件名和行号。

#progma 指令是另一种机制，用于支持因编译器而异的特性。它的语法也是因编译器而异。有

bbs.theithome.com

些环境可能提供一些#progma 指令，允许一些编译选项或其他任何方式无法实现的一些处理方式。例如，有些编译器使用#progma 指令在编译过程中打开或关闭清单显示，或者把汇编代码插入到 C 程序中。从本质上说，#progma 是不可移植的。预处理器将忽略它不认识的#progma 指令，两个不同的编译器可能以两种不同的方式解释同一条#progma 指令。

最后，无效指令(null directive)就是一个#符号开头，但后面不跟任何内容的一行。这类指令只是被预处理器简单地删除。下面例子中的无效指令通过把#include 与周围的代码分隔开来，凸显它的存在。

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

我们也可以通过插入空行取得相同的效果。

14.6 总结

编译一个 C 程序的第 1 个步骤就是对它进行预处理。预处理器共支持 5 个符号，它们在表 14.1 中描述。

#define 指令把一个符号名与一个任意的字符序列联系在一起。例如，这些字符可能是一个字面值常量、表达式或者程序语句。这个序列到该行的末尾结束。如果该序列较长，可以把它分开数行，但在最后一行之外的每一行末尾加一个反斜杠。宏就是一个被定义的序列，它的参数值将被替换。当一个宏被调用时，它的每个参数都被一个具体的值替换。为了防止可能出现于表达式中的与宏有关的错误，在宏完整定义的两边应该加上括号。同样，在宏定义中每个参数的两边也要加上括号。#define 指令可以用于“重写”C 语言，使它看上去像是其他语言。

#argument 结构由预处理器转换为字符串常量“argument”。##操作符用于把它两边的文本粘贴成同一个标识符。

有些任务既可以用宏也可以用函数实现。但是，宏与类型无关，这是一个优点。宏的执行速度快于函数，因为它不存在函数调用/返回的开销。但是，使用宏通常会增加程序的长度，但函数却不会。同样，具有副作用的参数可能在宏的使用过程中产生不可预料的结果，而函数参数的行为更容易预测。由于这些区别，使用一种命名约定，让程序员很容易地判断一个标识符是函数还是宏是非常重要的。

在许多编译器中，符号可以从命令行定义。#undef 指令将导致一个名字的原来定义被忽略。

使用条件编译，你可以从一组单一的源文件创建程序的不同版本。#if 指令根据编译时测试的结果，包含或忽略一个序列的代码。当同时使用#elif 和#else 指令时，你可以从几个序列的代码中选择其中之一进行编译。除了测试常量表达式之外，这些指令还可以测试某个符号是否已被定义。#ifdef 和#ifndef 指令也可以执行这个任务。

#include 指令用于实现文件包含。它具有两种形式。如果文件名位于一对尖括号中，编译器将在由编译器定义的标准位置查找这个文件。这种形式通常用于包含函数库头文件时。另一种形式，文件名出现在一对双引号内。不同的编译器可以用不同的方式处理这种形式。但是，如果用于处理本地头文件的任何特殊处理方法无法找到这个头文件，那么编译器接下来就使用标准查找过程来寻找它。这种形式通常用于包含你自己编写的头文件。文件包含可以嵌套，但很少需要进行超过一层或两层的文件包含嵌套。嵌套的包含文件将会增加多次包含同一个文件的危险，而且使我们更难以确定某个特定的源文件依赖的究竟是哪个头文件。

bbs.theithome.com

#error 指令在编译时产生一条错误信息，信息中包含的是你所选择的文本。#line 指令允许你告诉编译器下一行输入的行号，如果它加上了可选内容，它还将告诉编译器输入源文件的名字。因编译器而异的#progma 指令允许编译器提供不标准的处理过程，比如向一个函数插入内联的汇编代码。

14.7 警告的总结

1. 不要在一个宏定义的末尾加上分号，使其成为一条完整的语句。
2. 在宏定义中使用参数，但忘了在它们周围加上括号。
3. 忘了在整个宏定义的两边加上括号。

14.8 编程提示的总结

1. 避免用#define 指令定义可以用函数实现的很长序列的代码。
2. 在那些对表达式求值的宏中，每个宏参数出现的地方都应该加上括号，并且在整个宏定义的两边也加上括号。
3. 避免使用#define 宏创建一种新语言。
4. 采用命名约定，使程序员很容易看出某个标识符是否为#define 宏。
5. 只要合适就应该使用文件包含，不必担心它的额外开销。
6. 头文件只应该包含一组函数和（或）数据的声明。
7. 把不同集合的声明分离到不同的头文件中可以改善信息隐藏。
8. 嵌套的#include 文件使我们很难判断源文件之间的依赖关系。

14.9 问题

1. 预处理器定义了 5 个符号，给出了进行编译的文件名、文件的当前行号、当前日期和时间以及编译器是否为 ANSI C 编译器。为每个符号举出一种可能的用途。
2. 说出两个使用#define 定义的名字替代字面值常量的优点。
3. 编写一个用于调试的宏，打印出任意的表达式。它被调用时应该接受两个参数。第 1 个是 printf 格式码，第 2 个是需要打印的表达式。
4. 下面的程序将打印出什么？在展开#define 内容时必须非常小心！

```
#define MAX(a,b)          (a)>(b)?(a):(b)
#define SQUARE(x)         x*x
#define DOUBLE(x)         x+x

main()
{
        int     x, y, z;

        y = 2; z = 3;
        x = MAX(y,z);
/* a */ printf( "%d %d %d\n", x, y, z );

        y = 2; z = 3;
        x = MAX(++y,++z);
```


bbs.theithome.com

```
/* b */ printf( "%d %d %d\n", x, y, z );

        x = 2;
        y = SQUARE(x);
        z = SQUARE(x+6);
/* c */ printf( "%d %d %d\n", x, y, z );

        x = 2;
        y = 3;
        z = MAX(5*DOUBLE(x),++y);
/* d */ printf( "%d %d %d\n", x, y, z );
}
```

5. putchar 函数定义于文件 stdio.h 中，尽管它的内容比较长，但它是作为一个宏实现。你认为它为什么以这种方式定义？

6. 下列代码是否有错？如果有，错在何处？

```
/*
** Process all the values in the array.
*/
result = 0;
i = 0;
while( i < SIZE ){
        result += process( value[ i++ ] );
}
```

7. 下列代码是否有错？如果有，错在何处？

```
#define SUM( value )    ( ( value ) + ( value ) )
int     array[SIZE];
...
/*
** Sum all the values in the array.
*/
sum = 0;
i = 0;
while( i < SIZE )
        sum += SUM( array[ i++ ] );
```

8. 下列代码是否有错？如果有，错在何处？

```
在文件 header1.h 中:
#ifndef  _HEADER1_H
#define  _HEADER1_H
#include  "header2.h"
    其他声明
#endif

在文件 header2.h 中:
#ifndef  _HEADER2_H
#define  _HEADER2_H
#include  "header1.h"
    其他声明
#endif
```

9. 在一次提高程序可读性的尝试中，一位程序员编写了下面的声明。

```
#if sizeof( int ) == 2
        typedef long int32;
#else
        typedef int int32;
#endif
```


bbs.theithome.com

这段代码是否有错？如果有，错在何处？

14.10 编程练习

★★ 1. 你所在的公司向市场投放了一个程序，用于处理金融交易并打印它们的报表。为了扩展潜在的市场，这个程序以几个不同的版本进行销售，每个版本都有不同选项的组合——选项越多，价格就越高。你的任务是为一个打印函数实现代码，这样它可以很容易地进行编译，产生程序的不同版本。

你的函数名为 print_ledger。它接受一个 int 参数，没有返回值。它应该调用一个或多个下面的函数，具体依取决于该函数被编译时哪个符号（如果有的话）被定义。

如果这个符号被定义...	那么你就调用这个函数
OPTION_LONG	print_ledger_long
OPTION_DETAILED	print_ledger_detailed
(无)	print_ledger_default

每个函数都接受单个 int 参数。把你收到的值传递给你应该调用的函数。

★★ 2. 编写一个函数，返回一个值，提示运行这个函数的计算机的类型。这个函数将由一个能够运行于许多不同计算机的程序使用。

我们将使用条件编译来实现这个魔术。你的函数应该叫作 cpu_type，它不接受任何参数。当你的函数被编译时，在下面表中“已定义”列中的符号之一可能会被定义。你的函数应该从“返回值”列中返回对应的符号。如果左边列中的所有符号均未定义，那么函数就返回 CPU_UNKNOWN 这个值。如果超过一个的符号被定义，那么其结果就是未定义的。

定义符号	返回值
VAX	CPU_VAX
M68000	CPU_68000
M68020	CPU_68020
I80386	CPU_80386
X6809	CPU_6809
X6502	CPU_6502
U3B2	CPU_3B2
(无)	CPU_UNKNOWN

“返回值”列中的符号将被#define 定义为各种不同的整型值，其内容位于头文件 cpu_type.h 中。

bbs.theithome.com

bbs.theithome.com

第15章

输入/输出函数

ANSI C 和早期 C 相比的最大优点之一就是它在规范里所包含的函数库。每个 ANSI 编译器必须支持一组规定的函数，并具备规范所要求的接口，而且按照规定的行为工作。这种情况较之早期的 C 是一个巨大的改进。以前，不同的编译器可以通过修改或扩展普通函数库的功能来“改善”它们。这些改变可能在那个特定的作出修改的系统上很有用，但它们却限制了可移植性，因为依赖这些修改的代码在其他缺乏这些修改（或者具有不同修改）的编译器上将会失败。

ANSI 编译器并未被禁止在它们的函数库的基础上增加其他函数。但是，标准函数必须根据标准所定义的方式执行。如果你关心可移植性，只要避免使用任何非标准函数就可以了。

本章讨论 ANSI C 的输入和输出（I/O）函数。但是，我们首先学习两个非常有用的函数，它们用于报告错误以及对错误作出反应。

15.1 错误报告

perror 函数以一种简单、统一的方式报告错误。ANSI C 函数库的许多函数调用操作系统来完成某些任务，I/O 函数尤其如此。任何时候，当操作系统根据要求执行一些任务的时候，都存在失败的可能。例如，如果一个程序试图从一个并不存在的磁盘文件读取数据，操作系统除了提示发生了错误之外就没什么好做的了。标准库函数在一个外部整型变量 errno（在 errno.h 中定义）中保存错误代码之后把这个信息传递给用户程序，提示操作失败的准确原因。

perror 函数简化向用户报告这些特定错误的过程。它的原型定义于 stdio.h，如下所示：

```
void perror( char const *message );
```

如果 message 不是 NULL 并且指向一个非空的字符串，perror 函数就打印出这个字符串，后面跟一个分号和一个空格，然后打印出一条用于解释 errno 当前错误代码的信息。

提示：

perrno 最大的优点就是它容易使用。良好的编程实践要求任何可能产生错误的操作都应该在执行之后进行检查，确定它是否成功执行。即使是那些十拿九稳不会失败的操作也应该进行检查，因为它们迟早可能失败。这种检查需要稍许额外的工作，但与你可能付出的大量调试时间相比，它们

bbs.theithome.com

还是非常值得的。perror 将在本章许多地方以例子的方式进行说明。

注意，只有当一个库函数失败时，errno 才会被设置。当函数成功运行时，errno 的值不会被修改。这意味着我们不能通过测试 errno 的值来判断是否有错误发生。反之，只有当被调用的函数提示有错误发生时检查 errno 的值才有意义。

15.2 终止执行

另一个有用的函数是 exit，它用于终止一个程序的执行。它的原型定义于 stdlib.h，如下所示：

```
void  exit( int status );
```

status 参数返回给操作系统，用于提示程序是否正常完成。这个值和 main 函数返回的整型状态值相同。预定义符号 EXIT_SUCCESS 和 EXIT_FAILURE 分别提示程序的终止是成功还是失败。虽然程序也可以使用其他的值，但它们的具体含义将取决于编译器。

当程序发现错误情况使它无法继续执行下去时，这个函数尤其有用。你经常会在调用 perrno 之后再调用 exit 终止程序。尽管终止程序并非处理所有错误的正确方法，但和一个注定失败的程序继续执行以后再失败相比，这种做法更好一些。

注意这个函数没有返回值。当 exit 函数结束时，程序已经消失，所以它无处可返。

15.3 标准 I/O 函数库

K&R C 最早的编译器的函数库在支持输入和输出方面功能甚弱。其结果是，程序员如果需要使用比函数库所提供的 I/O 更为复杂的功能时，他不得不自己实现。

有了标准 I/O 函数库(Standard I/O Library)之后，这种情况得到了极大的改观。标准 I/O 函数库具有一组 I/O 函数，实现了在原先的 I/O 库基础上许多程序员自行添加实现的额外功能。这个函数库对现存的函数进行了扩展，例如为 printf 创建了不同的版本，可以用于各种不同的场合。函数库同时引进了缓冲 I/O 的概念，提高了绝大多数程序的效率。

这个函数库存在两个主要的缺陷。首先，它是在某台特定类型的机器上实现的，并没有对其他具有不同特性的机器作过多考虑。这就可能出现一种情况，就是在某台机器上运行良好的代码在另一台机器上无法运行，原因仅仅是两台机器之间的架构不同。第 2 个缺陷与第 1 个缺陷有直接有关系。当设计者发现上述问题后，他们试图通过修改库函数进行修正。但是，只要他们这样做了，这个函数库就不再“标准”，程序的可移植性就会降低。

ANSI C 函数库中的 I/O 函数是旧式标准 I/O 库函数的直接后代，只是这些 ANSI 版函数作了一些改进。在设计 ANSI 函数库时，可移植性和完整性是两个关键的考虑内容。但是，与现有程序的向后兼容性也不得不予以考虑。ANSI 版函数和它们的祖先之间的绝大多数区别就是那些在可移植性和功能性方面的改进。

对可移植性最后再说一句：这些函数是对原来的函数进行诸多完善之后的结果，但是它们仍可能进一步改进，使它们变得更完美。ANSI C 的一个主要优点就是这些修改将通过增加**不同**函数的方式实现，而不是通过对现存函数进行修改来实现。因此，程序的可移植性不会受到影响。

bbs.theithome.com

15.4 ANSI I/O 概念

头文件 stdio.h 包含了与 ANSI 函数库的 I/O 部分有关的声明。它的名字来源于旧式的标准 I/O 函数库。尽管不包含这个头文件也可以使用某些 I/O 函数，但绝大多数 I/O 函数在使用前都需要包含这个头文件。

15.4.1 流

当前的计算机具有大量不同的设备，很多都与 I/O 操作有关。CD-ROM 驱动器、软盘和硬盘驱动器、网络连接、通信端口和视频适配器就是这类很常见的设备。每种设备具有不同的特性和操作协议。操作系统负责这些不同设备的通信细节，并向程序员提供一个更为简单和统一的 I/O 接口。

ANSI C 进一步对 I/O 的概念进行了抽象。就 C 程序而言，所有的 I/O 操作只是简单地从程序移进或移出字节的事情。因此，毫不惊奇的是，这种字节流便被称为流(stream)。程序只需要关心创建正确的输出字节数据，以及正确地解释从输入读取的字节数据。特定 I/O 设备的细节对程序员是隐藏。

绝大多数流是完全缓冲的(fully buffered)，这意味着“读取”和“写入”实际上是从一块被称为缓冲区(buffer)的内存区域来回复制数据。从内存中来回复制数据是非常快速的。用于输出流的缓冲区只有当它写满时才会被刷新（flush，物理写入）到设备或文件中。一次性把写满的缓冲区写入和逐片把程序产生的输出分别写入相比效率更高。类似，输入缓冲区当它为空时通过从设备或文件读取下一块较大的输入，重新填充缓冲区。

使用标准输入和输出时，这种缓冲可能会引起混淆。所以，只有当操作系统可以断定它们与交互设备并无联系时才会进行完全缓冲。否则，它们的缓冲状态将因编译器而异。一个常见（但并不普遍）的策略是把标准输出和标准输入联系在一起，就是当请求输入时同时刷新输出缓冲区。这样，在用户必须进行输入之前，提示用户进行输入的信息和以前写入到输出缓冲区中的内容将出现在屏幕上。

警告：

尽管这种缓冲通常是我们所需的，但当你调试程序时仍可能引起混淆。一个常见的调试策略是把一些 printf 函数的调用散布于程序中，确定错误出现的具体位置。但是，这些函数调用的输出结果被写入到缓冲区中，并不立即显示于屏幕上。事实上，如果程序失败，缓冲输出可能不会被实际写入，这就可能使程序员得到关于错误出现位置的不正确结论。这个问题的解决方法就是在每个用于调试的 printf 函数之后立即调用 fflush，如下所示：

```
printf("something or other" );
fflush( stdout );
```

fflush（本章后面将有更多描述）迫使缓冲区的数据立即写入，不管它是否已满。

一、文本流

流分为两种类型，文本(text)流和二进制(binary)流。文本流的有些特性在不同的系统中可能不同。其中之一就是文本行的最大长度。标准规定至少允许 254 个字符。另一个可能不同的特性是文本行的结束方式。例如，在 MS-DOS 系统中，文本文件约定以一个回车符和一个换行符（或称为行反馈

bbs.theithome.com

符）结尾。但是，UNIX 系统只使用一个换行符结尾。

提示：

标准把文本行定义为零个或多个字符，后面跟一个表示结束的换行符。对于那些文本行的外在表现形式与这个定义不同的系统上，库函数负责外部形式和内部形式之间的翻译。例如，在 MS-DOS 系统中，在输出时，文本中的换行符被写成一对回车/换行符。在输入时，文本中的回车符被丢弃。这种不必考虑文本的外部形式而操纵文本的能力简化了可移植程序的创建。

二、二进制流

另一方面，二进制流中的字节将完全根据程序编写它们的形式写入到文件或设备中，而且完全根据它们从文件或设备读取的形式读入到程序中。它们并未作任何改变。这种类型的流适用于非文本数据，但是如果你不希望 I/O 函数修改文本文件的行末字符，也可以把它用于文本文件。

15.4.2 文件

stdio.h 所包含的声明之一就是 FILE 结构。请不要把它和存储于磁盘上的数据文件相混淆。FILE 是一个数据结构，用于访问一个流。如果你同时激活了几个流，每个流都有一个相应的 FILE 与它关联。为了在流上执行一些操作，你调用一些合适的函数，并向它们传递一个与这个流关联的 FILE 参数。

对于每个 ANSI C 程序，运行时系统必须提供至少三个流——标准输入(standard input)、标准输出(standard output)和标准错误(standard error)。这些流的名字分别为 stdin、stdout 和 stderr，它们都是一个指向 FILE 结构的指针。标准输入是缺省情况下输入的来源，标准输出是缺省的输出设置。具体的缺省值因编译器而异，通常标准输入为键盘设备，标准输出为终端或屏幕。

许多操作系统允许用户在程序执行时修改缺省的标准输入和输出设备。例如，MS-DOS 和 UNIX 系统都支持用下面这种方法进行输入/输出重定向：

```
$ program < data > answer
```

当这个程序执行时，它将从文件 data 而不是键盘作为标准输入进行读取，它将把标准输出写入到文件 answer 而不是屏幕上。请查阅你的系统文档中有关 I/O 重定向的细节。

标准错误就是错误信息写入的地方。perror 函数把它的输出也写到这个地方。在许多系统中，标准输出和标准错误在缺省情况下是相同的。但是，为错误信息准备一个不同的流意味着，即使标准输出重定向到其他地方，错误信息仍将出现在屏幕或其他缺省的输出设备上。

15.4.3 标准 I/O 常量

在 stdio.h 中定义了数量众多的与输入和输出有关的常量。你已经见过的 EOF 是许多函数的返回值，它提示到达了文件尾。EOF 所选择的实际值比一个字符要多几位，这是为了避免二进制值被错误地解释为 EOF。

一个程序同时最多能够打开多少个文件呢？它和编译器有关，但可以保证你能够同时打开至少 FOPEN_MAX 个文件。这个常量包括了三个标准流，它的值至少是 8。

常量 FILENAME_MAX 是一个整型值，用于提示一个字符数组应该多大以便容纳编译器所支持的最长合法文件名。如果对文件名的长度没有一个实际的限制，那个这个常量的值就是文件名的推荐最大长度。其余的一些常量将在本章剩余部分和使用它们的函数一起描述。

bbs.theithome.com

15.5 流 I/O 总览

标准库函数使我们在 C 程序中执行与文件相关的 I/O 任务非常方便。下面是关于文件 I/O 的一般概况。

1．程序为必须同时处于活动状态的每个文件声明一个指针变量，其类型为 FILE *。这个指针指向这个 FILE 结构，当它处于活动状态时由流使用。

2．流通过调用 fopen 函数打开。为了打开一个流，你必须指定需要访问的文件或设备以及它们的访问方式（例如，读、写或者既读又写）。fopen 和操作系统验证文件或设备确实存在（在有些操作系统中，还验证你是否允许执行你所指定的访问方式）并初始化 FILE 结构。

3．然后，根据需要对该文件进行读取或写入。

4．最后，调用 fclose 函数关闭流。关闭一个流可以防止与它相关联的文件被再次访问，保证任何存储于缓冲区的数据被正确地写到文件中，并且释放 FILE 结构使它可以用于另外的文件。

标准流的 I/O 更为简单，因为它们并不需要打开或关闭。

I/O 函数以三种基本的形式处理数据：单个字符、文本行和二进制数据。对于每种形式，都有一组特定的函数对它们进行处理。表 15.1 列出了用于每种 I/O 形式的函数或函数家族。函数家族在表中以斜体表示，它指一组函数中的每个都执行相同的基本任务，只是方式稍有不同。这些函数的区别在于获得输入的来源或输出写入的地方不同。这些变种用于执行下面的任务：

表 15.1　　执行字符、文本行和二进制 I/O 的函数

函数名或函数家族名

数据类型	输　入	输　出	描　述
字符	*getchar*	*putchar*	读取（写入）单个字符
文本行	*gets* *scanf*	*puts* *printf*	文本行未格式化的输入（输出） 格式化的输入（输出）
二进制数据	fread	fwrite	读取（写入）二进制数据

1．只用于 stdin 或 stdout。

2．随作为参数的流使用。

3．使用内存中的字符串而不是流。

需要一个流参数的函数将接受 stdin 或 stdout 作为它的参数。有些函数家族并不具备用于字符串的变种函数，因为使用其他语句或函数来实现相同的结果更为容易。表 15.2 列出了每个家族的函数。各个函数将在本章的后面详细描述。

表 15.2　　输入/输出函数家族

家 族 名	目　的	可用于所有的流	只用于 stdin 和 stdout	内存中的字符串
getchar	字符输入	fgetc, getc	getchar	①
putchar	字符输出	fputc, putc	putchar	①
gets	文本行输入	fgets	gets	②
puts	文本行输出	fputs	puts	②
scanf	格式化输入	fscanf	scanf	sscanf
printf	格式化输出	fprintf	printf	sprintf

① 对指针使用下标引用或间接访问操作从内存获得一个字符（或向内存写入一个字符）。

② 使用 strcpy 函数从内存读取文本行（或向内存写入文本行）。

bbs.theithome.com

15.6 打开流

fopen 函数打开一个特定的文件，并把一个流和这个文件相关联。它的原型如下所示：

```
FILE *fopen( char const *name, char const *mode );
```

两个参数都是字符串。name 是你希望打开的文件或设备的名字。创建文件名的规则在不同的系统中可能各不相同，所以 fopen 把文件名作为一个字符串而不是作为路径名、驱动器字母、文件扩展名等各准备一个参数。这个参数指定要打开的文件——FILE *变量的名字是程序用来保存 fopen 的返回值的，它并不影响哪个文件被打开。mode（模式）参数提示流是用于只读、只写还是既读又写，以及它是文本流还是二进制流。下面的表格列出了一些常用的模式。

	读　取	写　入	添　加
文本	“r”	“w”	“a”
二进制	“rb”	“wb”	“ab”

mode 以 r、w 或 a 开头，分别表示打开的流用于读取、写入还是添加。如果一个文件打开是用于读取的，那么它必须是原先已经存在的。但是，如果一个文件打开是用于写入的，如果它原先已经存在，那么它原来的内容就会被删除。如果它原先不存在，那么就创建一个新文件。如果一个打开用于添加的文件原先并不存在，那么它将被创建。如果它原先已经存在，它原先的内容并不会被删除。无论在哪一种情况下，数据只能从文件的尾部写入。

在 mode 中添加“a +”表示该文件打开用于更新，并且流既允许读也允许写。但是，如果你已经从该文件读入了一些数据，那么在你开始向它写入数据之前，你必须调用其中一个文件定位函数（fseek、fsetpos、rewind，它们将在本章稍后描述）。在你向文件写入一些数据之后，如果你又想从该文件读取一些数据，你首先必须调用 fflush 函数或者文件定位函数之一。

如果 fopen 函数执行成功，它返回一个指向 FILE 结构的指针，该结构代表这个新创建的流。如果函数执行失败，它就返回一个 NULL 指针，errno 会提示问题的性质。

警告：

你应该始终检查 fopen 函数的返回值！如果函数失败，它会返回一个 NULL 值。如果程序不检查错误，这个 NULL 指针就会传给后续的 I/O 函数。它们将对这个指针执行间接访问，并将失败。

下面的例子说明了 fopen 函数的用法。

```
FILE    *input;

input = fopen( "data3", "r" );
if( input == NULL ){
        perror( "data3" );
        exit( EXIT_FAILURE );
}
```

首先，fopen 函数被调用。这个被打开的文件名叫 data3，打开用于读取。这个步骤之后就是非常重要的对返回值的检查，确定文件打开是否成功。如果失败，错误就被报告给用户，程序也将终止。调用 perror 所产生的确切输出结果在不同的操作系统中可能各不相同，但它大致应该像下面这个样子：

```
data3: No such file or directory
```


bbs.theithome.com

这种类型的信息清楚地向用户报告有一个地方出了差错，并很好地提示了问题的性质。在那些读取文件名或者从命令行接受文件名的程序中，报告这些错误尤其重要。当用户输入一个文件名时，存在出错的可能性。显然，描述性的错误信息能够帮助用户判断哪里出了错以及如何修正它。

freopen 函数用于打开（或重新打开）一个特定的文件流。它的原型如下：

```
FILE *freopen( char const *filename, char const *mode,     FILE *stream );
```

最后一个参数就是需要打开的流。它可能是一个先前从 fopen 函数返回的流，也可能是标准流 stdin、stdout 或 stderr。

这个函数首先试图关闭这个流，然后用指定的文件和模式重新打开这个流。如果打开失败，函数返回一个 NULL 值。如果打开成功，函数就返回它的第 3 个参数值。

15.7 关闭流

流是用函数 fclose 关闭的，它的原型如下：

```
int  fclose( FILE *f );
```

对于输出流，fclose 函数在文件关闭之前刷新缓冲区。如果它执行成功，fclose 返回零值，否则返回 EOF。

程序 15.1 把它的命令行参数解释为一列文件名。它打开每个文件并逐个对它们进行处理。如果有任何一个文件无法打开，它就打印一条包含该文件名的错误信息。然后程序继续处理列表中的下一个文件名。退出状态(exit_status)取决于是否有错误发生。

我早先说过任何有可能失败的操作都应该进行检查，确定它是否成功执行。这个程序对 fclose 函数的返回值进行了检查，看看是否有什么地方出现了问题。许多程序员懒得执行这个测试，他们争辩说关闭文件没理由失败。更何况，此时对这个文件的操作已经结束，即使 fclose 函数失败也并无大碍。然而，这个分析并不完全正确。

```
/*
** 处理每个文件名出现于命令行的文件
*/
#include <stdlib.h>
#include <stdio.h>

int
main( int ac, char **av )
{
        int  exit_status = EXIT_SUCCESS;
        FILE      *input;

        /*
        ** 当还有更多的文件名时...
        */
        while( *++av != NULL ){
        /*
        ** 试图打开这个文件。
        */
              input = fopen( *av, "r" );
              if( input == NULL ){
```


bbs.theithome.com

```
                perror( *av );
                exit_status = EXIT_FAILURE;
                continue;
        }

    /*
    ** 在这里处理这个文件...
    */

    /*
    ** 关闭文件（期望这里不会发生什么错误）。
    */
    if( fclose( input ) != 0 ){
                perror( "fclose" );
                exit( EXIT_FAILURE );
            }
        }

        return exit_status;
}
```

程序 15.1　打开和关闭文件　　open_cls.c

input 变量可能因为 fopen 和 fclose 之间的一个程序 bug 而发生修改。这个 bug 无疑将导致程序失败。在那些并不检查 fopen 函数的返回值的程序中，input 的值甚至有可能是 NULL。在任何一种情况下，fclose 都将会失败，而且程序很可能在 fclose 被调用之前很早便已终止。

那么你是否应该对 fclose（或任何其他操作）进行错误检查呢？在你作出决定之前，首先问自己两个问题。

1．如果操作成功应该执行什么？

2．如果操作失败应该执行什么？

如果这两个问题的答案是不同的，那么你应该进行错误检查。只有当这两个问题的答案是相同时，跳过错误检查才是合理的。

15.8　字符 I/O

当一个流被打开之后，它可以用于输入和输出。它最简单的形式是字符 I/O。字符输入是由 getchar 函数家族执行的，它们的原型如下所示。

```
int fgetc( FILE *stream );
int getc( FILE *stream );
int getchar( void );
```

需要操作的流作为参数传递给 getc 和 fgetc，但 getchar 始终从标准输入读取。每个函数从流中读取下一个字符，并把它作为函数的返回值返回。如果流中不存在更多的字符，函数就返回常量值 EOF。

这些函数都用于读取字符，但它们都返回一个 int 型值而不是 char 型值。尽管表示字符的代码本身是小整型，但返回 int 型值的真正原因是为了允许函数报告文件的末尾（EOF）。如果返回值是 char 型，那么在 256 个字符中必须有一个被指定用于表示 EOF。如果这个字符出现在文件内部，那么这个字符以后的内容将不会被读取，因为它被解释为 EOF 标志。

bbs.theithome.com

让函数返回一个 int 型值就能解决这个问题。EOF 被定义为一个整型，它的值在任何可能出现的字符范围之外。这种解决方法允许我们使用这些函数来读取二进制文件。在二进制文件中，所有的字符都有可能出现，文本文件也是如此。

为了把单个字符写入到流中，你可以使用 putchar 函数家族。它们的原型如下：

```
int fputc( int character, FILE *stream );
int putc( int character, FILE *stream );
int putchar( int character );
```

第 1 个参数是要被打印的字符。在打印之前，函数把这个整型参数裁剪为一个无符号字符型值，所以

```
putchar('abc' );
```

只打印一个字符（至于是哪一个，不同的编译器可能不同）。

如果由于任何原因（如写入到一个已被关闭的流）导致函数失败，它们就返回 EOF。

15.8.1　字符 I/O 宏

fgetc 和 fputc 都是真正的函数，但 getc、putc、getchar 和 putchar 都是通过#define 指令定义的宏。宏在执行时间上效率稍高，而函数在程序的长度方面更胜一筹。之所以提供两种类型的方法，是为了允许你根据程序的长度和执行速度哪个更重要选择正确的方法。这个区别实际上不必太看重，通过对实际程序的观察，不论采用何种类型，其结果通常相差甚微。

15.8.2　撤销字符 I/O

在你实际读取之前，你并不知道流的下一个字符是什么。因此，偶尔你所读取的字符是自己想要读取的字符的后面一个字符。例如，假定你必须从一个流中逐个读入一串数字。由于在实际读入之前，你无法知道下一个字符，你必须连续读取，直到读入一个非数字字符。但是如果你不希望丢弃这个字符，那么你该如何处置它呢？

ungetc 函数就是为了解决这种类型的问题。下面是它的原型。

```
int  ungetc( int character, FILE *stream );
```

ungetc 把一个先前读入的字符返回到流中，这样它可以在以后被重新读入。程序 15.2 说明了 ungetc 的用法。它从标准输入读取字符并把它们转换为一个整数。如果没有 ungetc，这个函数将不得不把这个多余的字符返回给调用程序，后者负责把它发送到读取下一个字符的程序部分。处理这个额外字符所涉及的特殊情况和额外逻辑使程序的复杂性显著提高。

```
/*
** 把一串从标准输入读取的数字转换为整数。
*/

#include <stdio.h>
#include <ctype.h>

int
read_int()
{
        int  value;
```


bbs.theithome.com

```
    int    ch;

    value = 0;

    /*
    ** 转换从标准输入读入的数字，当我们得到一个非数字字符时就停止。
    */
    while( ( ch = getchar() ) != EOF && isdigit( ch ) ){
          value *= 10;
          value += ch - '0';
    }

    /*
    ** 把非数字字符退回到流中，这样它就不会丢失。
    */
    ungetc( ch, stdin );
    return value;
}
```

程序 15.2　把字符转换为整数　　char_int.c

每个流都允许至少一个字符被退回。如果一个流允许退回多个字符，那么这些字符再次被读取的顺序就以退回时的反序进行。注意把字符退回到流中和写入到流中并不相同。与一个流相关联的外部存储并不受 ungetc 的影响。

警告：

“退回”字符和流的当前位置有关，所以如果用 fseek、fsetpos 或 rewind 函数改变了流的位置，所有退回的字符都将被丢弃。

15.9　未格式化的行 I/O

行 I/O 可以用两种方式执行——未格式化的或格式化的。这两种形式都用于操纵字符串。区别在于未格式化的 I/O（unformatted line I/O）简单读取或写入字符串，而格式化的 I/O 则执行数字和其他变量的内部和外部表示形式之间的转换。在本节，我们将讨论未格式化的行 I/O。

gets 和 puts 函数家族是用于操作字符串而不是单个字符。这个特征使它们在那些处理一行行文本输入的程序中非常有用。这些函数的原型如下所示。

```
char *fgets( char *buffer, int buffer_size, FILE *stream );
char *gets( char *buffer );

int fputs( char const *buffer, FILE *stream );
int puts( char const *buffer );
```

fgets 从指定的 stream 读取字符并把它们复制到 buffer 中。当它读取一个换行符并存储到缓冲区之后就不再读取。如果缓冲区内存储的字符数达到 buffer_size-1 个时它也停止读取。在这种情况下，并不会出现数据丢失的情况，因为下一次调用 fgets 将从流的下一个字符开始读取。在任何一种情况下，一个 NUL 字节将被添加到缓冲区所存储数据的末尾，使它成为一个字符串。

如果在任何字符读取前就到达了文件尾，缓冲区就未进行修改，fgets 函数返回一个 NULL 指针。否则，fgets 返回它的第 1 个参数（指向缓冲区的指针）。这个返回值通常只用于检查是否到达了文件尾。

bbs.theithome.com

传递给 fputs 的缓冲区必须包含一个字符串，它的字符被写入到流中。这个字符串预期以 NUL 字节结尾，所以这个函数没有一个缓冲区长度参数。这个字符串是逐字写入的：如果它不包含一个换行符，就不会写入换行符。如果它包含了好几个换行符，所有的换行符都会被写入。因此，当 fgets 每次都读取一整行时，fputs 却既可以一次写入一行的一部分，也可以一次写入一整行，甚至可以一次写入好几行。如果写入时出现了错误，fputs 返回常量值 EOF，否则它将返回一个非负值。

程序 15.3 是一个函数，它从一个文件读取输入行并原封不动地把它们写入到另一个文件。常量 MAX_LINE_LENGTH 决定缓冲区的长度，也就是可以被读取的一行文本的最大长度。在这个函数中，这个值并不重要，因为不管长行是被一次性读取还是分段读取，它所产生的结果文件都是相同的。另一方面，如果函数需要计数被复制的行的数目，太小的缓冲区将产生一个不正确的计数，因为一个长行可能会被分成数段进行读取。我们可以通过增加代码，观察每段是否以换行符结尾来修正这个问题。

缓冲区长度的正确值通常是根据需要执行的处理过程的本质而作出的折衷。但是，即使溢出它的缓冲区，fgets 也绝不引起错误。

警告：

注意 fgets 无法把字符串读入到一个长度小于两个字符的缓冲区，因为其中一个字符需要为 NUL 字节保留。

gets 和 puts 函数几乎和 fgets 与 fputs 相同。之所以存在它们是为了允许向后兼容。它们之间的一个主要的功能性区别在于当 gets 读取一行输入时，它并不在缓冲区中存储结尾的换行符。当 puts 写入一个字符串时，它在字符串写入之后向输出再添加一个换行符。

警告：

另一个区别仅存在于 gets，这从函数的原型中就清晰可见：它没有缓冲区长度参数。因此 gets 无法判断缓冲区的长度。如果一个长输入行读到一个短的缓冲区，多出来的字符将被写入到缓冲区后面的内存位置，这将破坏一个或多个不相关变量的值。这个事实导致 gets 函数只适用于玩具程序，因为唯一防止输入缓冲区溢出的方法就是声明一个巨大的缓冲区。但不管它有多大，下一个输入行仍有可能比缓冲区更大，尤其是当标准输入被重定向到一个文件时。

```
/*
** 把标准输入读取的文本行逐行复制到标准输出。
*/
#include <stdio.h>

#define   MAX_LINE_LENGTH         1024 /* 我可以复制的最长行 */

void
copylines( FILE *input, FILE *output )
{
          char  buffer[MAX_LINE_LENGTH];

          while( fgets( buffer, MAX_LINE_LENGTH, input ) != NULL )
                fputs( buffer, output );
}
```

程序 15.3　从一个文件向另一个文件复制文本行　　copyline.c

bbs.theithome.com

15.10 格式化的行 I/O

“格式化的行 I/O”这个名字从某种意义上说并不准确，因为 scanf 和 printf 函数家族并不仅限于单行。它们也可以在行的一部分或多行上执行 I/O 操作。

15.10.1 scanf 家族

scanf 函数家族的原型如下所示。每个原型中的省略号表示一个可变长度的指针列表。从输入转换而来的值逐个存储到这些指针参数所指向的内存位置。

```
int fscanf( FILE *stream, char const *format, ... );
int scanf( char const *format, ... );
int sscanf( char const *string, char const *format, ... );
```

这些函数都从输入源读取字符并根据 format 字符串给出的格式代码对它们进行转换。fscanf 的输入源就是作为参数给出的流，scanf 从标准输入读取，而 sscanf 则从第 1 个参数所给出的字符串中读取字符。

当格式化字符串到达末尾或者读取的输入不再匹配格式字符串所指定的类型时，输入就停止。在任何一种情况下，被转换的输入值的数目作为函数的返回值返回。如果在任何输入值被转换之前文件就已到达尾部，函数就返回常量值 EOF。

警告：

为了能让这些函数正常运行，指针参数的类型必须是对应格式代码的正确类型。函数无法验证它们的指针参数是否为正确的类型，所以函数就假定它们是正确的，于是继续执行并使用它们。如果指针参数的类型是不正确的，那么结果值就会是垃圾，而邻近的变量有可能在处理过程中被改写。

警告：

现在，对于 scanf 函数的参数前面为什么要加一个&符号应该是比较清楚的了。由于 C 的传值参数传递机制，把一个内存位置作为参数传递给函数的唯一方法就是传递一个指向该位置的指针。在使用 scanf 函数时，一个非常容易出现的错误就是忘了加上&符号。省略这个符号将导致变量的值作为参数传递给函数，而 scanf 函数（或其他两个）却把它解释为一个指针。当它被解引用时，或者导致程序终止，或者导致一个不可预料的内存位置的数据被改写。

15.10.2 scanf 格式代码

scanf 函数家族中的 format 字符串参数可能包含下列内容：

- 空白字符——它们与输入中的零个或多个空白字符匹配，在处理过程中将被忽略。
- 格式代码——它们指定函数如何解释接下来的输入字符。
- 其他字符——当任何其他字符出现在格式字符串时，下一个输入字符必须与它匹配。如果匹配，该输入字符随后就被丢弃。如果不匹配，函数就不再读取直接返回。

scanf 函数家族的格式代码都以一个百分号开头，后面可以是(1)一个可选的星号，(2)一个可选的宽度，(3)一个可选的限定符，(4)格式代码。星号将使转换后的值被丢弃而不是进行存储。这个技巧可以用于跳过不需要的输入字符。宽度以一个非负的整数给出，它限制将被读取用于转换的输入字符的个数。如果未给出宽度，函数就连续读入字符直到遇见输入中的下一个空白字符。限定符用

bbs.theithome.com

于修改有些格式代码的含义，它们在表 15.3 中列出。

表 15.3　　scanf 限定符

使用限定符的结果

格式码	h	l	L
d, i, n	short	long	
o, u, x	unsigned short	unsigned long	
e, f, g		double	long double

警告：

限定符的目的是为了指定参数的长度。如果整型参数比缺省的整型值更短或更长时，在格式代码中省略限定符就是一个常见的错误。对于浮点类型也是如此。如果省略了限定符，可能会导致一个较长变量只有部分被初始化，或者一个较短变量的邻近变量也被修改，这些都取决于这些类型的相对长度。

提示：

在一个缺省的整型长度和 short 相同的机器上，在转换一个 short 值时限定符 h 并非必需。但是，对于那些缺省的整型长度比 short 长的机器上，这个限定符是必需的。因此，如果你在转换所有的 short 和 long 型整数值和 long double 型变量时都使用适当的限定符，你的程序将更具可移植性。

格式代码就是一个单字符，用于指定输入字符如何被解释。表 15.4 描述了这些代码。

让我们来看一些使用 scanf 函数家族的例子。同样，我只显示与这些函数有关的部分代码。我们的第 1 个例子非常简单明了。它从输入流成对地读取数字并对它们进行一些处理。当读取到文件末尾时，循环就终止。

```
int     a, b;

while( fscanf( input, "%d %d", &a, &b ) == 2 ){
        /*
        ** Process the values a and b.
        */
}
```

这段代码并不精致，因为从流中输入的任何非法字符都将导致循环终止。同样，由于 fscanf 跳过空白字符，所以它没有办法验证这两个值是位于同一行还是分属两个不同的输入行。解决这个问题可以使用一种技巧，它将在后面的例子中说明。

下一个例子使用了字段宽度。

```
nfields = fscanf( input, "%4d %4d %4d", &a, &b, &c )
```

这个宽度参数把整数值的宽度限制为 4 个数字或者更少。使用下面的输入，

```
1 2
```

a 的值将是 1，b 的值将是 2，c 的值没有改变，nfields 的值将是 2。但是，如果使用下面的输入，

```
12345 67890
```

a 的值将是 1234，b 的值是 5，c 的值是 6789，而 nfields 的值是 3。输入中的最后一个 0 将保持在未输入状态。

在使用 fscanf 时，在输入中保持行边界的同步是很困难的，因为它把换行符也当作空白字符跳

bbs.theithome.com

过。例如，假定有一个程序读取的输入是由 4 个值所组成的一组值。这些值然后通过某种方式进行处理，然后再读取接下来的 4 个值。在这类程序中准备输入的最简单方法是把每组的 4 个值放在一个单独的输入行，这就很容易观察哪些值形成一组。但如果某个行包含了太多或太少的值，程序就会产生混淆。例如，考虑下面这个输入，它的第 2 行包含了一个错误：

```
1 1 1 1
2 2 2 2 2
3 3 3 3
4 4 4 4
5 5 5 5
```

如果我们使用 fscanf 按照一次读取 4 个值的方式读取这些数据时，头两组数据是正确的，但第 3 组读取的数据将是 2, 3, 3, 3，接下来的各组数据也都将不正确。

表 15.4　　scanf 格式码

代　码	参　数	含　义
c	char *	读取和存储单个字符。前导的空白字符并不跳过。如果给出宽度，就读取和存储这个数目的字符。字符后面不会添加一个 NUL 字节。参数必须指向一个足够大的字符数组
i d	int *	一个可选的有符号整数被转换。d 把输入解释为十进制数；i 根据它的第 1 个字符决定值的基数，就像整型字面值常量的表示形式一样
u o x	unsigned *	一个可选的有符号整数被转换，但它按照无符号数存储。如果使用 u，值被解释为十进制数；如果使用 o，值被解释为八进制数；如果使用 x，值被解释为十六进制数。X 和 x 同义
e f g	float *	期待一个浮点值。它的形式必须像一个浮点型字面值常量，但小数点并非必需。E 和 G 分别与 e 和 g 同义
s	char *	读取一串非空白字符。参数必须指向一个足够大的字符数组。当发现空白时输入就停止，字符串后面会自动加上 NUL 终止符
[xxx]	char *	根据给定组合的字符从输入中读取一串字符。参数必须指向一个足够大的字符数组。当遇到第 1 个不在给定组合中出现的字符时，输入就停止。字符串后面会自动加上 NUL 终止符。代码%[abc]表示字符组合包括 a、b 和 c。如果列表中以一个^字符开头，表示字符组合是所列出的字符的补集，所以%[^abc]表示字符组合为 a、b、c 之外的所有字符。右方括号也可以出现在字符列表中，但它必须是列表的第 1 个字符。至于横杠是否用于指定某个范围的字符（例如%[a-z]），则因编译器而异
p	void *	输入预期为一串字符，诸如那些由 printf 函数的%p 格式代码所产生的输出。它的转换方式因编译器而异，但转换结果将和按照上面描述的进行打印所产生的字符的值是相同的
n	int *	到目前为止通过这个 scanf 函数的调用从输入读取的字符数被返回。%n 转换的字符并不计算在 scanf 函数的返回值之内。它本身并不消耗任何输入
%	(无)	这个代码与输入中的一个%相匹配，该%符号将被丢弃

程序 15.4 使用一种更为可靠的方法读取这种类型的输入。这个方法的优点在于现在的输入是逐步处理的。它不可能读入一组起始于某一行但结束于另一行的值。而且，通过尝试转换 5 个值，无论是输入行的值太多还是太少都会被检测出来。

```
/*
** 用 sscanf 处理行定向(line-oriented)的输入
*/
#include <stdio.h>
#define    BUFFER_SIZE    100   /* 我们将要处理的最长行 */

void
function( FILE *input )
{
```


bbs.theithome.com

```
        int  a, b, c, d, e;
        char buffer[ BUFFER_SIZE ];

        while( fgets( buffer, BUFFER_SIZE, input ) != NULL ){
                if( sscanf( buffer, "%d %d %d %d %d",
                    &a, &b, &c, &d, &e ) != 4 ){
                        fprintf( stderr, "Bad input skipped: %s",
                           buffer );
                        continue;
                }

        /*
        ** 处理这组输入
        */
        }
}
```

程序 15.4　用 sscanf 处理行定向的输入　　scanf1.c

一个相关的技巧用于读取可能以几种不同的格式出现的行定向输入。每个输入行先用 fgets 读取，然后用几个 sscanf（每个都使用一种不同的格式）进行扫描。输入行由**第 1 个** sscanf 决定，后者用于转换预期数目的值。例如，程序 15.5 检查一个以前读取的缓冲区的内容。它从一个输入行中提取或者 1 个或者 2 个或者 3 个值并对那些没有输入值的变量赋缺省的值。

```
/*
** 使用 sscanf 处理可变格式的输入
*/
#include <stdio.h>
#include <stdlib.h>

#define   DEFAULT_A 1    /* 或其他 ... */
#define   DEFAULT_B 2    /* 或其他 ... */

void
function( char *buffer )
{
        int   a, b, c;

        /*
        ** 看看 3 个值是否都已给出。
        */
        if( sscanf( buffer, "%d %d %d", &a, &b, &c ) != 3 ){
        /*
        ** 否，对 a 使用缺省值，看看其他两个值是否都已给出。
        */
                a = DEFAULT_A;
                if( sscanf( buffer, "%d %d", &b, &c ) != 2 ){
                /*
                ** 也为 b 使用缺省值，寻找剩余的值。
                */
                        b = DEFAULT_B;
                        if( sscanf( buffer, "%d", &c ) != 1 ){
                                fprintf( stderr, "Bad input: %s",
                                buffer );
```


bbs.theithome.com

```
                        exit( EXIT_FAILURE );
                    }
                }
            }
            /*
            ** 处理 a, b, c
            */
    }
```

程序 15.5　使用 sscanf 处理可变格式的输入　　scanf2.c

15.10.3　printf 家族

printf 函数家庭用于创建格式化的输出。这个家族共有 3 个函数：fprintf、printf 和 sprintf。它们的原型如下所示。

```
int fprintf( FILE *stream, char const *format, ... );
int printf( char const *format, ... );
int sprintf( char *buffer, char const *format, ... );
```

你在第 1 章就曾见过，printf 根据格式代码和 format 参数中的其他字符对参数列表中的值进行格式化。这个家族的另两个函数的工作过程也类似。使用 printf，结果输出送到标准输出。使用 fprintf，你可以使用任何输出流，而 sprintf 把它的结果作为一个 NUL 结尾的字符串存储到指定的 buffer 缓冲区而不是写入到流中。这 3 个函数的返回值是实际打印或存储的字符数。

警告：

sprintf 是一个潜在的错误根源。缓冲区的大小并不是 sprintf 函数的一个参数，所以如果输出结果很长溢出缓冲区时，就可能改写缓冲区后面内存位置中的数据。要杜绝这个问题，可以采取两种策略。第 1 种是声明一个非常巨大的缓冲区，但这个方案很浪费内存，而且尽管大型缓冲区能够减少溢出的可能性，但它并不能根除这种可能性。第 2 种方法是对格式进行分析，看看最大可能出现的值被转换后的结果输出将有多长。例如，在 4 位整型的机器上，最大的整数有 11 位（包括一个符号位），所以缓冲区至少能容纳 12 个字符（包括结尾的 NUL 字节）。字符串的长度并没有限制，但函数所生成的字符串的字符数目可以用格式代码中一个可选的字段来限制。

警告：

printf 函数家族的格式代码和 scanf 函数家族的格式代码用法不同。所以你必须小心谨慎，防止误用。两者的格式代码中的有些可选字段看上去是相同的，这使得问题变得更为困难。不幸的是，许多常见的格式代码，如%d 就属于这一类。

警告：

另一个错误来源是函数的参数类型与对应的格式代码不匹配。通常这个错误将导致输出结果是垃圾，但这种不匹配也可能导致程序失败。和 scanf 函数家族一样，这些函数无法验证一个值是否具有格式码所表示的正确类型，所以保证它们相互匹配是程序员的责任。

15.10.4　printf 格式代码

printf 函数原型中的 format 字符串可能包含格式代码，它使参数列表的下一个值根据指定的方

bbs.theithome.com

式进行格式化，至于其他的字符则原样逐字打印。格式代码由一个百分号开头，后面跟(1)零个或多个标志字符，用于修改有些转换的执行方式，(2)一个可选的最小字段宽度，(3)一个可选的精度，(4)一个可选的修改符，(5)转换类型。

标志和其他字段的准确含义取决于使用何种转换。表 15.5 描述了转换类型代码，表 15.6 描述了标志字符和它们的含义。

表 15.5　printf 格式代码

代　码	参　　数	含　　义
c	int	参数被裁剪为 unsigned char 类型并作为字符进行打印
d i	int	参数作为一个十进制整数打印。如果给出了精度而且值的位数少于精度位度，前面就用 0 填充
u o x,X	unsigned int	参数作为一个无符号值打印，u 使用十进制，o 使用八进制，x 或 X 使用十六进制，两者的区别是 x 约定使用 abcdef，而 X 约定使用 ABCDEF
e E	double	参数根据指数形式打印。例如，6.023000e23 是使用代码 e，6.023000E23 是使用代码 E。小数点后面的位数由精度字段决定，缺省值是 6
f	double	参数按照常规的浮点格式打印。精度字段决定小数点后面的位数，缺省值是 6
g G	double	参数以%f 或%e（如 G 则%E）的格式打印，取决于它的值。如果指数大于等于-4 但小于精度字段就使用%f 格式，否则使用指数格式
s	char *	打印一个字符串
p	void *	指针值被转换为一串因编译器而异的可打印字符。这个代码主要是和 scanf 中的%p 代码组合使用
n	int *	这个代码是独特的，因为它并不产生任何输出。相反，到目前为止函数所产生的输出字符数目将被保存到对应的参数中
%	(无)	打印一个%字符

表 15.6　printf 格式标志

标　志	含　　义
-	值在字段中左对齐，缺省情况下是右对齐
0	当数值为右对齐时，缺省情况下是使用空格填充值左边未使用的列。这个标志表示用零来填充，它可用于 d,i,u,o,x,X,e,E,f,g 和 G 代码。使用 d,i,u,o,x 和 X 代码时，如果给出了精度字段，零标志就被忽略。如果格式代码中出现了负号标志，零标志也没有效果
+	当用于一个格式化某个有符号值的代码时，如果值非负，正号标志就会给它加上一个正号。如果该值为负，就像往常一样显示一个负号。在缺省情况下，正号并不会显示
空格	只用于转换有符号值的代码。当值非负时，这个标志把一个空格添加到它的开始位置。注意这个标志和正号标志是相互排斥的，如果两个同时给出，空格标志便被忽略
#	选择某些代码的另一种转换形式。它们在表 15.8 中描述

字段宽度是一个十进制整数，用于指定将出现在结果中的最小字符数。如果值的字符数少于字段宽度，就对它进行填充以增加长度。标志决定填充是用空白还是零以及它出现在值的左边还是右边。

对于 d、i、u、o、x 和 X 类型的转换，精度字段指定将出现在结果中的最小的数字个数并覆盖零标志。如果转换后的值的位数小于宽度，就在它的前面插入零。如果值为零且精度也为零，则转换结果就不会产生数字。对于 e、E 和 f 类型的转换，精度决定将出现在小数点之后的数字位数。对于 g 和 G 类型的转换，它指定将出现在结果中的最大有效位数。当使用 s 类型的转换时，精度指定将被转换的最多字符数。精度以一个句点开头，后面跟一个可选的十进制整数。如果未

bbs.theithome.com

给出整数，精度的缺省值为零。

如果用于表示字段宽度和/或精度的十进制整数由一个星号代替，那么 printf 的下一个参数（必须是个整数）就提供宽度和（或）精度。因此，这些值可以通过计算获得而不必预先指定。

当字符或短整数值作为 printf 函数的参数时，它们在传递给函数之前先转换为整数。有时候转换可以影响函数产生的输出。同样，在一个长整数的长度大于普通整数的环境里，当一个长整数作为参数传递给函数时，printf 必须知道这个参数是个长整数。表 15.7 所示的修改符用于指定整数和浮点数参数的准确长度，从而解决了这个问题。

表 15.7　　printf 格式代码修改符

修　改　符	用于...时	表示参数是...
h	d, i, u, o, x, X	一个（可能是无符号）short 型整数
h	n	一个指向 short 型整数的指针
l	d, i, u, o, x, X	一个（可能是无符号）long 型整数
l	n	一个指向 long 型整数的指针
L	e, E, f, g, G	一个 long double 型值

在有些环境里，int 和 short int 的长度相等，此时 h 修改符就没有效果。否则，当 short int 作为参数传递给函数时，这个被转换的值将升级为（无符号）int 类型。这个修改符在转换发生之前使它被裁剪回原先的 short 形式。在十进制转换中，一般并不需要进行剪裁。但在有些八进制或十六进制的转换中，h 修改符将保证适当位数的数字被打印。

警告：

在 int 和 long int 长度相同的机器上，l 修改符并无效果。在所有其他机器上，需要使用 l 修改符，因为这些机器上的长整型分为两部分传递到运行时堆栈。如果这个修改符并未给出，那就只有第 1 部分被提取用于转换。这样，不仅转换将产生不正确的结果，而且这个值的第 2 部分被解释为一个单独的参数，这样就破坏了后续参数和它们的格式代码之间的对应关系。

#标志可以用于几种 printf 格式代码，为转换选择一种替代形式。这些形式的细节列于表 15.8。

表 15.8　　printf 转换的其他形式

用于...	#标志...
o	保证产生的值以一个零开头
x, X	在非零值前面加 0x 前缀（%X 则为 0X）
e, E, f	确保结果始终包含一个小数点，即使它后面没有数字
g, G	和上面的 e,E 和 f 代码相同。另外，缀尾的 0 并不从小数中去除

提示：

由于有些机器在打印长整数值时要求 l 修改符而另外一些机器可能不需要。所以，当你打印长整数值时，最好坚持使用 l 修改符。这样，当你把程序移植到任何一台机器上时，就不太需要进行改动。

printf 函数可以使用丰富的格式代码、修改符、限定符、替代形式和可选字段，这使得它看上去极为复杂。但是，它们能够在格式化输出时提供极大的灵活性。所以，你应该耐心一些，把它们全部学会要花一些时间！这里有一些例子，帮助你学习它们。

bbs.theithome.com

图 15.1 显示了格式化字符串可能产生的一些变型。只有显示出来的字符才被打印。为了避免歧义，符号¤用于表示一个空白。图 15.2 显示了用不同的整数格式代码格式化一些整数值的结果。图 15.3 显示了浮点值被格式化的一些可能方法。最后，图 15.4 显示了用与前图相同的那些格式代码来格式化一个非常大的浮点数的结果。在前两个输出中出现了明显的错误，因为它们所打印的有效数字的位数超出了指定内存位置所能存储的位数。

格式代码	转换后的字符串		
	A	ABC	ABCDEFGH
%s	A	ABC	ABCDEFGH
%5s	¤¤¤¤A	¤¤ABC	ABCDEFGH
%.5s	A	ABC	ABCDE
%5.5s	¤¤¤¤A	¤¤ABC	ABCDE
%-5s	A¤¤¤¤	ABC¤¤	ABCDEFGH

图 15.1　用 printf 格式字符串

格式代码	转换后的数值			
	1	-12	12345	123456789
%d	1	-12	12345	123456789
%6d	¤¤¤¤¤1	¤¤¤-12	¤12345	123456789
%.4d	0001	-0012	12345	123456789
%6.4d	¤¤0001	¤-0012	¤12345	123456789
%-4d	1¤¤¤	-12¤	12345	123456789
%04d	0001	-012	12345	123456789
%+d	+1	-12	+12345	+123456789

图 15.2　用 printf 格式化整数

格式代码	转换后的数值			
	1	.01	.00012345	12345.6789
%f	1.000000	0.010000	0.000123	12345.678900
%10.2f	¤¤¤¤¤¤1.00	¤¤¤¤¤¤0.01	¤¤¤¤¤¤0.00	¤¤12345.68
%e	1.000000e+00	1.000000e-02	1.234500e-04	1.234568e+04
%.4e	1.0000e+00	1.0000e-02	1.2345e-04	1.2346e+04
%g	1	0.01	0.00012345	12345.7

图 15.3　用 printf 格式化浮点值

格式代码	转换后的数值
	6.023e23
%f	602299999999999975882752.000000
%10.2f	602299999999999975882752.00
%e	6.023000e+23
%.4e	6.0230e+23
%g	6.023e+23

图 15.4　用 printf 格式化大浮点值

bbs.theithome.com

15.11 二进制 I/O

把数据写到文件效率最高的方法是用二进制形式写入。二进制输出避免了在数值转换为字符串过程中所涉及的开销和精度损失。但二进制数据并非人眼所能阅读，所以这个技巧只有当数据将被另一个程序按顺序读取时才能使用。

fread 函数用于读取二进制数据，fwrite 函数用于写入二进制数据。它们的原型如下所示：

```
size_t  fread( void *buffer, size_t size, size_t count,  FILE *stream );
size_t  fwrite( void *buffer, size_t size, size_t count, FILE *stream );
```

buffer 是一个指向用于保存数据的内存位置的指针，size 是缓冲区中每个元素的字节数，count 是读取或写入的元素数，当然 stream 是数据读取或写入的流。

buffer 参数被解释为一个或多个值的数组。count 参数指定数组中有多少个值，所以读取或写入一个标量时，count 的值应为 1。函数的返回值是实际读取或写入的**元素**（并非字节）数目。如果输入过程中遇到了文件尾或者输出过程中出现了错误，这个数字可能比请求的元素数目要小。

让我们观察一个使用这些函数的代码段。

```
struct  VALUE    {
        long     a;
        float    b;
        char     c[SIZE];
} values[ARRAY_SIZE];
...
n_values = fread( values, sizeof( struct VALUE ),
    ARRAY_SIZE, input_stream );
(处理数组中的数据)
fwrite( values, sizeof( struct VALUE ),
    n_values, output_stream );
```

这个程序从一个输入文件读取二进制数据，对它执行某种类型的处理，把结果写入到一个输出文件。前面提到过，这种类型的 I/O 效率很高，因为每个值中的位直接从流读取或向流写入，不需要任何转换。例如，假定数组中的一个长整数的值是 4,023,817。代表这个值的位是 0x003d6609——这些位将被写入到流中。二进制信息非人眼所能阅读，因为这些位并不对应任何合理的字符。如果它们解释为字符，其值将是\0=f\t，这显然不能很好地向我们传达原数的值。

15.12 刷新和定位函数

在处理流时，另外还有一些函数也较为有用。首先是 fflush，它迫使一个输出流的缓冲区内的数据进行物理写入，不管它是不是已经写满。它的原型如下：

```
int  fflush( FILE *stream );
```

当我们需要立即把输出缓冲区的数据进行物理写入时，应该使用这个函数。例如，调用 fflush 函数保证调试信息实际打印出来，而不是保存在缓冲区中直到以后才打印。

在正常情况下，数据以线性的方式写入，这意味着后面写入的数据在文件中的位置是在以前所有写入数据的后面。C 同时支持随机访问 I/O，也就是以任意顺序访问文件的不同位置。随机访问是通过在读取或写入先前定位到文件中需要的位置来实现的。有两个函数用于执行这项操作，它们的原型如下：

bbs.theithome.com

```
long  ftell( FILE *stream );
int  fseek( FILE *stream, long offset, int from );
```

ftell 函数返回流的当前位置，也就是说，下一个读取或写入将要开始的位置距离文件起始位置的偏移量。这个函数允许你保存一个文件的当前位置，这样你可能在将来会返回到这个位置。在二进制流中，这个值就是当前位置距离文件起始位置之间的字节数。

在文本流中，这个值表示一个位置，但它并不一定准确地表示当前位置和文件起始位置之间的字符数，因为有些系统将对行末字符进行翻译转换。但是，ftell 函数返回的值总是可以用于 fseek 函数中，作为一个距离文件起始位置的偏移量。

fseek 函数允许你在一个流中定位。这个操作将改变下一个读取或写入操作的位置。它的第 1 个参数是需要改变的流。它的第 2 和第 3 个参数标识文件中需要定位的位置。表 15.9 描述了三种第 2 个和第 3 个参数可以使用的方法。

试图定位到一个文件的起始位置之前是一个错误。定位到文件尾之后并进行写入将扩展这个文件。定位到文件尾之后并进行读取将导致返回一条“到达文件尾”的信息。在二进制流中，从 SEEK_END 进行定位可能不被支持，所以应该避免。在文本流中，如果 from 是 SEEK_CUR 或 SEEK_END，offset 必须是零。如果 from 是 SEEK_SET，offset 必须是一个从同一个流中以前调用 ftell 所返回的值。

表 15.9　　fseek 参数

如果 from 是...	你将定位到...
SEEK_SET	从流的起始位置起 offset 个字节，offset 必须是一个非负值
SEEK_CUR	从流的当前位置起 offset 个字节，offset 的值可正可负
SEEK_END	从流的尾部位置起 offset 个字节，offset 的值可正可负。如果它是正值，它将定位到文件尾的后面

之所以存在这些限制，部分原因是文本流所执行的行末字符映射。由于这种映射的存在，文本文件的字节数可能和程序写入的字节数不同。因此，一个可移植的程序不能根据实际写入字符数的计算结果定位到文本流的一个位置。

用 fseek 改变一个流的位置会带来三个副作用。首先，行末指示字符被清除。其次，如果在 fseek 之前使用 ungetc 把一个字符返回到流中，那么这个被退回的字符会被丢弃，因为在定位操作以后，它不再是“下一个字符”。最后，定位允许你从写入模式切换到读取模式，或者回到打开的流以便更新。

程序 15.6 使用 fseek 访问一个学生信息文件。记录数参数的类型是 size_t，这是因为它不可能是个负值。需要定位的文件位置通过将记录数和记录长度相乘得到。只有当文件中的所有记录都是同一长度时，这种计算方法才是可行的。最后，fread 的结果被返回，这样调用程序就可以判断操作是否成功。

另外还有三个额外的函数，用一些限制更严的方式执行相同的任务。它们的原型如下：

```
void  rewind( FILE *stream );
int   fgetpos( FILE *stream, fpos_t *position );
int   fsetpos( FILE *stream, fpos_t const *position );
```

rewind 函数将读/写指针设置回指定流的起始位置。它同时清除流的错误提示标志。fgetpos 和 fsetpos 函数分别是 ftell 和 fseek 函数的替代方案。

它们的主要区别在于这对函数接受一个指向 fpos_t 的指针作为参数。fgetpos 在这个位置存储文件的当前位置，fsetpos 把文件位置设置为存储在这个位置的值。

bbs.theithome.com

用 fpos_t 表示一个文件位置的方式并不是由标准定义的。它可能是文件中的一个字节偏移量，也可能不是。因此，使用一个从 fgetpos 函数返回的 fpos_t 类型的值唯一安全的用法是把它作为参数传递给后续的 fsetpos 函数。

```
/*
** 从一个文件读取一个特定的记录。参数分别是进行读取的流、需要读取的记录数和** 指向放置数据的缓冲区的指针。
*/
#include <stdio.h>
#include "student_info.h"

int
read_random_record( FILE *f, size_t rec_number, StudentInfo *buffer )
{
        fseek( f, (long)rec_number * sizeof( StudentInfo ),
               SEEK_SET );
        return fread( buffer, sizeof( StudentInfo ), 1, f );
}
```

程序 15.6　随机文件访问　　rd_rand.c

15.13　改变缓冲方式

在流上执行的缓冲方式有时并不合适，下面两个函数可以用于对缓冲方式进行修改。这两个函数只有当指定的流被打开但还没有在它上面执行任何其他操作前才能被调用。

```
void setbuf( FILE *stream, char *buf );
int  setvbuf( FILE *stream, char *buf, int mode, size_t size );
```

setbuf 设置了另一个数组，用于对流进行缓冲。这个数组的字符长度必须为 BUFSIZ（它在 stdio.h 中定义）。为一个流自行指定缓冲区可以防止 I/O 函数库为它动态分配一个缓冲区。如果用一个 NULL 参数调用这个函数，setbuf 函数将关闭流的所有缓冲方式。字符准确地将程序所规引的方式进行读取和写入[1]。

警告：

为流缓冲区使用一个自动数组是很危险的。如果在流关闭之前，程序的执行流离开了数组声明所在的代码块，流就会继续使用这块内存，但此时它可能已经分配给了其他函数另作它用。

setvbuf 函数更为通用。mode 参数用于指定缓冲的类型。_IOFBF 指定一个完全缓冲的流，_IONBF 指定一个不缓冲的流，_IOLBF 指定一个行缓冲流。所谓行缓冲，就是每当一个换行符写入到缓冲区时，缓冲区便进行刷新。

buf 和 size 参数用于指定需要使用的缓冲区。如果 buf 为 NULL，那么 size 的值必须是 0。一般而言，最好用一个长度为 BUFSIZ 的字符数组作为缓冲区。尽管使用一个非常大的缓冲区可能可以稍稍提高程序的效率，但如果使用不当，它也有可能降低程序的效率。例如，绝大多数操作系统在内部对磁盘的输入/输出进行缓冲操作。如果你自行指定了一个缓冲区，但它的长度却不是操作系统内部使用的缓冲区的整数倍，就可能需要一些额外的磁盘操作，用于读取或写入一个内存块的一部

[1] 在宿主式运行时环境中，操作系统可能执行自己的缓冲方式，不依赖于流。因此，仅仅调用 setbuf 将不允许程序从键盘即输即读入字符，因为操作系统通常对这些字符进行缓冲，用于实现退格编辑。

bbs.theithome.com

分。如果你需要使用一个很大的缓冲区，它的长度应该是 BUFSIZ 的整数倍。在 MS-DOS 机器中，缓冲区的大小如果和磁盘簇的大小相匹配，可能会提高一些效率。

15.14　流错误函数

下面的函数用于判断流的状态。

```
int  feof( FILE *stream );
int  ferror( FILE *stream );
void clearerr( FILE *stream );
```

如果流当前处于文件尾，feof 函数返回真。这个状态可以通过对流执行 fseek、rewind 或 fsetpos 函数来清除。ferror 函数报告流的错误状态，如果出现任何读/写错误函数就返回真。最后，clearerr 函数对指定流的错误标志进行重置。

15.15　临时文件

偶尔，为了方便起见，我们会使用一个文件来临时保存数据。当程序结束时，这个文件便被删除，因为它所包含的数据不再有用。tmpfile 函数就是用于这个目的的。

```
FILE *tmpfile( void );
```

这个函数创建了一个文件，当文件被关闭或程序终止时这个文件便自动删除。该文件以 wb+模式打开，这使它可用于二进制和文本数据。

如果临时文件必须以其他模式打开或者由一个程序打开但由另一个程序读取，就不适合用 tmpfile 函数创建。在这些情况下，我们必须使用 fopen 函数，而且当结果文件不再需要时必须使用 remove 函数（稍后描述）显式地删除。

临时文件的名字可以用 tmpnam 函数创建，它的原型如下：

```
char  *tmpnam( char *name );
```

如果传递给函数的参数为 NULL，那么这个函数便返回一个指向静态数组的指针，该数组包含了被创建的文件名。否则，参数便假定是一个指向长度至少为 L_tmpnam 的字符数组的指针。在这种情况下，文件名在这个数组中创建，返回值就是这个参数。

无论哪种情况，这个被创建的文件名保证不会与已经存在的文件名同名[1]。只要调用次数不超过 TMP_MAX 次，tmpnam 函数每次调用时都能产生一个新的不同名字。

15.16　文件操纵函数

有两个函数用于操纵文件但不执行任何输入/输出操作。它们的原型如下所示。如果执行成功，这两个函数都返回零值。如果失败，它们都返回非零值。

---

[1] 注意：这个用于保证唯一性的方法可能会在多程序系统(multiprogramming system)或那些共享一个网络文件服务器的系统中失败。问题的根源是名字被创建和该名字所标识的文件被创建之间的时间延迟。如果几个程序恰好都创建了一个相同的名字，并在任何文件被实际创建之前测试是否存在这个名字的文件，此时测试结果是否定的，于是每个程序都以为这是个唯一的名字。在文件名被创建之后立即创建文件可以减少（但不能根除）这种潜在的冲突。

bbs.theithome.com

```
int  remove( char const *filename );
int  rename( char const *oldname, char const *newname );
```

remove 函数删除一个指定的文件。如果当 remove 被调用时文件处于打开状态，其结果则取决于编译器。

rename 函数用于改变一个文件的名字，从 oldname 改为 newname。如果已经有一个名为 newname 的文件存在，其结果取决于编译器。如果这个函数失败，文件仍然可以用原来的名字进行访问。

15.17 总结

标准规定了标准函数库中的函数的接口和操作，这有助于提高程序的可移植性。一种编译器可以在它的函数库中提供额外的函数，但不应修改标准要求提供的函数。

perror 函数提供了一种向用户报告错误的简单方法。当检测到一个致命的错误时，你可以使用 exit 函数终止程序。

stdio.h 头文件包含了使用 I/O 库函数所需要的声明。所有的 I/O 操作都是一种在程序中移进或移出字节的事务。函数库为 I/O 所提供的接口称为流。在缺省情况下，流 I/O 是进行缓冲的。二进制流主要用于二进制数据，字节不经修改地从二进制流读取或向二进制流写入。另一方面，文本流则用于字符。文本流能够允许的最大文本行因编译器而异，但至少允许 254 个字符。根据定义，行由一个换行符结尾。如果宿主操作系统使用不同的约定结束文本行，I/O 函数必须在这种形式和文本行的内部形式之间进行翻译转换。

FILE 是一种数据结构，用于管理缓冲区和存储流的 I/O 状态。运行时环境为每个程序提供了三个流——标准输入、标准输出和标准错误。最常见的情况是把标准输入缺省设置为键盘，其他两个流缺省设置为显示器。错误信息使用一个单独的流，这样即使标准输出的缺省值重定向为其他位置，错误信息仍能够显示在它的缺省位置。FOPEN_MAX 是你能够同时打开的最多文件数，具体数目因编译器而异，但不能小于 8。FILENAME_MAX 是用于存储文件名的字符数组的最大限制长度。如果不存在长度限制，这个值就是推荐最大长度。

为了对一个文件执行流 I/O 操作，首先必须用 fopen 函数打开文件，它返回一个指向 FILE 结构的指针，这个 FILE 结构指派给进行操作的流。这个指针必须在一个 FILE *类型的变量中保存。然后，这个文件就可以进行读取和（或）写入。读写完毕后，应该关闭文件。许多 I/O 函数属于同一个家族，它们在本质上执行相同的任务，但在从何处读取或何处写入方面存在一些微小的差别。通常一个函数家族的各个变型包括接受一个流参数的函数，一个只用于标准流之一的函数以及一个使用内存中的缓冲区而不是流的函数。

流用 fopen 函数打开。它的参数是需要打开的文件名和需要采用的流模式。模式指定流用于读取、写入还是添加，它同时指定流为二进制流还是文本流。freopen 函数用于执行相同的任务，但你可以自己指定需要使用的流。这个函数最常用于重新打开一个标准流。你应该始终检查 fopen 或 freopen 函数的返回值，看看有没有发生错误。在结束了一个流的操作之后，你应该使用 fclose 函数将它关闭。

逐字符的 I/O 由 getchar 和 putchar 函数家族实现。输入函数 fgetc 和 getc 都接受一个流参数，getchar 则只从标准输入读取。第 1 个以函数的方式实现，后两个则以宏的方式实现。它们都返回一个用整型值表示的单字符。除了用于执行输出而不是输入之外，fputc、putc 和 putchar 函数具有和对应的输入函数相同的属性。ungetc 用于把一个不需要的字符退回到流中。这个被退回的字符将是

bbs.theithome.com

下一个输入操作所返回的第 1 个字符。改变流的位置（定位）将导致这个退回的字符被丢弃。

行 I/O 既可以是格式化的，也可以是未格式化的。gets 和 puts 函数家族执行未格式化的行 I/O。fgets 和 gets 都从一个指定的缓冲区读取一行。前者接受一个流参数，后者从标准输入读取。fgets 函数更为安全，它把缓冲区长度作为参数之一，因此可以保证一个长输入行不会溢出缓冲区。而且数据并不会丢失——长输入行的剩余部分（超出缓冲区长度的那部分）将被 fgets 函数的下一次调用读取。fputs 和 puts 函数把文本写入到流中。它们的接口类似对应的输入函数。为了保证向后兼容，gets 函数将去除它所读取的行的换行符，puts 函数在写入到缓冲区的文本后面加上一个换行符。

scanf 和 printf 函数家族执行格式化的 I/O 操作。输入函数共有三种，fscanf 接受一个流参数，scanf 从标准输入读取，sscanf 从一个内存中的缓冲区接收字符。printf 家族也有三个函数，它们的属性也类似。scanf 家族的函数根据一个格式字符串对字符进行转换。一个指针参数列表用于提示结果值的存储地点。函数的返回值是被转换的值的个数，如果没有任何值被转换就遇到文件尾，函数就返回 EOF。printf 家族的函数根据一个格式字符串把值转换为字符形式。这些值是作为参数传递给函数的。

使用二进制流写入二进制数据（如整数和浮点数）比使用字符 I/O 效率更高。二进制 I/O 直接读写值的各个位，而不必把值转换为字符。但是，二进制输出的结果非人眼所能阅读。fread 和 fwrite 函数执行二进制 I/O 操作。每个函数都接受 4 个参数：指向缓冲区的指针、缓冲区中每个元素的长度、需要读取或写入的元素个数以及需要操作的流。

在缺省情况下，流是顺序读取的。但是，你可以通过在读取或写入之前定位到一个不同的位置实现随机 I/O 操作。fseek 函数允许你指定文件中的一个位置，它用一个偏移量表示，参考位置可以是文件起始位置，也可以是文件当前位置，还可以是文件的结尾位置。ftell 函数返回文件的当前位置。fsetpos 和 fgetpos 函数是前两个函数的替代方案。但是，fsetpos 函数的参数只有当它是先前从一个作用于同一个流的 fgetpos 函数的返回值时才是合法的。最后，rewind 函数返回到文件的起始位置。

在执行任何流操作之前，调用 setbuf 函数可以改变流所使用的缓冲区。用这种方式指定一个缓冲区可以防止系统为流动态分配一个缓冲区。向这个函数传递一个 NULL 指针作为缓冲区参数表示禁止使用缓冲区。setvbuf 函数更为通用。使用它，你可以指定一个并非标准长度的缓冲区。你也可以选择你所希望的缓冲方式：全缓冲、行缓冲或不缓冲。

ferror 和 clearerr 函数和流的错误状态有关，也就是说，是否出现了任何读/写错误。第 1 个函数返回错误状态，第 2 个函数重置错误状态。如果流当前位于文件的末尾，那么 feof 函数就返回真。

tmpfile 函数返回一个与一个临时文件关联的流。当流被关闭之后，这个文件被自动删除。tmpnam 函数为临时文件创建一个合适的文件名。这个名字不会与现存的文件名冲突。把文件名作为参数传递给 remove 函数可以删除这个文件。rename 函数用于修改一个文件的名字。它接受两个参数，文件的当前名字和文件的新名字。

15.18　警告的总结

1. 忘了在一条调试用的 printf 语句后面跟一个 fflush 调用。
2. 不检查 fopen 函数的返回值。
3. 改变文件的位置将丢弃任何被退回到流的字符。
4. 在使用 fgets 时指定太小的缓冲区。
5. 使用 gets 的输入溢出缓冲区且未被检测到。

bbs.theithome.com

6. 使用任何 scanf 系列函数时，格式代码和参数指针类型不匹配。

7. 在任何 scanf 系列函数的每个非数组、非指针参数前忘了加上&符号。

8. 注意在使用 scanf 系列函数转换 double、long double、short 和 long 整型时，在格式代码中加上合适的限定符。

9. sprintf 函数的输出溢出了缓冲区且未检测到。

10. 混淆 printf 和 scanf 格式代码。

11. 使用任何 printf 系列函数时，格式代码和参数类型不匹配。

12. 在有些长整数长于普通整数的机器上打印长整数值时，忘了在格式代码中指定 l 修改符。

13. 使用自动数组作为流的缓冲区时应多加小心。

15.19 编程提示的总结

1. 在可能出现错误的场合，检查并报告错误。
2. 操纵文本行而无需顾及它们的外部表示形式这个能力有助于提高程序的可移植性。
3. 使用 scanf 限定符提高可移植性。
4. 当你打印长整数时，即使你所使用的机器并不需要，坚持使用 l 修改符可以提高可移植性。

15.20 问题

1. 如果对 fopen 函数的返回值不进行错误检查可能会出现什么后果？
2. 如果试图对一个从未打开过的流进行 I/O 操作会发生什么情况？
3. 如果一个 fclose 调用失败，但程序并未对它的返回值进行错误检查可能会出现什么后果？
4. 如果一个程序在执行时它的标准输入已重定向到一个文件，程序如何检测到这个情况？
5. 如果调用 fgets 函数时使用一个长度为 1 的缓冲区会发生什么？长度为 2 呢？
6. 为了保证下面这条 sprintf 语句所产生的字符串不溢出，缓冲区至少应该有多大？假定你的机器的上整数的长度为 2 个字节。

```
sprintf( buffer, "%d %c %x", a, b, c );
```

7. 为了保证下面这条 sprintf 语句所产生的字符串不溢出，缓冲区至少应该有多大？

```
sprintf( buffer, "%s", a );
```

8. %f 格式代码所打印的最后一位数字是经过四舍五入呢？还是未打印的数字被简单地截掉？
9. 你如何得到 perror 函数可能打印的所有错误信息列表？
10. 为什么 fprintf、fscanf、fputs 和 fclose 函数都接受一个指向 FILE 结构的指针作为参数而不是 FILE 结构本身。
11. 你希望打开一个文件进行写入，假定（1）你不希望文件原先的内容丢失，（2）你希望能够写入到文件的任何位置。那么你该怎样设置打开模式呢？
12. 为什么需要 freopen 函数？

bbs.theithome.com

13. 对于绝大多数程序，你觉得有必要考虑 fgetc(stdin)或 getchar 哪个更好吗？
14. 在你的系统上，下面的语句将打印什么内容？

```
printf("%d\n", 3.14 );
```

15. 请解释使用%-6.10s 格式代码将打印出什么形式的字符串。
16. 当一个特定的值用格式代码%.3f 打印时，其结果是 1.405。但这个值用格式代码%.2f 打印时，其结果是 1.40。似乎出现了明显错误，请解释其原因。

15.21 编程练习

★1. 编写一个程序，把标准输入的字符逐个复制到标准输出。

★2. 修改你对练习 1 的解决方案，使它每次读写一整行。你可以假定文件中每一行所包含的字符数不超过 80 个（不包括结尾的换行符）。

★★3. 修改你对练习 2 的解决方案，去除每行 80 个字符的限制。处理这个文件时，你仍应该每次处理一行，但对于那些长于 80 个字符的行，你可以每次处理其中的一段。

★★★4. 修改你对练习 3 的解决方案，提示用户输入两个文件名，并从标准输入读取它们。第 1 个作为输入文件，第 2 个作为输出文件。这个修改后的程序应该打开这两个文件并把输入文件的内容按照前面的方式复制到输出文件。

★★★5. 修改你对练习 4 的解决方案，使它寻找那些以一个整数开始的行。这些整数值应该进行求和，其结果应该写入到输出文件的末尾。除了这个修改之外，这个修改后的程序的其他部分应该和练习 4 一样。

★★6. 在第 9 章，你编写了一个称为 palindrome 的函数，用于判断一个字符串是否是一个回文。在这个练习中，你需要编写一个函数，判断一个整型变量的值是不是回文。例如，245 不是回文，但 14741 却是回文。这个函数的原型应该如下：

```
int  numeric_palindrome( int value );
```

如果 value 是回文，函数返回真，否则返回假。

★★★7. 某个数据文件包含了家庭成员的年龄。一个家庭各个成员的年龄都位于同一行，由空格分隔。例如，下面的数据

```
45  42  22
36  35  7  3  1
22   20
```

描述了三个家庭的所有成员的年龄，它们分别有 3 个、5 个和 2 个成员。

编写一个程序，计算用这种文件表示的每个家庭所有成员的平均年龄。程序应该用格式代码%5.2f 打印出平均年龄，后面是一个冒号和输入数据。你可以假定每个家庭的成员数量都不超过 10 个。

★★★★8. 编写一个程序，产生一个文件的十六进制倾印码(dump)。它应该从命令行接受单个参数，也就是需要进行倾印的文件名。如果命令行中未给出参数，程序就打印标准输入的倾印码。

倾印码的每行都应该具有下面的格式。

bbs.theithome.com

列	内　　容
1-6	文件的当前偏移位置，用十六进制表示，前面用零填充
9-43	文件接下来 16 个字节的十六进制表示形式。它们分成 4 组，每组由 8 个十六进制数字组成，每组之间以一个空格分隔
46	一个星号
47-62	文件中上述 16 个字节的字符表示形式。如果某个字符是不可打印字符或空白，就打印一个句点
63	一个星号

所有的十六进制数应该使用大写的 A-F 而不是小写的 a-f。

下面是一些样例行，用于说明这种格式。

```
000200  D405C000 82102004 91D02000 9010207F  *...... ... ... .*
000210  82102001 91D02000 0001C000 2F757372  *.. ... ...../usr*
000220  2F6C6962 2F6C642E 736F002F 6465762F  */lib/ld.so./dev/*
```

★★★ 9. UNIX 的 fgrep 程序从命令行接受一个字符串和一系列文件名作为参数。然后，它逐个查看每个文件的内容。对于文件中每个包含命令行中给定字符串的文本行，程序将打印出它所在的文件名、一个冒号和包含该字符串的行。

编写这个程序。首先出现的是字符串参数，它不包含任何换行字符。然后是文件名参数。如果没有给出任何文件名，程序应该从标准输入读取。在这种情况下，程序所打印的行不包括文件名和冒号。你可以假定各文件所有文本行的长度都不会超过 510 个字符。

★★★★ 10. 编写一个程序，计算文件的检验和(checksum)。该程序按照下面的方式进行调用：

```
$ sum [ -f ] [ file ... ]
```

其中，-f 选项是可选的。稍后我将描述它的含义。

接下来是一个可选的文件名列表，如果未给出任何文件名，程序就处理标准输入。否则，程序根据各个文件在命令行中出现的顺序逐个对它们进行处理。“处理文件”就是计算和打印文件的检验和。

计算检验和的算法是很简单的。文件中的每个字符都和一个 16 位的无符号整数相加，其结果就是检验和的值。不过，虽然它很容易实现，但这个算法可不是个优秀的错误检测方法。在文件中对两个字符进行互换将不会被这种方法检测出是个错误。

正常情况下，当到达每个文件的文件尾时，检验和就写入到标准输出。如果命令行中给出了-f 选项，检验和就写入到一个文件而不是标准输出。如果输入文件的名字是 file，那么这个输出文件的名字应该是 file.cks。当程序从标准输入读取时，这个选项是非法的，因为此时并不存在输入文件名。

下面是这个程序运行的几个例子。它们在那些使用 ASCII 字符集的系统中是有效的。文件 hw 包含了文本行“Hello, World!”，后面跟一个换行符。文件 hw2 包含了两个这样的文本行。所有的输入都不包含任何缀尾的空格或制表符。

```
$ sum
hi
^D
```


bbs.theithome.com

```
219
$ sum hw
1095
$ sum -f
-f illegal when reading standard input
$ sum -f hw2
$
```

(File hw2.cks now contains 2190)

★★★★★ 11. 编写一个程序，保存零件和它们的价值的存货记录。每个零件都有一份描述信息，其长度为 1～20 个字符。当一个新零件被添加到存货记录文件时，程序将下一个可用的零件号指定给它。第 1 个零件的零件号为 1。程序应该存储每个零件的当前数量和总价值。

这个程序应该从命令行接受单个参数，也就是存货记录文件的名字。如果这个文件并不存在，程序就创建一个空的存货记录文件。然后程序要求用户输入需要处理的事务类型并逐个对它们进行处理。

程序允许处理下列交易。

```
new description, quantity, cost-each
```

new 交易向系统添加一个新零件。descrption 是该零件的描述信息，它的长度不超过 20 个字符。quantity 是保存到存货记录文件中该零件的数量，它不可以是个负数。cost-each 是每个零件的单价。一个新零件的描述信息如果和一个现有的零件相同并不是错误。程序必须计算和保存这些零件的总价值。对于每个新增加的零件，程序为其指定下一个可用的零件号。零件号从 1 开始，线性递增。被删除零件的零件号可以重新分配给新添加的零件。

```
buy part-number, quantity, cost-each
```

buy 交易为存货记录中一个现存的零件增加一定的数量。part-number 是该零件的零件号，quantity 是购入的零件数量（它不能是负数），cost-each 是每个零件的单价。程序应该把新的零件数量和总价值添加到原先的存货记录中。

```
sell part-number, quantity, price-each
```

sell 交易从存货记录中一个现存的零件减去一定的数量。part-number 是该零件的零件号，quantity 是出售的零件数量（它不能是负数，也不能超过该零件的现有数量），price-each 是每个零件出售所获得的金额。程序应该从存货记录中减去这个数量，并减少该零件的总价值。然后，它应该计算销售所获得的利润，也就是零件的购买价格和零件的出售价格之间的差价。

```
delete part-number
```

这个交易从存货记录文件中删除指定的零件。

```
print part-number
```

这个交易打印指定零件的信息，包括描述信息、现存数量和零件的总价值。

```
print all
```

这个交易以表格的形式打印记录中所有零件的信息。

```
total
```


bbs.theithome.com

```
* 这个交易计算和打印记录中所有零件的总价值。
    end
```

这个交易终止程序的执行。

当零件以不同的购买价格获得时，计算存货记录的真正价值将变得很复杂，而且取决于首先使用的是最便宜的零件还是最昂贵的零件。这个程序所使用的方法比较简单：只保存每种零件的总价值，每种零件的单价被认为是相等的。例如，假定 10 个纸夹原先以每个$1.00 的价格购买。这个零件的总价值便是$10.00。以后，又以每个$1.25 的价格购入另外 10 个纸夹，这样这个零件的总价值便成了$22.50。此时，每个纸夹的当前单价便是$1.125。存货记录并不保存每批零件的购买记录，即使它们的购买价格不同。当纸夹出售时，利润根据上面计算所得的当前单价进行计算。

这里有一些关于设计这个程序的提示。首先，使用零件号判断存货记录文件中一个零件的写入位置。第 1 个零件号是 1，这样记录文件中零件号为 0 的位置可以用于保存一些其他信息。其次，你可以在删除零件时把它的的描述信息设置为空字符串，便于以后检测该零件是否已被删除。

bbs.theithome.com

第16章

标准函数库

标准函数库是一个工具箱，它极大地扩展了C程序员的能力。但是，在你使用这个能力之前，你必须熟悉库函数。忽略函数库相当于你只学习怎样使用油门、方向盘和刹车来开车，却不想费神学习使用自动恒速器、收音机和空调。虽然你仍然能够驾车到达你想去的地方，但过程要艰难一些，乐趣也要少很多。

本章描述前面章节未曾覆盖的一些库函数。各小节的标题中包括了获得这些函数原型必须用#include指令包含的文件名。

16.1 整型函数

这组函数返回整型值。这些函数分为三类：算术、随机数和字符串转换。

16.1.1 算术 <stdlib.h>

标准函数库包含了4个整型算术函数。

```
int abs( int value );
long int labs( long int value );
div_t div( int numerator, int denominator );
ldiv_t ldiv( long int numer, long int denom );
```

abs函数返回它的参数的绝对值。如果其结果不能用一个整数表示，这个行为是未定义的。labs用于执行相同的任务，但它的作用对象是长整数。

div函数把它的第2个参数（分母）除以第1个参数（分子），产生商和余数，用一个div_t结构返回。这个结构包含下面两个字段，

```
int  quot;   //  商
int  rem;   //  余数
```

但这两个字段并不一定以这个顺序出现。如果不能整除，商将是所有小于代数商的整数中最靠近它的那个整数。注意/操作符的除法运算结果并未精确定义。当/操作符的任何一个操作数为负而不能整除时，到底商是最大的那个小于等于代数商的整数还是最小的那个大于等于代数商的整数，这取决于编译器。ldiv所执行的任务和div相同，但它作用于长整数，其返回值是一

bbs.theithome.com

个 ldiv_t 结构。

16.1.2 随机数<stdlib.h>

有些程序每次执行时不应该产生相同的结果，如游戏和模拟，此时随机数就非常有用。下面两个函数合在一起使用能够产生**伪随机数**(pseudo-random number)。之所以如此称呼是因为它们通过计算产生随机数，因此有可能重复出现，所以并不是真正的随机数。

```
int  rand( void );
void  srand( unsigned int seed );
```

rand 返回一个范围在 0 和 RAND_MAX（至少为 32,767）之间的伪随机数。当它重复调用时，函数返回这个范围内的其他数。为了得到一个更小范围的伪随机数，首先把这个函数的返回值根据所需范围的大小进行取模，然后通过加上或减去一个偏移量对它进行调整。

为了避免程序每次运行时获得相同的随机数序列，我们可以调用 srand 函数。它用它的参数值对随机数发生器进行初始化。一个常用的技巧是使用每天的时间作为随机数产生器的**种子**(seed)，如下面的程序所示：

```
srand( (unsigned int)time( 0 ) );
```

time 函数将在本章后面描述。

程序 16.1 中的函数使用整数来表示游戏用的牌并使用随机数在“牌桌”上“洗”指定数目的牌。

```
/*
** 使用随机数在牌桌上洗“牌”。第 2 个参数指定牌的数字。当这个函数第 1 次调用
** 时，调用 srand 函数初始化随机数发生器。
*/
#include <stdlib.h>
#include <time.h>
#define   TRUE      1
#define   FALSE     0

void shuffle( int *deck, int n_cards )
{
      int i;
      static   int  first_time = TRUE;

/*
** 如果尚未进行初始化，用当天的当前时间作为随机数发生器。
**
*/
if( first_time ){
        first_time = FALSE;
        srand( (unsigned int)time( NULL ) );
}

 /*
** 通过交换随机对的牌进行“洗牌”。
*/
for( i = n_cards - 1; i > 0; i -= 1 ){
         int   where;
         int   temp;

         where = rand() % i;
```


bbs.theithome.com

```
            temp = deck[ where ];
            deck[ where ] = deck[ i ];
            deck[ i ] = temp;
        }
    }
```

程序 16.1　用随机数洗牌　　shuffle.c

16.1.3　字符串转换 <stdlib.h>

字符串转换函数把字符串转换为数值。其中最简单的函数 atoi 和 atol，执行基数为 10 的转换。strtol 和 strtoul 函数允许你在转换时指定基数，同时它们还允许你访问字符串的剩余部分。

```
int atoi( char const *string );
long int atol( char const *string );
long int strtol( char const *string, char **unused, int base );
unsigned long int strtoul( char const *string, char **unused,
    int base );
```

如果任何一个上述函数的第 1 个参数包含了前导空白字符，它们将被跳过。然后函数把合法的字符转换为指定类型的值。如果存在任何非法缀尾字符，它们也将被忽略。

atoi 和 atol 分别把字符转换为整数和长整数值。strtol 和 atol 同样把参数字符串转换为 long。但是，strtol 保存一个指向转换值后面第 1 个字符的指针。如果函数的第 2 个参数并非 NULL，这个指针便保存在第 2 个参数所指向的位置。这个指针允许字符串的剩余部分进行处理而无需推测转换在字符串的哪个位置终止。strtoul 和 strtol 的执行方式相同，但它产生一个无符号长整数。

这两个函数的第 3 个参数是转换所执行的基数。如果基数为 0，任何在程序中用于书写整数字面值的形式都将被接受，包括指定数字基数的形式，如 0x2af4 和 0377。否则，基数值应该在 2 到 36 的范围内——然后转换根据这个给定的基数进行。对于基数 11 到 36，字母 A 到 Z 分别被解释为数值 10 到 35。在这个上下文环境中，小写字母 a-z 被解释为与对应的大写字母相同的意思。因此，

```
x = strtol("   590bear", next, 12 );
```

的返回值为 9947，并把一个指向字母 e 的指针保存在 next 所指向的变量中。转换在 b 处终止，因为在基数为 12 时 e 不是一个合法的数字。

如果这些函数的 string 参数中并不包含一个合法的数值，函数就返回 0。如果被转换的值无法表示，函数便在 errno 中存储 ERANGE 这个值，并返回表 16.1 中的一个值。

表 16.1　　strtol 和 strtoul 返回的错误值

函　数	返　回　值
strtol	如果值太大且为负数，返回 LONG_MIN。如果值太大且为正数，返回 LONG_MAX
strtoul	如果值太大，返回 ULONG_MAX

16.2　浮点型函数

头文件 math.h 包含了函数库中剩余的数学函数的声明。这些函数的返回值以及绝大多数参数都是 double 类型。

警告：

一个常见的错误就是在使用这些函数时忘了包含这个头文件，如下所示：

bbs.theithome.com

```
double  x;
x = sqrt( 5.5 );
```

编译器在此之前未曾见到过 sqrt 函数的原型，因此错误地假定它返回一个整数，然后错误地把这个值的类型转换为 double。这个结果值是没有意义的。

如果一个函数的参数不在该函数的定义域之内，称为**定义域错误**(domain error)。例如：

```
sqrt( -5.0 );
```

就是个定义域错误，因为负值的平方根是未定义的。当出现一个定义域错误时，函数返回一个由编译器定义的错误值，并且在 errno 中存储 EDOM 这个值。如果一个函数的结果值过大或过小，无法用 double 类型表示，这称为**范围错误**(range error)。例如：

```
exp( DBL_MAX )
```

将产出一个范围错误，因为它的结果值太大。在这种情况下，函数将返回 HUGE_VAL，它是一个在 math.h 中定义的 double 类型的值。如果一个函数的结果值太小，无法用一个 double 表示，函数将返回 0。这种情况也属于范围错误，但 errno 会不会设置为 ERANGE 则取决于编译器。

16.2.1 三角函数 <math.h>

标准函数库提供了常见的三角函数。

```
double sin( double angle );
double cos( double angle );
double tan( double angle );
double asin( double value );
double acos( double value );
double atan( double value );
double atan2( double x, double y );
```

sin、cos 和 tan 函数的参数是一个用弧度表示的角度，这些函数分别返回这个角度的正弦、余弦和正切值。

asin、acos 和 atan 函数分别返回它们的参数的反正弦、反余弦和反正切值。如果 asin 和 acos 的参数并不位于-1 和 1 之间，就出现一个定义域错误。asin 和 atan 的返回值是范围在$-\pi/2$ 和 $\pi/2$ 之间的一个弧度，acos 的返回值是一个范围在 0 和 π 之间的一个弧度。

atan2 函数返回表达式 y/x 的反正切值，但它使用这两个参数的符号来决定结果值位于哪个象限。它的返回值是一个范围在$-\pi$ 和 π 之间的弧度。

16.2.2 双曲函数 <math.h>

```
double sinh( double angle );
double cosh( double angle );
double tanh( double angle );
```

这些函数分别返回它们的参数的双曲正弦、双曲余弦和双曲正切值。每个函数的参数都是一个以弧度表示的角度。

16.2.3 对数和指数函数 <math.h>

标准函数库存在一些直接处理对数和指数的函数。

bbs.theithome.com

```
double exp( double x );
double log( double x );
double log10( double x );
```

exp 函数返回 e 值的 x 次幂，也就是 e^x。

log 函数返回 x 以 e 为底的对数，也就是常说的自然对数。log10 函数返回 x 以 10 为底的对数。注意 x 以任意一个以 b 为底的对数可以通过下面的公式进行计算：

$$\log b^x = \frac{\log e^x}{\log e^b}$$

如果它们的参数为负数，两个对数函数都将出现定义域错误。

16.2.4　浮点表示形式 <math.h>

这三个函数提供了一种根据一个编译器定义的格式存储一个浮点值的方法。

```
double frexp( double value, int *exponent );
double ldexp( double fraction, int exponent );
double modf( double value, double *ipart );
```

frexp 函数计算一个**指数**(exponent)和**小数**(fraction)，这样 fraction × $2^{exponent}$ = value，其中 0.5 ≤ fraction < 1，exponent 是一个整数。exponent 存储于第 2 个参数所指向的内存位置，函数返回 fraction 的值。与它相关的函数 ldexp 的返回值是 fraction × $2^{exponent}$，也就是它原先的值。当你必须在那些浮点格式不兼容的机器之间传递浮点数时，这些函数是非常有用的。

modf 函数把一个浮点值分成整数和小数两个部分，每个部分都具有和原值一样的符号。整数部分以 double 类型存储于第 2 个参数所指向的内存位置，小数部分作为函数的返回值返回。

16.2.5　幂 <math.h>

这个家族共有两个函数。

```
double pow( double x, double y );
double sqrt( double x );
```

pow 函数返回 x^y 的值。由于在计算这个值时可能要用到对数，所以如果 x 是一个负数且 y 不是一个整数，就会出现一个定义域错误。

sqrt 函数返回其参数的平方根。如果参数为负，就会出现一个定义域错误。

16.2.6　底数、顶数、绝对值和余数 <math.h>

这些函数的原型如下所示。

```
double floor( double x );
double ceil( double x );
double fabs( double x );
double fmod( double x, double y );
```

floor 函数返回不大于其参数的最大整数值。这个值以 double 的形式返回，这是因为 double 能够表示的范围远大于 int。ceil 函数返回不小于其参数的最小整数值。

fabs 函数返回其参数的绝对值。fmod 函数返回 x 除以 y 所产生的余数，这个除法的商被限制为一个整数值。

bbs.theithome.com

16.2.7 字符串转换 <stdlib.h>

这些函数和整型字符串转换函数类似，只不过它们返回浮点值。

```
double atof( char const *string );
double strtod( char const *string, char **unused );
```

如果任一函数的参数包含了前导的空白字符，这些字符将被忽略。函数随后把合法的字符转换为一个 double 值，忽略任何缀尾的非法字符。这两个函数都接受程序中所有浮点数字面值的书写形式。

strtod 函数把参数字符串转换为一个 double 值，其方法和 atof 类似，但它保存一个指向字符串中被转换的值后面的第 1 个字符的指针。如果函数的第 2 个参数不是 NULL，那么这个被保存的指针就存储于第 2 个参数所指向的内存位置。这个指针允许对字符串的剩余部分进行处理，而不用猜测转换会在字符串中的什么位置结束。

如果这两个函数的字符串参数并不包含任何合法的数值字符，函数就返回零。如果转换值太大或太小，无法用 double 表示，那么函数就在 errno 中存储 ERANGE 这个值，如果值太大（无论是正数还是负数），函数返回 HUGE_VAL。如果值太小，函数返回零。

16.3 日期和时间函数

函数库提供了一组非常丰富的函数，用于简化日期和时间的处理。它们的原型位于 time.h。

16.3.1 处理器时间 <time.h>

clock 函数返回从程序开始执行起处理器所消耗的时间。

```
clock_t  clock( void );
```

注意这个值可能是个近似值。如果需要更精确的值，你可以在 main 函数刚开始执行时调用 clock，然后把以后调用 clock 时所返回的值减去前面这个值。如果机器无法提供处理器时间，或者如果时间值太大，无法用 clock_t 变量表示，函数就返回-1。

clock 函数返回一个数字，它是由编译器定义的。通常它是处理器时钟滴答的次数。为了把这个值转换为秒，你应该把它除以常量 CLOCKS_PER_SEC。

警告：

在有些编译器中，这个函数可能只返回程序所使用的处理器时间的近似值。如果宿主操作系统不能追踪处理器时间，函数可以返回已经流逝的实际时间数量。在有些一次不能运行超过一个程序的简单操作系统中，就可能出现这种情况。本章的练习之一就是探索如何判断你的系统在这方面的表现方式。

16.3.2 当天时间 <time.h>

time 函数返回当前的日期和时间。

```
time_t  time( time_t *returned_value );
```

如果参数是一个非 NULL 的指针，时间值也将通过这个指针进行存储。如果机器无法提供当前

bbs.theithome.com

的日期和时间，或者时间值太大，无法用 time_t 变量表示，函数就返回-1。

标准并未规定时间的编码方式，所以你不应该使用字面值常量，因为它们在不同的编译器中可能具有不同的含义。一种常见的表示形式是返回从一个任意选定的时刻开始流逝的秒数。在 MS-DOS 和 UNIX 系统中，这个时刻是 1970 年 1 月 1 日 00:00:00[1]。

警告：

调用 time 函数两次并把两个值相减，由此判断期间所流逝的时间是很有诱惑力的。但这个技巧是很危险的，因为标准并未要求函数的结果值用秒来表示。difftime 函数（下一节描述）可以用于这个目的。

日期和时间的转换 <time.h>

下面的函数用于操纵 time_t 值。

```
char *ctime( time_t const *time_value );
double difftime( time_t time1, time_t time2 );
```

ctime 函数的参数是一个指向 time_t 的指针，并返回一个指向字符串的指针，字符串的格式如下所示：

```
Sun Jul 4 04:02:48 1976\n\0
```

字符串内部的空格是固定的。一个月的每一天总是占据两个位置，即使第 1 个是空格。时间值的每部分都用两个数字表示。标准并未提及存储这个字符串的内存类型，许多编译器使用一个静态数组。因此，下一次调用 ctime 时，这个字符串将被覆盖。因此，如果你需要保存它的值，应该事先为其复制一份。注意 ctime 实际上可能以下面这种方式实现：

```
asctime( localtime( time_value ) );
```

difftime 函数计算 time1-time2 的差，并把结果值转换为秒。注意它返回的是一个 double 类型的值。

接下来的两个函数把一个 time_t 值转换为一个 tm 结构，后者允许我们很方便地访问日期和时间的各个组成部分。

```
struct  tm *gmtime( time_t  const *time_value );
struct  tm *localtime( time_t  const *time_value );
```

gmtime 函数把时间值转换为**世界协调时间**(Coordinated Universal Time, UTC)。UTC 以前被称为**格林尼治标准时间**(Greenwich Mean Time)，这也是 gmtime 这个名字的来历。正如其名字所提示的那样，localtime 函数把一个时间值转换为当地时间。标准包含了这两个函数，但它并没有描述 UTC 和当地时间的实现之间的关系。

tm 结构包含了表 16.2 所列出的字段，不过这些字段在结构中出现的顺序并不一定如此。

警告：

使用这些值最容易出现的错误就是错误地解释月份。这些值表示从 1 月开始的月份，所以 0 表示 1 月，11 表示 12 月。尽管初看上去很不符合直觉，这种编号方式被证明是一种行之有效的月份

[1] 在许多编译器中，time_t 被定义为一个有符号的 32 位量。2038 年应该是比较有趣的：从 1970 年开始计数的秒数将在该年溢出 time_t 变量。

bbs.theithome.com

编码方式，因为它允许你把这些值作为下标值使用，访问一个包含月份名称的数组。

表 16.2 tm 结构的字段

类型 & 名称	范　围	含　义
int tm_sec;	0-61	分之后的秒数*
int tm_min;	0-59	小时之后的分数
int tm_hour;	0-23	午夜之后的小时数
int tm_mday;	1-31	当月的日期
int tm_mon;	0-11	1 月之后的月数
int tm_year;	0-??	1900 之后的年数
int tm_wday;	0-6	星期天之后的天数
int tm_yday;	0-365	1 月 1 日之后的天数
int tm_isdat;		夏令时标志

* 我们必须赞美制订 C++标准的 ANSI 标准委员会考虑之周详，它允许偶尔出现的“闰秒”加到每年的最后一分钟，对我们的时间标准进行调整，以适应地球旋转的细微变慢现象。

警告：

接下来一个常见的错误就是忘了 tm_year 这个值只是 1900 年之后的年数。为了计算实际的年份，这个值必须与 1900 相加。

当你拥有了一个 tm 结构之后，你既可以直接使用它的值，也可以把它作为参数传递给下面的函数之一。

```
char *asctime( struct tm const *tm_ptr );
size_t strftime( char *string, size_t maxsize, char const *format,
    struct tm const *tm_ptr );
```

asctime 函数把参数所表示的时间值转换为一个以下面的格式表示的字符串：

```
Sun Jul 4 04:02:48 1976\n\0
```

这个格式和 ctime 函数所使用的格式一样，后者在内部很可能调用了 asctime 来实现自己的功能。

strftime 函数把一个 tm 结构转换为一个根据某个格式字符串而定的字符串。这个函数在格式化日期方面提供了令人难以置信的灵活性。如果转换结果字符串的长度小于 maxsize 参数，那么该字符串就被复制到第 1 个参数所指向的数组中，strftime 函数返回字符串的长度。否则，函数返回-1 且数组的内容是未定义的。

格式字符串包含了普通字符和格式代码。普通字符被复制到它们原先在字符串中出现的位置。格式代码则被一个日期或时间值代替。格式代码包括一个%字符，后面跟一个表示所需值的字符。表 16.3 列出了已经实现的格式代码。如果%字符后面是一个其他任何字符，其结果是未定义的，这就允许各个编译器自由地定义额外的格式代码。你应该避免使用这种自定义的格式代码，除非你不怕牺牲代码的可移植性。特定于 locale 的值由当前的 locale 决定，它将在本章的后面讨论。%U 和%W 代码基本相同，区别在于前者把当年的第 1 个星期日作为第 1 个星期的开始而后者把当年的第 1 个星期一作为第 1 个星期的开始。如果无法判断时区，%Z 代码就由一个空字符串代替。

bbs.theithome.com

表 16.3　strftime 格式代码

代　码	被...代替
%%	一个%字符
%a	一星期的某天，以当地的星期几的简写形式表示
%A	一星期的某天，以当地的星期几的全写形式表示
%b	月份，以当地月份名的简写形式表示
%B	月份，以当地月份名的全写形式表示
%c	日期和时间，使用%x %X
%d	一个月的第几天(01-31)
%H	小时，以 24 小时的格式(00-23)
%I	小时，以 12 小时的格式(00-12)
%J	一年的第几天(001-366)
%M	月数(01-12)
%M	分钟（00～59）
%P	AM 或 PM（不论哪个合适）的当地对等表示形式
%S	秒(00-61)
%U	一年的第几星期(00-53)，以星期日为第 1 天
%w	一星期的第几天，星期日为第 0 天
%W	一年的第几星期(00-53)，以星期一为第 1 天
%x	日期，使用本地的日期格式
%X	时间，使用本地的时间格式
%y	当前世纪的年份(00-99)
%Y	年份的全写形式(例如，1984)
%Z	时区的简写

最后，mktime 函数用于把一个 tm 结构转换为一个 time_t 值。

```
time_t  mktime( struct tm *tm_ptr );
```

tm 结构中 tm_wday 和 tm_yday 的值被忽略，其他字段的值也无需限制在它们的通常范围内。在转换之后，该 tm 结构会进行规格化，因此 tm_wday 和 tm_yday 的值将是正确的，其余字段的值也都位于它们通常的范围之内。这个技巧是一种简单的用于判断某个特定的日期属于星期几的方法。

16.4　非本地跳转 <setjmp.h>

setjmp 和 longjmp 函数提供了一种类似 goto 语句的机制，但它并不局限于一个函数的作用域之内。这些函数常用于深层嵌套的函数调用链。如果在某个低层的函数中检测到一个错误，你可以立即返回到顶层函数，不必向调用链中的每个中间层函数返回一个错误标志。

为了使用这些函数，你必须包含头文件 setjmp.h。这两个函数的原型如下所示：

```
int  setjmp( jmp_buf  state );
void  longjmp( jump_buf  state, int value );
```


bbs.theithome.com

你声明一个 jmp_buf 变量，并调用 setjmp 函数对它进行初始化，setjmp 的返回值为零。setjmp 把程序的状态信息（例如，堆栈指针的当前位置和程序的计数器）保存到跳转缓冲区[1]。你调用 setjmp 时所处的函数便成为你的“顶层”函数。

以后，在顶层函数或其他任何它所调用的函数（不论是直接调用还是间接调用）内的任何地方调用 longjmp 函数，将导致这个被保存的状态重新恢复。longjmp 的效果就是使执行流通过再次从 setjmp 函数返回，从而立即跳回到顶层函数中。

你如何区别从 setjmp 函数的两种不同返回方式呢？当 setjmp 函数第 1 次被调用时，它返回 0。当 setjmp 作为 longjmp 的执行结果再次返回时，它的返回值是 longjmp 的第 2 个参数，它必须是个非零值。通过检查它的返回值，程序可以判断是否调用了 longjmp。如果存在多个 longjmp，也可以由此判断哪个 longjmp 被调用。

16.4.1 实例

程序 16.2 使用 setjmp 来处理它所调用的函数检测到的错误，但无需使用寻常的返回和检查错误代码的逻辑。setjmp 的第 1 次调用确立了一个地点，如果调用 longjmp，程序的执行流将在这个地点恢复执行。setjmp 的返回值为 0，这样程序便进入事务处理循环。如果 get_trans、process_trans 或其他任何被这些函数调用的函数检测到一个错误，它将像下面这样调用 longjmp：

```
longjmp( restart, 1 );
```

执行流将立即在 restart 这个地点重新执行，setjmp 的返回值为 1。

这个例子可以处理两种不同类型的错误：一种是阻止程序继续执行的致命错误；另一种是只破坏正在处理的事务的小错误。这个对 longjmp 的调用属于后者。当 setjmp 返回 1 时，程序就打印一条错误信息，并再次进入事务处理循环。为了报告一个致命错误，可以用任何其他值调用 longjmp，程序将保存它的数据并退出。

```
/*
** 一个说明 setjmp 用法的程序
*/
#include "trans.h"
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

/*
** 用于存储 setjmp 的状态信息的变量。
*/
jmp_buf   restart;

int
main()
{
     int  value;
     Trans     *transaction;

     /*
```

[1] 程序当前正在执行的指令的地址。

bbs.theithome.com

```
    ** 确立一个我们希望在 longjmp 的调用之后执行流恢复执行的地点。
    */
    value = setjmp( restart );

    /*
    ** 从 longjmp 返回后判断下一步执行什么。
    */
    switch( setjmp( restart ) ){
    default:
        /*
        **longjmp 被调用 -- 致命错误
        */
        fputs( "Fatal error.\n", stderr );
        break;

    case 1:
        /*
        **longjmp 被调用 -- 小错误
        */
        fputs( "Invalid transaction.\n", stderr );
        /* FALL THROUGH 并继续进行处理 */

    case 0:
        /*
        ** 最初从 setjmp 返回的地点: 执行正常的处理。
        */
        while( (transaction = get_trans()) != NULL )
                process_trans( transaction );

    }

    /*
    ** 保存数据并退出程序
    */
    write_data_to_file();

    return value == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

程序 16.2　setjmp 和 longjmp 实例　　setjmp.c

16.4.2　何时使用非本地跳转

setjmp 和 longjmp 并不是绝对必需的，因为你总是可以通过返回一个错误代码并在调用函数中对其进行检查来实现相同的效果。返回错误代码的方法有时候不是很方便，特别当函数已经返回了一些值的时候。如果存在一长串的函数调用链，即使只有最深层的那个函数发现了错误，调用链中的所有函数都必须返回并检查错误代码。在这种情况下使用 setjmp 和 longjmp 去除了中间函数的错误代码逻辑，从而对它们进行了简化。

警告：

当顶层函数（调用 setjmp 的那个）返回时，保存在跳转缓冲区的状态信息便不再有效。在此之

bbs.theithome.com

后调用 longjmp 很可能失败，而它的症状很难调试。这就是为什么 longjmp 只能在顶层函数或者在顶层函数所调用的函数中进行调用的原因。只有这个时候保存在跳转缓冲区的状态信息才是有效的。

提示：

由于 setjmp 和 longjmp 有效地实现了 goto 语句的功能，所以你在使用它们时必须遵循某些诫律。在程序 16.2 例子的情况下，这两个函数有助于编写更清晰、复杂度更低的代码。但是，如果 setjmp 和 longjmp 用于在一个函数内部模拟 goto 语句或者程序中存在许多执行流可能返回的跳转缓冲区时，那么程序的逻辑就会变得更加难以理解，程序将会变得更难调试和维护，另外失败的可能性也变得更大。你可以使用 setjmp 和 longjmp，但你应该合理地使用它们。

16.5 信号

程序中所发生的事件绝大多数都是由程序本身所引发的，例如执行各种语句和请求输入。但是，有些程序必须遇到的事件却不是程序本身所引发的。一个常见的例子就是用户中断了程序。如果部分计算好的结果必须进行保存以避免数据的丢失，程序必须预备对这类事件作出反应，虽然它并没有办法预测什么时候会发生这种情况。

信号就是用于这种目的。**信号**(signal)表示一种事件，它可能异步地发生，也就是并不与程序执行过程的任何事件同步。如果程序并未安排怎样处理一个特定的信号，那么当该信号出现时程序就作出一个缺省的反应。标准并未定义这个缺省反应是什么，但绝大多数编译器都选择终止程序。另外，程序可以调用 signal 函数，或者忽略这个信号，或者设置一个**信号处理函数**(signal handler)，当信号发生时程序就调用这个函数。

16.5.1 信号名 <signal.h>

表 16.4 列出了标准所定义的信号，但编译器并不需要实现所有这些信号，而且如果它觉得合适，也可以定义其他的信号。

SIGABRT 是一个由 abort 函数所引发的信号，用于终止程序。至于哪些错误将引发 SIGFPE 信号则取决于编译器。常见的有算术上溢或下溢以及除零错误。有些编译器对这个信号进行了扩展，提供了关于引发这个信号的操作的特定信息。使用这个信息可以允许程序对这个信号作出更智能的反应，但这样做将影响程序的可移植性。

表 16.4　　信　号

信　号	含　义
SIGABRT	程序请求异常终止
SIGFPE	发生一个算术错误
SIGILL	检测到非法指令
SIGSEGV	检测到对内存的非法访问
SIGINT	收到一个交互性注意信号
SIGTERM	收到一个终止程序的请求

SIGILL 信号提示 CPU 试图执行一条非法的指令。这个错误可能由于不正确的编译器设置所导致。例如，用 Intel 80386 指令编译一个程序，但把这个程序运行于一台 80286 计算机上。另一个可

bbs.theithome.com

能的原因是程序的执行流出现了错误，例如使用一个未初始化的函数指针调用一个函数，导致 CPU 试图执行实际上是数据的东西（把数据段当成了代码段）。SIGSEGV 信号提示程序试图非法访问内存。这个信号有两个最常见的原因，其中一个是程序试图访问未安装于机器上的内存或者访问操作系统未曾分配给这个程序的内存，另一个是程序违反了内存访问的边界要求。后者可能在那些要求数据边界对齐的机器上发生。例如，如果整数要求位于偶数的边界（存储的起始位置是编号为偶数的地址），一条指定在奇数边界访问一个整数的指令将违反边界规则。未初始化的指针常常会引起这类错误。

前面几个信号是同步的，因为它们都是在程序内部发生的。尽管你无法预测一个算术错误何时将会发生，如果你使用相同的数据反复运行这个程序，每次在相同的地方将出现相同的错误。最后两个信号，SIGINT 和 SIGTERM 则是异步的。它们在程序的外部产生，通常是由程序的用户所触发，表示用户试图向程序传达一些信息。

SIGINT 信号在绝大多数机器中都是当用户试图中断程序时发生的。SIGTERM 则是另一种用于请求终止程序的信号。在实现了这两个信号的系统里，一种常用的策略是为 SIGINT 定义一个信号处理函数，目的是执行一些日常维护工作(housekeeping)并在程序退出前保存数据。但是，SIGTERM 则不配备信号处理函数，这样当程序终止时便不必执行这些日常维护工作。

16.5.2　处理信号 <signal.h>

通常，我们关心的是怎样处理那些自主发生的信号，也就是无法预测其什么时候会发生的信号。raise 函数用于显式地引发一个信号。

```
int  raise( int sig );
```

调用这个函数将引发它的参数所指定的信号。程序对这类信号的反应和那些自主发生的信号是相同的。你可以调用这个函数对信号处理函数进行测试。但如果误用，它可能会实现一种非局部的 goto 效果，因此要避免以这样方式使用它。

当一个信号发生时，程序可以使用三种方式对它作出反应。缺省的反应是由编译器定义的，通常是终止程序。程序也可以指定其他行为对信号作出反应：信号可以被忽略，或者程序可以设置一个信号处理函数，当信号发生时调用这个函数。signal 函数用于指定程序希望采取的反应。

```
void ( *signal( int sig, void ( *handler )( int ) ) )( int );
```

这个函数的原型看上去有些吓人，所以让我们对它进行分析。首先，我将省略返回类型，这样我们可以先对参数进行研究：

```
signal( int sig, void ( *handler )( int ) )
```

第 1 个参数是表 16.4 所列的信号之一，第 2 个参数是你希望为这个信号设置的信号处理函数。这个处理函数是一个函数指针，它所指向的函数接受一个整型参数且没有返回值。当信号发生时，信号的代码作为参数传递给信号处理函数。这个参数允许一个处理函数处理几种不同的信号。

现在我将从原型中去掉参数，这样函数的返回类型看上去就比较清楚。

```
void ( *signal() )( int );
```

siganl 是一个函数，它返回一个函数指针，后者所指向的函数接受一个整型参数且没有返回值。事实上，signal 函数返回一个指向该信号以前的处理函数的指针。通过保存这个值，你可以为信号设置一个处理函数并在将来恢复为先前的处理函数。如果调用 signal 失败，例如由于非法的信号代

bbs.theithome.com

码所致，函数将返回 SIG_ERR 值。这个值是个宏，它在 signal.h 头文件中定义。

signal.h 头文件还定义了另外两个宏，SIG_DFL 和 SIG_IGN，它们可以作为 signal 函数的第 2 个参数。SIG_DFL 恢复对该信号的缺省反应，SIG_IGN 使该信号被忽略。

16.5.3 信号处理函数

当一个已经设置了信号处理函数的信号发生时，系统首先恢复对该信号的缺省行为[1]。这样做是为了防止如果信号处理函数内部也发生这个信号可能导致的无限循环。然后，信号处理函数被调用，信号代码作为参数传递给函数。

信号处理函数可能执行的工作类型是很有限的。如果信号是异步的，也就是说不是由于调用 abort 或 raise 函数引起的，信号处理函数便不应调用除 signal 之外的任何库函数，因为在这种情况下其结果是未定义的。而且，信号处理函数除了能向一个类型为 volatile sig_atomic_t 的静态变量（volatile 在下一节描述）赋一个值以外，可能无法访问其他任何静态数据。为了保证真正的安全，信号处理函数所能做的就是对这些变量之一进行设置然后返回。程序的剩余部分必须定期检查变量的值，看看是否有信号发生。

这些严格的限制是由于信号处理的本质产生的。信号通常用于提示发生了错误。在这些情况下，CPU 的行为是精确定义的，但在程序中，错误所处的上下文环境可能很不相同，因此它们并不一定能够良好定义。例如，当 strcpy 函数正在执行时如果产生一个信号，可能当时目标字符串暂时未以 NUL 字节终结；或者当一个函数被调用时如果产生一个信号，当时堆栈可能处于不完整的状态。如果依赖这种上下文环境的库函数被调用，它们就可能以不可预料的方式失败，很可能引发另一个信号。

访问限制定义了在信号处理函数中保证能够运行的最小功能。类型 sig_atomic_t 定义了一种 CPU 可以以原子方式访问的数据类型，也就是不可分割的访问单位。例如，一台 16 位的机器可以以原子方式访问一个 16 位整数，但访问一个 32 位整数可能需要两个操作。在访问非原子数据的中间步骤时如果产生一个信号可能导致不一致的结果，在信号处理函数中把数据访问限制为原子单位可以消除这种可能性。

警告：

标准表示信号处理函数可以通过调用 exit 终止程序。用于处理除了 SIGABRT 之外所有信号的处理函数也可以通过调用 abort 终止程序。但是，由于这两个都是库函数，所以当它们被异步信号处理函数调用时可能无法正常运行。如果你必须用这种方式终止程序，注意仍然存在一种微小的可能性导致它失败。如果发生这种情况，函数的失败可能破坏数据或者表现出奇怪的症状，但程序最终将终止。

一、volatile 数据

信号可能在任何时候发生，所以由信号处理函数修改的变量的值可能会在任何时候发生改变。因此，你不能指望这些变量在两条相邻的程序语句中肯定具有相同的值。volatile 关键字告诉编译器这个事实，防止它以一种可能修改程序含义的方式“优化”程序。考虑下面的程序段：

---

[1] 编译器可以选择当信号处理函数正在执行时“阻塞”信号而不是恢复缺省行为。请参阅有关文档。

bbs.theithome.com

```
if( value ){
        printf( "True\n" );
}
else {
        printf( "False\n" );
}
if( value ){
        printf( "True\n" );
}
else {
        printf( "False\n" );
}
```

在普通情况下，你会认为第 2 个测试和第 1 个测试具有相同的结果。如果信号处理函数修改了这个变量，第 2 个测试的结果可能不同。除非变量被声明为 volatile，否则编译器可能会用下面的代码进行替换，从而对程序进行“优化”。这些语句在通常情况下是正确的：

```
if( value ){
        printf( "True\n" );
        printf( "True\n" );
}
else {
        printf( "False\n" );
        printf( "False\n" );
}
```

二、从信号处理函数返回

从一个信号处理函数返回导致程序的执行流从信号发生的地点恢复执行。这个规则的例外情况是 SIGFPE。由于计算无法完成，从这个信号返回的效果是未定义的。

警告：

如果你希望捕捉将来同种类型的信号，从当前这个信号的处理函数返回之前注意要调用 signal 函数重新设置信号处理函数。否则，只有第 1 个信号才会被捕捉。接下来的信号将使用缺省反应进行处理。

提示：

由于各种计算机对不可预料的错误的反应各不相同，因此信号机制的规范也比较宽松。例如，编译器并不一定要使用标准定义的所有信号，而且在调用某个信号的处理函数之前可能会也可能不会重新设置信号的缺省行为。另一方面，对信号处理函数所施加的严重限制反映了不同的硬件和软件环境所施加的限制的交集。

这些限制和平台依赖性的结果就是使用信号处理函数的程序比不使用信号处理函数的程序可移植性弱一些。只有当需要时才使用信号以及不违反信号处理函数的规则有助于使这种类型的程序内部固有的可移植性问题降低到最低限度。

16.6　打印可变参数列表 <stdarg.h>

这组函数用于可变参数列表必须被打印的场合。注意：它们要求包含头文件 stdio.h 和 stdarg.h。

bbs.theithome.com

```
int vprintf( char const *format, va_list arg );
int vfprintf( FILE *stream, char const *format, va_list arg );
int vsprintf( char *buffer, char const *format, va_list arg );
```

这些函数与它们对应的标准函数基本相同，但它们使用了一个可变参数列表（请参阅第 7 章关于可变参数列表的详细内容）。在调用这些函数之前，arg 参数必须使用 va_start 进行初始化。这些函数都不需要调用 va_end。

16.7 执行环境

这些函数与程序的执行环境进行通信或者对后者施加影响。

16.7.1 终止执行 <stdlib.h>

这三个函数与正常或不正常的程序终止有关。

```
void abort( void )
void atexit( void (func)( void ) );
void exit( int status );
```

abort 函数用于不正常地终止一个正在执行的程序。由于这个函数将引发 SIGABRT 信号，你可以在程序中为这个信号设置一个信号处理函数，在程序终止（或干脆不终止）之前采取任何你想采取的动作，甚至可以不终止程序。

atexit 函数可以把一些函数注册为**退出函数**(exit function)。当程序将要正常终止时（或者由于调用 exit，或者由于 main 函数返回），退出函数将被调用。退出函数不能接受任何参数。

exit 函数在第 15 章已经作了描述，它用于正常终止程序。如果程序以 main 函数返回一个值结束，那么其效果相当于用这个值作用参数调用 exit 函数。

当 exit 函数被调用时，所有被 atexit 函数注册为退出函数的函数将按照它们所注册的顺序被反序依次调用。然后，所有用于流的缓冲区被刷新，所有打开的文件被关闭。用 tmpfile 函数创建的文件被删除。然后，退出状态返回给宿主环境，程序停止执行。

警告：

由于程序停止执行，所以 exit 函数绝不会返回到它的调用处。但是，如果任何一个用 atexit 注册为退出函数的函数再次调用了 exit，其效果是未定义的。这个错误可能导致一个无限循环，很可能只有当堆栈的内存耗尽后才会终止。

16.7.2 断言<assert.h>

断言就是声明某种东西应该为真。ANSI C 实现了一个 assert 宏，它在调试程序时很有用。它的原型如下所示[1]。

```
void assert( int expression );
```

当它被执行时，这个宏对表达式参数进行测试。如果它的值为假（零），它就向标准错误打印一条诊断信息并终止程序。这条信息的格式是由编译器定义的，但它将包含这个表示式和源文件的名

[1] 由于它是一个宏而不是函数，assert 实际上并不具有原型。但是，这个原型说明了 assert 的用法。

bbs.theithome.com

字以及断言所在的行号。如果表达式为真（非零），它不打印任何东西，程序继续执行。

这个宏提供了一种方便的方法，对应该是真的东西进行检查。例如，如果一个函数必须用一个不能为 NULL 的指针参数进行调用，那么函数可以用断言验证这个值：

```
assert( value != NULL );
```

如果函数错误地接受了一个 NULL 参数，程序就会打印一条类似下面形式的信息：

```
Assertion failed: value != NULL, file.c line 274
```

提示：

用这种方法使用断言使调试变得更容易，因为一旦出现错误，程序就会停止。而且，这条信息准确地提示了症状出现的地点。如果没有断言，程序可能继续运行，并在以后失败，这就很难进行调试。

注意 assert 只适用于验证必须为真的表达式。由于它会终止程序，所以你无法用它检查那些你试图进行处理的情况，例如检测非法的输入并要求用户重新输入一个值。

当程序被完整地测试完毕之后，你可以在编译时通过定义 NDEBUG 消除所有的断言[1]。你可以使用-DNDEBUG 编译器命令行选项或者在源文件中头文件 assert.h 被包含之前增加下面这个定义

```
#define NDEBUG
```

当 NDEBUG 被定义之后，预处理器将丢弃所有的断言，这样就消除了这方面的开销，而不必从源文件中把所有的断言实际删除。

16.7.3　环境 <stdlib.h>

环境(environment)就是一个由编译器定义的名字/值对的列表，它由操作系统进行维护。getenv 函数在这个列表中查找一个特定的名字，如果找到，返回一个指向其对应值的指针。程序不能修改返回的字符串。如果名字未找到，函数就返回一个 NULL 指针。

```
char *getenv( char const *name );
```

注意标准并未定义一个对应的 putenv 函数。有些编译器以某种方式提供了这个函数，不过如果你需要考虑程序的可移植性，最好还是避免使用它。

16.7.4　执行系统命令 <stdlib.h>

system 函数把它的字符串参数传递给宿主操作系统，这样它就可以作为一条命令，由系统的命令处理器执行。

```
void system( char const *command );
```

这个任务执行的准确行为因编译器而异，system 的返回值也是如此。但是，system 可以用一个 NULL 参数调用，用于询问命令处理器是否实际存在。在这种情况下，如果存在一个可用的命令处理器，system 返回一个非零值，否则它返回零。

---

[1] 可以把它定义为任何值，编译器只关心是否定义了 NDEBUG。

bbs.theithome.com

16.7.5 排序和查找<stdlib.h>

qsort 函数在一个数组中以升序的方式对数据进行排序。由于它是和类型无关的，所以你可以使用 qsort 排序任意类型的数据，只是数组中元素的长度是固定的。

```
void qsort( void *base, size_t n_elements, size_t el_size,
     int (*compare)(void const *, void const * ) );
```

第 1 个参数指向需要排序的数组，第 2 个参数指定数组中元素的数目，第 3 个参数指定每个元素的长度（以字符为单位）。第 4 个参数是一个函数指针，用于对需要排序的元素类型进行比较。在排序时，qsort 调用这个函数对数组中的数据进行比较。通过传递一个指向合适的比较函数的指针，你可以使用 qsort 排序任意类型值的数组。

比较函数接受两个参数，它们是指向两个需要进行比较的值的指针。函数应该返回一个整数，大于零、等于零和小于零分别表示第 1 个参数大于、等于和小于第 2 个参数。

由于这个函数与类型无关的性质，参数被声明为 void *类型。在比较函数中必须使用强制类型转换把它们转换为合适的指针类型。程序 16.3 说明了一个元素类型为一个关键字值和其他一些数据的结构的数组是如何被排序的。

```
/*
** 使用 qsort 对一个元素为某种结构的数组进行排序
*/
#include <stdlib.h>
#include <string.h>

typedef    struct    {
           char key[ 10 ];     /* 数组的排序关键字 */
           int  other_data;    /* 与关键字关联的数据 */
} Record;

/*
**  比较函数: 只比较关键字的值。
*/
int r_compare( void const *a, void const *b ){
       return strcmp( ((Record *)a)->key, ((Record *)b)->key );
}

int
main()
{
     Record    array[ 50 ];

     /*
     ** 用 50 个元素填充数组的代码
     */
     qsort( array, 50, sizeof( Record ), r_compare );

     /*
     ** 现在，数组已经根据结构的关键字字段排序完毕
     */
```


bbs.theithome.com

```
    return EXIT_SUCCESS;
}
```

程序 16.3　用 qsort 排序一个数组　　qsort.c

bsearch 函数在一个已经排好序的数组中用二分法查找一个特定的元素。如果数组尚未排序，其结果是未定义的。

```
void *bsearch(void const *key, void const *base,     size_tn_elements,
     size_t el_size, int (*compare)(void const *, void const * ) );
```

第 1 个参数指向你需要查找的值，第 2 个参数指向查找所在的数组，第 3 个参数指定数组中元素的数目，第 4 个参数是每个元素的长度（以字符为单位）。最后一个参数是和 qsort 中相同的指向比较函数的指针。bsearch 函数返回一个指向查找到的数组元素的指针。如果需要查找的值不存在，函数返回一个 NULL 指针。

注意关键字参数的类型必须与数组元素的类型相同。如果数组中的结构包含了一个关键字字段和其他一些数据，你必须创建一个完整的结构并填充关键字字段。其他字段可以留空，因为比较函数只检查关键字字段。bsearch 函数的用法如程序 16.4 所示。

```
/*
** 用 bearch 在一个元素类型为结构的数组中查找
*/
#include <stdlib.h>
#include <string.h>

typedef   struct    {
          char key[ 10 ];     /* 数组的排序关键字 */
          int  other_data;    /* 与关键字关联的数据 */
} Record;

/*
**  比较函数: 只比较关键字的值。
*/
int r_compare( void const *a, void const *b ){
      return strcmp( ((Record *)a)->key, ((Record *)b)->key );
}

int
main()
{
      Record   array[ 50 ];
      Record   key;
      Record   *ans;

      /*
      ** 用 50 个元素填充数组并进行排序的代码
      */

      /*
      ** 创建一个关键字结构（只用需要查找的值填充关键字字段），
      ** 并在数组中查找。
      */
      strcpy( key.key, "value" );
      ans = bsearch( &key, array, 50, sizeof( Record ),
```


bbs.theithome.com

```
            r_compare );

        /*
        **ans 现在指向关键字字段与值匹配的数据元素，如果无匹配，ans 为 NULL
        */

        return EXIT_SUCCESS;
    }
```

程序 16.4　用 bsearch 在数组中查找　　bsearch.c

16.8　locale

为了使 C 语言在全世界的范围内更为通用，标准定义了 locale，这是一组特定的参数，每个国家可能各不相同。在缺省情况下是“C”locale，编译器也可以定义其他的 locale。修改 locale 可能影响库函数的运行方式。修改 locale 的效果在本节的最后进行描述。

setlocale 函数的原型如下所示，它用于修改整个或部分 locale。

```
char *setlocale( int category, char const *locale );
```

category 参数指定 locale 的哪个部分需要进行修改。它所允许出现的值列于表 16.5。

如果 setlocale 的第 2 个参数为 NULL，函数将返回一个指向给定类型的当前 locale 的名字的指针。这个值可能被保存并在后续的 setlocale 函数中使用，用来恢复以前的 locale。如果第 2 个参数不是 NULL，它指定需要使用的新 locale。如果函数调用成功，它将返回新 locale 的值，否则返回一个 NULL 指针，原来的 locale 不受影响。

表 16.5　　setlocale 类型

值	修　改
LC_ALL	整个 locale
LC_COLLATE	对照序列，它将影响 strcoll 和 strxfrm 函数的行为
LC_CTYPE	定义于 ctype.h 中的函数所使用的字符类型分类信息
LC_MONETARY	在格式化货币值时使用的字符
LC_NUMERIC	在格式化非货币值时使用的字符。同时修改由格式化输入/输出函数和字符串转换函数所使用的小数点符号
LC_TIME	strftime 函数的行为

16.8.1　数值和货币格式 <locale.h>

格式数值和货币值的规则在全世界的不同地方可能并不相同。例如，在美国，一个写作 1,234.56 的数字在许多欧洲国家将被写成 1.234,56。localeconv 函数用于获得根据当前的 locale 对非货币值和货币值进行合适的格式化所需要的信息。注意这个函数并不实际执行格式化任务，它只是提供一些如何进行格式化的信息。

```
struct lconv *localeconv( void );
```

lconv 结构包含两种类型的参数：字符和字符指针。字符参数为非负值。如果一个字符参数为 CHAR_MAX，那个这个值就在当前的 locale 中不可用（或不使用）。对于字符指针参数，如果它指

bbs.theithome.com

向一个空字符串，它表示的意义和上面相同。

一、数值格式化

表 16.6 列出的参数用于格式化非货币的数值量。grouping 字符串按照下面的方式进行解释。该字符串的第 1 个值指定小数点左边多少个数字组成一组。第 2 个值指定再往左边一组数字的个数，以下依此类推。有两个值具有特别的意义：CHAR_MAX 表示剩余的数字并不分组，0 表示前面的值适用于数值中剩余的各组数字。

表 16.6　　格式化非货币数值的参数

字段和类型	含　义
char *decimal_point	用作小数点的字符。这个值绝不能是个空字符串
char *thousands_sep	用作分隔小数点左边各组数字的符号
char *grouping	指定小数点左边多少个数字组成一组

典型的北美格式是用下面的参数指定的：

```
decimal_point="."
thousands_sep=","
grouping="\3"
```

grouping 字符串包含一个 3[1]，后面是一个 0（也就是用于结尾的 NUL 字节）。这些值表示小数点左边的第 1 组数字将包括三个数字，其余的各组也将包括三个数字。值 1234567.89 根据这些参数进行格式化以后将以 1 234 567.89 的形式出现。

下面是另外一个例子。

```
grouping = "\4\3"
thousands_sep = "-"
```

这些值表示格式化北美地区电话号码的规则。根据这些参数，值 2125551234 将被格式化为 212-555-1234 的形式。

二、货币格式化

格式化货币值的规则要复杂得多。这是由于存在许多不同的提示正值和负值的方法、货币符号相对于值的位置等。另外，当货币值的格式化用于国际化时，规则又有所修改。首先，我们研究一些用于格式化本地（非国际）货币量的参数，见表 16.7。

表 16.7　　格式化本地货币值的参数

字段和类型	含　义
char *currency_symbol	本地货币符号
char *mon_decimal_point	小数点字符
char *mon_thousands_sep	用于分隔小数点左边各组数字的字符
char *mon_grouping	指定出现在小数点左边每组数字的数字个数
char *positive_sign	用于提示非负值的字符串
char *negative_sign	用于提示负值的字符串

[1] 注意这个数字是二进制的 3，而不是字符 3。

bbs.theithome.com

续表

字段和类型	含　义
char frac_digits	出现在小数点右边的数字个数
char p_cs_precedes	如果 currency_symbol 出现在一个非负值之前，其值为 1；如果出现在后面，其值为 0
char n_cs_precedes	如果 currency_symbol 出现在一个负值之前，其值为 1；如果出现在后面，其值为 0
char p_sep_by_space	如果 currency_symbol 和非负值之间用一个空格分隔，其值为 1，否则为 0
char n_sep_by_space	如果 currency_symbol 和负值之间用一个空格分隔，其值为 1，否则为 0
char p_sign_posn	提示 positive_sign 出现在一个非负值的位置。允许下列值： 0 货币符号和值两边的括号 1 正号出现在货币符号和值之前 2 正号出现在货币符号和值之后 3 正号紧邻货币符号之前 4 正号紧随货币符号之后
char n_sign_posn	提示 negative_sign 出现在一个负值中的位置。用于 p_sign_posn 的值也可用于此处

当按照国际化的用途格式化货币值时，字符串 int-curr_symbol 替代了 currency_symbol，字符 int_frac_digits 替代了 frac_digits。国际货币符号是根据 ISO 4217:1987 标准形成的。这个字符串的头三个字符是字母形式的国际货币符号，第 4 个字符用于分隔符号和值。

下面的值用一种可以被美国接受的方式对货币进行格式化。

```
currency_symbol="$"      p_cs_precedes='\1'
mon_decimal_point="."    n_cs_precedes='\1'
mon_thousands_sep=","    p_sep_by_space='\0'
mon_grouping="\3"        n_sep_by_space='\0'
positive_sign=""         p_sign_posn='\1'
negative_sign="CR"       n_sign_posn='\2'
frac_digits='\2'
```

使用上面这些参数，值 1234567890 和 -1234567890 将分别以 $1 234 567 890.00 和 $1 234 567 890.00CR 的形式出现。

设置 n_sign_posn='\0'可以使上面的负值以($1 234 567 890.00)的形式出现。

16.8.2 字符串和 locale <string.h>

一台机器的字符集的对照序列是固定的，但 locale 提供了一种方法指定不同的序列。当你必须使用一个并非缺省的对照序列时，可以使用下列两个函数。

```
int  strcoll( char const *s1, char const *s2 );
size_t  strxfrm( char *s1, char const *s2, size_t size );
```

strcoll 函数对两个根据当前 locale 的 LC_COLLATE 类型参数指定的字符串进行比较。它返回一个大于、等于或小于零的值，分别表示第 1 个参数大于、等于或小于第 2 个参数。

注意这个比较可能比 strcmp 需要多得多的计算量，因为它需要遵循一个并非是本地机器的对照序列。当字符串必须以这种方式反复进行比较时，我们可以使用 strxfrm 函数减少计算量。它把根据当前的 locale 解释的第 2 个参数转换为另一个不依赖于 locale 的字符串。尽管转换后的字符串的内容是未确定的，但使用 strcmp 函数对这种字符串进行比较和使用 strcoll 函数对原先的字符串进行比较的结果是相同的。

bbs.theithome.com

16.8.3　改变 locale 的效果

除了前面描述的那些效果之外，改变 locale 还会产生一些另外的效果。

1．locale 可能向正在执行的程序所使用的字符集增加字符（但可能不会改变现存字符的含义）。例如，许多欧洲语言使用了能够提示重音、货币符号和其他特殊符号的扩展字符集。

2．打印的方向可能会改变。尤其是，locale 决定一个字符应该根据前面一个被打印的字符的哪个方向进行打印。

3．printf 和 scanf 函数家族使用当前 locale 定义的小数点符号。

4．如果 locale 扩展了正在使用的字符集，isalpha、islower、isspace 和 isupper 函数可能比以前包括更多的字符。

5．正在使用的字符集的对照序列可能会改变。这个序列由 strcoll 函数使用，用于字符串之间的相互比较。

6．strftime 函数所产生的日期和时间格式的许多方面都是特定于 locale 的，前面已有所描述。

16.9　总结

标准函数库包含了许多有用的函数。第 1 组函数返回整型结果。abs 和 labs 函数返回它们的参数的绝对值。div 和 ldiv 函数用于执行整数除法。和/操作符不同，当其中一个参数为负时，商的值是精确定义的。rand 函数返回一个伪随机数。调用 srand 允许你从一串伪随机值中的任意一个位置开始产生随机数。atoi 和 atol 函数把一个字符串转换为整型值。strtol 和 strtoul 执行相同的转换，但它们可以给你更多的控制。

下一组函数中的绝大部分接受一个 double 参数并返回 double 结果。标准库提供了常用的三角函数 sin、cos、tan、asin、acos、atan 和 atan2。头三个函数接受一个以弧度表示的角度参数，分别返回该角度对应的正弦、余弦、正切值。接下来的三个函数分别返回与它们的参数对应的反正弦、反余弦和反正切值。最后一个函数根据 x 和 y 参数计算反正切值。双曲正弦、双曲余弦和双曲正切分别由 sinh、cosh 和 tanh 函数进行计算。exp 函数返回以 e 值为底，其参数为幂的指数值。log 函数返回其参数的自然对数，log10 函数返回以 10 为底的对数。

frexp 和 ldexp 函数在创建与机器无关的浮点数表示形式方面是很有用的。frexp 函数用于计算一个给定值的表示形式。ldexp 函数用于解释一个表示形式，恢复它的原先值。modf 函数用于把一个浮点值分割成整数和小数部分。pow 函数计算以第 1 个参数为底，第 2 个参数为幂的指数值。sqrt 函数返回其参数的平方根。floor 函数返回不大于其参数的最大整数，ceil 函数返回不小于其参数的最小整数。fabs 函数返回其参数的绝对值。fmod 函数接受两个参数，返回第 2 个参数除以第 1 个参数的余数。最后，atof 和 strtod 函数把字符串转换为浮点值。后者能够在转换时提供更多的控制。

接下来的一组函数用于处理日期和时间。clock 函数返回从程序执行开始到调用这个函数之间所花费的处理器时间。time 函数用一个 time_t 值返回当前的日期和时间。ctime 函数把一个 time_t 值转换为人眼可读的日期和时间表示形式。difftime 函数计算两个 time_t 值之间以秒为单位的时间差。gmtime 和 localtime 函数把一个 time_t 值转换为一个 tm 结构，tm 结构包含了日期和时间的所有组成部分。gmtime 函数使用世界协调时间，localtime 函数使用本地时间。asctime 和 strftime 函数把一个

bbs.theithome.com

tm 结构值转换为人眼可读的日期和时间的表示形式。strftime 函数对转换结果的格式提供了强大的控制。最后，mktime 把存储于 tm 结构中的值进行规格化，并把它们转换为一个 time_t 值。

非本地跳转由 setjmp 和 longjmp 函数提供。调用 setjmp 在一个 jmp_buf 变量中保存处理器的状态信息。接着，后续的 longjmp 调用将恢复这个被保存的处理器状态。在调用 setjmp 的函数返回之后，可能无法再调用 longjmp 函数。

信号表示在一个程序的执行期间可能发生的不可预料的事件，诸如用户中断程序或者发生一个算术错误。当一个信号发生时系统所采取的缺省反应是由编译器定义的，但一般都是终止程序。你可以通过定义一个信号处理函数并使用 signal 函数对其进行设置，从而改变信号的缺省行为。你可以在信号处理函数中执行的工作类型是受到严格限制的，因为程序在信号出现之后可能处于不一致的状态。volatile 数据的值可能会改变，而且很可能是由于自身所致。例如，一个在信号处理函数中修改的变量应该声明为 volatile。raise 函数产生一个由它的参数指定的信号。

vprintf、vfprintf 和 vsprintf 函数和 printf 函数家族执行相同的任务，但需要打印的值以可变参数列表的形式传递给函数。abort 函数通过产生 SIGABRT 信号终止程序。atexit 函数用于注册退出函数，它们在程序退出前被调用。assert 宏用于断言，当一个应该为真的表达式实际为假时，它就会终止程序。当调试完成之后，你可以通过定义 NDEBUG 符号去除程序中的所有断言，而不必把它们物理性地从源代码中删除。getenv 从操作系统环境中提取值。system 接受一个字符串参数，把它作为命令用本地命令处理器执行。

qsort 函数把一个数组中的值按照升序进行排序，bsearch 函数用于在一个已经排好序的数组中用二分法查找一个特定的值。由于这两个函数都是与类型无关的，所以它们可以用于任何数据类型的数组。

locale 就是一组参数，根据世界各国的约定差异对 C 程序的行为进行调整。setlocale 函数用于修改整个或部分 locale。locale 包括了一些用于定义数值如何进行格式化的参数。它们描述的值包括非货币值、本地货币值和国际货币值。locale 本身并不执行任何形式的格式化，它只是简单地提供格式化的规范。locale 可以指定一个和机器的缺省序列不同的对照序列。在这种情况下，strxcoll 用于根据当前的对照序列对字符串进行比较。它所返回的值类型类似 strcmp 函数的返回值。strxfrm 函数把一个当前对照序列的字符串转换为一个位于缺省对照序列的字符串。用这种方式转换的字符串可以用 strcmp 函数进行比较，比较的结果和用 strxcoll 比较原先的字符串的结果相同。

16.10 警告的总结

1. 忘了包含 math.h 头文件可能导致数学函数产生不正确的结果。
2. clock 函数可能只产生处理器时间的近似值。
3. time 函数的返回值并不一定是以秒为单位的。
4. tm 结构中月份的范围并不是从 1 到 12。
5. tm 结构中的年是从 1900 年开始计数的年数。
6. longjmp 不能返回到一个已经不再处于活动状态的函数。
7. 从异步信号的处理函数中调用 exit 或 abort 函数是不安全的。
8. 当每次信号发生时，你必须重新设置信号处理函数。
9. 避免 exit 函数的多重调用。

bbs.theithome.com

16.11 编程提示的总结

1. 滥用 setjmp 和 longjmp 可能导致晦涩难懂的代码。
2. 对信号进行处理将导致程序的可移植性变差。
3. 使用断言可以简化程序的调试。

16.12 问题

1. 下面的函数调用返回什么？

```
strtol("12345", NULL, -5 );
```

2. 如果说 rand 函数产生的“随机”数并不是真正的随机数，那么事实上它们能不能满足我们的需要呢？

3. 在你的系统上，下面的程序是什么结果？

```
#include <stdlib.h>
int
main()
{
    int  i;
    for( i = 0; i < 100; i += 1 )
        printf( "%d\n", rand() % 2 );
}
```

4. 你怎样编写一个程序，判断在你的系统中 clock 函数衡量 CPU 时间用的是 CPU 使用时间还是总流逝时间？

5. 下面的代码段试图用军事格式(military format)打印当前时间。它有什么错误？

```
#include <time.h>
struct tm *tm;
time_t now;
...
now = time();
tm = localtime( now );
printf( "%d:%02d:%02d %d/%02d/%02d\n",
    tm->tm_hour, tm->tm_min, tm->tm_sec,
    tm->tm_mon, tm->tm_mday, tm->tm_year );
```

6. 下面的程序有什么错误？当它在你的系统上执行时会发生什么？

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

jmp_buf jbuf;

void
set_buffer()
{
        setjmp( jbuf );
}

int
main( int ac, char **av )
{
```


bbs.theithome.com

```
    int     a = atoi( av[ 1 ] );
    int     b = atoi( av[ 2 ] );

    set_buffer();
    printf( "%d plus %d equals %d\n",
        a, b, a + b );
    longjmp( jbuf, 1 );
    printf( "After longjmp\n" );
    return EXIT_SUCCESS;
}
```

7. 编写一个程序，判断一个整数除以零或者一个浮点数除以零会不会产生 SIGFPE 信号。你如何解释这个结果？
8. qsort 函数所使用的比较函数在第 1 个参数小于第 2 个参数的情况下应该返回一个负值，在第 1 个参数大于第 2 个参数的情况下应该返回一个正值。如果比较函数返回相反的值，对 qsort 的行为有没有什么影响？

16.13 编程练习

★ 1. 计算机人群中颇为流行的一个笑话是“我 29 岁，但我不告诉你这个数字的基数！”如果基数是 16，这个人实际上是 41 岁。编写一个程序，接受一个年龄作为命令行参数，并在 2～36 的范围中计算那个字面值小于等于 29 的最小基数。例如，如果用户输入 41，程序应该计算出这个最小基数为 16。因为在 16 进制中，十进制 41 的值是 29。

★★ 2. 编写一个函数，通过返回一个范围为 1 至 6 的随机整数来模拟掷骰子。注意这 6 个值出现的概率应该相同。当这个函数第 1 次调用时，它应该用当天的当前时间作为种子来产生随机数。

★★ 3. 编写一个程序，以一种三岁小孩的方式来说明当前的时间（例如，时针在 6 上面，分针在 12 上面）。

★★ 4. 编写一个程序，接受三个整数为命令行参数，把它们分别解释为月（1～12）、日（1～31）和年（0～？）。然后，它应该打印出这个日子是星期几（或将是星期几）。对于哪个范围的年份，这个程序的结果才是正确的？

★★ 5. 冬天的天气预报常常会给出“风寒(wind chill)”这个词，它的意思是一个特定的温度或风速所感觉到的寒冷度。例如，如果气温为摄氏-5 度（华氏 23 度），并且风速每秒 10 米（22.37mph，即每小时 22.37 英里），那么风寒度便是摄氏-22.3 度（华氏-8.2 度）。

编写一个函数，使用下面的原型，计算风寒度。

```
double  wind_chill( double temp, double velocity );
```

temp 是摄氏气温的度数，velocity 是风速（米/秒）。函数返回摄氏形式的风寒度。风寒度是用下面的公式计算的：

$$Windchill = \frac{(A + B\sqrt{V} + CV)\Delta t}{A + B\sqrt{X} + CX}$$

对于一个给定的气温和风速。这个公式给出在风速为 4mph（风寒度标准）的情况下产生相同寒冷感的温度。V 是以米/秒计的风速，Δt 是 33-temp，也就是中性皮肤

bbs.theithome.com

温度（摄氏 33 度）和气温之间的温度差。常量 A=10.45，B=10，C=-1。X=1.78816，它是 4mph 转换为米/秒的值。

★★ 6. 用于计算抵押的月付金额的公式是：

$$P=\frac{AI}{1-(1+I)^{-N}}$$

A 是贷款的数量，I 是每个时段的利率（小数形式，而不是百分数形式），N 是贷款需要支付的时段数。例如，一笔$100 000 的 20 年期利率 8%的贷款每月需要支付$836.44（20 年共有 240 个支付时段，每个支付时段的利率为 0.66667）。

编写一个函数，它的原型如下所示，计算每月支付的贷款。

```
double payment( double amount, double interest,  int years );
```

years 指定货款的时期，amount 是贷款的数量，interest 是用百分数形式（例如，12%）表示的年利率。函数应该计算并返回贷款的月付金额，四舍五入至美分。

★★★ 7. 设计良好的随机数生成函数所生成的值看上去很像随机数，但随着时间的延长，其结果会显示出一致性。从随机值派生而来的数字也具有这些属性。例如，一个设计欠佳的随机数生成函数的返回值看上去像是随机数，但实际上却是奇数和偶数交替出现。如果对这些看似的随机数对 2 取模（例如，用于模拟抛硬币的结果），其结果将是一个 0 和 1 交叉的序列。另一种较差的随机数生成函数只返回奇数值。把这些值对 2 取模的结果将是一个连续的 0 序列。这两类值都无法作为随机数使用，因为它们不够“随机”。

编写一个程序，在你的系统中测试随机数生成函数。你应该生成 10 000 个随机数并执行两种类型的测试。首先是频率测试，把每个随机数对 2 取模，看看结果 0 和 1 的次数各有多少。然后对 3 到 10 做同样的测试。这些结果将不会具有精确的一致性，但各个余数在频率上的峰谷差异不应该太大。

其次是周期性频率测试，取每个随机数和它之前的那个随机数，将它们对 2 取模。使用这两个余数作为一个二维数组的下标并增加指定位置的值。对 3～10 重复进行上面的取模测试。同样，这些结果将不会具有很严格的规律，但应该具有近似的一致性。

修改你的程序，这样你可以为随机数生成函数提供不同的种子，并对使用几个不同的种子所产生的随机值进行测试。你的随机数生成函数是不是足够优秀？

★★★ 8. 某个文件包含了家庭成员的年龄。同一个家庭成员的年龄位于同一行，中间由一个空格分隔。例如，下面的数据

```
45  42  22
36  35  7  3  1
22  20
```

描述了三个分别具有 3 个、5 个和 2 个成员的家庭的年龄。

编写一个程序，计算用这种文件形式表示的每个家庭的平均年龄。它应该使用%5.2f 格式打印平均年龄，后面跟一个冒号和输入数据。这个问题和前一章的编程练习类似，但它没有家庭成员的数量限制！但是，你可以假定每个输入行的长度不超过 512 个字符。

★★★ 9. 在一个有 30 名学生的班级里，两个学生的生日是同一天的概率有多大？如果一群

bbs.theithome.com

人中两个成员的生日是同一天的概率为 50%，那么这个人群应该有多少人？

编写一个程序，回答这些问题。取 30 个随机数，并把它们对 365 取模，分别表示一年内的各天（忽略闰年）。然后对这些值进行检查，看看有没有相同的。重复这个测试 10 000 次，对这个频率作一个估计。

为了回答第 2 个问题，对程序进行修改，它把人数作为一个命令行参数，把当天的时间作为随机数生成函数的种子，数次运行这个程序，以获得这个概率较为精确的估计值。

★★★★ 10. 插入排序(insertion sort)就是逐个把值插入到一个数组中。第 1 个值存储于数据的起始位置。每个后续的值在数组中寻找合适的插入位置，如果需要的话，对数组中原有的值进行移动以留出空间，然后再插入该值。

编写一个名叫 insertion_sort 的函数执行这个任务。它的原型应该和 qsort 函数一样。

提示：考虑把数组的左边作为已排序的部分，右边作为未排序的部分。最初已排序部分为空。当你的函数插入每个值时，已排序部分和未排序部分的边界向右移动，以便插入。当所有的元素都被插入时，未排序部分便为空，数组排序完毕。

bbs.theithome.com

第17章 经典抽象数据类型

有些抽象数据类型(ADT)是C程序员不可或缺的工具，这是由于它们的属性决定的。这类ADT有链表、堆栈、队列和树等。第12章已经讨论了链表，本章我们将讨论剩余的ADT。

本章的第1部分描述了这些结构的属性和基本实现方法。在本章的最后，我们将探讨如何提高它们在实现上的灵活性以及由此导致的安全性能的妥协。

17.1 内存分配

所有的ADT都必须确定一件事情——如何获取内存来存储值。有三种可选的方案：静态数组、动态分配的数组和动态分配的链式结构。

静态数组要求结构的长度固定。而且，这个长度必须在编译时确定。但是，这个方案最为简单，而且最不容易出错。

如果你使用动态数组，你可以在运行时才决定数组的长度。而且，如果需要的话，你可以通过分配一个新的、更大的数组，把原来数组的元素复制到新数组中，然后删除原先的数组，从而达到动态改变数组长度的目的。在决定是否采用动态数组时，你需要在由此增加的复杂性和随之产生的灵活性（不需要一个固定的、预定确定的长度）之间作一番权衡。

最后，链式结构提供了最大程度的灵活性。每个元素在需要时才单独进行分配，所以除了不能超过机器的可用内存之外，这种方式对元素的数量几乎没有什么限制。但是，链式结构的链接字段需要消耗一定的内存，在链式结构中访问一个特定元素的效率不如数组。

17.2 堆栈

堆栈（stack）这种数据最鲜明的特点就是其后进先出(Last-In First-Out, LIFO)的方式。参加Party的人们对堆栈是很熟悉的。主人的车道就是一个汽车的堆栈，最后一辆进入车道的汽车必须首先开出，第1辆进入车道的汽车只有等其余所有车辆都开走后才能开出。

bbs.theithome.com

17.2.1 堆栈接口

基本的堆栈操作通常被称为 push 和 pop。push 就是把一个新值压入到堆栈的顶部，pop 就是把堆栈顶部的值移出堆栈并返回这个值。堆栈只提供对它的顶部值的访问。

在传统的堆栈接口中，访问顶部元素的唯一方法就是把它移除。另一类堆栈接口提供三种基本的操作：push、pop 和 top。push 操作和前面描述的一样，pop 只把顶部元素从堆栈中移除，它并不返回这个值。top 返回顶部元素的值，但它并不把顶部元素从堆栈中移除。

提示：

传统的 pop 函数具有一个副作用：它将改变堆栈的状态。它也是访问堆栈顶部元素的唯一方法。top 函数允许你反复访问堆栈顶部元素的值而不必把它保存在一个局部变量中。这个例子再次说明了设计不带副作用的函数的好处。

我们需要两个额外的函数来使用堆栈。一个空堆栈不能执行 pop 操作，所以我们需要一个函数告诉我们堆栈是否为空。在实现堆栈时如果存在最大长度限制，那么我们也需要另一个函数告诉我们堆栈是否已满。

17.2.2 实现堆栈

堆栈是最容易实现的 ADT 之一。它的基本方法是当值被 push 到堆栈时把它们存储于数组中连续的位置上。你必须记住最近一个被 push 的值的下标。如果需要执行 pop 操作，你只需要简单地减少这个下标值就可以了。程序 17.1 的头文件描述了一个堆栈模块的非传统接口。

提示：

注意接口只包含了用户使用堆栈所需要的信息，特别是它并没有展示堆栈的实现方式。事实上，对这个头文件稍作修改（稍后讨论），它可以用于所有三种实现方式。用这种方式定义接口是一种好方法，因为它防止用户以为它依赖于某种特定的实现方式。

```
/*
** 一个堆栈模块的接口
*/

#define STACK_TYPE int/* 堆栈所存储的值的类型 */

/*
**  push
**把一个新值压入到堆栈中。它的参数是需要被压入的值。
*/
voidpush( STACK_TYPE value );

/*
**  pop
**从堆栈弹出一个值，并将其丢弃。
*/
void pop( void );

/*
```


bbs.theithome.com

```
**  top
**    返回堆栈顶部元素的值，但不对堆栈进行修改。
*/
STACK_TYPE top( void );

/*
**  is_empty
**    如果堆栈为空，返回 TRUE，否则返回 FALSE。
*/
int is_empty( void );

/*
**  is_full
**  如果堆栈已满，返回 TRUE，否则返回 FALSE。
*/
int is_full( void );
```

程序 17.1　堆栈接口　　stack.h

提示：

这个接口的一个有趣特性是存储于堆栈中的值的类型的声明方式。在编译这个堆栈模块之前，用户可以修改这个类型以适合自己的需要。

一、数组堆栈

在程序 17.2 中，我们的第 1 种实现方式是使用一个静态数组。堆栈的长度以一个#define 定义的形式出现，在模块被编译之前用户必须对数组长度进行设置。我们后面所讨论的堆栈实现方案就没有这个限制。

提示：

所有不属于外部接口的内容都被声明为 static，这可以防止用户使用预定义接口之外的任何方式访问堆栈中的值。

```
/*
** 用一个静态数组实现的堆栈。数组的长度只能通过修改#define 定义
** 并对模块重新进行编译来实现。
*/

#include "stack.h"
#include <assert.h>

#define   STACK_SIZE    100/* 堆栈中值数量的最大限制 */

/*
** 存储堆栈中值的数组和一个指向堆栈顶部元素的指针。
*/
static    STACK_TYPE    stack[ STACK_SIZE ];
static    int           top_element = -1;

/*
**  push
*/
void
```


bbs.theithome.com

```
push( STACK_TYPE value )
{
     assert( !is_full() );
     top_element += 1;
     stack[ top_element ] = value;
}


/*
** pop
*/
void
pop( void )
{
     assert( !is_empty() );
     top_element -= 1;
}


/*
**  top
*/
STACK_TYPE top( void )
{
     assert( !is_empty() );
     return stack[ top_element ];
}


/*
**   is_empty
*/
int
is_empty( void )
{
     return top_element == -1;
}


/*
**   is_full
*/
int
is_full( void )
{
     return top_element == STACK_SIZE - 1;
}
```

程序 17.2　用静态数组实现堆栈　　a_stack.c

变量 top_element 保存堆栈顶部元素的下标值。它的初始值为-1，提示堆栈为空。push 函数在存储新元素前先增加这个变量的值，这样 top_element 始终包含顶部元素的下标值。如果它的初始值为 0，top_element 将指向数组的下一个可用位置。这种方式当然也可行，但它的效率稍差一些，因此它需要执行一次减法运算才能访问顶部元素。

一种简单明了的传统 pop 函数的写法如下所示：

bbs.theithome.com

```
STACK_TYPE
pop( void )
{
        STACK_TYPE temp;

        assert( !is_empty() );
        temp = stack[ top_element ];
        top_element -= 1;
        return temp;
}
```

这些操作的顺序是很重要的。top_element 在元素被复制出数组之后才减 1，这和 push 相反，后者是在被元素复制到数组之前先加 1。我们可以通过消除这个临时变量以及随之带来的复制操作来提高效率：

```
assert( !is_empty() );
return stack[ top_element-- ];
```

pop 函数不需要从数组中删除元素——只减少顶部指针的值就足矣，因为用户此时已不能再访问这个旧值了。

提示：

这个堆栈模块的一个值得注意的特性是它使用了 assert 来防止非法操作，诸如从一个空堆栈弹出元素或者向一个已满的堆栈压入元素。这个断言调用 is_full 和 is_empty 函数而不是测试 top_element 本身。如果你以后决定以不同的方法来检测空堆栈和满堆栈，使用这种方法显然要容易很多。

对于用户无法消除的错误，使用断言是很合适的。但如果用户希望确保程序不会终止，那么程序向堆栈压入一个新值之前必须检测堆栈是否仍有空间。因此，断言必须只能够对那些用户自己也能进行检查的内容进行检查。

二、动态数组堆栈

接下来的这种实现方式使用了一个动态数组，但我们首先需要在接口中定义两个新函数：

```
/*
** create_stack
** 创建堆栈。参数指定堆栈可以保存多少个元素。
** 注意：这个函数并不用于静态数组版本的堆栈。
*/
void  create_stack( size_t  size );

/*
** destroy_stack
** 销毁堆栈。它释放堆栈所使用的内存。
** 注意：这个函数也不用于静态数组版本的堆栈。
*/
void  destroy_stack( void );
```

第 1 个函数用于创建堆栈，用户向它传递一个参数，用于指定数组的长度。第 2 个函数用于删除堆栈，为了避免内存泄漏，这个函数是必需的。

这些声明可以添加到 stack.h 中，尽管前面的堆栈实现中并没有定义这两个函数。注意，用户在使用静态数组类型的堆栈时并不存在错误地调用这两个函数的危险，因为它们在那个模块中并

bbs.theithome.com

不存在。

提示：

一个更好的方法是把不需要的函数在数组模块中以存根的形式实现。如此一来，这两种实现方式的接口将是相同的，因此从其中一个转换到另一个会容易一些。

有趣的是，使用动态分配数组在实现上改动得并不多（见程序 17.3）。数组由一个指针代替，程序引入 stack_size 变量保存堆栈的长度。它们在缺省情况下都初始化为零。

create_stack 函数首先检查堆栈是否已经创建。然后分配所需数量的内存并检查分配是否成功。destroy_stack 在释放内存之后把长度和指针变量重新设置为零，这样它们可以用于创建另一个堆栈。

模块剩余部分的唯一改变是在 is_full 函数中与 stack_size 变量进行比较而不是与常量 STACK_SIZE 进行比较，并且在 is_full 和 is_empty 函数中都增加了一条断言。这条断言可以防止任何堆栈函数在堆栈被创建前就被调用。其余的堆栈函数并不需要这条断言，因为它们都调用了这两个函数之一。

```
/*
** 一个用动态分配数组实现的堆栈
** 堆栈的长度在创建堆栈的函数被调用时给出，该函数必须在任何其他操作堆栈的函数被调用之前调用。
*/
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
/*
** 用于存储堆栈元素的数组和指向堆栈顶部元素的指针。
*/
static    STACK_TYPE    *stack;
static    size_t        stack_size;
static    int           top_element = -1;

/*
**  create_stack
*/
void
create_stack( size_t size )
{
     assert( stack_size == 0 );
     stack_size = size;
     stack = malloc( stack_size * sizeof( STACK_TYPE ) );
     assert( stack != NULL );
}

/*
**  destroy_stack
*/
void
destroy_stack( void )
{
     assert( stack_size > 0 );
     stack_size = 0;
     free( stack );
```


bbs.theithome.com

```
        stack = NULL;
}

/*
**  push
*/
void
push( STACK_TYPE value )
{
        assert( !is_full() );
        top_element += 1;
        stack[ top_element ] = value;
}

/*
**    pop
*/
void
pop( void )
{
        assert( !is_empty() );
        top_element -= 1;
}

/*
**    top
*/
STACK_TYPE top( void )
{
        assert( !is_empty() );
        return stack[ top_element ];
}

/*
**    is_empty
*/
int
is_empty( void )
{
        assert( stack_size > 0 );
        return top_element == -1;
}

/*
**    is_full
*/
int
is_full( void )
{
        assert( stack_size > 0 );
        return top_element == stack_size - 1;
}
```

程序 17.3　用动态数组实现堆栈　　d_stack.c

警告：

在内存有限的环境中，使用 assert 检查内存分配是否成功并不合适，因此它很可能导致程序终

bbs.theithome.com

止，这未必是你希望的结果。一种替代策略是从 create_stack 函数返回一个值，提示内存分配是否成功。当这个函数失败时，用户程序可以用一个较小的长度再试一次。

三、链式堆栈

由于只有堆栈的顶部元素才可以被访问，所以使用单链表就可以很好地实现链式堆栈。把一个新元素压入到堆栈是通过在链表的起始位置添加一个元素实现的。从堆栈中弹出一个元素是通过从链表中移除第 1 个元素实现的。位于链表头部的元素总是很容易被访问。

在程序 17.4 所示的实现中，不再需要 create_stack 函数，但可以实现 destroy_stack 函数用于清空堆栈。由于用于存储元素的内存是动态分配的，它必须予以释放以避免内存泄漏。

```
/*
**  一个用链表实现的堆栈。这个堆栈没有长度限制。
*/
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>

#define   FALSE 0

/*
**  定义一个结构以存储堆栈元素，其中 link 字段将指向堆栈的下一个元素。
*/
typedef   struct    STACK_NODE {
          STACK_TYPE     value;
          struct STACK_NODE *next;
} StackNode;

/*
**  指向堆栈中第一个节点的指针。
*/
static    StackNode*stack;

/*
**    create_stack
*/
void
create_stack( size_t size )
{
}

/*
**    destroy_stack
*/
void
destroy_stack( void )
{
     while( !is_empty() )
          pop();
```


bbs.theithome.com

```
}

/*
**    push
*/
void
push( STACK_TYPE value )
{
      StackNode     *new_node;

      new_node = malloc( sizeof( StackNode ) );
      assert( new_node != NULL );
      new_node->value = value;
      new_node->next = stack;
      stack = new_node;
}

/*
**    pop
*/
void
pop( void )
{
      StackNode*first_node;

      assert( !is_empty() );
      first_node = stack;
      stack = first_node->next;
      free( first_node );
}

/*
**    top
*/
STACK_TYPE top( void )
{
      assert( !is_empty() );
      return stack->value;
}

/*
**    is_empty
*/
int
is_empty( void )
{
      return stack == NULL;
}

/*
**    is_full
*/
int
is_full( void )
{
```


bbs.theithome.com

```
    return FALSE;
    }
```

程序 17.4　用链表实现堆栈　　l_stack.c

STACK_NODE 结构用于把一个值和一个指针捆绑在一起，而 stack 变量是一个指向这些结构变量之一的指针。当 stack 指针为 NULL 时，堆栈为空，也就是初始时的状态。

提示：

destroy_stack 函数连续从堆栈中弹出元素，直到堆栈为空。同样，注意这个函数使用了现存的 is_empty 和 pop 函数而不是重复那些用于实际操作的代码。

create_stack 是一个空函数，由于链式堆栈不会填满，所以 is_full 函数始终返回假。

17.3 队列

队列和堆栈的顺序不同：队列是一种先进先出(First-IN First-OUT, FIFO)的结构。排队就是一种典型的队列。首先轮到的是排在队伍最前面的人，新入队的人总是排在队伍的最后。

17.3.1 队列接口

和堆栈不同，在队列中，用于执行元素的插入和删除的函数并没有被普遍接受的名字，所以我们将使用 insert 和 delete 这两个名字。同样，对于插入应该在队列的头部还是尾部也没有完全一致的意见。从原则上说，你在队列的哪一端插入并没有区别。但是，在队列的尾部插入以及在头部删除更容易记忆一些，因为它准确地描述了人们在排队时的实际体验。

在传统的接口中，delete 函数从队列的头部删除一个元素并将其返回。在另一种接口中，delete 函数从队列的头部删除一个元素，但并不返回它。first 函数返回队列第 1 个元素的值但并不将它从队列中删除。

程序 17.5 的头文件定义了后面那种接口。它包括链式和动态分配实现的队列需要使用的 create_queue 和 destroy_queue 函数的原型。

```
/*
**  一个队列模块的接口
*/

#include <stdlib.h>

#define   QUEUE_TYPE    int/* 队列元素的类型 */

/*
**  create_queue
**    创建一个队列，参数指定队列可以存储的元素的最大数量。
**  注意: 这个函数只适用于使用动态分配数组的队列。
*/
void create_queue( size_t size );

/*
**  destroy_queue
**    销毁一个队列。注意: 这个函数只适用于链式和动态分配数组的队列。
```


bbs.theithome.com

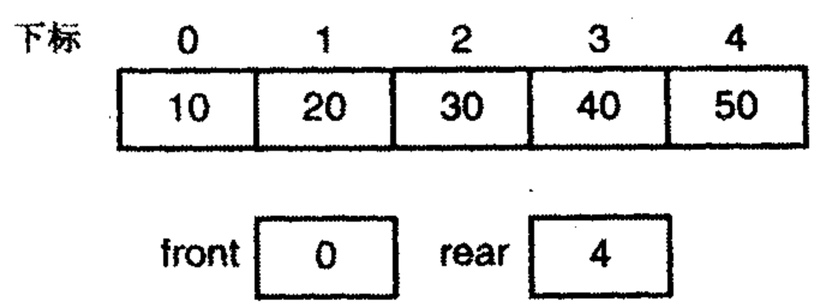
```
*/
void destroy_queue( void );

/*
** insert
**    向队列添加一个新元素，参数就是需要添加的元素。
*/
void insert( QUEUE_TYPE value );

/*
** delete
**    从队列中移除一个元素并将其丢弃。
*/
void delete( void );

/*
** first
**    返回队列中第一个元素的值，但不修改队列本身。
*/
QUEUE_TYPE first( void );

/*
** is_empty
**    如果队列为空，返回 TRUE，否则返回 FALSE。
*/
int  is_empty( void );

/*
** is_full
**    如果队列已满，返回 TRUE，否则返回 FALSE。
*/
int  is_full( void );
```

程序 17.5　队列接口　　queue.h

17.3.2　实现队列

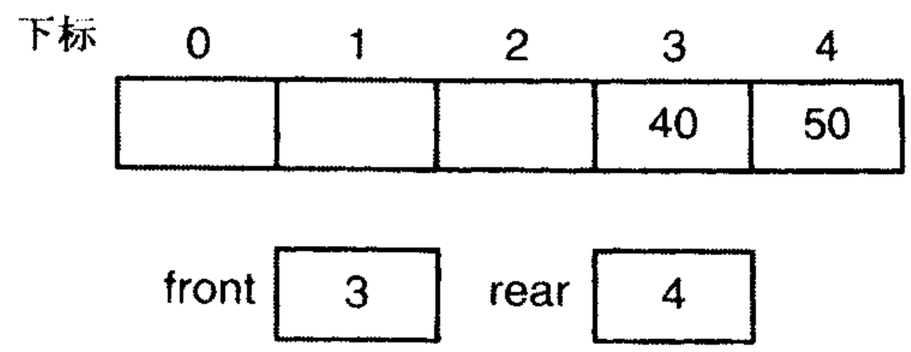
队列的实现比堆栈要难得多。它需要两个指针——一个指向队头，一个指向队尾。同时，数组并不像适合堆栈那样适合队列的实现，这是由于队列使用内存的方式决定的。

堆栈总是扎根于数据的一端。但是，当队列的元素插入和删除时，它所使用的是数组中的不同元素。考虑一个用 5 个元素的数组实现的队列。下面的图是 10、20、30、40 和 50 这几个值插入队列以后队列的样子。



经过三次删除之后，队列的样子如下所示：

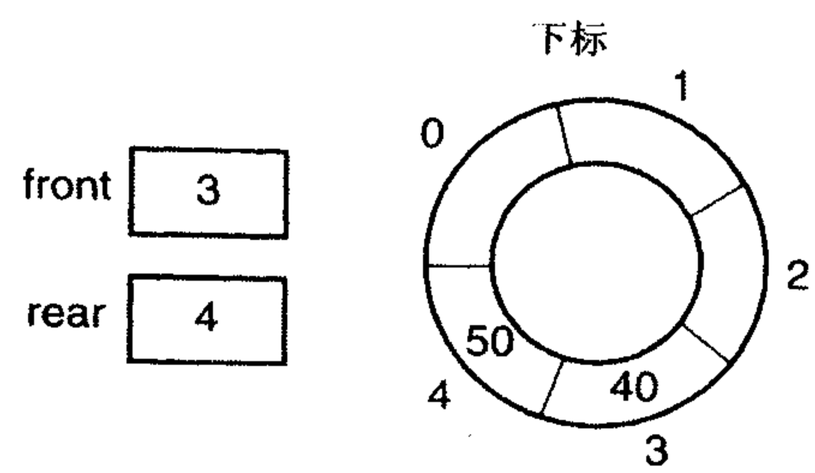
bbs.theithome.com



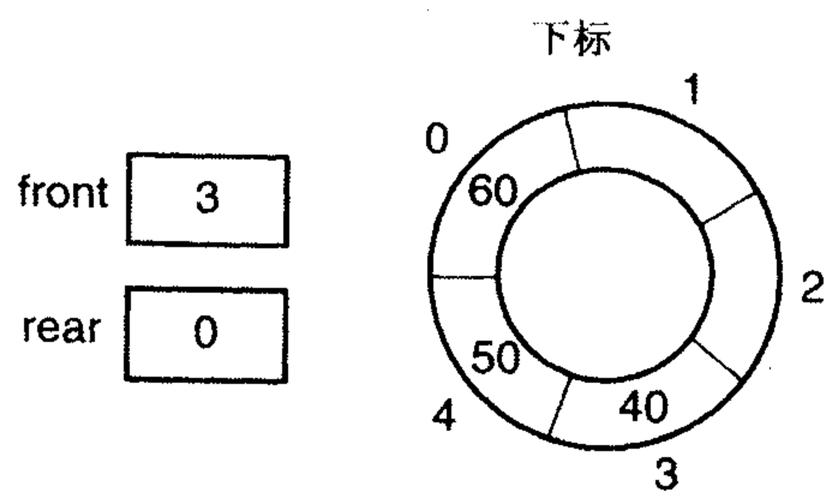
数组并未满，但它的尾部已经没有空间，无法再插入新的元素。

这个问题的一种解决方法是当一个元素被删除之后，队列中的其余元素朝数组起始位置方向移动一个位置。由于复制元素所需的开销，这种方法几乎不可行，尤其是那些较大的队列。

一个好一点的方案是让队列的尾部“环绕”到数组的头部，这样新元素就可以存储到以前删除元素所留出来的空间中。这个方法常常被称为循**环数组**(circular array)。下图说明了这个概念。



插入另一个元素之后的结果如下：



循环数组很容易实现——当尾部下标移出数组尾部时，把它设置为 0。用下面的代码便可以实现。

```
rear += 1;
if( rear >= QUEUE_SIZE )
        rear = 0;
```

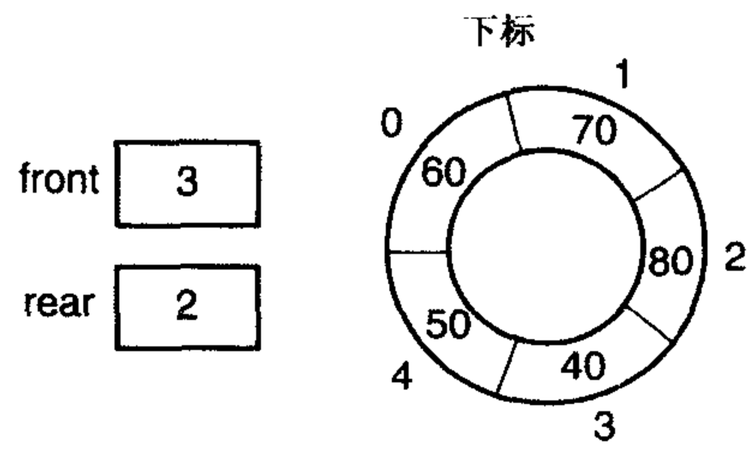
下面的方法具有相同的结果。

```
rear = ( rear + 1 ) % QUEUE_SIZE;
```

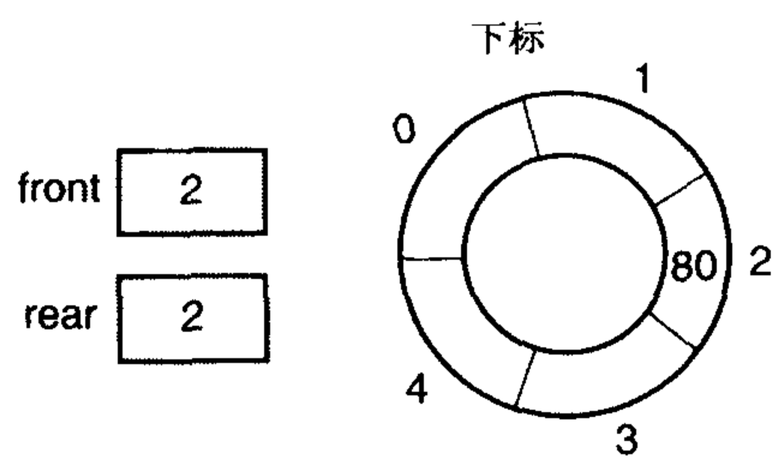
在对 front 增值时也必须使用同一个技巧。

但是，循环数组自身也引入了一个问题。它使得判断一个循环数组是否为空或者已满更为困难。假定队列已满，如下图所示：

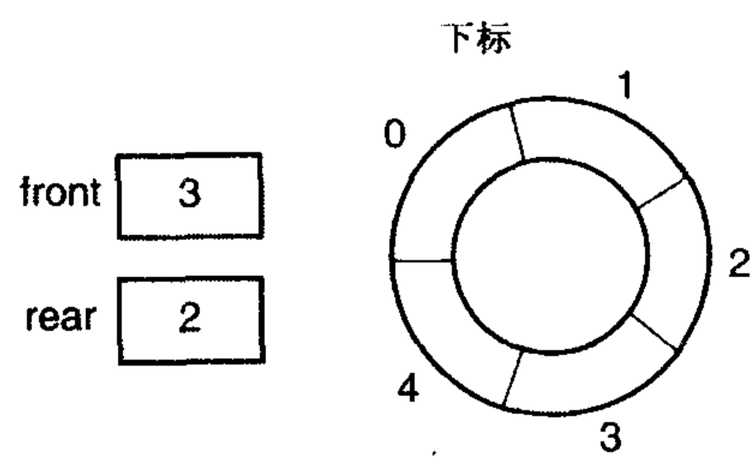
bbs.theithome.com



注意 front 和 rear 的值：分别是 3 和 2。如果有 4 个元素从队列中删除，front 将增值 4 次，队列中的情况如下图所示：



当最后一个元素被删除时，队列中的情况如下图所示：



问题是现在 front 和 rear 的值是相同的，这和队列已满时的情况是一样的。当队列空或者满时对 front 和 rear 进行比较，其结果都是真。所以，我们无法通过比较 front 和 rear 来测试队列是否为空。

有两种方法可以解决这个问题。第 1 种是引入一个新变量，用于记录队列中的元素数量。它在每次插入元素时加 1，在每次删除元素时减 1。对这个变量的值进行测试就可以很容易分清队列空间为空还是已满。

第 2 种方法是重新定义“满”的含义。如果使数组中的一个元素始终保留不用，这样当队列“满”时 front 和 rear 的值便不相同，可以和队列为空时的情况区分开来。通过不允许数组完全填满，问题便得以避免。

不过还是留下一个小问题：当队列为空时，front 和 rear 的值应该是什么？当队列只有一个元素时，我们需要使 front 和 rear 都指向这个元素。一次插入操作将增加 rear 的值，所以为了使 rear 在第 1 次插入后指向这个插入的元素，当队列为空时 rear 的值必须比 front 小 1。幸运的是，从队列中删除最后一个元素后的状态也是如此，因此删除最后一个元素并不会造成一种特殊情况。

当满足下面的条件时，队列为空：

bbs.theithome.com

```
( rear + 1 ) % QUEUE_SIZE == front
```

由于在 front 和 rear 正好满足这个关系之前，我们必须停止插入元素，所以当满足下列条件时，队列必须认为已“满”。

```
( rear + 2 ) % QUEUE_SIZE == front
```

一、数组队列

程序 17.6 用一个静态数组实现了一个队列。它使用“不完全填满数组”的技巧来区分空队列和满队列。

```
/*
**  一个用静态数组实现的队列。数组的长度只能通过修改#define 定义并重新编译模块来调整。
*/

#include "queue.h"
#include <stdio.h>
#include <assert.h>

#define   QUEUE_SIZE     100/* 队列中元素的最大数量 */
#define   ARRAY_SIZE     ( QUEUE_SIZE + 1 )/* 数组的长度 */

/*
**  用于存储队列元素的数组和指向队列头和尾的指针。
*/
static    QUEUE_TYPE     queue[ ARRAY_SIZE ];
static    size_t         front = 1;
static    size_t         rear = 0;

/*
**    insert
*/
void
insert( QUEUE_TYPE value )
{
     assert( !is_full() );
     rear = ( rear + 1 ) % ARRAY_SIZE;
     queue[ rear ] = value;
}

/*
**    delete
*/
void
delete( void )
{
     assert( !is_empty() );
     front = ( front + 1 ) % ARRAY_SIZE;
}

/*
**    first
*/
QUEUE_TYPE first( void )
```


bbs.theithome.com

```
{
     assert( !is_empty() );
     return queue[ front ];
}

/*
**   is_empty
*/
int
is_empty( void )
{
     return ( rear + 1 ) % ARRAY_SIZE == front;
}

/*
**   is_full
*/
int
is_full( void )
{
     return ( rear + 2 ) % ARRAY_SIZE == front;
}
```

程序 17.6　用静态数组实现队列　　a_queue.c

QUEUE_SIZE 常量设置为用户希望队列可以容纳的元素的最大数量。由于这种实现方式永远不会真正填满队列，ARRAY_SIZE 的值被定义为比 QUEUE_SIZE 大 1。这些函数是我们所讨论的那些技巧的简单明了的实现。

我们可以使用任何值初始化 front 和 rear，只要 rear 比 front 小 1。程序 17.6 所使用的初始值使数组的第 1 个元素保留不用，直到 rear 第 1 次“环绕”至数组头部，猜猜接下来会怎样？

二、动态数组队列和链式队列

用动态数组实现队列所需要的修改和堆栈的情况类似。因此，它的实现留作练习。

链式队列在几个方面比数组形式的队列简单。它不使用数组，所以不存在循环数组的问题。测试队列是否为空只是简单测试链表是否为空就可以了。测试队列是否已满的结果总是假，链式队列的实现也留作练习。

17.4　树

对树的所有种类作完整的描述超出了本书的范围。但是，通过描述一种非常有用的树：**二叉搜索树**(binary search tree)，可以很好地说明实现树的技巧。

树是一种数据结构，它要么为空，要么具有一个值并具有零个或多个**孩子**(child)，每个孩子本身也是树。这个递归的定义正确地提示了一棵树的高度并没有内在的限制。**二叉树**(binary tree)是树的一种特殊形式，它的每个节点至多具有两个孩子，分别称为**左孩子**(left)和**右孩子**(right)。二叉搜索树具有一个额外的属性：每个节点的值比它的左子树的所有节点的值都要大，但比它的右子树的所有节点的值都要小。注意这个定义排除了树中存在值相同的节点的可能性。这些属性使二叉搜索树成为一种用关键值快速查找数据的优秀工具。图 17.1 是二叉搜索树的一个例子。这棵树的每个节

bbs.theithome.com

点都正好具有一个双亲节点（它的上层节点），零个、一个或两个孩子（直接在它下面的节点）。唯一的例外是最上面的那个节点，称为树根，它没有双亲节点。没有孩子的节点被称为**叶节点**(leaf node)或**叶子**(leaf)。在绘制树时，根位于顶端，叶子位于底部[1]。

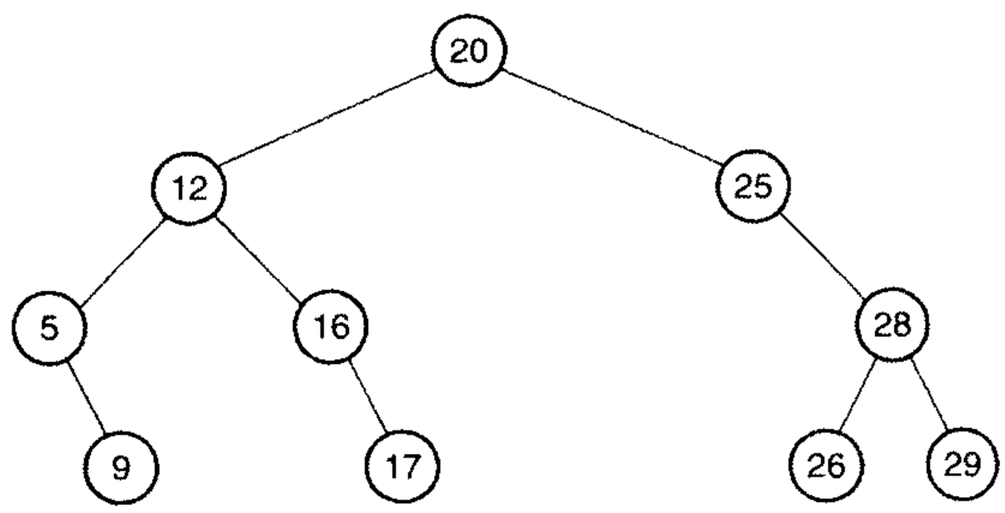


图 17.1　二叉搜索树

17.4.1　在二叉搜索树中插入

当一个新值添加到一棵二叉搜索树时，它必须被放在合适的位置，继续保持二叉搜索树的属性。幸运的是，这个任务是很简单的。其基本算法如下所示：

```
如果树为空:
    把新值作为根节点插入
否则:
    如果新值小于当前节点的值:
        把新值插入到当前节点的左子树
    否则:
        把新值插入到当前节点的右子树
```

这个算法的递归表达正是树的递归定义的直接结果。

为了把 15 插入到图 17.1 的树，把 15 和 20 比较。15 更小，所以它被插入到左子树。左子树的根为 12，因此重复上述过程：把 15 和 12 比较。这次 15 更大，所以它被插入到 12 的右子树。现在我们把 15 和 16 比较。15 更小，所以插入到节点 16 的左子树。但这个子树是空的，所以包含 15 的节点便成为节点 16 的新左子树的根节点。

提示：

由于递归在算法的尾部出现（尾部递归），所以我们可以使用迭代更有效地实现这个算法。

17.4.2　从二叉搜索树删除节点

从树中删除一个值比从堆栈或队列删除一个值更为困难。从一棵树的中部删除一个节点将导致它的子树和树的其余部分断开——我们必须重新连接它们，否则它们将会丢失。

我们必须处理三种情况：删除没有孩子的节点；删除只有一个孩子的节点；删除有两个孩子的节点。第 1 个情况很简单，删除一个叶节点不会导致任何子树断开，所以不存在重新连接的问题。

[1] 注意这和自然世界中根在底叶在上的树实际上是颠倒的。

bbs.theithome.com

删除只有一个孩子的节点几乎同样容易：把这个节点的双亲节点和它的孩子连接起来就可以了。这个解决方法防止了子树的断开，而且仍能维持二叉搜索树的次序。

最后一种情况要困难得多。如果一个节点有两个孩子，它的双亲不能连接到它的两个孩子。解决这个问题的一种策略是不删除这个节点，而是删除它的左子树中值最大的那个节点，并用这个值代替原先应被删除的那个节点的值。删除函数的实现留作练习。

17.4.3　在二叉搜索树中查找

由于二叉搜索树的有序性，所以在树中查找一个特定的值是非常容易的。下面是它的算法：

```
如果树为空:
    这个值不存在于树中
否则:
    如果这个值和根节点的值相等:
        成功找到这个值
    否则:
        如果这个值小于根节点的值:
            查找左子树
        否则:
            查找右子树
```

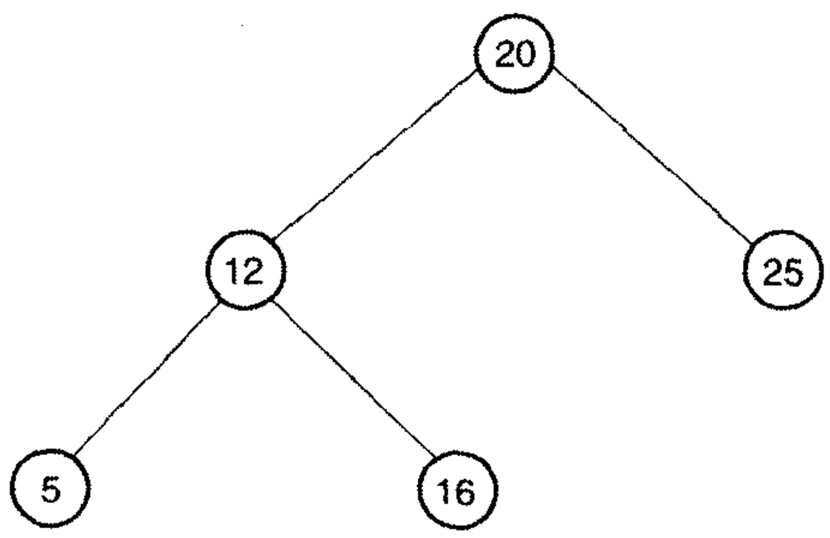
这个递归算法也属于尾部递归，所以采用迭代方案来实现效率更高。

当值被找到时你该做些什么呢？这取决于用户的需要。有时，用户只需要确定这个值是否存在于树中。这时，返回一个真/假值就足够了。如果数据是一个由一个关键值字段标识的结构，用户需要访问这个查找到的结构的非关键值成员，这就要求函数返回一个指向该结构的指针。

17.4.4　树的遍历

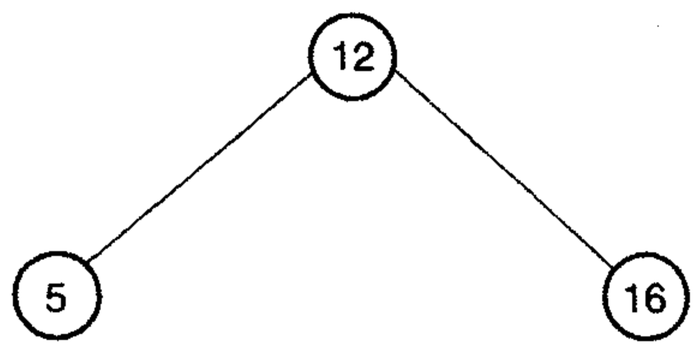
和堆栈和队列不同，树并未限制你只能访问一个值。因此树具有另一个基本操作——**遍历**(traversal)。当你检查一棵树的所有节点时，你就在遍历这棵树。遍历树的节点有几种不同的次序，最常用的是**前序**(pre-order)、**中序**(in-order)、**后序**(post-order)和**层次遍历**(breadth-first)。所有类型的遍历都是从树的根节点或你希望开始遍历的子树的根节点开始。

前序遍历检查节点的值，然后递归地遍历左子树和右子树。例如，下面这棵树的前序遍历



将从处理 20 这个值开始。然后我们再遍历它的左子树：

bbs.theithome.com



在处理完 12 这个值之后，我们继续遍历它的左子树

并处理 5 这个值。它的左右子树皆为空，所以我们就完成了这棵子树的遍历。

在完成节点 12 的左子树遍历之后，我们继续遍历它的右子树：

并处理 16 这个值。它的左右子树皆为空，这意味着我们已经完成了根为 16 的子树和根为 12 的子树的遍历。

在完成了节点 20 的左子树遍历之后，下一个步骤就是处理它的右子树：

处理完 25 这个值以后便完成了整棵树的遍历。

对于一个较大的例子，考虑图 17.1 的二叉搜索树。当检查每个节点时打印出它的值，那么它的前序遍历的输出结果将是：20，12，5，9，16，17，25，28，26，29。

中序遍历首先遍历左子树，然后检查当前节点的值，最后遍历右子树。图 17.1 的树的中序遍历结果将是：5，9，12，16，17，20，25，26，28，29。

后序遍历首先遍历左右子树，然后检查当前节点的值。图 17.1 的树的后序遍历结果将是：9，5，17，16，12，26，29，28，25，20。

最后，层次遍历逐层检查树的节点。首先处理根节点，接着是它的孩子，再接着是它的孙子，依次类推。用这种方法遍历图 17.1 的树的次序是：20，12，25，5，16，28，9，17，26，29。虽然前三种遍历方法可以很容易地使用递归来实现，但最后这种层次遍历要采用一种使用队列的迭代算法。本章的练习对它有更详细的描述。

17.4.5 二叉搜索树接口

程序 17.7 的接口提供了用于把值插入到一棵二叉搜索树的函数的原型。它同时包含了一个 find 函数用于查找树中某个特定的值，它的返回值是一个指向找到的值的指针。它只定义了一个遍历函数，因为其余遍历函数的接口只是名字不同而已。

```
/*
**  二叉搜索树模块的接口
*/
```


bbs.theithome.com

```
#define   TREE_TYPE int     /* 树的值类型 */

/*
** insert
**    向树添加一个新值。参数是需要被添加的值，它必须原先并不存在于树中。
*/
void insert( TREE_TYPE value );

/*
** find
**    查找一个特定的值，这个值作为第 1 个参数传递给函数。
*/
TREE_TYPE *find( TREE_TYPE value );

/*
** pre_order_traverse
**执行树的前序遍历。它的参数是一个回调函数指针，它所指向的函数将在树中处理每
**个节点被调用，节点的值作为参数传递给这个函数。
*/
void pre_order_traverse(void (*callback)( TREE_TYPE value ));
```

程序 17.7　二叉搜索树接口　　tree.h

17.4.6　实现二叉搜索树

尽管树的链式实现是最为常见的，但将二叉搜索树存储于数组中也是完全可能的。当然，数组的固定长度限制了你可以插入到树中的元素的数量，但如果你使用动态数组，当原先的数组溢出时，你可以创建一个更大的空间并把值复制给它。

一、数组形式的二叉搜索树

用数组表示树的关键是使用下标来寻找某个特定值的双亲和孩子。规则很简单：

节点 N 的双亲是节点 N/2。
节点 N 的左孩子是节点 2N。
节点 N 的右孩子是节点 2N+1。

双亲节点的公式是成立的，因为整除操作符将截去小数部分。

警告：

唉！这里有个小问题。这些规则假定树的根节点是第 1 个节点，但 C 的数组下标从 0 开始。最容易的解决方案是忽略数组的第 1 个元素。如果元素非常大，这种方法将浪费很多空间，如果这样你可以使用基于零下标数组的另一套规则：

节点 N 的双亲节点是节点(N + 1)/2-1。
节点 N 的左孩子节点是节点 2N+1。
节点 N 的右孩子节点是节点 2N+2。

程序 17.8 是一个用静态数组实现的二叉搜索树。这个实现方法有几个有趣之处。它使用第 1 种更简单的规则来确定孩子节点，这样数组声明的长度比宣称的长度大 1，它的第 1 个元素被忽略。它定义了一些函数访问一个节点的左右孩子。尽管计算很简单，这些函数名还是让使用这些函数的代码看上去更清晰。这些函数同时简化了修改模块以便使用其他规则的任务。

bbs.theithome.com

这种实现方法使用 0 这个值提示一个节点未被使用。如果 0 是一个合法的数据值，那就必须另外挑选一个不同的值，而且数组元素必须进行动态初始化。另一个技巧是使用一个比较数组，它的元素是布尔型值，用于提示哪个节点被使用。

数组形式的树的问题在于数组空间常常利用得不够充分。空间之所以被浪费是由于新值必须插入到树中特定的位置，无法随便放置到数组中的空位置。

为了说明这种情况，假定我们使用一个拥有 100 个元素的数组来容纳一棵树。如果值 1，2，3，4，5，6 和 7 以这个次序插入，它们将分别存储在数组中 1，2，4，8，16，32 和 64 的位置。但现在值 8 不能被插入，因为 7 的右孩子将存储于位置 128，数组的长度没有那么长。这个问题会不会实际发生取决于值插入的顺序。如果相同的值以这样的顺序插入：4，2，1，3，6，5 和 7，它们将占据数组 1 至 7 的位置，这样插入 8 这个值便毫无困难。

使用动态分配的数组，当需要更多空间时我们可以对数组进行重新分配。但是，对于一棵不平衡的树，这个技巧并不是一个好的解决方案，因为每次新插入都将导致数组的大小扩大一倍，这样可用于动态分配的内存很快便会耗尽。一个更好的方法是使用链式二叉树而不是数组。

```
/*
** 一个使用静态数组实现的二叉搜索树。数组的长度只能通过修改#define 定义
** 并对模块进行重新编译来实现。
*/
#include "tree.h"
#include <assert.h>
#include <stdio.h>

#define   TREE_SIZE 100 /* Max # of values in the tree */
#define   ARRAY_SIZE    ( TREE_SIZE + 1 )

/*
**  用于存储树的所有节点的数组。
*/
static    TREE_TYPE tree[ ARRAY_SIZE ];

/*
** left_child
** 计算一个节点左孩子的下标。
*/
static int
left_child( int current )
{
     return current * 2;
}

/*
** right_child
**    计算一个节点右孩子的下标。
*/
static int
right_child( int current )
{
     return current * 2 + 1;
}
```


bbs.theithome.com

```
/*
** insert
*/
void
insert( TREE_TYPE value )
{
      int  current;

      /*
      ** 确保值为非零，因为零用于提示一个未使用的节点。
      */
      assert( value != 0 );

      /*
      **  从根节点开始。
      */
      current = 1;

      /*
      **  从合适的子树开始，直到到达一个叶节点。
      */
      while( tree[ current ] != 0 ){
      /*
      ** 根据情况，进入叶节点或右子树（确信未出现重复的值）。
      */
            if( value < tree[ current ] )
                  current = left_child( current );
            else {
                  assert( value != tree[ current ] );
                  current = right_child( current );
            }
            assert( current < ARRAY_SIZE );
      }

      tree[ current ] = value;
}

/*
** find
*/
TREE_TYPE *
find( TREE_TYPE value )
{
      int  current;

      /*
      ** 从根节点开始。直到找到那个值，进入合适的子树。
      */
      current = 1;

      while( current < ARRAY_SIZE && tree[ current ] != value ){
      /*
      ** 根据情况，进入左子树或右子树。
      */
         if( value < tree[ current ] )
               current = left_child( current );
```


bbs.theithome.com

```
            else
                current = right_child( current );
        }
        if( current < ARRAY_SIZE )
            return tree + current;
        else
            return 0;
}

/*
** do_pre_order_traverse
**   执行一层前序遍历，这个帮助函数用于保存我们当前正在处理的节点的信息，
**   它并不是用户接口的一部分。
*/
static void
do_pre_order_traverse( int current,
    void (*callback)( TREE_TYPE value ) )
{
        if( current < ARRAY_SIZE && tree[ current ] != 0 ){
            callback( tree[ current ] );
            do_pre_order_traverse( left_child( current ),
                callback );
            do_pre_order_traverse( right_child( current ),
                callback );
        }
}

/*
** pre_order_traverse
*/
void
pre_order_traverse( void (*callback)( TREE_TYPE value ) )
{
    do_pre_order_traverse( 1, callback );
}
```

程序 17.8　用静态数组实现二叉搜索树　　a_tree.c

二、链式二叉搜索树

队列的链式实现消除了数组空间利用不充分的问题，这是通过为每个新值动态分配内存并把这些结构链接到树中实现的。因此，不存在不使用的内存。

程序 17.9 是二叉搜索树的链式实现方法。请将它和程序 17.8 的数组实现方法进行比较。由于树中的每个节点必须指向它的左右孩子，所以节点用一个结构来容纳值和两个指针。数组由一个指向树根节点的指针代替。这个指针最初为 NULL，表示此时为一棵空树。

insert 函数使用两个指针[1]。第 1 个用于检查树中的节点，寻找新值插入的合适位置。第 2 个指针指向另一个节点，后者的 link 字段指向当前正在检查的节点。当到达一个叶节点时，这个指针必须进行修改以插入新节点。这个函数自上而下，根据新值和当前节点值的比较结果选择进入左子树或右子树，直到到达叶节点。然后，创建一个新节点并链接到树中。这个迭代算法在插入第 1 个节

[1] 我们使用了和第 12 章的函数中把值插入到一个有序的单链表的相同技巧。如果你沿着从根到叶的路径观察插入发生的位置，你就会发现它本质上就是一个单链表。

bbs.theithome.com

点时也能正确处理，不会造成特殊情况。

三、树接口的变型

find 函数只用于验证值是否存在于树中。返回一个指向找到元素的指针并无大用，因为调用程序已经知道这个值：它就是传递给函数的参数嘛！

假定树中的元素实际上是一个结构，它包括一个关键值和一些数据。现在我们可以修改 find 函数，使它更加实用。通过它的关键值查找一个特定的节点并返回一个指向该结构的指针可以向用户提供更多的信息——与这个关键值相关联的数据。但是，为了取得这个结果，find 函数必须设法只比较每个节点元素的关键值部分。解决办法是编写一个函数执行这个比较，并把一个指向该函数的指针传递给 find 函数，就像我们在 qsort 函数中所采取的方法一样。

有时候用户可能要求自己遍历整棵树，例如，计算每个节点的孩子数量。因此，TreeNode 结构和指向树根节点的指针都必须声明为公用，以便用户遍历该树。最安全的方法是通过函数向用户提供根指针，这样可以防止用户自行修改根指针，从而导致丢失整棵树。

```
/*
** 一个使用动态分配的链式结构实现的二叉搜索树。
*/
#include "tree.h"
#include <assert.h>
#include <stdio.h>
#include <malloc.h>

/*
**  TreeNode 结构包含了值和两个指向某个树节点的指针。
*/
typedef struct TREE_NODE {
        TREE_TYPE   value;
        struct TREE_NODE *left;
        struct TREE_NODE *right;
} TreeNode;

/*
**  指向树根节点的指针。
*/
static    TreeNode  *tree;

/*
**    insert
*/
void
insert( TREE_TYPE value )
{
      TreeNode *current;
      TreeNode **link;

      /*
      **  从根节点开始。
      */
      link = &tree;
```


bbs.theithome.com

```
        /*
        ** 持续查找值，进入合适的子树。
        */
        while( (current = *link) != NULL ){
        /*
        **  根据情况，进入左子树或右子树（确认没有出现重复的值）
        */
            if( value < current->value )
                link = &current->left;
            else {
                assert( value != current->value );
                link = &current->right;
            }
    }

    /*
    ** 分配一个新节点，使适当节点的 link 字段指向它。
    */
    current = malloc( sizeof( TreeNode ) );
    assert( current != NULL );
    current->value = value;
    current->left = NULL;
    current->right = NULL;
    *link = current;
    }

    /*
    **   find
    */
    TREE_TYPE *
    find( TREE_TYPE value )
    {
       TreeNode    *current;

       /*
       **  从根节点开始，直到找到这个值，进入合适的子树。
       */
       current = tree;

       while( current != NULL && current->value != value ){
       /*
       **  根据情况，进入左子树或右子树。
       */
          if( value < current->value )
                  current = current->left;
          else
                  current = current->right;
        }

        if( current != NULL )
                  return &current->value;
        else
             return NULL;
       }

       /*
```


bbs.theithome.com

```
** do_pre_order_traverse
**  执行一层前序遍历。这个帮助函数用于保存我们当前正在处理的节点的信息。
**  这个函数并不是用户接口的一部分。
*/
static void
do_pre_order_traverse( TreeNode *current,
   void (*callback)( TREE_TYPE value ) )
{
   if( current != NULL ){
       callback( current->value );
       do_pre_order_traverse( current->left, callback );
       do_pre_order_traverse( current->right, callback );
   }
}

/*
**   pre_order_traverse
*/
void
pre_order_traverse( void (*callback)( TREE_TYPE value ) )
{
    do_pre_order_traverse( tree, callback );
}
```

程序 17.9　链式二叉搜索树　　l_tree.c

让每个树节点拥有一个指向它的双亲节点的指针常常是很有用的。用户可以利用这个双亲节点指针在树中上下移动。这种更为开放的树的 find 函数可以返回一个指向这个树节点的指针而不是节点值，这就允许用户利用这个指针执行其他形式的遍历。

程序的最后一个可供改进之处是用一个 destroy_tree 函数释放所有分配给这棵树的内存。这个函数的实现留作练习。

17.5　实现的改进

本章的实现方法说明了不同的 ADT 是如何工作的。但是，当它们用于现实的程序时，它们在好几个方面是不够充分的。本节的目的是找出这些问题并建议如何解决它们。我们使用数组形式的堆栈作为例子，但这里所讨论的技巧适用于其他所有 ADT。

17.5.1　拥有超过一个的堆栈

到目前为止的实现中，最主要的一个问题是它们把用于保存结构的内存和那些用于操纵它们的函数都封装在一起了。这样一来，一个程序便不能拥有超过一个的堆栈！

这个限制很容易解决，只要从堆栈的实现模块中去除数组和 top_element 的声明，并把它们放入用户代码即可。然后，它们通过参数被堆栈函数访问，这些函数便不再固定于某个数组。用户可以创建任意数量的数组，并通过调用堆栈函数将它们作为堆栈使用。

警告：

这个方法的危险之处在于它失去了封装性。如果用户拥有数据，他可以直接访问它。非法的访问，例如在一个错误的位置向数组增加一个新值或者增加一个新值但并不调整 top_element，都有可

bbs.theithome.com

能导致数据丢失或者非法数据或者导致堆栈函数运行失败。

一个相关的问题是当每个堆栈函数被调用时，用户应该确保向它传递正确的堆栈和 top_element 参数。如果这些参数发生混淆，其结果就是垃圾。我们可以通过把堆栈数组和它的 top_element 值捆绑在一个结构里来减少这种情况发生的可能性。

当堆栈模块包含数据时，就不存在出现上述两种问题的危险性。本章的练习部分描述了一个修改方案，允许堆栈模块管理超过一个的堆栈。

17.5.2 拥有超过一种的类型

即使前面的问题得以解决，存储于堆栈的值的类型在编译时便已固定，它就是 stack.h 头文件中所定义的类型。如果你需要一个整数堆栈和一个浮点数堆栈，你就没那么幸运了。

解决这个问题最简单的方法是另外编写一份堆栈函数的拷贝，用于处理不同的数据类型。这种方法可以达到目的，但它涉及大量重复代码，这就使得程序的维护工作变得更为困难。

一种更为优雅的方法是把整个堆栈模块实现为一个#define 宏，把目标类型作为参数。这个定义然后便可以用于创建每种目标类型的堆栈函数。但是，为了使这种解决方案得以运作，我们必须找到一种方法为不同类型的堆栈函数产生独一无二的函数名，这样它们相互之间就不会冲突。同时，你必须小心在意，对于每种类型只能创建一组函数，不管你实际需要多少个这种类型的堆栈。这种方法的一个例子在第 17.5.4 节描述。

第 3 种方法是使堆栈与类型无关，方法是让它存储 void *类型的值。将整数和其他数据都按照一个指针的空间进行存储，使用强制类型转换把参数的类型转换为 void *后再执行 push 函数，top 函数返回的值再转换回原先的类型。为了使堆栈也适用于较大的数据（例如结构），你可以在堆栈中存储指向数据的指针。

警告：

这种方法的问题是它绕过了类型检查。我们没有办法证实传递给 push 函数的值正是堆栈所使用的正确类型。如果一个整数意外地压入到一个元素类型为指针的堆栈中，其结果几乎肯定是一场灾难。

使树模块与类型无关更为困难一些，因为树函数必须比较树节点的值。但是，我们可以向每个树函数传递一个指向由用户编写的比较函数的指针。同样，传递一个错误的指针也会造成灾难性的后果。

17.5.3 名字冲突

堆栈和队列模块都拥有 is_full 和 is_empty 函数，队列和树模块拥有 insert 函数。如果你需要向树模块增加一个 delete 函数，它就会与原先存在于队列模块的 delete 函数发生冲突。

为了使它们共存于程序中，所有这些函数的名字都必须是独一无二的。但是，人们有一种强烈的愿望，在尽可能的情况下，让那些和每个数据结构关联的函数都保持“标准”名字。这个问题的解决方法是一种妥协方案：选择一种命名约定，使它既可以为人们所接受又能保证唯一性。例如，is_queue_empty 和 is_stack_empty 名字就解决了这个问题。它们的不利之处在于这些长名字使用起来不太方便，它们并未传递任何附加信息。

bbs.theithome.com

17.5.4　标准函数库的 ADT

计算机科学虽然不是一门古老的学科，但我们对它的研究显然已经花费了相当长的时间，对堆栈和队列的行为的方方面面已经研究得相当透彻了。那么，为什么每个人还需要自己编写堆栈和队列函数呢？为什么这些 ADT 不是标准函数库的一部分呢？

其原因正是我们刚刚讨论过的三个问题。名字冲突问题很容易解决，但是，类型安全性的缺乏以及让用户直接操纵数据的危险性使得用一种通用而又安全的方式编写实现堆栈的库函数变得极不可行。

解决这个问题就要求实现泛型(genericity)，它是一种编写一组函数，但数据的类型暂时可以不确定的能力。这组函数随后用用户需要的不同类型进行实例化(instantiated)或创建。C 语言并未提供这种能力，但我们可以使用#define 定义近似地模拟这种机制。

程序 17.10a 包含了一个#define 宏，它的宏体是一个数组堆栈的完整实现。这个#define 宏的参数是需要存储的值的类型、一个后缀以及需要使用的数组长度。后缀用于粘贴到由实现定义的每个函数名的后面，用于避免名字冲突。

程序 17.10b 使用程序 10.7a 的声明创建两个堆栈，一个可以容纳 10 个整数，另一个可以容纳 5 个浮点数。当每个#define 宏被扩展时，一组新的堆栈函数被创建，用于操作适当类型的数据。但是，如果需要两个整数堆栈，这种方法将会创建两组相同的函数。

我们将程序 17.10a 进行改写，把它分成三个独立的宏：一个用于声明接口，一个用于创建操纵数据的函数，一个用于创建数据。当我们需要第 1 个整数堆栈时，所有三个宏均被使用。当我们还需要另外的整数堆栈时，通过重复调用最后一个宏来实现。堆栈的接口也应该进行修改。函数必须接受一个附加的参数用于指定进行操作的堆栈。这些修改都留作练习。

这个技巧使得创建泛型抽象数据类型库成为可能。但是，这种灵活性是要付出代价的。用户需要承担几个新的责任。现在，他必须：

1. 采用一种命名约定，避免不同类型间堆栈的名字冲突。
2. 必须保证为每种不同类型的堆栈只创建一组堆栈函数。
3. 在访问堆栈时，必须保证使用适当的名字（例如，push_int 或 push_float 等）。
4. 确保向函数传递正确的堆栈数据结构。

毫不吃惊的是，用 C 语言实现泛型是相当困难的，因为它的设计远早于泛型这个概念被提出之时。泛型是面向对象编程语言处理得比较完美的问题之一。

```
/*
** 用静态数组实现一个泛型的堆栈。数组的长度当堆栈实例化时作为参数给出。
*/
#include <assert.h>

#define   GENERIC_STACK( STACK_TYPE, SUFFIX, STACK_SIZE )        \

          static    STACK_TYPE     stack##SUFFIX[ STACK_SIZE ];  \
          static    int            top_element##SUFFIX = -1;     \
                                                                  \
          int                                                     \
          is_empty##SUFFIX( void )                                \
          {                                                       \
```


bbs.theithome.com

```
        return top_element##SUFFIX == -1;                    \
    }                                                        \
                                                             \
    int                                                      \
    is_full##SUFFIX( void )                                  \
    {                                                        \
        return top_element##SUFFIX == STACK_SIZE - 1;        \
    }                                                        \
                                                             \
    void                                                     \
    push##SUFFIX( STACK_TYPE value )                         \
    {                                                        \
        assert( !is_full##SUFFIX() );                        \
        top_element##SUFFIX += 1;                            \
        stack##SUFFIX[ top_element##SUFFIX ] = value;        \
    }                                                        \
                                                             \
    void                                                     \
    pop##SUFFIX( void )                                      \
    {                                                        \
        assert( !is_empty##SUFFIX() );                       \
        top_element##SUFFIX -= 1;                            \
    }                                                        \
                                                             \
    STACK_TYPE top##SUFFIX( void )                           \
    {                                                        \
        assert( !is_empty##SUFFIX() );                       \
        return stack##SUFFIX[ top_element##SUFFIX ];         \
    }
```

程序 17.10a　泛型数组堆栈　　g_stack.h

```
/*
**  一个使用泛型堆栈模块创建两个容纳不同类型数据的堆栈的用户程序。
*/
#include <stdlib.h>
#include <stdio.h>
#include "g_stack.h"

/*
**  创建两个堆栈，一个用于容纳整数，另一个用于容纳浮点数。
*/
GENERIC_STACK( int, _int, 10 )
GENERIC_STACK( float, _float, 5 )

int
main()
{
    /*
    **  往每个堆栈压入几个值。
    */
    push_int( 5 );
    push_int( 22 );
    push_int( 15 );
    push_float( 25.3 );
    push_float( -40.5 );
```


bbs.theithome.com

```
    /*
    ** 清空整数堆栈并打印这些值。
    */
    while( !is_empty_int() ){
         printf( "Popping %d\n", top_int() );
         pop_int();
    }

    /*
     ** 清空浮点数堆栈并打印这些值。
     */
    while( !is_empty_float() ){
         printf( "Popping %f\n", top_float() );
         pop_float();
    }

         return EXIT_SUCCESS;
    }
```

程序 17.10b　使用泛型数组堆栈　　g_client.c

17.6 总结

为 ADT 分配内存有三种技巧：静态数组、动态分配的数组和动态分配的链式结构。静态数组对结构施加了预先确定固定长度这个限制。动态数组的长度可以在运行时计算，如果需要数组也可以进行重新分配。链式结构对值的最大数量并未施加任何限制。

堆栈是一种后进先出的结构。它的接口提供了把新值压入堆栈的函数和从堆栈弹出值的函数。另一类接口提供了第 3 个函数，它返回栈顶元素的值但并不将其中堆栈中弹出。堆栈很容易使用数组来实现，我们可以使用一个变量，初始化为-1，用它记住栈顶元素的下标。为了把一个新值压入到堆栈中，这个变量先进行增值，然后这个值被存储到数组中。当弹出一个值时，在访问栈顶元素之后，这个变量进行减值。我们需要两个额外的函数来使用动态分配的数组。一个用于创建指定长度的堆栈，另一个用于销毁它。单链表也能很好地实现堆栈。通过在链表的头部插入，可以实现堆栈的压入。通过删除第 1 个元素，可以实现堆栈的弹出。

队列是一种先进先出的结构。它的接口提供了插入一个新值和删除一个现有值的函数。由于队列对它的元素所施加的次序限制，用循环数组来实现队列要比使用普通数组合适得多。当一个变量被当作循环数组的下标使用时，如果它处于数组的末尾再增值时，它的值就“环绕”到零。为了判断数组是否已满，你可以使用一个用于计数已经插入到队列中的元素数量的变量。为了使用队列的 front 和 rear 指针来检测这种情况，数组应始终至少保留一个空元素。

二叉搜索树(BST)是一种数据结构，它或者为空，或者具有一个值并拥有零个、一个或两个孩子（分别称为左孩子和右孩子），它的孩子本身也是一棵 BST。BST 树节点的值大于它的左孩子所有节点的值，但小于它的右孩子所有节点的值。由于这种次序关系的存在，在 BST 中查找一个值是非常高效的——如果节点并未包含需要查找的值，你总是可以知道接下来应该查找它的哪棵子树。为了向 BST 插入一个值，你首先进行查找。如果值未找到，就把它插入到查找失败的位置。当你从 BST 删除一个节点时，必须小心防止把它的子树同树的其他部分断开。树的遍历就是以某种次序处理它

bbs.theithome.com

的所有节点。有 4 种常见的遍历次序。前序遍历先处理节点，然后遍历它的左子树和右子树。中序遍历先遍历节点的左子树，然后处理该节点，最后遍历节点的右子树。后序遍历先遍历节点的左子树和右子树，最后处理该节点。层次遍历从根到叶逐层从左向右处理每个节点。数组可以用于实现 BST，但如果树不平衡，这种方法会浪费很多内存空间。链式 BST 可以避免这种浪费。

这些 ADT 的简单实现方法带来了三个问题。首先，它们只允许拥有一个堆栈、一个队列或一棵树。这个问题可以通过把为这些结构分配内存的操作从操纵这些结构的函数中分离出来。但这样做导致封装性的损失，增加了出错机会。第 2 个问题是无法声明不同类型的堆栈、队列和树。为每种类型单独创建一份 ADT 函数使代码的维护变得更为困难。一个更好的办法是用#define 宏实现代码，然后用目标类型对它进行扩展。不过，使用这种方法，你必须小心选择一种命名约定。另一种方法是通过把需要存储到 ADT 的值强制转换为 void *。这种策略的一个缺点是它绕过了类型检查。第 3 个问题是避免不同 ADT 之间以及同种 ADT 用于处理不同类型数据的各个版本之间避免名字冲突。我们可以创建 ADT 的泛型实现，但为了正确使用它们，用户必须承担更多的责任。

17.7 警告的总结

1. 使用断言检查内存是否分配成功是危险的。
2. 数组形式的二叉树节点位置计算公式假定数组的下标从 1 开始。
3. 把数据封装于对它进行操纵的模块可以防止用户不正确地访问数据。
4. 与类型无关的函数没有类型检查，所以应该小心，确保传递正确类型的数据。

17.8 编程提示的总结

1. 避免使用具有副作用的函数可以使程序更容易理解。
2. 一个模块的接口应该避免暴露它的实现细节。
3. 将数据类型参数化，使它更容易修改。
4. 只有模块对外公布的接口才应该是公用的。
5. 使用断言来防止非法操作。
6. 几个不同的实现使用同一个通用接口使模块具有更强的可互换性。
7. 复用现存的代码而不是对它进行改写。
8. 迭代比尾部递归效率更高。

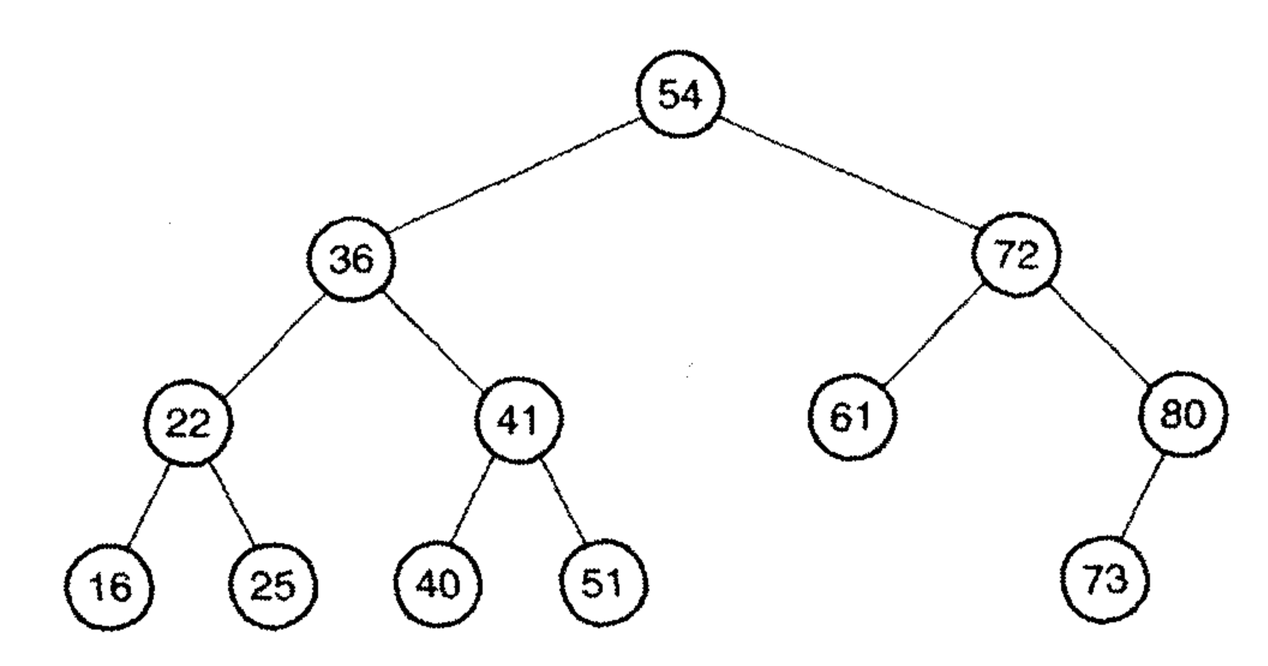
17.9 问题

1. 假定你有一个程序，它读取一系列名字，但必须以反序将它们打印出来。哪种 ADT 更适合完成这个任务？
2. 在超级市场的货架上摆放牛奶时，使用哪种 ADT 更为合适？你即需要考虑顾客购买牛奶，也需要考虑超级市场新到货一批牛奶的情况。
3. 在堆栈的传统接口中，pop 函数返回它从堆栈中删除的那个元素的值。在一个模块中提供两种接口是不是有可能？
4. 如果堆栈模块具有一个 empty 函数，用于删除堆栈中所有的值，你觉得模块的功能

bbs.theithome.com

是不是变得明显更为强大？

5. 在 push 函数中，top_element 在存储值之前先增值。但在 pop 函数中，它却在返回栈顶值后再减值。如果这两个次序弄反，会产生什么后果？
6. 如果在一个使用静态数组的堆栈模块中删除所有的断言，会产生什么后果？
7. 在堆栈的链式实现中，为什么 destroy_stack 函数从堆栈中逐个弹出每个元素。
8. 链式堆栈实现的 pop 函数声明了一个局部变量称为 first_node。这个变量可不可以省略？
9. 当一个循环数组已满时，front 和 rear 值之间的关系和堆栈为空时一样。但是，满和空是两种不同的状态。从概念上说，为什么会出现这种情况？
10. 有两种方法可用于检测一个已满的循环数组：(1)始终保留一个数组元素不使用。(2)另外增加一个变量，记录数组中元素的个数。哪种方法更好一些？
11. 编写语句，根据 front 和 rear 的值计算队列中元素的数量。
12. 实现队列可以使用单链表，也可以使用双链表，哪个更适合？
13. 画一棵树，它是根据下面的顺序把这些值依次插入到一棵二叉搜索树而形成的：20，15，18，32，5，91，-4，76，33，41，34，21，90。
14. 按照升序或降序把一些值插入到一棵二叉搜索树将导致树不平衡。在这样一棵树中查找一个值的效率如何？
15. 使用前序遍历，下面这棵树各节点的访问次序是怎么样的？中序遍历呢？后序遍历呢？层次遍历呢？



16. 改写 do_pre_order_traversal 函数，用于执行树的中序遍历。
17. 改写 do_pre_order_traversal 函数，用于执行树的后序遍历。
18. 二叉搜索树的哪种遍历方法可以以升序依次访问树中所有的节点？哪种遍历方法可以以降序依次访问树中所有的节点？
19. destroy_tree 函数通过释放所有分配给树中节点的内存来删除这棵树，这意味着所有的树节点必须以某个特定的次序进行处理。哪种类型的遍历最适合这个任务？

17.10 编程练习

★ 1. 在动态分配数组的堆栈模块中增加一个 resize_stack 函数。这个函数接受一个参数：

bbs.theithome.com

堆栈的新长度。

★★ 2. 把队列模块转换为使用动态分配的数组形式，并增加一个 resize_queue 函数（类似于第 1 题）。

★★★ 3. 把队列模块转换为使用链表实现。

★★★ 4. 堆栈、队列和树模块如果可以处理超过一个的堆栈、队列和树，它们会更加实用。修改动态数组堆栈模块，使它最多可以处理 10 个不同的堆栈。你将不得不对堆栈函数的接口进行修改，使它们接受另一个参数——需要使用的堆栈的索引。

★★ 5. 编写一个函数，计算一棵二叉搜索树的节点数量。你可以选择任何一种你喜欢的二叉搜索树实现形式。

★★★ 6. 编写一个函数，执行数组形式的二叉搜索树的层次遍历。使用下面的算法：

```
向一个队列添加根节点。
while 队列非空时:
    从队列中移除第 1 个节点并对它进行处理。
    把这个节点所有的孩子添加到队列中。
```

★★★★ 7. 编写一个函数，检查一棵树是不是二叉搜索树。你可以选择任何一种你喜欢的树实现形式。

★★★★★ 8. 为数组形式的树模块编写一个函数，用于从树中删除一个值。如果需要删除的值并未在树中找到，函数可以终止程序。

★★ 9. 为链式实现的二叉搜索树编写一个 destroy_tree 函数。函数应该释放树使用的所有内存。

★★★★★ 10. 为链式实现的树模块编写一个函数，用于从树中删除一个值。如果需要删除的值并未在树中找到，函数可以终止程序。

★★★★ 11. 修改程序 17.10a 的#define 定义，让它拥有三个单独的定义。

a. 一个用于声明堆栈接口

b. 一个用于创建堆栈函数的实现

c. 一个用于创建堆栈使用的数据

你必须修改堆栈的接口，把堆栈数据作为显式的参数传递给函数（把堆栈数据包装于一个结构中会更方便）。这些修改将允许一组堆栈函数操纵任意个对应类型的堆栈。

bbs.theithome.com

第18章

运行时环境

本章，我们将研究由某个特定的编译器为某个特定的计算机所产生的汇编语言代码，目的是学习一些关于这个编译器的运行时环境的几个有趣的内容。我们需要回答的几个问题是“我的运行时环境的限制是什么？”和“我如何使C程序和汇编语言程序一起工作？”

18.1 判断运行时环境

你的编译器或环境和我们在这里所看到的肯定不同，所以你将需要自己执行类似这样的试验以便找出在你的机器上它们是如何运作的。

第1个步骤是从你的编译器获得一个汇编语言代码列表。在UNIX系统中，编译器选项-S使编译器把每个源文件的汇编代码写到一个具有.s后缀的文件中。Borland编译器也支持这种选项，不过它使用的是.asm后缀。请参阅相关文档，获得其他系统的特定细节。

你还需要阅读你的机器上的汇编语言代码。你并不一定要成为一个熟练的汇编语言程序员，但你需要对每条指令的工作过程以及如何解释地址模型有一个基本的了解。一本描述你的计算机指令集的手册是完成这个任务的绝佳参考材料。

本章并不讲授汇编语言，因为这不是本书的要点。你的机器所产生的汇编语言很可能和本书的不一样。但是，如果你编译测试程序，我在这里对本书的汇编语言的解释可能有助于你分析你的机器上的汇编语言，因为这两种汇编程序实现了相同的源代码。

18.1.1 测试程序

让我们观察程序18.1，也就是测试程序。它包含了各种不同的代码段，它们的实现颇有意思。这个程序并没有实现任何有用的功能，但它并不需要如此——我们需要的只是观察编译器为它所产生的汇编代码。如果你希望研究你的运行时环境的其他方面，你可以修改这个程序，包含这些方面的例子。

```
/*
** 判断C运行时环境的程序。
*/
```


bbs.theithome.com

```
/*
** 静态初始化
*/
int  static_variable = 5;

void
f()
{
      register int  i1, i2, i3, i4, i5,
                    i6, i7, i8, i9, i10;
      register char*c1, *c2, *c3, *c4, *c5,
                  *c6, *c7, *c8, *c9, *c10;
      extern   inta_very_long_name_to_see_how_long_they_can_be;
      double   dbl;
      intfunc_ret_int();
      double  func_ret_double();
      char    *func_ret_char_ptr();

      /*
      ** 寄存器变量的最大数量。
      */
      i1 = 1; i2 = 2; i3 = 3; i4 = 4; i5 = 5;
      i6 = 6; i7 = 7; i8 = 8; i9 = 9; i10 = 10;
      c1 = (char *)110; c2 = (char *)120;
      c3 = (char *)130; c4 = (char *)140;
      c5 = (char *)150; c6 = (char *)160;
      c7 = (char *)170; c8 = (char *)180;
      c9 = (char *)190; c10 = (char *)200;

      /*
      **  外部名字
      */
      a_very_long_name_to_see_how_long_they_can_be = 1;

      /*
      **  函数调用/返回协议，堆栈帧（过程活动记录）
      */
      i2 = func_ret_int( 10, i1, i10 );
      dbl = func_ret_double();
      c1 = func_ret_char_ptr( c1 );
}

int
func_ret_int( int a, int b, register int c )
{
      intd;

      d = b - 6;
      return a + b + c;
}

double
func_ret_double()
{
      return 3.14;
}
```


bbs.theithome.com

```
char *
func_ret_char_ptr( char *cp )
{
        return cp + 1;
}
```

程序 18.1　测试程序　　runtime.c

程序 18.2 的汇编代码是由一台使用 Motorola 68000 处理器家族的计算机产生的。我对代码进行了编辑，使它看上去更清晰，我还去掉了一些不相关的声明。

这是一个很长的程序。和绝大部分的编译器输出一样，它没有包含帮助读者阅读的注释。但你不要被它吓倒！我将逐行解释绝大部分代码。我采用的方法是分段解释，先显示一小段 C 代码，后面是根据它产生的汇编代码。完整的代码列表只是作为参考而给出，这样你可以观察所有这些小段例子是如何组成一个整体的。

```
        .data
        .even
        .globl    _static_variable
_static_variable:
        .long     5
        .text

        .globl    _f
_f:     linka6,#-88
        moveml    #0x3cfc,sp@
        moveq     #1,d7
        moveq     #2,d6
        moveq     #3,d5
        moveq     #4,d4
        moveq     #5,d3
        moveq     #6,d2
        movl      #7,a6@(-4)
        movl      #8,a6@(-8)
        movl      #9,a6@(-12)
        movl      #10,a6@(-16)
        movl      #110,a5
        movl      #120,a4
        movl      #130,a3
        movl      #140,a2
        movl      #150,a6@(-20)
        movl      #160,a6@(-24)
        movl      #170,a6@(-28)
        movl      #180,a6@(-32)
        movl      #190,a6@(-36)
        movl      #200,a6@(-40)
        movl      #1,_a_very_long_name_to_see_how_long_they_can_be
        movl      a6@(-16),sp@-
        movl      d7,sp@-
        pea       10
        jbsr      _func_ret_int
        lea       sp@(12),sp
        movl      d0,d6
        jbsr      _func_ret_double
        movl      d0,a6@(-48)
```


bbs.theithome.com

```
        movl      d1,a6@(-44)
        pea       a5@
        jbsr      _func_ret_char_ptr
        addqw     #4,sp
        movl      d0,a5
        moveml    a6@(-88),#0x3cfc
        unlk      a6
        rts

        .globl    _func_ret_int
_func_ret_int:
        link      a6,#-8
        moveml    #0x80,sp@
        movl      a6@(16),d7
        movl      a6@(12),d0
        subql     #6,d0
        movl      d0,a6@(-4)
        movl      a6@(8),d0
        addl      a6@(12),d0
        addl      d7,d0
        moveml    a6@(-8),#0x80
        unlk      a6
        rts

        .globl    _func_ret_double
_func_ret_double:
        link      a6,#0
        moveml    #0,sp@
        movl      L2000000,d0
        movl      L2000000+4,d1
        unlk      a6
        rts
L2000000:.long      0x40091eb8,0x51eb851f

        .globl    _func_ret_char_ptr
_func_ret_char_ptr:
        link      a6,#0
        moveml    #0,sp@
        movl      a6@(8),d0
        addql     #1,d0
        unlk      a6
        rts
```

程序 18.2　测试程序的汇编语言代码　　runtime.s

18.1.2　静态变量和初始化

测试程序所执行的第 1 项任务是在静态内存中声明并初始化一个变量。

```
/*
**  静态初始化
*/
int       static_variable = 5;
```


bbs.theithome.com

```
.data
.enen
.global  _static_variable
_static_variable:
.long    5
```

汇编代码的一开始是两个指令，分别表示进入程序的数据区以及确保变量开始于内存的偶数地址。68000 处理器要求边界对齐。然后变量被声明为全局类型。注意变量名以一个下划线开始。许多（但不是所有）C 编译器会在 C 代码所声明的外部名字前加一个下划线，以免与各个库函数所使用的名字冲突。最后，编译器为变量创建空间，并用适当的值对它进行初始化。

18.1.3　堆栈帧

接下来是函数 f。一个函数分成三个部分：**函数序**(prologue)、**函数体**(body)和**函数跋**(epilogue)。函数序用于执行函数启动需要的一些工作，例如为局部变量保留堆栈中的内存。函数跋用于在函数即将返回之前清理堆栈。当然，函数体是用于执行有用工作的地方。

```
void
f()
{
        register int    i1, i2, i3, i4, i5,
                        i6, i7, i8, i9, i10;
        register char   *c1, *c2, *c3, *c4, *c5,
                        *c6, *c7, *c8, *c9, *c10;
        extern  int     a_very_long_name_to_see_...
        double  dbl;
        int     func_ret_int( );
        double  func_ret_double();
        char    *func_ret_char_ptr( );
```

```
        .text
        .globl    _f
_f:     link      a6, #-88
        moveml    #0x3cfc,sp@
```

这些指令的第 1 条表示进入程序的代码（文本）段，紧随其后的是函数名的全局声明。注意在名字前面也有一条下划线。第 1 条可执行指令开始为函数创建**堆栈帧**(stack frame)。堆栈帧是堆栈中的一个区域，函数在那里存储变量和其他值。link 指令将在稍后详细解释，现在你只需要记住它在堆栈中保留了 88 个字节的空间，用于存储局部变量和其他值。

这个代码序列中的最后一条指令把选定寄存器中的值复制到堆栈中。68000 处理器有 8 个用于操纵数据的寄存器，它们的名字是从 d0 至 d7。还有 8 个寄存器用于操纵地址，它们的名字是从 a0 至 a7。值 0x3cfc 表示寄存器 d2 至 d7、a2 至 a5 中的值需要被存储，这些值就是前面提到的“其他值”。稍后你就会明白为什么这些寄存器的值需要进行保存。

局部变量声明和函数原型并不会产生任何汇编代码。但如果任何局部变量在声明时进行了初始化，那么这里也会出现指令用于执行赋值操作。

18.1.4　寄存器变量

接下来便是函数体。测试程序的这部分代码的目的是判断寄存器里可以存储多少个变量。它声明了许多寄存器变量，每个都用不同的值进行初始化。汇编代码通过显示每个值在何处存储来回答这个问题。

bbs.theithome.com

```
/*
**  寄存器变量的最大数量。
*/
i1 = 1; i2 = 2; i3 = 3; i4 = 4; i5 = 5;
i6 = 6; i7 = 7; i8 = 8; i9 = 9; i10 = 10;
c1 = (char *)110; c2 = (char *)120;
c3 = (char *)130; c4 = (char *)140;
c5 = (char *)150; c6 = (char *)160;
c7 = (char *)170; c8 = (char *)180;
c9 = (char *)190; c10 = (char *)200;
```

```
moveq    #1,d7
moveq    #2,d6
moveq    #3,d5
moveq    #4,d4
moveq    #5,d3
moveq    #6,d2
movl     #7,a6@(-4)
movl     #8,a6@(-8)
movl     #9,a6@(-12)
movl     #10,a6@(-16)
movl     #110,a5
movl     #120,a4
movl     #130,a3
movl     #140,a2
movl     #150,a6@(-20)
movl     #160,a6@(-24)
movl     #170,a6@(-28)
movl     #180,a6@(-32)
movl     #190,a6@(-36)
movl     #200,a6@(-40)
```

整型变量首先进行初始化。注意值 1 至 6 被存放在数据寄存器，但 7 至 10 却被存放在其他地方。这段代码显示了最多只能有 6 个整型值可以被存放在数据寄存器。那么其他不是整型的数据又如何呢？有些编译器不会把字符型变量存放在寄存器中。在有些机器上，double 的长度太长，无法存放在寄存器中。有些机器具有特殊的寄存器，用于存放浮点值。我们可以很容易地对测试程序进行修改来发现这些细节。

接下来的几条指令对指针变量进行初始化。前 4 个值被存放在寄存器，最后那个值被存放在其他地方。因此，这个编译器最多允许 4 个指针变量存放在寄存器中。那么其他类型的指针变量又是如何呢？同样，我们也需要进行试验。但是，在许多机器上，不管指针指向什么类型的东西，它的长度是固定的。所以你可能会发现任何类型的指针都可以存放在寄存器中。

那么其他变量存放在什么地方呢？机器使用的地址模型执行间接寻址和索引操作。这种组合工作颇似数组的下标引用。寄存器 a6 称为**帧指针**(frame pointer)，它指向堆栈帧内部的一个“引用”位置。堆栈帧中的所有值都是通过这个引用位置再加上一个偏移量进行访问的。a6@(-28)指定了一个偏移地址-28。注意偏移位置从-4 开始，每次增长 4。这台机器上的整型值和指针都占据 4 个字节的内存。使用这些偏移地址，你可以建立一张映射表，准确地显示堆栈中的每个值相对于帧指针 a6 的位置。

我们已经见到寄存器 d2 至 d7、a2 至 a5 用于存放寄存器变量，现在很清楚为什么这些寄存器需要在函数序中进行保存。函数必须对任何将用于存储寄存器变量的寄存器进行保存，这样它们原先的值可以在函数返回到调用函数前恢复，这样就能保留调用函数的寄存器变量。

关于寄存器变量最后还要提一点：为什么寄存器 d0-d1、a0-a1 以及 a6-a7 并未用于存放寄存器

bbs.theithome.com

变量呢？在这台机器上，a6 用作帧指针，而 a7 是堆栈指针（这个汇编语言给它取了个别名 sp）。后面有个例子将显示 d0 和 d1 用于从函数返回值，所以它们不能用于存放寄存器变量。

但是，在这个程序的代码里并没有明确显示 a0 或 a1 的用途。显而易见的结论是它们将用于某种目的，但这个测试程序并不包含这种类型的代码。要回答这个问题需要进行进一步的试验。

18.1.5　外部标识符的长度

接下来的测试用于确定外部标识符所允许的最大长度。这个测试看上去够简单了：用一个长名字声明并使用一个变量，看看会发生什么。

```
/*
**  外部名字
*/
a_very_long_name_to_see_how_long_they_can_be = 1
```

```
movl   #1, a_very_long_name_to_see_how_long_they_can_be
```

从这段代码似乎可以看出，名字的长度并没有限制。更精确地说，这个名字未超出限制。为了找出这个限制，你可以不断加长这个名字，直到发现汇编程序把这个名字截短。

警告：

事实上，这个测试是不够充分的。外部名字的最终限制是链接器施加的，它很可能愉快地接受任何长度的名字但忽略除前几个字符以外的其他字符。标准要求外部名字至少区分前 6 个字符（但并不要求区分大小写）。为了测试链接器做了些什么，我们只要简单地链接程序并检查一下结果的装入映像表(load map)和名字列表。

18.1.6　判断堆栈帧布局

运行时堆栈保存了每个函数运行时所需要的数据，包括它的自动变量和返回地址。接下来的几个测试将确定两个相关的内容：堆栈帧的组织形式，调用和从函数返回的协议。它们的结果显示了如何提供 C 和汇编程序的接口。

一、传递函数参数

这个例子从调用一个函数开始。

```
/*
** 函数调用/返回协议、堆栈帧。
*/
i2=func_ret_int(10,i1,i10);
```

```
movl      a6@(-16), sp@-
movl      d7, sp@-
pea       10
jbsr      _func_ret_int
```

前 3 条指令把函数的参数压入到堆栈中。被压入的第 1 个参数存储于 a6@(-16)：这个我们原先

bbs.theithome.com

讨论过的偏移地址显示这个值就是变量 i10。然后被压入的是 d7，它包含了变量 i1。最后一个参数的压入方式和前两个不同。pea 指令简单地把它的操作数压入到堆栈中，这是一种高效的压入字面值常量的方法。为什么参数要以它们在参数列表中的相反次序逐个压到堆栈中？我们很快就能找到这个答案。

这些指令一开始创建属于即将被调用的函数的堆栈帧。通过跟踪指令并记住它们的效果，我们可以勾勒一幅关于堆栈帧的完整的图。如果你需要从汇编语言的层次追踪一个 C 程序的执行过程，这幅图可以向你提供一些有用的信息。图 18.1 显示了到目前为止所创建的内容。图中显示低内存地址位于顶部而高内存地址位于底部。当值压入堆栈时，堆栈向低地址方向生长（向上）。在原先的堆栈指针以下的堆栈内容是未知的，所以在图中以一个问号显示。

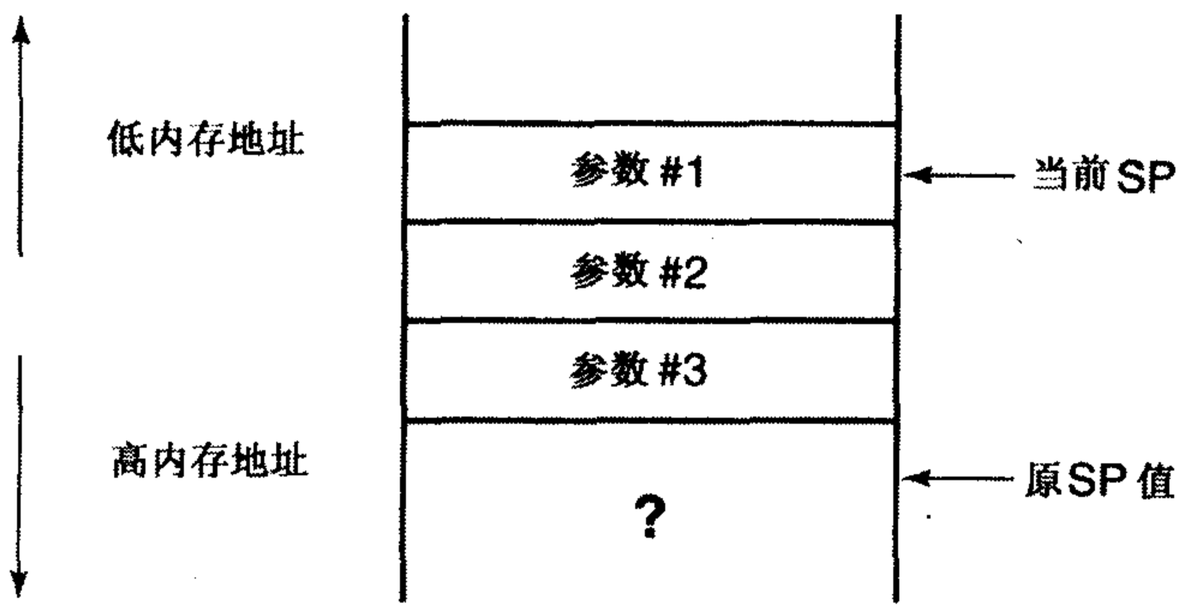


图 18.1　压入参数后的堆栈帧

接下来的指令是一个“跳转子程序(jump subroutine)”。它把返回地址压入到堆栈中，并跳转到 _func_ret_int 的起始位置。当被调用函数结束任务后需要返回到它的调用位置时，就需要使用这个压入到堆栈中的返回地址。现在，堆栈的情况如图 18.2 所示。

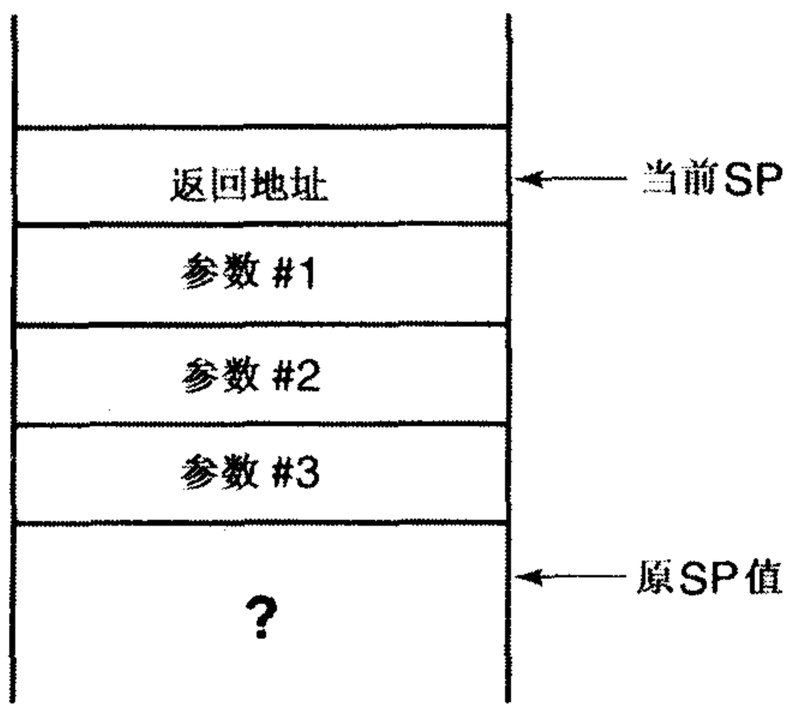


图 18.2　在跳转子程序指令之后的堆栈帧

二、函数序

接下来，执行流来到被调用函数的函数序：

bbs.theithome.com

```
int
func_ret_int( int a, int b, register int c )
{
        int     d;
```

```
        .globl  _func_ret_int
_func_ret_int:
        link    a6,#-8
        moveml  #0x80,sp@
        movl    a6@(16),d7
```

这个函数序类似于我们前面观察的那个。我们对指令必须进行更详细的研究以便完整地弄清整个堆栈帧的映像。link 指令分成几个步骤。首先，a6 的内容被压入到堆栈中。其次，堆栈指针的当前值被复制到 a6。图 18.3 显示了这个结果。

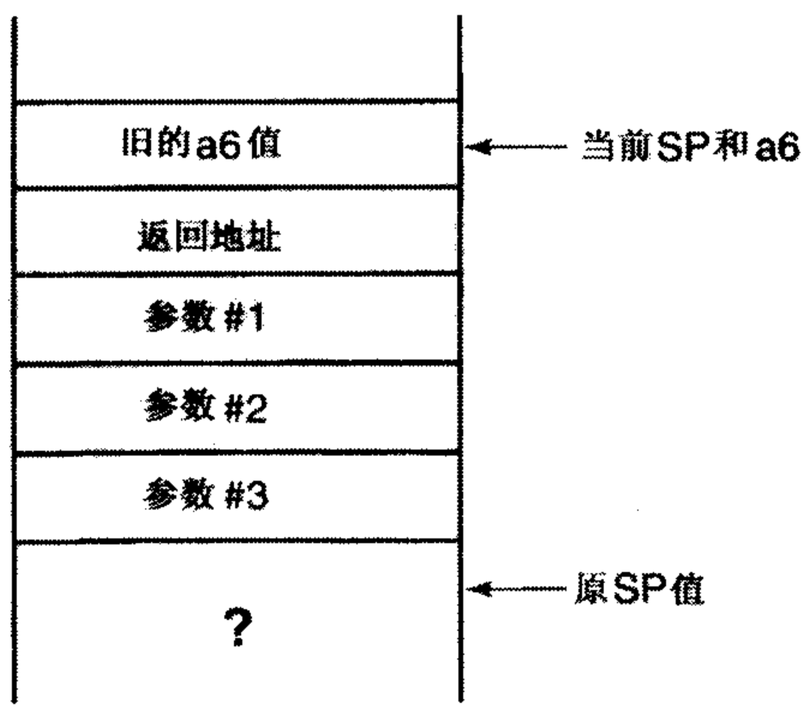


图 18.3　link 指令期间的堆栈帧

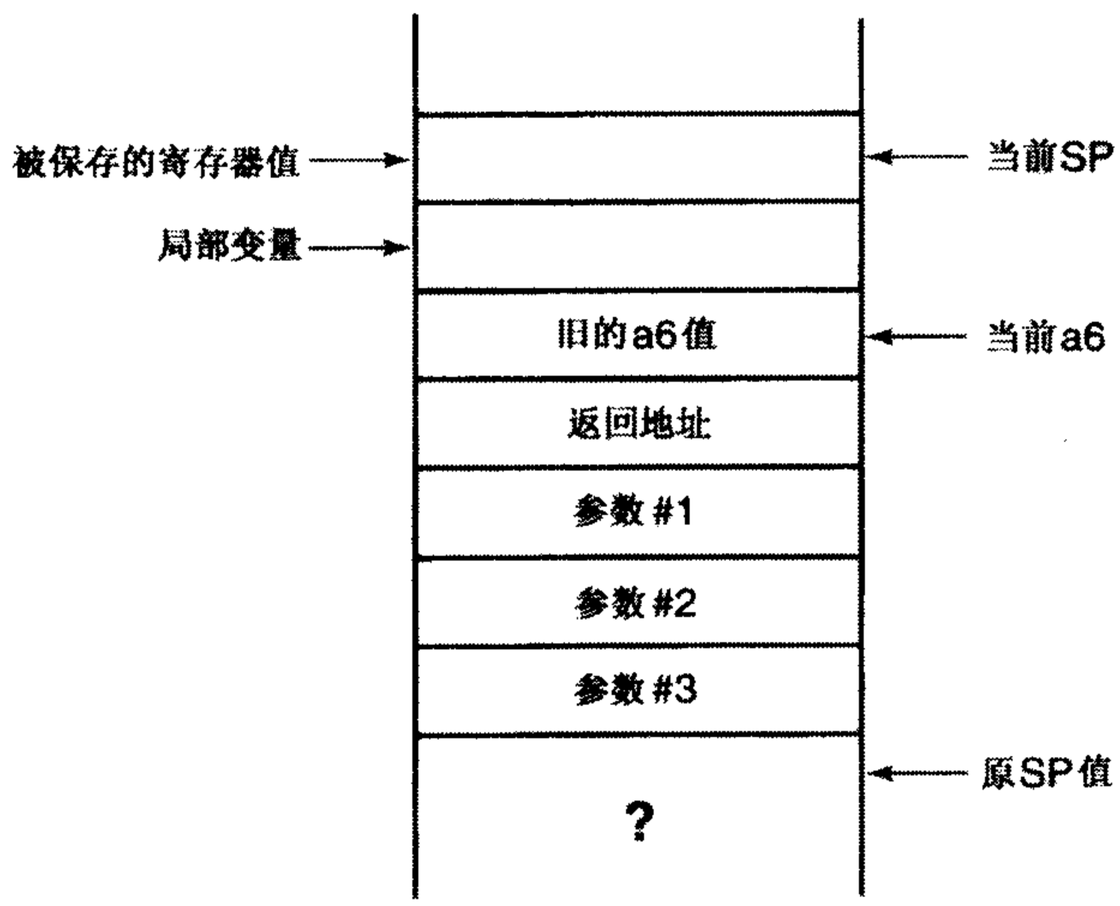


图 18.4　link 指令之后的堆栈帧

最后，link 指令从堆栈指针中减去 8。和以前一样，这将创建空间用于保存局部变量和被保存的寄存器值。下一条指令把一个单一的寄存器保存到堆栈帧。操作数 0x80 指定寄存器 d7。寄存器

bbs.theithome.com

存储在堆栈的顶部，它提示堆栈帧的顶部就是寄存器值保存的位置。堆栈帧剩余的部分必然是局部变量存储的地方。图 18.4 显示了到目前为止我们所知道的堆栈帧的情况。

函数序所执行的最后一个任务是从堆栈复制一个值到 d7。函数把第 3 个参数声明为寄存器变量，这第 3 个参数的位置是从帧指针往下 16 个字节。在这台机器上，寄存器变量在函数序中正常地通过堆栈传递并复制到一个寄存器。这条额外的指令带来了一些开销——如果函数中并没有很多指令使用这个参数，那么它在时间或空间上的节约将无法弥补把参数复制到寄存器而带来的开销。

三、堆栈中的参数次序

我们现在可以推断出为什么参数要按参数列表相反的顺序压入到堆栈中。被调用函数使用帧指针加一个偏移量来访问参数。当参数以反序压入到堆栈时，参数列表的**第 1 个**参数便位于堆栈中这堆参数的顶部，它距离帧指针的偏移量是一个常数。事实上，**任何**一个参数距离帧指针的偏移量都是一个常数，这和堆栈中压入多少个参数并无关系。

如果参数以相反的顺序压入到堆栈中又会怎样呢（也就是按照参数列表的顺序）？这样一来，第 1 个参数距离帧指针的偏移量就和压入到堆栈的参数数量有关。编译器可以计算出这个值，但还是存在一个问题——实际传递的参数数量和函数期望接受的参数数量可能并不相同。在这种情况下，这个偏移量就是不正确的，当函数试图访问一个参数时，它实际所访问的将不是它想要的那个。

那么在反序方案中，额外的参数是如何处理的呢？堆栈帧的图显示任何额外的参数都将位于前几个参数的下面，第 1 个参数距离帧指针的距离将保持不变。因此，函数可以正确地访问前三个参数，对于额外的参数可以简单地忽略。

提示：

如果函数知道存在额外的参数，在这台机器上，函数可以通过取最后一个参数的地址并增加堆栈指针的值来访问它们的值。但更好的方法是使用 stdarg.h 文件定义的宏，它们提供了一个可移植的接口来访问可变参数。

四、最终的堆栈帧布局

这个编译器所产生的堆栈帧的映像到此就完成了，它在图 18.5 中显示。

让我们继续观察这个函数：

```
        d = b - 6;
        return a + b + c;
}
```

```
movl      a6@(12), d0
subql     #6, d0
movl      d0, a6@(-4)
movl      a6@(8), d0
addl      a6@(12), d0
addl      d7, d0
moveml    a6@(-8), #0x80
unlk      a6
rts
```

通过堆栈帧映像，我们很容易判断第 1 条 movl 指令是把第 2 个参数复制到 d0。下一条指令将这个值减去 6，第 3 条指令把结果存储到局部变量 d。d0 的作用是计算过程中的“中间结果暂存器”

bbs.theithome.com

或临时位置。这也是它不能用于存放寄存器变量的原因之一。

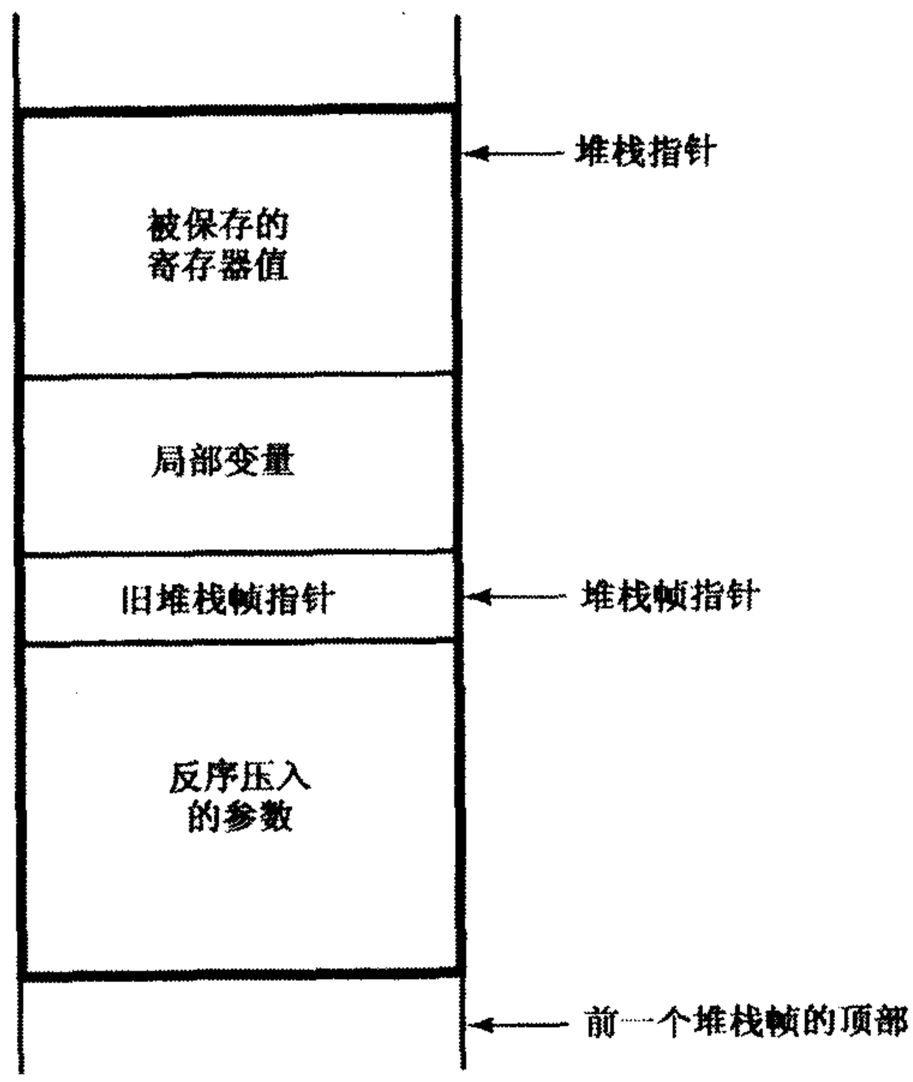


图 18.5　堆栈帧布局

接下来的三条指令对 return 语句进行求值。这个值就是我们希望返回给调用函数的值。但在这里，结果值存放在 d0 中。记住这个细节，以后会用到。

五、函数跋

这个函数的函数跋以一条 moveml 指令开始，它用于恢复以前被保存的寄存器值。然后 unkl(unlink)指令把 a6 的值复制给堆栈指针并把从堆栈中弹出的 a6 的旧值装入到 a6 中。这个动作的效果就是清除堆栈帧中返回地址以上的那部分内容。最后，rts 指令通过把返回地址从堆栈中弹出到程序计数器，从而从该函数返回。

现在，执行流从调用程序的地点继续。注意此时堆栈尚未被完全清理。

```
i2 = func_ret_int( 10, i1, i10 );
```

```
lea     sp@(12),sp
movl    d0,d6
```

当我们返回到调用程序之后执行的第 1 条指令就是把 12 加到堆栈指针。这个加法运算有效地把参数值从堆栈中弹出。现在，堆栈的状态就和调用函数前的状态完全一样了。

有趣的是，**被调用**函数并没有从堆栈中完全清除它的整个堆栈帧：参数还留在那里等待**调用**函数清除。同样，它的原因和可变参数列表有关。调用函数把参数压到堆栈上，所以只有它才知道堆栈中到底有多少个参数。因此，只有调用函数可以安全地清除它们。

六、返回值

函数跋并没有使用 d0，因此它依然保存着函数的返回值。第 2 条指令在从函数返回后执行，它

bbs.theithome.com

把 d0 的值复制到 d6，后者是变量 i2 的存放位置，也就是结果所在的位置。

在这个编译器中，函数返回一个值时把它存放在 d0，调用函数从被调用函数返回之后从 d0 获取这个值。这个协议是 d0 不能用于存放寄存器变量的另一个原因。

下一个被调用的函数返回一个 double 值。

```
dbl = func_ret_double();
c1 = func_ret_char_ptr( c1 );
```

```
jbsr    _func_ret_double
movl    d0,a6@(-48)
movl    d1,a6@(-44)

pea     a5@
jbsr    _func_ret_char_ptr
addqw   #4,sp
movl    d0,a5
```

这个函数并没有任何参数，所以没有什么东西压入到堆栈中。在这个函数返回之后，d0 和 d1 的值都被保存。在这台机器上，double 的长度是 8 个字节，无法放入一个寄存器中。因此，要返回这种类型的值，必须同时使用 d0 和 d1 寄存器。

最后那个函数调用说明了指针变量是如何从函数中返回的：它们也是通过 d0 进行传递的。不同的编译器可能通过 a0 或其他寄存器来传递它们。这个程序的剩余指令属于这个函数的函数序部分。

18.1.7 表达式的副作用

在第 4 章，我曾提到如果像下面这样的表达式

```
y + 3;
```

出现在程序中，它将会被求值但不会对程序产生影响，因为它的结果并未保存。接着我在一个脚注里说明它实际上**可以**以一种微妙的方式对程序的执行产生影响。

考虑程序 18.3，它被认为将返回 a+b 的值。这个函数计算一个结果但并不返回任何东西，因为这个表达式被错误地从 return 语句中省略。但使用这个编译器，这个函数实际上可以返回这个值！d0 被用于计算 x，并且由于这个表达式是最后进行求值的，所以当函数结束时 d0 仍然保存了这个结果值。所以这个函数很意外地向调用函数返回了正确的值。

```
/*
** 尽管存在一个巨大错误，但仍能在某些机器上正确运行的函数。
*/
int
erroneous( int a, int b )
{
        intx;

        /*
        ** 计算答案，并返回它
        */
        x = a + b;
        return;
}
```

程序 18.3 一个意外地返回正确值的函数 no_ret.c

bbs.theithome.com

现在假定我们在 return 语句之前插入了这样一个表达式：

```
a + 3;
```

这个新表达式将修改 d0 的值。即使这个表达式的结果并未存储于任何变量中，但它还是影响了程序的执行，因为它修改了这个函数的返回值。

类似的问题也可以由于调试语句引起。如果你增加了一条语句

```
printf( "Function returns the value %d\n", x );
```

把它插入到 return 语句之前，函数也将不会返回正确的值。如果删除了这条语句，函数又能正确运行。当你发现一条调试语句也能改变程序的行为时，你心中的挫折感可想而知！

之所以可能出现这些效果，其罪魁祸首是原先存在的那个错误——return 语句省略了表达式。这种现象听上去好像不太可能，但令人吃惊的是，在一些老式的编译器里经常出现这种情况，这是因为当它们发现一个函数应该返回某个值但实际上并未返回任何值时并不会向程序员发出警告。

18.2　C 和汇编语言的接口

这个试验已经显示了编写能够调用 C 程序或者被 C 程序调用的汇编语言程序所需要的内容。与这个环境相关的结果总结如下——**你的环境肯定在某些方面与它不同！**

首先，汇编程序中的名字必须遵循外部标识符的规则。在这个系统中，它必须以一个下划线开始。

其次，汇编程序必须遵循正确的函数调用/返回协议。有两种情况：从一个汇编语言程序调用一个 C 程序和从一个 C 程序调用一个汇编程序。为了从汇编语言程序调用 C 程序：

1．如果寄存器 d0、d1、a0 或 a1 保存了重要的值，它们必须在调用 C 程序之前进行保存，因为 C 函数不会保存它们的值。

2．任何函数的参数必须以参数列表相反的顺序压入到堆栈中。

3．函数必须由一条“跳转子程序”类型的指令调用，它会把返回地址压入到堆栈中。

4．当 C 函数返回时，汇编程序必须清除堆栈中的任何参数。

5．如果汇编程序期望接受一个返回值，它将保存在 d0（如果返回值的类型为 double，它的另一半将位于 d1）。

6．任何在调用之前进行过保存的寄存器此时可以恢复。

为了编写一个由 C 程序调用的汇编程序：

1．保存任何你希望修改的寄存器（除 d0、d1、a0 和 a1 之外）。

2．参数值从堆栈中获得，因为调用它的 C 函数把参数压入在堆栈中。

3．如果函数应该返回一个值，它的值应保存在 d0 中（在这种情况下，d0 **不能**进行保存和恢复）。

4．在返回之前，函数必须清除任何它压入到堆栈中的内容。

在你的汇编程序中创建一个完全 C 风格的堆栈帧并无必要。你所要做的就是调用一个能够以正确的方式压入参数并当它返回时能够正确地执行清理任务的函数。在一个由 C 程序调用的汇编程序里，你必须访问 C 函数放置在那里的参数。

在你实际编写汇编函数之前，你需要知道你机器上的汇编语言。一些简陋的能够让我们明白一个现有的汇编程序是如何工作的知识对于编写新程序是远远不够的。

程序 18.4 和 18.5 是两个从 C 函数调用汇编函数以及从汇编函数调用 C 函数的例子。虽然它们

bbs.theithome.com

都是特定于这个环境的，但对于说明这方面的情况还是非常有用的。第 1 个例子是一个汇编语言程序，它返回 3 个整型参数的和。这个函数并没有费心完成堆栈帧，它只是计算参数的和并返回。我们将以下面的方式从一个 C 函数中调用这个函数：

```
sum = sum_three_values( 25, 14, -6 );
```

第 2 个例子显示了一段汇编语言程序，它需要打印 3 个值，它调用 printf 函数来完成这项工作。

```
|
| 对三个整数求和，并返回这个值。
|
        .text

        .globl    _sum_three_values
_sum_three_values:
        movl      sp@(4),d0       |Get 1st arg,
        addl      sp@(8),d0       |add 2nd arg,
        addl      sp@(12),d0      |add last arg.
        rts                       |Return.
```

程序 18.4　对 3 个整数求和的汇编语言程序　　sum.s

```
|
| 需要打印三个值，x,y 和 z。
|
        movl      z,sp@-            | Push args on the
        movl      y,sp@-            | stack in reverse
        movl      x,sp@-            | order: format, x,
        movl      #format,sp@-      | y, and z.
        jbsr      _printf           | Now call printf
        addl      #16,sp            | Clean up stack
        \&...
        .data
format:.ascii  "x = %d, y = %d,   and z = %d"
       .byte   012, 0               | Newline and null
       .even
x:     .long   25
y:     .long   45
z:     .long   50
```

程序 18.5　调用 printf 函数的汇编语言程序　　printf.s

18.3　运行时效率

什么时候一个程序在老式的计算机上会“太大”呢？当程序增长后的容量超过了内存的数量时，它就无法运行，因此它就属于“太大”。即使在一些现代的机器上，一个必须存储于 ROM 的程序必须相当小才有可能装入到有限的内存空间中[1]。

但许多现代的计算机系统在这方面的限制大不如前，这是因为它们提供了**虚拟内存**(virtual memory)。虚拟内存是由操作系统实现的，它在需要时把程序的活动部分放入内存并把不活动的部分复制到磁盘中，这样就允许系统运行大型的程序。但程序越大，需要进行的复制就越多。所以大

[1] 只读内存(ROM, Read Only Memory)就是无法进行修改的内存。它通常用于存储那些在计算机上控制一些设备的程序。

bbs.theithome.com

型程序不是像以前那样根本无法运行，而是随着程序的增大，它的执行效率逐渐降低。所以，什么时候程序显得“太大”呢？就是当它运行得太慢的时候。

程序的执行速度显然与它的体积有关。程序执行的速度越慢，使用这个程序就会显得越不舒服。我们很难界定究竟在哪一点一个程序突然会被扣上一顶“太慢”的帽子。除非它必须对一些它自身无法控制的物理事件作出反应。例如，一个操作 CD 播放器的程序如果处理数据的速度无法赶上数据从 CD 传送过来的速度，它显然就太慢了。

提高效率

现代的经过优化的编译器在从一个 C 程序产生高效的目标代码方面做得非常好。因此，你把时间花在对代码进行一些小的修改以便使它效率更高常常并不是很合算。

提示：

如果一个程序太大或太慢，较之钻研每个变量，看看把它们声明为 register 能不能提高效率，选择一种效率更高的算法或数据结构往往效果要满意得多。然而，这并不是说你可以在代码中胡作非为，因为风格恶劣的代码总是会把事情弄得更糟。

如果一个程序太大，你很容易想到从哪里着手可以使程序变得更小：最大的函数和数据结构。但如果一个程序太慢，你该从何处着手提高它的速度呢？答案是对程序进行性能评测，简单地说就是测算程序的每个部分在执行时所花费的时间。花费时间最多的那部分程序显然是优化的目标。程序中使用最频繁的那部分代码运行速度如果能更快一些，将能够大大提高程序的整体运行速度。

绝大多数 UNIX 系统都具有性能评测工具，这些工具在许多其他操作系统中也有。图 18.6 是其中一个这类工具的输出的一部分。它显示了在某个特定程序的执行期间每个函数所耗费时间的名次

```
 Seconds    #Calls   Function Name
-------------------------------------
    4.94   293423   malloc
    3.21   272593   free
    2.85   658973   _nextch_from_chrlst
    2.82   272593   _insert
    2.69   791309   _check_traverse
    2.57     9664   _lookup_macro
    1.35   372915   _append_to_chrlst
    1.23   254501   _interpolate
    1.10   302714   _next_input_char
    1.09   285031   _input_filter
    0.91   197235   demote
    0.90   272419   putfreehdr
    0.82   285031   _nextchar
    0.79     7620   _lookup_number_register
    0.77    63946   _new_character
    0.65   292822   allocate
    0.57   272594   _getfreehdr
    0.51    34374   _next_text_char
    0.46   151006   _duplicate_char
    0.41     6473   _expression
    0.37     8843   _sub_expression
    0.35    23774   _skip_white_space
    0.34   203535   _copy_interpolate
    0.32    10984   _copy_function
    0.31   133032   _duplicate_ascii_char
    0.31      604   _process_filled_text
    0.31    52627   _next_ascii_char
```

图 18.6　性能评测样例信息

bbs.theithome.com

以及它所耗费的时间（以秒为单位）。这个程序的总共执行时间是 32.95 秒。我们可以从这个列表中发现三个有趣的地方。

1．在耗费时间最多的函数中，有些是库函数。在这个例子里，malloc 和 free 占据了前两位。你无法修改它们的实现方式，但在重新设计程序时，如果能够不用或少用动态内存分配，程序的执行速度在最多情况下可以提高 25%。

2．有些函数之所以耗费了大量的时间是因为它们被调用的次数非常多。即使每次单独调用时它的速度很快，由于调用次数多，所以总的时间不少。_nextch_from_chrlst 就是其中一例。这个函数每次调用所耗费的时间只有 4.3 微秒。由于它是如此之短，所以你通过对函数进行改进大幅度提高它的执行速度的可能性非常之小。但是，就是因为它的调用次数非常多，所以它还是值得加以关注。加上几个明智的 register 声明稍微提高函数的效率，对程序的总体性能可能还是会有较大的改善。

3．有些函数调用的次数并不多，但每次调用所花费的时间却很长。例如，_loopup_macro 平均每次调用要花费 265 微秒的时间。为这个函数寻找一种更快的算法最多可以使程序的速度提高 7.75%。[1]

作为最后一招，你可以对单个函数用汇编语言重新编码，函数越小，重新编码就越容易。这种方法的效果可能很好，因为在小型函数中，C 的函数序和函数跋所耗费的固定开销在执行时间中所占的比例不小。对较大的函数进行重新编码要困难得多，因此把你的时间花在这个地方效率不是很高。

性能评测常常并不能告诉你原先不知道的东西，但有时候它的结果可能相当出人意料。性能评测的优点在于你可能弄清你正在花时间研究的那部分程序可能会带来最大程度的性能提高。

18.4 总结

我们在这台机器上研究的有些任务在许多其他环境中也是以这些方式实现的。例如，绝大多数环境都创建某种类型的堆栈帧，函数用它来保存它们的数据。堆栈帧的细节可能各不相同，但它们的基本思路是相当一致的。

其他一些任务在不同的环境中可能差异较大。有些计算机具有特殊的硬件用于保存函数的参数，所以它们的处理方式和我们所看到的可能大不一样。其他机器在传递函数值时也可能采用不同的方式。

警告：

事实上，不同的编译器可能在相同的机器上产生不同的代码。另一种在我们的测试机器上使用的编译器能够使用 9 至 14 个寄存器变量（具体数目取决于一些其他情况）。不同的编译器可能具有不同的堆栈帧约定或者在函数的调用和返回上使用不兼容的协议。因此，在通常情况下，你不能使用不同的编译器编译同一个程序的不同片段。

提高程序效率的最好方法是为它选择一种更好的算法。接下来的一种提高程序执行速度的最佳手段是对程序进行性能评测，看看程序的哪个地方花费的时间最多。你把优化措施集中在程序的这部分将产生最好的结果。

[1] 事实上我们还需要注意第 4 点。malloc 的调用次数比 free 多了 20 833 次，所以有些内存被泄漏了。

bbs.theithome.com

提示：

学习机器的运行时环境既有益处又存在危险——说它有用是因为你获得的知识允许你做一些其他方法无法完成的事情，说它危险是因为程序中如果存在任何依赖于这方面知识的东西，可能会损害程序的可移植性。现在这个时代，计算机发展的速度很快，许多机器还没有摆到货架上就已经过时。因此，程序从一台机器转换到另一台机器的可能性是非常现实的，所以我们非常希望代码具有良好的可移植性。

18.5 警告的总结

1. 是链接器而不是编译器决定外部标识符的最大长度。
2. 你无法链接由不同编译器产生的程序。

18.6 编程提示的总结

1. 使用 stdarg 实现可变参数列表。
2. 改进算法比优化代码更有效率。
3. 使用某种环境特有的技巧会导致程序不可移植。

18.7 问题

1. 在你的环境中，堆栈帧的样子是什么样的？
2. 在你的系统中，有意义的外部标识符最长可以有多少个字符？
3. 在你的环境中，寄存器可以存储多少个变量？对于指针和非指针值，它是不是进行了任何区分？
4. 在你的环境中，参数是如何传递给函数的？值是如何从函数返回的？
5. 在本章我们所使用的这台机器上，如果一个函数把它的一个或多个参数声明为寄存器变量，那么这个函数的参数在函数序中和平常一样被压入到堆栈中，然后再复制到正确的寄存器中。如果这些参数能够直接保存到寄存器，函数的效率会更高一些。这种参数传递技巧能够实现吗？如果能，怎么实现呢？
6. 在我们所讨论的环境中，调用函数负责清除它压入到堆栈中的参数。那么，能不能由被调用函数来完成这项任务呢？如果不能，那么在满足什么条件下它才可能呢？
7. 如果说汇编语言程序比 C 程序效率更高，那么为什么不用汇编语言来编写所有程序呢？

18.8 编程练习

★ 1. 为你的系统编写一个汇编语言函数，它接受 3 个整型参数并返回它们的和。

★ 2. 编写一个汇编语言程序，创建 3 个整型值并调用 printf 函数把它们打印出来。

★★ 3. 假定 stdarg.h 文件被意外地从你的系统中删除。请编写一组第 7 章所描述的 stdarg 宏。

bbs.theithome.com

bbs.theithome.com

附录　部分问题答案

本书的附录部分节选了各章的一些问题和编程练习的答案。对于编程练习，除了这里给出的答案，应该还有很多其他正确的答案。

第 1 章　问题

1.2　声明只需要编写一次，这样以后维护和修改它时会更容易。同样，声明只编写一次消除了在多份拷贝中出现写法不一致的机会。

1.5　`scanf( "%d %d %s", &quantity, &price, department );`

1.8　当一个数组作为函数的参数进行传递时，函数无法知道它的长度。因此，gets 函数没有办法防止一个非常长的输入行，从而导致 input 数组溢出。fgets 函数要求数组的长度作为参数传递给它，因此不存在这个问题。

第 1 章　编程练习

1.2　通过从输入中逐字符进行读取而不是逐行进行读取，可以避免行长度限制。在这个解决方案中，如果定义了 TRUE 和 FALSE 符号，程序的可读性会更好一些，但这个技巧在本章尚未讨论。

```
/*
** 从标准输入复制到标准输出，并对输出行标号
*/

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

int
main()
      {
      int ch;
      int line;
      int at_beginning;

      line = 0;
```


bbs.theithome.com

```
        at_beginning = 1;
        /*
        **  读取字符并逐个处理它们。
        */
        while( (ch = getchar()) != EOF ){
            /*
            **  如果我们位于一行的起始位置，打印行号。
            */
            if( at_beginning == 1 ){
                at_beginning = 0;
                line += 1;
                printf( "%d ", line );
            }

            /*
            **  打印字符，并对行尾进行检查。
            */
            putchar( ch );
            if( ch == '\n' )
                at_beginning = 1;
        }

        return EXIT_SUCCESS;
    }
```

解决方案 1.2 number.c

1.5 当输出行已满时，我们仍然可以中断循环，但在其他情况下循环必须继续。我们必须同时检查每个范围内已经复制了多少个字符，以防止一个 NUL 字节过早地被复制到输出缓冲区。这里是一个修改方案，用于完成这项工作。

```
/*
** 处理一个输入行，方法是把指定列的字符连接在一起。输出行用 NUL 结尾。
*/

void
rearrange( char *output, char const *input,
    int const n_columns, int const columns[] )
{
        int     col;            /* columns 数组的下标 */
        int     output_col;     /* 输出列计数器 */
        int     len;            /* 输入行的长度 */

        len = strlen( input );
        output_col = 0;

        /*
        ** 处理每对列号
        */
        for( col = 0; col < n_columns; col += 2 ){
            int  nchars = columns[col + 1] - columns[col] + 1;

            /*
            **  如果输入行没这么长，跳过这个范围。
            */
```


bbs.theithome.com

```
        if( columns[col] >= len )
            continue;

        /*
        ** 如果输出数组已满，任务就完成。
        */
        if( output_col == MAX_INPUT - 1 )
            break;

        /*
        ** 如果输出数组空间不够，只复制可以容纳的部分。
        */
        if( output_col + nchars > MAX_INPUT - 1 )
            nchars = MAX_INPUT - output_col - 1;

        /*
        ** 观察输入行中多少个字符在这个范围里面。如果它小于 nchars，
        ** 对 nchars 的值进行调整。
        */
        if( columns[col] + nchars - 1 >= len )
            nchars = len - columns[col];

        /*
        ** 复制相关的数据。
        */
        strncpy( output + output_col, input + columns[col],
            nchars );
        output_col += nchars;
    }

    output[output_col] = '\0';
}
```

解决方案 1.5　　rearran2.c

第 2 章　问题

2.4　假定系统使用的是 ASCII 字符集，存在下面的相等关系。

\40 = 32 = 空格字符

\100 = 64 = '@'

\x40 = 64 = '@'

\x100 占据 12 位（尽管前三位为零）。在绝大多数机器上，这个值过于庞大，无法存储于一个字符内，所以它的结果因编译器而异。

\0123 由两个字符组成，'\012'和'3'。其结果值因编译器而异。

\x0123 过于庞大，无法存储于一个字符内，其结果值因编译器而异。

2.7　有对有错。对：除了预处理指令之外，语言并没有对程序应该出现的外观施加任何规则。错：风格恶劣的程序难以维护或无法维护，所以除了极为简单的程序之外，绝大多数程序的编写风格是非常重要的。

2.8　这两个程序的 while 循环都缺少一个用于结束语句的右花括号。但是，第 2 个程序更容易发现这个错误。这个例子说明了在函数中对语句进行缩进的价值。

bbs.theithome.com

2.11 当一个头文件被修改时，所有包含它的文件都必须重新编译。

如果这个文件被修改	这些文件必须重新编译
list.c	list.c
list.h	list.c, table.c, main.c
table.h	table.c, main.c

Borland C/C++编译器的 Windows 集成开发环境在各个文件中寻找这些关系并自动只编译那些需要重新编译的文件。UNIX 系统有一个称为 make 的工具，用于执行相同的任务。但是，要使用这个工具，你必须创建一个“makefile”，它用于描述各个文件之间的关系。

第 2 章 编程练习

2.2 这个程序很容易通过一个计数器实现。但是，它并没有像初看上去那么简单。使用“}{”这个输入测试你的解决方案。

```
/*
** 检查一个程序的花括号对。
*/

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

int
main()
{
        int ch;
        int braces;

        braces = 0;

        /*
        ** 逐字符读取程序。
        */
        while( (ch = getchar()) != EOF ){
            /*
            ** 左花括号始终是合法的。
            */
            if( ch == '{' )
                braces += 1;

            /*
            ** 右花括号只有当它和一个左花括号匹配时才是合法的。
            */
            if( ch == '}' )
                if( braces == 0 )
                    printf( "Extra closing brace!\n" );
                else
                    braces -= 1;
        }

        /*
        ** 没有更多输入: 验证不存在任何未被匹配的左花括号。
```


bbs.theithome.com

```
        */
        if( braces > 0 )
            printf( "%d unmatched opening brace(s)!\n", braces );

        return EXIT_SUCCESS;
}
```

解决方案 2.2　　braces.c

第 3 章　问题

3.3　声明整型变量名，使变量的类型必须有一个确定的长度（如 int8、int16、int32）。对于你希望成为缺省长度的整数，根据它所能容纳的最大值，使用类似 defint8、defint16 或 defint32 这样的名字。然后为每台机器创建一个名为 int_sizes.h 的文件，它包含一些 typedef 声明，为你创建的类型名字选择最合适的整型长度。在一台典型的 32 位机器上，这个文件将包含：

```
typedef signed char     int8;
typedef short int       int16;
typedef int             int32;
typedef int             defint8;
typedef int             defint16;
typedef int             defint32;
```

在一台典型的 16 位整数机器上，这个文件将包含：

```
typedef signed char     int8;
typedef int             int16;
typedef long int        int32;
typedef int             defint8;
typedef int             defint16;
typedef long int        defint32;
```

你也可以使用#define 指令。

3.7　变量 jar 是一个枚举类型，但它的值实际上是个整数。但是，printf 格式代码%s 用于打印字符串而不是整数。结果，我们无法判断它的输出会是什么样子。如果格式代码是%d，那么输出将会是：

```
32
48
```

3.10　否。任何给定的 n 个位的值只有 2^n 个不同的组合。一个有符号值和无符号值仅有的区别在于它的一半值是如何解释的。在一个有符号值中，它们是负值。在一个无符号值中，它们是一个更大的正值。

3.11　float 的范围比 int 大，但如果它的位数不比 int 更多，它并不能比 int 表示更多不同的值。前一个问题的答案已经提示了它们应该能够表示的不同值的数量是相同的，但在绝大多数浮点系统中，这个答案是错误的。零通常有许多种表示形式，而且通过使用不规范的小数形式，其他值也具有多种不同的表示形式。因此，float 能够表示的不同值的数量比 int 少。

3.21　是的，这是可能的，但你不应该指望它。而且，即使不存在其他的函数调用，它们的值也很可能不同。在有些架构的机器上，一个硬件中断将把机器的状态信息压到堆栈上，它们将破坏这些变量。

bbs.theithome.com

第 4 章　问题

4.1　它是合法的，但它不会影响程序的状态。这些操作符都不具有副作用，并且它们的计算结果并没有赋值给任何变量。

4.4　使用空语句

```
if( condition )
        ;
else {
        statements
}
```

你可以对条件进行修改，省略空的 then 子句。它们的效果是一样的。

```
if( ! ( condition ) ){
        statements
}
```

4.9　由于不存在 break 语句，所以对于每个偶数，这两条信息都将打印出来。

```
odd
even
odd
odd
even
odd
```

4.12　如果一开始处理最为特殊的情况，以后再处理更为普通的情况，你的任务会更轻松一些。

```
if( year % 400 == 0 )
        leap_year = 1;
else if( year % 100 == 0 )
        leap_year = 0;
else if( year % 4 == 0 )
        leap_year = 1;
else
        leap_year = 0;
```

第 4 章　编程练习

4.1　必须使用浮点变量，而且程序应该对负值输入进行检查。

```
/*
** 计算一个数的平方根。
*/

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

int
main()
{
        float   new_guess;
        float   last_guess;
        float   number;

        /*
        ** 催促用户输入，读取数据并对它进行检查。
        */
        printf( "Enter a number: " );
        scanf( "%f", &number );
```


bbs.theithome.com

```
    if( number < 0 ){
            printf( "Cannot compute the square root of a "
                "negative number!\n" );
        return EXIT_FAILURE;
    }

    /*
    ** 计算平方根的近似值，直到它的值不再变化。
    */
    new_guess = 1;
    do {
        last_guess = new_guess;
        new_guess = ( last_guess + number / last_guess ) / 2;
        printf( "%.15e\n", new_guess );
    } while( new_guess != last_guess );

    /*
    ** 打印结果。
    */
    printf( "Square root of %g is %g\n", number, new_guess );

    return EXIT_SUCCESS;
}
```

解决方案 4.1 sqrt.c

4.4　src 向 dst 的赋值可以蕴含在 if 语句内部。

```
/*
** 从 src 中的字符串向 dst 数组准确地复制 N 个字符（如果需要，用 NUL 进行填充）。
*/
void
copy_n( char dst[], char src[], int n )
{
        int dst_index, src_index;

        src_index = 0;

        for( dst_index = 0; dst_index < n; dst_index += 1 ){
            dst[dst_index] = src[src_index];
            if( src[src_index] != 0 )
                src_index += 1;
        }
}
```

解决方案 4.4 copy_n.c

第 5 章　问题

5.2　这是一个狡猾的问题。比较明显的回答是-10(2 – 3 * 4)，但实际上它因编译器而异。乘法运算必须在加法运算之前完成，但并没有规则规定函数调用完成的顺序。因此，下面几个答案都是正确的：

bbs.theithome.com

```
-10   ( 2 - 3 * 4 ) or ( 2 - 4 * 3 )
 -5   ( 3 - 2 * 4 ) or ( 3 - 4 * 2 )
 -2   ( 4 - 2 * 3 ) or ( 4 - 3 * 2 )
```

5.4　不，它们都执行相同的任务。如果你比较吹毛求疵，使用 if 的那个方案看上去稍微臃肿一些，因为它具有两条存储到 i 的指令。但是，它们之间只有一条指令才会执行，所以在速度上并无区别。

5.6　()操作符本身并无任何副作用，但它所调用的函数可能有副作用。

操作符	副 作 用
++, --	不论是前缀还是后缀形式，这些操作符都会修改它们的操作数
=	包括所有其他的复合赋值符：它们都修改作为左值的左操作数

第 5 章　编程练习

5.1　应该提倡的转换字母大小写的方法是使用 tolower 库函数。如下所示：

```
/*
** 将标准输入复制到标准输出，将所有大写字母转换为小写字母。注意，它依赖于
** 这个事实：如果参数并非大写字母，tolower 函数将不修改它的参数，直接返回
** 它的值。
*/
#include <stdio.h>
#include <ctype.h>

int
main( void )
{
     int  ch;

     while( (ch = getchar()) != EOF )
          putchar( tolower( ch ) );
}
```

解决方案 5.1a　　uc_lc.c

不过，我们此时还没有讨论这个函数，所以下面是另一种方案：

```
/*
** 将标准输入复制到标准输出，把所有的大写字母转换为小写字母。
*/
#include <stdio.h>

int
main( void )
{
     int  ch;

     while( (ch = getchar()) != EOF ){
          if( ch >= 'A' && ch <= 'Z' )
               ch += 'a' - 'A';
          putchar( ch );
```


bbs.theithome.com

```
    }
}
```

解决方案 5.1b　　uc_lc_b.c

这第 2 个程序在使用 ASCII 字符集的机器上运行良好。但在那些大写字母并不连续的字符集（如 EBCDIC）中，它就会对非字母字符进行转换，从而违反了题目的规定，所以最好的方法还是使用库函数。

5.3　对位的计数不使用硬编码，可以避免可移植性问题。这个解决方案使用一个位在一个无符号整数中进行移位来控制创建答案的循环。

```
    /*
    ** 在一个无符号整数值中翻转位的顺序。
    */

    unsigned int
    reverse_bits( unsigned int value )
{
    unsigned int  answer;
    unsigned int  i;

    answer = 0;

    /*
    ** 只是 i 不是 0 就继续进行。这就使循环与机器的字长无关，从而避免了可移植性问题。
    */
    for( i = 1; i != 0; i <<= 1 ){
        /*
        ** 把旧的 answer 左移 1 位，为下一个位留下空间;
        ** 如果 value 的最后一位是 1，answer 就与 1 进行 OR 操作;
        ** 然后将 value 右移至下一个位。
        */
        answer <<= 1;
        if( value & 1 )
            answer |= 1;
        value >>= 1;
    }

    return answer;
}
```

解决方案 5.3　　reverse.c

第 6 章　问题

6.1　机器无法作出判断。编译器根据值的声明类型创建适当的指令，机器只是盲目地执行这些指令而已。

6.4　这是很危险的。首先，解引用一个 NULL 指针的结果因编译器而异，所以程序不应该这样做。允许程序在这样的访问之后还能继续运行是很不幸的，因为这时程序很可能并没有正确运行。

6.6　有两个错误。对增值后的指针进行解引用时，数组的第 1 个元素并没有被清零。另外，指针在越过数组的右边界以后仍然进行解引用，它将把其他某个内存地址的内容清零。

bbs.theithome.com

注意 pi 在数组之后立即声明。如果编译器恰好把它放在紧跟数组后面的内存位置，结果将是灾难性的。当指针移到数组后面的那个内存位置时，那个最后被清零的内存位置就是保存指针的位置。这个指针（现在变成了零）因此仍然小于&array[ARRAY_SIZE]，所以循环将继续执行。指针在它被解引用之前增值，所以下一个被破坏的值就是存储于内存位置 4 的变量（假定整数的长度为 4 个字节）。如果硬件并没有捕捉到这个错误并终止程序，这个循环将快乐地继续下去，指针在内存中欢快地前行，破坏它遇见的所有值。当它再一次到达这个数组的位置时，就会重复上面这个过程，从而导致一个微妙的无限循环。

第 6 章　编程练习

6.3　这个算法的关键是当两个指针相遇或擦肩而过时就停止。否则，这些字符将翻转两次，实际上相当于没有任何效果。

```
/*
** 翻转参数字符串。
*/

void reverse_string( char *str )
{
char*last_char;

/*
** 把 last_char 设置为指向字符串的最后一个字符。
*/
for( last_char = str; *last_char != '\0'; last_char++ )
     ;

last_char--;

/*
** 交换 str 和 last_char 指向的字符，然后 str 前进一步，last_char 后退一
** 步，在两个指针相遇或擦肩而过之前重复这个过程。
*/
while( str < last_char ){
     char temp;

     temp = *str;
     *str++ = *last_char;
     *last_char-- = temp;
 }
}
```

解决方案 6.3　　rev_str.c

第 7 章　问题

7.1　当存根函数被调用时，打印一条消息，显示它已被调用，或者也可以打印作为参数传递给它的值。

7.7　这个函数假定当它被调用时传递给它的正好是 10 个元素的数组。如果参数数组更大一些，

bbs.theithome.com

它就会忽略剩余的元素。如果传递一个不足 10 个元素的数组，函数将访问数组边界之外的值。

7.8　递归和迭代都必须设置一些目标，当达到这些目标时便终止执行。每个递归调用和循环的每次迭代必须取得一些进展，进一步靠近这些目标。

第 7 章　编程练习

7.1　Hermite polynomials 用于物理学和统计学。它们也可以作为递归练习在程序中使用。

```
/*
** 计算 Hermite polynomial 的值
**
**    输入:
**          n, x: 用于标识值
**
**    输出:
**          polynomial 的值 (返回值)
*/

int
hermite( int n, int x )
{
      /*
      ** 处理不需要递归的特殊情况。
      */
      if( n <= 0 )
            return 1;
      if( n == 1 )
            return 2 * x;

      /*
      ** 否则，递归地计算结果值。
      */
      return 2 * x * hermite( n - 1, x ) -
         2 * ( n - 1 ) * hermite( n - 2, x );
}
```

解决方案 7.1　　hermite.c

7.3　这个问题应该用迭代方法解决，而不应采用递归方法。

```
/*
** 把一个数字字符串转换为一个整数。
*/

int
ascii_to_integer( char *string )
{
      int  value;

      value = 0;

      /*
      ** 逐个把字符串的字符转换为数字。
```


bbs.theithome.com

```
        */
        while( *string >= '0' && *string <= '9' ){
             value *= 10;
             value += *string - '0';
             string++;
        }

        /*
        ** 错误检查: 如果由于遇到一个非数字字符而终止，把结果设置为 0。
        */
        if( *string != '\0' )
             value = 0;

        return value;
    }
```

解决方案 7.3 atoi.c

第 8 章 问题

8.1 其中两个表达式的答案无法确定，因为我们不知道编译器选择在什么地方存储 ip。

ints	100	ip	112
ints[4]	50	ip[4]	80
ints + 4	116	ip + 4	128
*ints + 4	14	*ip + 4	44
*(ints + 4)	50	*(ip + 4)	80
ints[-2]	非法	ip[-2]	20
&ints	100	&ip	未知
&ints[4]	116	&ip[4]	128
&ints + 4	116	&ip + 4	未知
&ints[-2]	非法	&ip[-2]	104

8.5 经常，一个程序 80%的运行时间用于执行 20%的代码，所以其他 80%的代码的语句对效率并不是特别敏感，所以使用指针获得的效率上的提高抵不上其他方面的损失。

8.8 在第 1 个赋值中，编译器认为 a 是一个指针变量，所以它提取存储在那里的指针值，并加上 12（3 和整型的长度相乘），然后对这个结果执行间接访问操作。但 a 实际上是整型数组的起始位置，所以作为“指针”获得的这个值实际上是数组的第 1 个整型元素。它与 12 相加，其结果解释为一个地址，然后对它进行间接访问。作为结果，它或者将提取一些任意内存位置的内容，或者由于某种地址错误而导致程序失败。

在第 2 个赋值中，编译器认为 b 是个数组名，所以它把 12（3 的调整结果）加到 b 的存储地址，然后间接访问操作从那里获得值。事实上，b 是个指针变量，所以从内存中提取的后面三个字实际上是从另外的任意变量中取得的。这个问题说明了指针和数组虽然存在关联，但绝不是相同的。

8.12 当执行任何“按照元素在内存中出现的顺序对元素进行访问”的操作时。例如，初始化一个数组、读取或写入超过一个的数组元素、通过移动指针访问数组的底层内存“压扁”数组等都属于这类操作。

8.17 第 1 个参数是个标量，所以函数得到值的一份拷贝。对这份拷贝的修改并不会影响原先的参数，所以 const 关键字的作用并不是防止原先的参数被修改。

bbs.theithome.com

第 2 个参数实际上是一个指向整型的指针。传递给函数的是指针的拷贝，对它进行修改并不会影响指针参数本身，但函数可以通过对指针执行间接访问修改调用程序的值。const 关键字用于防止这种修改。

第 8 章　编程练习

8.2　由于这个表相当短，所以也可以使用一系列的 if 语句实现。我们使用的是一个循环，它既可以用于短表，也适用于长表。这个表（类似于税务指南这样的小册子）把许多值都显示了不止一次，目的是为了使指令更加清楚。这里给出的解决方案并没有存储这些冗余值。注意数据被声明为 static，这是为了防止用户程序直接访问它。如果数据存储于结构而不是数组中，程序会更好一些，但我们现在还没有学习结构。

```
/*
** 计算 1995 年美国联邦政府对每位公民征收的个人收入所得税。
*/

#include <float.h>

static    double    income_limits[]
 = { 0,    23350,    56550,  117950,  256500,  DBL_MAX };
static    float     base_tax[]
 = { 0,    3502.5, 12798.5, 31832.5, 81710.5 };
static    float     percentage[]
 = { .15, .28,  .31,       .36,       .396 };

double
single_tax( double income )
{
     int  category;

     /*
     ** 找到正确的收入类别。DBL_MAX 被添加到这个列表的末尾，保证循环不会进
     ** 行得太久。
     */
     for( category = 1;
         income >= income_limits[ category ];
         category += 1 )
           ;
     category -= 1;

     /*
     ** 计算税。
     */
     return base_tax[ category ] + percentage[ category ] *
         ( income - income_limits[ category ] );
}
```

解决方案 8.2　　sing_tax.c

8.5　考虑到程序实际完成的工作，它实际上是相当紧凑的。由于它和矩阵的大小无关，所以这个函数不能使用下标——这个程序是一个使用指针的好例子。但是，从技术上说它是非法的，因为

bbs.theithome.com

它将压扁数组。

```
/*
** 将两个矩阵相乘。
*/

void
matrix_multiply( int *m1, int *m2, register int *r,
    int x, int y, int z )
{
     register int  *m1p;
     register int  *m2p;
     register int  k;
     int  row;
     int  column;

     /*
     ** 外层的两个循环逐个产生结果矩阵的元素。由于这是按照存在顺序进行的。
     ** 我们可以通过对 r 进行间接访问来访问这些元素。
     */
     for( row = 0; row < x; row += 1 ){
          for( column = 0; column < z; column += 1 ){
               /*
               ** 计算结果的一个值。这是通过获得指向 m1 和 m2 的合适元素的指针，
               ** 当我们进行循环时，使它们前进来实现的。
               */
               m1p = m1 + row * y;
               m2p = m2 + column;
               *r = 0;

               for( k = 0; k < y; k += 1 ){
                    *r += *m1p * *m2p;
                    m1p += 1;
                    m2p += z;
               }

               /*
               ** r 前进一步，指向下一个元素。
               */
               r++;
          }
     }
}
```

解决方案 8.5 matmult.c

第 9 章　问题

9.1　这个问题存在争议（虽然我作出了一个结论）。目前这种方法的优点是操纵字符数组的效率和访问的灵活性。它的缺点是有可能引起错误：溢出数组，使用的下标超出了字符串的边界，无法改变任何用于保存字符串的数组的长度等。

我的结论是从现代的面向对象的技术引出的。字符串类毫无例外地包括了完整的错误检查、用于字符串的动态内存分配和其他一些防护措施。这些措施都会造成效率上的损失。但是，如果

bbs.theithome.com

程序无法运行，效率再高也没有什么意义。而且，较之设计 C 语言的时代，现代软件项目的规模要大得多。

因此，在数年前，缺少显式的字符串类型还能被看成是一个优点。但是，由于这个方法内在的危险性，所以使用现代的高级的、完整的字符串类还是物有所值的。如果 C 程序员愿意循规蹈矩地使用字符串，也可以获得这些优点。

9.4　使用其中一个操纵内存的库函数：

```
memcpy( y, x, 50 );
```

重要的是不要使用任何 str---函数，因为它们将在遇见第 1 个 NUL 字节时停止。如果你想自己编写循环，那要复杂得多，而且在效率上也不太可能压倒这个方案。

9.8　如果缓冲区包含了一个字符串，memchr 将在内存中 buffer 的起始位置开始查找第 1 个包含 0 的字节并返回一个指向该字节的指针。将这个指针减去 buffer 获得存储在这个缓冲区中的字符串的长度。strlen 函数完成相同的任务，不过 strlen 的返回值是个无符号(size_t)类型的值，而指针减法的值应该是个有符号类型(ptrdiff_t)。

但是，如果缓冲区内的数据并不是以 NULL 字节结尾，memchr 函数将返回一个 NULL 指针。将这个值减去 buffer 将产生一个无意义的结果。另一方面，strlen 函数在数组的后面继续查找，直到最终发现一个 NUL 字节。

尽管使用 strlen 函数可以获得相同的结果，但一般而言使用字符串函数不可能查找到 NUL 字节，因为这个值用于终止字符串。如果它是你需要查找的字节，你应该使用内存操纵函数。

第 9 章　编程练习

9.2　非常不幸！标准函数库并没有提供这个函数。

```
/*
** 安全的字符串长度函数。它返回一个字符串的长度，即使字符串并未以 NUL 字节结
** 尾。'size'是存储字符串的缓冲区的长度。
*/

#include <string.h>
#include <stddef.h>

size_t
my_strnlen( char const *string, int size )
{
     register size_t    length;

     for( length = 0; length < size; length += 1 )
          if( *string++ == '\0' )
               break;

     return length;
}
```

解决方案 9.2　　mstrnlen.c

9.6　这个问题有两种解决方法。第 1 种是简单但效率稍差的方案。

bbs.theithome.com

```
/*
** 字符串拷贝函数，返回一个指向目标参数末尾的指针（版本 1）。
*/

#include <string.h>

char *
my_strcpy_end( char *dst, char const *src )
{
     strcpy( dst, src );

     return dst + strlen( dst );
}
```

解决方案 9.2a mstrcpe1.c

用这种方案解决问题，最后一次调用 strlen 函数所消耗的时间不会少于省略那个字符串连接函数所节省的时间。

第 2 种方案避免使用库函数。register 声明用于提高函数的效率。

```
*
** 字符串拷贝函数，返回一个指向目标参数末尾的指针，不使用任何标准库字符处理
** 函数（版本 2）。
*/

#include <string.h>

char *
my_strcpy_end( register char *dst, register char const *src )
{
     while( ( *dst++ = *src++ ) != '\0' )
     ;

     return dst - 1;
}
```

解决方案 9.2b mstrcpe2.c

用这个方案解决问题并没有充分利用有些实现了特殊的字符串处理指令的机器所提供的额外效率。

9.11 一个长度为 101 个字节的缓冲区数组，用于保存 100 个字节的输入和 NUL 终止符。strtok 函数用于逐个提取单词。

```
/*
** 计算标准输入中单词“the”出现的次数。字母是区分大小写的，输入中的单词由** 一个或多次空白字符分隔。
*/

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

char const          whitespace[] = " \n\r\f\t\v";
```


bbs.theithome.com

```
int
main()
{
      char buffer[101];
      int  count;

      count = 0;

      /*
      ** 读入文本行，直到发现EOF。
      */
      while( gets( buffer ) ){
           char*word;

           /*
           ** 从缓冲区逐个提取单词，直到缓冲区内不再有单词。
           */
           for( word = strtok( buffer, whitespace );
              word != NULL;
              word = strtok( NULL, whitespace ) ){
               if( strcmp( word, "the" ) == 0 )
                    count += 1;
           }
      }

      printf( "%d\n", count );

      return EXIT_SUCCESS;
}
```

解决方案 9.11　　the.c

9.15　尽管没有在规范中说明，但这个函数应该对两个参数都进行检查，确保它们不是 NULL。程序包含了 stdio.h 文件，因为它定义了 NULL。如果参数能够通过测试，我们只能假定输入字符串已被正确地加上了终止符。

```
/*
** 把数字字符串'src'转换为美元和美分的格式，并存储于'dst'。
*/

#include <stdio.h>

void
dollars( register char *dst, register char const *src )
{
      int  len;

      if( dst == NULL || src == NULL )
           return;

      *dst++ = '$';
      len = strlen( src );

      /*
      ** 如果数字字符串足够长，复制将出现在小数点左边的数字，在适当的位置添
```


bbs.theithome.com

```
** 加逗号。如果字符串短于 3 个数字，在小数点前面再添加一个'0'。
*/
if( len >= 3 ){
     int i;

     for( i = len - 2; i > 0; ){
          *dst++ = *src++;
          if( --i > 0 && i % 3 == 0 )
               *dst++ = ',';
     }
} else
     *dst++ = '0';

/*
** 存储小数字，然后存储'src'中剩余的数字。如果'src'中的数字少于 2 个数
** 字，用'0'填充。然后在'dst'中添加 NUL 终止符。
*/
     *dst++ = '.';
     *dst++ = len < 2 ? '0' : *src++;
     *dst++ = len < 1 ? '0' : *src;
     *dst = 0;
}
```

解决方案 9.15 dollars.c

第 10 章　问题

10.2　结构是一个标量。和其他任何标量一样，当结构名在表达式中作为右值使用时，它表示存储在结构中的值。当它作为左值使用时，它表示结构存储的内存位置。但是，当数组名在表达式中作为右值使用时，它的值是一个指向数组第 1 个元素的指针。由于它的值是一个常量指针，所以数组名不能作为左值使用。

10.7　其中有一个答案无法确定，因为我们不知道编译器会选择在什么位置存储 np。

表达式	值
nodes	200
nodes.a	非法
nodes[3].a	12
nodes[3].c	200
nodes[3].c->a	5
*nodes	{5, nodes+3, NULL}
*nodes.a	非法
(*nodes).a	5
nodes->a	5
nodes[3].b->b	248
*nodes[3].b->b	{18, nodes+12, nodes+1 }
&nodes	200
&nodes[3].a	236
&nodes[3].c	244
&nodes[3].c->a	200
&nodes->a	200
np	224
np->a	22

bbs.theithome.com

np->c->c->a	15
npp	216
npp->a	非法
*npp	248
**npp	{18, nodes+2, nodes+1}
*npp->a	非法
(*npp)->a	18
&np	未知
&np->a	224
&np->c->c->a	212

10.11　x 应该被声明为整型（或无符号整型），然后使用移位和屏蔽存储适当的值。单独翻译每条语句给出了下面的代码：

```
x &= 0x0fff;
x |= ( aaa & 0xf ) << 12;
x &= 0xf00f;
x |= ( bbb & 0xff ) << 4;
x &= 0xfff1;
x |= ( ccc & 0x7 ) << 1;
x &= 0xfffe;
x |= ( dddd & 0x1 );
```

如果你只关心最终结果，下面的代码效率更高：

```
x = ( aaa & 0xf ) << 12 | \
    ( bbb & 0xff ) << 4 | \
    ( ccc & 0x7 ) << 1 | \
    ( ddd & 0x1 );
```

下面是另外一种方法：

```
x = aaa & 0xf;
x <<= 8;
x |= bbb & 0xff;
x <<= 3;
x |= ccc & 0x7;
x <<= 1;
x |= ddd & 1;
```

第 10 章　编程练习

10.1　虽然这个问题并没有明确要求，但正确的方法是为电话号码声明一个结构，然后使用这个结构表示付账信号结构的三个成员。

```
/*
** 表示长途电话付账记录的结构。
*/
struct PHONE_NUMBER {
    short area;
    short exchange;
    short station;
};
```


bbs.theithome.com

```
    struct LONG_DISTANCE_BILL {
        short       month;
        short       day;
        short       year;
        int         time;
        struct      PHONE_NUMBER  called;
        struct      PHONE_NUMBER  calling;
        struct      PHONE_NUMBER  billed;
};
```

解决方案 10.2a phone1.h

另一种方法是使用一个长度为 PHONE_NUMBERS 的数组，如下所示：

```
/*
**表示长途电话付账记录的结构。
*/
enum  PN_TYPE{ CALLED, CALLING, BILLED };

struct LONG_DISTANCE_BILL {
    short       month;
    short       day;
    short       year;
    int         time;
    struct      PHONE_NUMBER  numbers[3];
};
```

解决方案 10.2b phone2.h

第 11 章　问题

11.3　如果输入包含在一个文件中，它肯定是由其他程序（例如编辑器）放在那儿的。如果是这种情况，最长行的长度是由编辑器程序支持的，它会作出一个合乎逻辑的选择，确定你的输入缓冲区的大小。

11.4　主要的优点是当分配内存的函数返回时，这块内存会被自动释放。这个属性是由于堆栈的工作方式决定的，它可以保证不会出现内存泄漏。但这种方法也存在缺点。由于当函数返回时被分配的内存将消失，所以它不能用于存储那些回传给调用程序的数据。

11.5

a. 用字面值常量 2 作为整型值的长度。这个值在整型值长度为 2 个字节的机器上能正常工作。但在 4 字节整数的机器上，实际分配的内存将只是所需内存的一半。所以应该换用 sizeof。

b. 从 malloc 函数返回的值未被检查。如果内存不足，它将是 NULL。

c. 把指针退到数组左边界的左边来调整下标的范围或许行得通，但它违背了标准关于指针不能越过数组左边界的规定。

d. 指针经过调整之后，第 1 个元素的下标变成了 1，接着 for 循环将错误地从 0 开始。在许多系统中，这个错误将破坏 malloc 所使用的用于追踪堆的信息，常常导致程序崩溃。

e. 数组增值前并未检查输入值是否位于合适的范围内。非法的输入值可能会以一种有趣的方式导致程序崩溃。

f. 如果数组应该被返回，它不能被 free 函数释放。

bbs.theithome.com

第 11 章　编程练习

11.2　这个函数分配一个数组，并在需要时根据一个固定的增值对数组进行重新分配。增量 DELTA 可以进行微调，用于在效率和内存浪费之间作一平衡。

```
/*
** 从标准输入读取一列由 EOF 结尾的整数并返回一个包含这些值的动态分配的数组。
数组的第 1 个元素是数组所包含的值的数量。
*/

#include <stdio.h>
#include <malloc.h>

#define   DELTA          100

int *
readints()
{
    int    *array;
    int    size;
    int    count;
    int    value;

    /*
    ** 获得最初的数组，大小足以容纳 DELTA 个值。
    */
    size = DELTA;
    array = malloc( ( size + 1 ) * sizeof( int ) );
    if( array == NULL )
         return NULL;

    /*
    ** 从标准输入获得值。
    */
    count = 0;
    while( scanf( "%d", &value ) == 1 ){
         /*
         ** 如果需要，使数组变大，然后存储这个值。
         */
         count += 1;
         if( count > size ){
              size += DELTA;
              array = realloc( array,
                   ( size + 1 ) * sizeof( int ) );
              if( array == NULL )
                   return NULL;
    }
    array[ count ] = value;
}

/*
** 改变数组的长度，使其刚刚正好，然后存储计数值并返回这个数组。
** 这样做绝不会使数组更大，所以它绝不应该失败（但还是应该进行检查！）。
*/
```


bbs.theithome.com

```
    if( count < size ){
        array = realloc( array,
            ( count + 1 ) * sizeof( int ) );
        if( array == NULL )
            return NULL;
    }
    array[ 0 ] = count;
    return array;
    }
```

解决方案 11.2 readints.c

第 12 章 问题

12.2 和不用处理任何特殊情况代码的 sll_insert 函数相比，这种使用头结点的技巧没有任何优越之处。而且自相矛盾的是，这个声称用于消除特殊情况的技巧实际上将引入用于处理特殊情况的代码。当链表被创建时，必须添加哑节点。其他操纵这个链表的函数必须跳过这个哑节点。最后，这个哑节点还会浪费内存。

12.4 如果根节点是动态分配内存的，我们可以通过只为节点的一部分分配内存来达到目的。

```
Node  *root;
 root = malloc( sizeof(Node) - sizeof(ValueType) );
```

一种更安全的方法是声明一个只包含指针的结构。根指针就是这类结构之一，每个节点只包含这类结构中的一个。这种方法的有趣之处在于结构之间的相互依赖，每个结构都包含了一个对方类型的字段。这种相互依赖性就在声明它们时产生了一个“先有鸡还是先有蛋”的问题：哪个结构先声明呢？这个问题只是通过其中一个结构标签的不完整声明来解决。

```
struct  DLL_NODE;

struct  DLL_POINTERS    {
        struct DLL_NODE *fwd;
        struct DLL_NODE *bwd;
};

struct  DLL_NODE        {
        struct DLL_POINTERS     pointers;
        int     value;
};
```

12.7 在多个链表的方案中进行查找比在一个包含所有单词的链表中进行查找效率要高得多。例如，查找一个以字母 b 开头的单词，我们就不需要在那些以 a 开头的单词中进行查找。在 26 个字母中，如果每个字母开头的单词出现频率相同，这种多个链表方案的效率几乎可以提高 26 倍。不过实际改进的幅度要比这小一些。

第 12 章 编程练习

12.1 这个函数很简单，虽然它只能用于它被声明的那种类型的节点——你必须知道节点的内部结构。下一章将讨论解决这个问题的技巧。

```
/*
** 在单链表中计数节点的个数。
*/
```


bbs.theithome.com

```
#include "singly_linked_list_node.h"
#include <stdio.h>

int
sll_count_nodes( struct NODE *first )
{
        int  count;

        for( count = 0; first != NULL; first = first->link ){
             count += 1;
          }

        return count;
}
```

解决方案 12.1　　sll_cnt.c

如果这个函数被调用时传递给它的是一个指向链表中间位置某个节点的指针，那么它将对链表中这个节点以后的节点进行计数。

12.5　首先，这个问题的答案：接受一个指向我们希望删除的节点的指针可以使函数和存储在链表中的数据类型无关。所以通过对不同的链表包含不同的头文件，相同的代码可以作用于任何类型的值。另一方面，如果我们并不知道哪个节点包含了需要被删除的值，我们首先必须对它进行查找。

```
/*
** 从一个单链表删除一个指定的节点。第 1 个参数指向链表的根指针，第 2 个参数
** 指向需要被删除的节点。如果它可以被删除，函数返回 TRUE，否则返回 FALSE。
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "singly_linked_list_node.h"

#define   FALSE     0
#define   TRUE      1

int
sll_remove( struct NODE **linkp, struct NODE *delete )
{
        register Node*current;

        assert( delete != NULL );

        /*
        ** 寻找要求删除的节点。
        */
        while( ( current = *linkp ) != NULL && current != delete )
            linkp = &current->link;

        if( current == delete ){
            *linkp = current->link;
            free( current );
```


bbs.theithome.com

```
            return TRUE;
        }
        else
            return FALSE;
    }
```

解决方案 12.5 sll_remv.c

注意让这个函数用 free 函数删除节点将限制它只适用于动态分配节点的链表。另一种方案是如果函数返回真，由调用程序负责删除节点。当然，如果调用程序没有删除动态分配的节点，将导致内存泄漏。

一个讨论问题：为什么这个函数需要使用 assert？

第 13 章　问题

13.1　a. Ⅷ, b. Ⅲ, c. Ⅹ, d. Ⅺ, e. Ⅳ, f. Ⅸ, g. ⅩⅥ, h. Ⅶ, i. Ⅵ, j. ⅪⅩ
k. ⅩⅪ, l. ⅩⅩⅢ, m. ⅩⅩⅤ

13.4　把 trans 声明为寄存器变量可能有所帮助，这取决于你使用的环境。在有些机器上，把指针放入寄存器的好处相当突出。其次，声明一个保存 trans->product 值的局部变量。如下所示：

```
register Product *the_product;

the_product = trans->product;
the_product->orders += 1;
the_product->quantity_on_hand -= trans->quantity;
the_product->supplier->reorder_quantity
    += trans->quantity;
if( the_product->export_restricted ){
        ...
}
```

这个表达式可以被多次使用，但不需要每次重新计算。有些编译器会自动为你做这两件事，但有些编译器不会。

13.7　它的唯一优点如此明显，你可能没有对它多加思考，这也是编写这个函数的理由——这个函数使处理命令行参数更为容易。但这个函数的其他方面都是不利因素。你只能使用这个函数所支持的方式处理参数。由于它并不是标准的一部分，所以使用 getopt 将会降低程序的可移植性。

13.11　首先，有些编译器把字符串常量存放在无法进行修改的内存区域，如果你试图对这类字符串常量进行修改，就会导致程序终止。其次，即使一个字符串常量在程序中使用的地方不止一处，有些编译器只保存这个字符串常量的一份拷贝。修改其中一个字符串常量将影响程序中这个字符串常量所有出现的地方，这使得调试工作极为困难。例如，如果一开始执行了下面这条语句

```
strcpy("hello\n", "Bye!\n" );
```

然后再执行下面这条语句：

```
printf("hello\n" );
```

将打印出 Bye!。

第 13 章　编程练习

13.1　这个问题是在第 9 章给出的，但那里没有对 if 语句施加限制。这个限制的意图是促使你考虑其他实现方法。函数 is_not_print 的结果是 isprint 函数返回值的负值，它避免了主循环处理特殊

bbs.theithome.com

情况的需要，每个元素保存函数指针、标签以及每种类型的计数值。

```
/*
** 计算从标准输入的几类字符的百分比。
*/
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

/*
**   定义一个函数，判断一个字符是否为可打印字符。这可以消除下面代码中这种类
**  型的特殊情况。
*/
int is_not_print( int ch )
{
        return !isprint( ch );
}

/*
**    用于区别每种类型的分类函数的跳转表。
*/
static    int(*test_func[])( int ) = {
          iscntrl,
          isspace,
          isdigit,
          islower,
          isupper,
          ispunct,
          is_not_print
};
#define   N_CATEGORIES\
             ( sizeof( test_func ) / sizeof( test_func[ 0 ] ) )

/*
**  每种字符类型的名字。
*/
char*label[] = {
    "control",
    "whitespace",
    "digit",
    "lower case",
    "upper case",
    "punctuation",
    "non-printable"
};

/*
**   目前见到的每种类型的字符数以及字符的总量。
*/
int  count[ N_CATEGORIES ];
int  total;

main()
{
     int  ch;
```


bbs.theithome.com

```
        int  category;

    /*
    ** 读取和处理每个字符。
    */
    while( (ch = getchar()) != EOF ){
        total += 1;

        /*
        ** 为这个字符调用每个测试函数。如果结果为真，增加对应计数器的值。
        */
        for( category = 0; category < N_CATEGORIES;
            category += 1 ){
              if( test_func[ category ]( ch ) )
                    count[ category ] += 1;
        }
    }

    /*
    ** 打印结果。
    */
    if( total == 0 ){
        printf( "No characters in the input!\n" );
    }
    else {
        for( category = 0; category < N_CATEGORIES;
            category += 1 ){
                printf( "%3.0f%% %s characters\n",
                count[ category ] * 100.0 / total,
                label[ category ] );
        }
    }

    return EXIT_SUCCESS;
}
```

解决方案 13.1 char_cat.c

第 14 章　问题

14.1　在打印错误信息时，文件名和行号可能是很有用的，尤其是在调试的早期阶段。事实上，assert 宏使用它们来实现自己的功能。__DATE__和__TIME__可以把版本信息编译到程序中。最后，__STDC__可以用于条件编译中，用于在必须由两种类型的编译器进行编译的源代码中选择 ANSI 和前 ANSI 结构。

14.6　我们无法通过给出的源代码进行判断。如果 process 以宏的方式实现，并且对它的参数求值超过一次，增加下标值的副作用可能会导致不正确的结果。

14.7　这段代码有几个地方存在错误，其中几处比较微妙。它的主要问题是这个宏依赖于具有副作用（增加下标值）的参数。这种依赖性是非常危险的，由于宏的名字并没有提示它实际所执行的任务（这是第 2 个问题），这种危险性进一步加大了。假定循环后来改写为：

```
for( i = 0; i < SIZE; i += 1 )
        sum += SUM( array[ i ] );
```


bbs.theithome.com

尽管看上去相同，但程序此时将会失败。最后一个问题是：由于宏始终访问数组中的两个元素，所以如果 SIZE 是个奇数值，程序就会失败。

第 14 章　编程练习

14.1　这个问题唯一棘手之处在于两个选项都有可能被选择。这种可能性排除了使用#elif 指令帮助你确定哪一个未被定义。

```
/*
** 打印风格由预定义符号指定的分类账户。
*/

void
print_ledger( int x )
{
#ifdef    OPTION_LONG
#         define    OK   1
          print_ledger_long( x );
#endif

#ifdef    OPTION_DETAILED
#         define    OK   1
          print_ledger_detailed( x );
#endif

#ifndef   OK
          print_ledger_default( x );
#endif
}
```

解决方案 14.1　　prt_ldgr.c

第 15 章　问题

15.1　如果由于任何原因导致打开失败，函数的返回值将是 NULL。当这个值传递给后续的 I/O 函数时，该函数就会失败。至于程序是否失败，则取决于编译器。如果程序并不终止，那么 I/O 操作可能会修改内存中有些不可预料的位置的内容。

15.2　程序将会失败，因为你试图使用的 FILE 结构没有被适当地初始化。某个不可预料的内存地址的内容可能会被修改。

15.4　不同的操作系统提供不同的机制来检测这种重定向，但程序通常并不需要知道输入来自于文件还是键盘。操作系统负责处理绝大多数与设备无关的输入操作的许多方面，剩余部分则由库 I/O 函数负责。对于绝大多数应用程序，程序从标准输入读取的方式相同，不管输入实际来自何处。

15.16　如果实际值是 1.4049，格式代码%.3f将导致缀尾的 4 四舍五入至 5，但使用格式代码%.2f，缀尾的 0 并没有进行四舍五入至 1，因为它后面被截掉的第 1 个数字是 4。

第 15 章　编程练习

15.2　输入行有长度限制这个条件极大地简化了问题。如果使用 gets，缓冲区的长度至少为 81

bbs.theithome.com

个字节以便保存 80 个字符加一个结尾的 NUL 字节。如果使用 fgets，缓冲区的长度至少为 82 个字节，因为还需要存储一个换行符。

```
/*
** 将标准输入复制到标准输出，每次复制一行。每行的长度不超过 80 个字符。
*/

#include <stdio.h>

#define   BUFSIZE   81/* 80 个数据字节加上 NUL 字节 */

main()
{
      charbuf[BUFSIZE];

      while( gets( buf ) != NULL )
            puts( buf );

      return EXIT_SUCCESS;
}
```

解决方案 15.2 prog2.c

15.9 字符串不能包含换行符的限制意味着程序可以从文件中一次读取一行。程序并不需要尝试匹配错行的字符串。这个限制意味着查找文本行可以使用 strstr 函数。输入行长度的限制简化了解决方案。使用动态分配的数组应该可以去除这个长度限制，因为当程序发现一个长度大于缓冲区的输入行时，重新为缓冲区指定长度。程序的主要内容用于处理获得文件名并打开文件。

```
/*
** 在指定的文件中，查找并打印所有包含指定字符串的文本行。
**
**    用法:
**          fgrep string file [ file ... ]
*/

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

#define   BUFFER_SIZE   512

void
search( char *filename, FILE *stream, char *string )
{
       char  buffer[ BUFFER_SIZE ];

       while( fgets( buffer, BUFFER_SIZE, stream ) != NULL ){
          if( strstr( buffer, string ) != NULL ){
               if( filename != NULL )
                    printf( "%s:", filename );
               fputs( buffer, stdout );
          }
       }
```


bbs.theithome.com

```
}

int
main( int ac, char **av )
{
        char*string;

        if( ac <= 1 ){
                fprintf( stderr, "Usage: fgrep string file ...\n" );
                exit( EXIT_FAILURE );
        }

        /*
        ** 得到字符串。
        */
        string = *++av;

        /*
        ** 处理文件。
        */
        if( ac <= 2 )
                search( NULL, stdin, string );
            else {
                while( *++av != NULL ){
                        FILE*stream;

                        stream = fopen( *av, "r" );
                        if( stream == NULL )
                                perror( *av );
                        else {
                                search( *av, stream, string );
                                fclose( stream );
                }
            }
        }

        return EXIT_SUCCESS;
}
```

解决方案 15.9　　fgrep.c

第 16 章　问题

16.1　这个情况标准并未定义，所以你不得不自己尝试一下并观察结果。但即使它看上去会产生一些有用的结果，**不要使用它**！否则你的代码将失去可移植性。

16.3　它取决于你的编译器所提供的随机数生成函数的质量。在理想情况下，它应该产生一个随机序列的 0 和 1。但有些随机数生成函数并没有如此优秀，它生成的是交替出现的 0 和 1 序列——这看上去可不是很随机。如果你的编译器也属于这种类型，你可能会发现高字节的位比低字节的位更为随机。

16.5　首先，一个 NULL 指针必须传递给 time 函数。但此处并没有传递，所以编译器将抱怨这个调用与原型不匹配。其次，一个指向时间值的指针必须传递给 localtime 函数，编译器应该也能捕捉到这种情况。第三，月份应该是一个 0~11 的范围，但此处它作为输出的日期部分直接被打印。在

bbs.theithome.com

打印之前它的值应该加上 1。第四，2000 年以后，打印出来的年份的样子将很奇怪。

第 16 章　编程练习

16.2　除了“概率相等”这个要求之外，这个问题的其他部分非常简单。这里有个例子。普通情况下你将把一个随机数对 6 取模，产生一个 0~5 的值，将这个值加上 1 并返回。但是，如果随机数生成函数所返回的最大值是 32 767，那么这些值就不是“概率相等”。从 0~32 765 返回的值所产生的 0~5 之间各个值的概率相等。但是，最后两个值，32 766 和 32 767 的返回值将分别是 0 和 1，这使它们的出现概率有所增加（是 5 462/32 768 而不是 5 461/32 768）。由于我们需要的答案的范围很窄，所以这个差别是非常小的。如果这个函数试图产生一个范围在 1~30 000 之间的随机数时，那么前 2 768 个值的出现概率将两倍于后面那些值。程序中的循环用于消除这种错误，方法是一旦出现最后两个值，就产生另一个随机值。

```
/*
** 通过返回一个范围为 1 至 6 的值，模拟掷一个六边的骰子。
*/
#include <stdlib.h>
#include <stdio.h>

/*
**    计算将产生 6 作为骰子值的随机数生成函数所返回的最大数。
*/
#define MAX_OK_RAND\
    (int)( ( ( (long)RAND_MAX + 1 ) / 6 ) * 6 - 1 )

int
throw_die( void ){
        static int is_seeded = 0;
        int value;

        if( !is_seeded ){
                is_seeded = 1;
                srand( (unsigned int)time( NULL ) );
        }

        do {
                value = rand();
        } while( value > MAX_OK_RAND );

        return value % 6 + 1;
}
```

解决方案 16.2　　**die.c**

16.7　这个程序从本质上来说是一个一次性程序，这个不优雅的解决方案用于完成这个任务是绰绰有余了。

```
/*
** 测试 rand 函数所产生的值的随机程度。
*/
#include <stdlib.h>
```


bbs.theithome.com

```
#include <stdio.h>

/*
**    用于计数各个数字相对频率的数组。
*/
int   frequency2[2];
int   frequency3[3];
int   frequency4[4];
int   frequency5[5];
int   frequency6[6];
int   frequency7[7];
int   frequency8[8];
int   frequency9[9];
int   frequency10[10];

/*
**    用于计数各个数字周期性频率的数组。
*/
int   cycle2[2][2];
int   cycle3[3][3];
int   cycle4[4][4];
int   cycle5[5][5];
int   cycle6[6][6];
int   cycle7[7][7];
int   cycle8[8][8];
int   cycle9[9][9];
int   cycle10[10][10];


/*
**    用于为一个特定的数字同时计数频率和周期性频率的宏。
*/
#define   CHECK( number, f_table, c_table )                          \
                 remainder = x % number;                             \
                 f_table[ remainder ] += 1;                          \
                 c_table[ remainder ][ last_x % number ] += 1

/*
**    用于打印一个频率表的宏。
*/
#define   PRINT_F( number, f_table )                                 \
          printf( "\nFrequency of random numbers modulo %d\n\t",     \
              number );                                              \
          for( i = 0; i < number; i += 1 )                           \
              printf( " %5d", f_table[ i ] );                        \
          printf( "\n" )

/*
**    用于打印一个周期性频率表的宏。
*/
#define PRINT_C( number, c_table )                                   \
        printf( "\nCyclic frequency of random numbers modulo %d\n",  \
           number );                                                 \
        for( i = 0; i < number; i += 1 ){                            \
             printf( "\t" );                                         \
             for( j = 0; j < number; j += 1 )                        \
```


bbs.theithome.com

```
            printf( " %5d", c_table[ i ][ j ] );                    \
        printf( "\n" );                                             \
}


int
main( int ac, char **av )
{
        int i;
        int j;
        int x;
        int last_x;
        int remainder;

        /*
        ** 如果给出了种子，就为随机数生成函数设置种子。
        */
        if( ac > 1 )
            srand( atoi( av[ 1 ] ) );

        last_x = rand();

        /*
        ** 运行测试。
        */
        for( i = 0; i < 10000; i += 1 ){
                x = rand();
                CHECK( 2, frequency2, cycle2 );
                CHECK( 3, frequency3, cycle3 );
                CHECK( 4, frequency4, cycle4 );
                CHECK( 5, frequency5, cycle5 );
                CHECK( 6, frequency6, cycle6 );
                CHECK( 7, frequency7, cycle7 );
                CHECK( 8, frequency8, cycle8 );
                CHECK( 9, frequency9, cycle9 );
                CHECK( 10, frequency10, cycle10 );
                last_x = x;
        }

        /*
        ** 打印结果。
        */
        PRINT_F( 2, frequency2 );
        PRINT_F( 3, frequency3 );
        PRINT_F( 4, frequency4 );
        PRINT_F( 5, frequency5 );
        PRINT_F( 6, frequency6 );
        PRINT_F( 7, frequency7 );
        PRINT_F( 8, frequency8 );
        PRINT_F( 9, frequency9 );
        PRINT_F( 10, frequency10 );

        PRINT_C( 2, cycle2 );
        PRINT_C( 3, cycle3 );
        PRINT_C( 4, cycle4 );
        PRINT_C( 5, cycle5 );
```


bbs.theithome.com

```
        PRINT_C( 6, cycle6 );
        PRINT_C( 7, cycle7 );
        PRINT_C( 8, cycle8 );
        PRINT_C( 9, cycle9 );
        PRINT_C( 10, cycle10 );

        return EXIT_SUCCESS;
}
```

解决方案 16.7　　testrand.c

第 17 章　问题

17.3　传统接口和替代形式的接口很容易共存。top 函数返回栈顶元素值但并不实际移除它，pop 函数移除栈顶元素并返回它。希望使用传递方式的用户可以用传统的方式使用 pop 函数。如果希望使用替代方案，用户可以用 top 函数获得栈顶元素的值，而且在使用 pop 函数时忽视它的返回值。

17.7　由于它们中的每一个都是用 malloc 函数单独分配的，逐个将它们弹出可以保证每个元素均被释放。用于释放它们的代码在 pop 函数中已经存在，所以调用 pop 函数比复制那些代码更好。

17.9　考虑一个具有 5 个元素的数组，它可以出现 6 种不同的状态：它可能为空，也可能分别包含 1 个、2 个、3 个、4 个或 5 个元素。但 front 和 rear 始终必须指向数组中的 5 个元素之一。所以对于任何给定值的 front，rear 只可能出现 5 种不同的情况：它可能等于：front、front+1、front+2、front+3 或 front+4（记住，front+5 实际上就是 front，因为它已经环绕到这个位置）。我们不可能用只能表示 5 个不同状态的变量来表示 6 种不同的状态。

17.12　假定你拥有一个指向链表尾部的指针，单链表就完全可以达到目的。队列绝不会反向遍历，由于双链表具有一个额外的链字段开销，所以它用于这个场合并无优势。

17.18　中序遍历可以以升序访问一棵二叉搜索树的各个节点。没有一种预定义的遍历方法以降序访问二叉搜索树的各个节点，但我们可以对中序遍历稍作修改，使它先遍历右子树然后遍历左子树就可以实现这个目的。

第 17 章　编程练习

17.3　这个转换类似链式堆栈，但是当最后一个元素被移除时，rear 指针也必须被设置为 NULL。

```
/*
** 一个用链表形式实现的队列，它没有长度限制。
*/
#include "queue.h"
#include <stdio.h>
#include <assert.h>

/*
**    定义一个结构用地保存一个值。link 字段将指向队列中的下一个节点。
*/
typedef    struct    QUEUE_NODE {
           QUEUE_TYPE     value;
           struct QUEUE_NODE *next;
} QueueNode;
```


bbs.theithome.com

```
/*
**    指向队列第 1 个和最后一个节点的指针。
*/
static    QueueNode*front;
static    QueueNode*rear;

/*
**    destroy_queue
*/
void
destroy_queue( void )
{
        while( !is_empty() )
              delete();
}

/*
**    insert
*/
void
insert( QUEUE_TYPE value )
{
      QueueNode*new_node;

      /*
      ** 分配一个新节点，并填充它的各个字段。
      */
      new_node = (QueueNode *)malloc( sizeof( QueueNode ) );
      assert( new_node != NULL );
      new_node->value = value;
      new_node->next = NULL;

      /*
      ** 把它插入到队列的尾部。
      */
      if( rear == NULL ){
              front = new_node;
      }
      else {
              rear->next = new_node;
      }
      rear = new_node;
}

/*
**    delete
*/
void
delete( void )
{
        QueueNode  *next_node;

        /*
        ** 从队列的头部删除一个节点，如果它是最后一个节点，
        ** 将 rear 也设置为 NULL。
        */
```


bbs.theithome.com

```
    assert( !is_empty() );
    next_node = front->next;
    free( front );
    front = next_node;
    if( front == NULL )
        rear = NULL;
}

/*
**    first
*/
QUEUE_TYPE first( void )
{
    assert( !is_empty() );
    return front->value;
}

/*
**    is_empty
*/
int
is_empty( void )
{
    return front == NULL;
}

/*
**    is_full
*/
int
is_full( void )
{
    return 0;
}
```

解决方案 17.3　　l_queue.c

17.6　如果使用队列模块，我们必须解决名字冲突问题。

```
/*
** 对一个数组形式的二叉搜索树执行层次遍历。
*/
void
breadth_first_traversal( void (*callback)( TREE_TYPE value ) )
{
    int  current;
    int  child;

    /*
    ** 把根节点插入到队列中。
    */
    queue_insert( 1 );

    /*
    ** 当队列还没有空时...
    */
```


bbs.theithome.com

```
    while( !is_queue_empty() ){
        /*
        ** 从队列中取出第 1 个值并对它进行处理。
        */
        current = queue_first();
        queue_delete();
        callback( tree[ current ] );

        /*
        ** 将该节点的所有孩子添加到队列中。
        */
        child = left_child( current );
        if( child < ARRAY_SIZE && tree[ child ] != 0 )
            queue_insert( child );
        child = left_child( current );
        if( child < ARRAY_SIZE && tree[ child ] != 0 )
            queue_insert( child );
    }
}
```

解决方案 17.6 breadth.c

第 18 章　问题

18.5　这个主意听上去不错，但它无法实现。在函数的原型中，register 关键字是可选的，所以调用函数并没有一种可靠的方法知道哪些参数（如果有的话）是被这样声明的。

18.6　不，这是不可能的。只有调用函数才知道有多少个参数被实际压入到堆栈中。但是，如果在堆栈中压入一个参数计数器，被调用函数就可以清除所有参数。不过，它先要弹出返回地址并进行保存。

第 18 章　编程练习

18.3　这个答案实际上取决于特定的环境。不过这里的解决方案适用于本章所讨论的环境。用户必须提供经历标准类型转换之后的参数的实际类型。真正的 stdarg.h 宏就是这样做的。

```
/*
** 标准库文件 stdarg.h 所定义的宏的替代品。
*/

/*
** va_list
**    为一个保存一个指向参数列表可变部分的指针的变量进行类型定义。 这里使用的
**    是 char * ，因为作用于它们之上的运算并没有经过调整。
*/
typedef   char*va_list;

/*
** va_start
**    用于初始化一个 va_list 变量的宏，使它指向堆栈中第 1 个可变参数。
*/
#define   va_start(arg_ptr,arg)  arg_ptr = (char *)&arg + sizeof( arg )
```


bbs.theithome.com

```
/*
** va_arg
**    用于返回堆栈中下一个变量值的宏，它同时增加 arg_ptr 的值，使它指向下一个参数。
*/
#define   va_arg(arg_ptr,type)    *((type *)arg_ptr)++

/*
** va_end
**    在可变参数最后的访问之后调用。在这个环境中，它不需要执行任何任务。
*/
#define   va_end(arg_ptr)
```

解决方案 18.3　　mystdarg.h

bbs.theithome.com

bbs.theithome.com

索　　引

#argument 283
#define 5, 279
#elif 287
#else 287
#endif 4, 286
#error 291
#if 4, 286
#ifdef 287
#ifndef 287
#include 5, 288
#line 291
#progma 291
#undef 285
_IOFBF 318
_IOLBF 318
_IONBF 318
__DATE__ 279
__FILE__ 279
__LINE__ 279
__STDC__ 279
__TIME__ 279
abort 338, 342
abs 327
absolute pathname 绝对路径名 290
absolute value 绝对值 327
Abstract Data Type 抽象数据类型 124, 355
acos 330
aggregate data type 聚合数据类型 195
alloca 232
argc 265
argument 参数 5, 122
argv 265
arithmetic of pointer 指针运算 107
arithmetic of shift 移位运算 67
usual arithmetic conversion 寻常算术转换 80, 119
argument promotion 参数提升 124, 137
array 数组 194
circular array 循环数组 366
array of pointer 指针数组 162
asctime 333, 334
asin 330
assert 342
assert.h 342
assertion 断言 342
associativity 结合性 80
atan 330
atan2 330
atexit 342
atof 332
atoi 329
atol 329
auto 43
backtracking 回溯法 173
binary search 二分法 345
binary search tree 二叉搜索树 369, 370, 371
binary tree 二叉树 369
bit field 位段 209
black box 黑盒 124
block 代码块 39, 40, 117
boundary alignment 边界对齐 92, 206, 222
breadth traversal 层次遍历 371
break 13, 54, 55, 58
bsearch 345
buffer 缓冲区 299
BUFSIZ 318
bus error 总线错误 95
call by reference 传址调用 6, 122, 137
call by value 传值调用 122, 150, 207
callback function 回调函数 261
calloc 222
case 51
cast 强制类型转换 73
ceil 331

bbs.theithome.com

char 29
character I/O 字符 I/O 304
child 孩子 369
clearerr 319
clock 332
CLOCKS_PER_SEC 332
clock_t 332
command line argument 命令行参数 265
compilation process 编译过程 19
compliling and linking 编译和链接 20
compound operator 复合赋值符 71
concordance list 索引表 254
conditional 条件运算符 75
conditional compilaton 条件编译 286
const 2, 31, 184
constant expression 常量表达式 58, 149, 286
named constant 命名常量 31
pointer constant 指针常量 34
continue 54, 58
Coordinated Universal Time 世界协调时间 333
cos 330
cosh 349
ctime 349
ctype.h 184
basic declaration 基本声明 35
implicit declaration 隐式声明 37
expression declaration 表达式声明 258
structure declaration 结构声明 195
incomplete declaration 不完整声明 199
default 59
dereference 解引用 94
device controller 设备控制器 210
difftime 333
div 327
div_t 327
do 56
domain error 定义域错误 330
double 33
double linked list 双链表 245
dynamic memory allocation 动态内存分配 228
EDOM 330
Eight Queens Problem 八皇后问题 173
else 53
empty statement 空语句 12, 51
enum 32
enumerated type 枚举类型 32
execution environment 执行环境 19, 342
freestanding environment 独立环境 19
hosted environment 宿主式环境 26
translation environment 翻译环境 19
EOF 11, 70, 300, 305
ERANGE 329, 330, 332
errno 184, 297, 329, 330
errno.h 297
range error 范围错误 330
escape sequence 转义序列 22
exclusive OR 异或 69
exit 298
exit function 退出函数 342
EXIT_FAILURE 5, 298
EXIT_SUCCESS 5, 298
exp 331
exponent 指数 331
pointer expression 指针表达式 99
typedef 38
extern 42
external identifier 外部标识符 393
fabs 331
factorial 阶乘 131
fclose 301, 303
feof 319
ferror 319
fflush 299, 316
fgetc 304, 305, 320
fgetpos 317
fgets 306, 307
Fibonacci number 菲波那契数 133
FIFO 364
FILE 300
head file 头文件 15
file inclusion 文件包含 288
temporary file 临时文件 319
filename convention 文件名约定 20
FILENAME_MAX 300
float 33
float.h 33
floor 349
flush 刷新 299
fmod 331
fopen 302
FOPEN_MAX 300
for 13, 51
form feed 格式反馈 22
monetary formatting 货币格式化 347
numeric formatting 数值格式化 347
fpos_t 317
fprintf 301, 312
fputc 301, 305, 320
fputs 301, 307
fread 316, 317
free 221
free form 自由格式 4
freopen 303
frexp 331
fscanf 301, 308, 321
fseek 317, 319, 321
fsetpos 317, 318, 319, 321
ftell 317
function 函数 5
function body 函数体 391

bbs.theithome.com

function declaration 函数声明 117
function definition 函数定义 117
function epilogue 函数跋 391
function name 函数名 41
pointer of function 函数指针 260, 261
function prologue 函数序 391
function prototype 函数原型 5, 119
fwrite 316, 321
genericity 泛型 381
getc 304, 305
getchar 11, 304, 305
getenv 343
gets 7, 306, 307
global 全局 46
GMT 格林尼治标准时间 333
gmtime 333
goto 41, 60, 335
greatest common divisor 最大公约数 139
HUGE_VAL 330
I/O 297
二进制 I/O 316
character I/O macro 字符 I/O 宏 305
I/O device I/O 设备 299
line I/O 行 I/O 306
formatted line I/O 格式化的行 I/O 308
unformatted line I/O 未格式化的行 I/O 306
random I/O 随机 I/O 321
I/O redirection I/O 重定向 300
identifier 标识符 24
identity matrix 单位矩阵 170
if 52
in-order traversal 中序遍历 372
indirection 间接访问 72, 77, 94, 142, 198
initialization 初始化 8, 35
integral promotion 整型提升 80
isalnum 187
isalpha 187
iscntrl 187
isdigit 187
isgraph 187
islower 187
isprint 187
ispunct 187
isspace 187
isupper 187
isxdigit 187
jmp_buf 336
L-value 左值 73, 79, 97
LIFO 355
ldexp 331
ldiv 327
ldiv_t 328
lexical rule 词法规则 21
limits.h 30
linefeed 行反馈符 300
linker 链接器 19
linked list 链表 235
singly linked list 单链表 235
literal 字面值 31
character literal 字符常量 31
string literal 字符串常量 34
local 局部 39
locale 346
locale.h 346
localeconv 346
log 349
localtime 333
log10 331
logical shift 逻辑移位 68
long 31
long double 33
long int 30
longjmp 335
L_tmpnam 319
macro 宏 281
macro name convention 宏命名约定 285
macro 与. function 宏 与. 函数 283
main 5
malloc 221
mask 屏蔽 211
math.h 329
member 成员 195
memchr 186
memcmp 186
memcpy 185
memmove 186
memory leak 内存泄漏 226
memset 186
mktime 335
modf 331
NDEBUG 343
newline 换行符 7, 22, 59, 300, 321
node 节点 235
NUL 7, 34, 177
NULL 7, 96, 165, 222, 223
null directive 无效指令 292
object code 目标代码 19
offsetof 206
operating system 操作系统
..... 19, 184, 297, 318, 400
operator 运算符(操作符) 67
add + ... 67
subtract - 67
multiply * 67
divide / 67
modulo % 67
= 赋值符 70
great than > 73
great than or equal >= 73
less than < 73
less than or equal <= 73
not equal != 73

bbs.theithome.com

equal == 73
not ! 72
increment ++ 72
unary minus - 72
address of & 72
sizeof 72
one's complement ~ 72
decrement -- 72
unary plus + 72
indirection * 72
cast (类型) 72
precedence 优先级 80
subcript 下标引用 77
dot operator 点操作符 197
arraw operator 箭头操作符 198
comma , 76
optimizer 优化器 19
OR 69
AND 69
palindrome 回文 189, 323
perror 297
pointer 指针 5, 33
pointer addition 指针加法 102
pointer arithmetic 指针运算 107
pointer subtraction 指针减法 110
pointer to pointer 指针的指针 98, 257
post-order traversal 后序遍历 372
pow 331
pre-order traversal 前序遍历 371
preprocessor 预处理器 4, 19
preprocessor directives 预处理指令 4
prime number 质数 64
printf 7, 312
profile 性能评测 401
pseudo-random number 伪随机数 328
ptrdiff_t 110
putc 305
putchar 14, 305
puts 307, 321
qsort 344
queue 队列 364
R-value 右值 79
raise 339
rand 328
random access 随机访问 316
random number 随机数 328
RAND_MAX 328
realloc 222
recursion 递归 127
tail recursion 尾部递归 132
register 43, 151
remove 320
rename 320
return type 返回类型 5
rewind 317
row major order 行主序 155
runtime stack 运行时堆栈 393
scanf 8, 308, 321
scope 作用域 39
block scope 代码块作用域 40
file scope 文件作用域 41
function scope 函数作用域 41
prototype scope 原型作用域 41
segmentation violation 段违例 95
setbuf 318
setjmp 335
setjmp.h 335
setlocale 346
setvbuf 318
arithmetic shift 算术移位 68
short 35
short int 30
side effect 副作用 284
short-circuited evaluation 短路求值 75
signal 339
signal handler 信号处理函数 338
signal.h 338
signed 30
sig_atomic_t 340
sin 330
sinh 349
size_t 176
sprintf 312
sqrt 331
srand 328
sscanf 321
stack 堆栈 355
stack frame 堆栈帧 391
frame pointer 帧指针 392
stack frame layout 堆栈帧布局 393
push 356
pop 356
top 356
standard error 标准错误 300
standard input 标准输入 300
standard output 标准输出 300
standard I/O Library 标准 I/O 库 5
standard vibrary 标准函数库 423
static 21, 42
stdarg.h 135, 341
stddef.h 176
stderr 300
stdin 300
stdout 300
stdlib.h 5, 327
stdio.h 5
storage class 存储类型 43
storage order 存储顺序 154
strcat 14, 178
strchr 14, 180
strcmp 179
strcoll 348

bbs.theithome.com

strcpy 14, 177
strcspn 182
strdup 228
stream 流 299
binary stream 二进制流 300
text stream 文本流 299
full buffered 全缓冲 318
unbuffered stream 不缓冲 318
strerror 184
strftime 334, 335
string 字符串 7
string.h 175
strlen 175
strncat 179
strncmp 180
strncpy 186
strpbrk 186
strrchr 180
strpbrk 181
strrstr 181
strspn 182
strstr 181
strtod 332
strtok 183, 420
strtol 329
strtoul 329
structure 结构 195
strxfrm 348
stub 存根 117
switch 57
system 343
tan 330
tanh 349
time 332
time.h 332
time_t 333
tm 333
tmpfile 319
tmpnam 319
TMP_MAX 319
token 标记 21
tolower 184
toupper 184
typedef 196
UCHAR_MAX 30
UINT_MAX 30
ULONG_MAX 30
ungetc 305
unsigned 29
USHRT_MAX 30
global variable 全局变量 47
local variable 局部变量 12
static variable 静态变量 43
pointer variable 指针变量 93
variable argument list 可变参数列表 134
variant record 变体记录 212
va_arg 135
va_end 135
va_list 135
va_start 135
vertical tab 垂直制表符 22
vfprintf 350
virtual memory 虚拟内存 400
void 5, 118
volatile 340, 350
vprintf 350
vsprintf 350
while 7, 51, 53, 54
XOR 69
NULL 318

bbs.theithome.com

参考文献

[ANSI 90]1990. *American National Standard for Programming Languages-C*. ANSI/ISO 9899-1990. New York, NY: American National Standars Institution.

[KERN 89]Kernighan, Brian 和 Dennis Ritchie.1989. *The C Programming Language, Sencond Edition*. Englewood Cliffs, NJ:Prentice Hall.

[KERN 78] Kernighan, Brian 和 Dennis Ritchie.1978. *The C Programming Language,* Englewood Cliffs, NJ:Prentice Hall.

[REEK 89] ReeK,Kenneth A. 1989. "The TRY System—or—How to Avoid Testing Student Programs." *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education* Volume 21: 112~116.

[REEK 96] Reek, Kenneth A. 1996. "A Software Infrastructure to Support Introductory Computer Science Courses." *Proceddings of the Twenty Seventh SIGCSE Technical Symposium on Computer Science Education* Volume 28: 125~129.

bbs.theithome.com