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;
```

```
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。
        */
```

```
    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。

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。
```

```
        */
        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。

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 );
```

```
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：

```
-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 );
```

```
    }
}
```

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。

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，

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。
```

```
        */
        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。

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

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

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。

```
/*
** 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";
```

```
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
```

```
** 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

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;
};
```

```
    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。

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！）。
*/
```

```
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。
*/
```

```
#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 );
```

```
            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

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;
```

```
        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 ] );
```

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

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 );
          }
       }
```

```
}

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

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>
```

```
#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 )                          \
```

```
            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 );
```

```
        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;
```

```
/*
**    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。
        */
```

```
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...
      */
```

```
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 )
```

```
/*
** 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