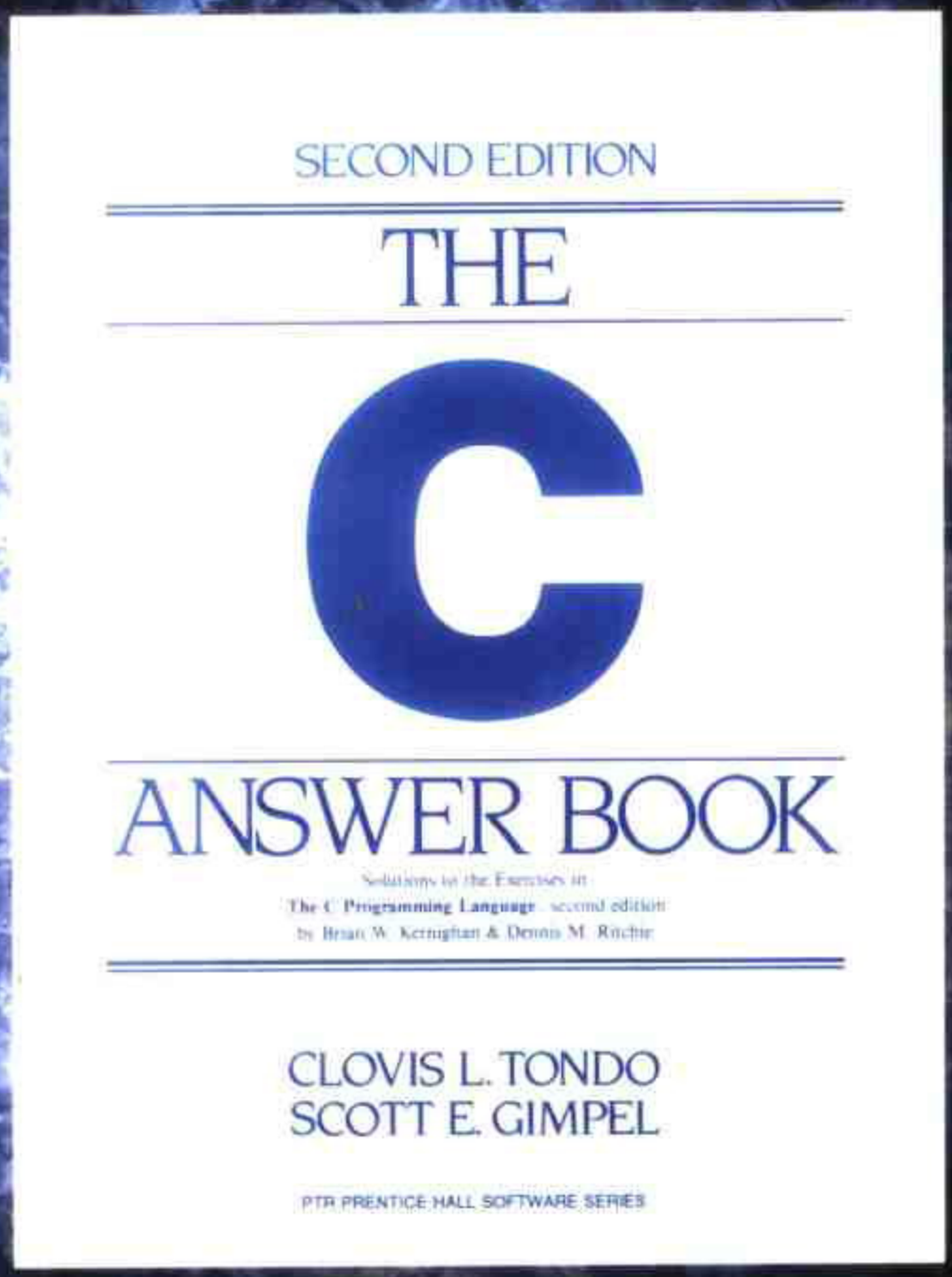
计 算 机 科 学 丛 书

原书第2版

C程序设计语言（第2版·新版）习题解答

(美) Clovis L. Tondo
Scott E. Gimpel 著 杨涛 等译



The C Answer Book

Second Edition

机械工业出版社
China Machine Press

前 言

The C Answer Book

这本习题解答对Brian W. Kernighan和Dennis M. Ritchie所著的《The C Programming Language》（第2版，Prentice Hall, 1988）（以下简称为“教材”）中所有的练习题都进行了解答。教材的中文版《C程序设计语言（第2版·新版）》已由机械工业出版社华章公司于2003年11月出版。

在美国国家标准协会（American National Standards Institute，ANSI）推出C语言的ANSI标准之后，Kernighan和Ritchie两位作者对《The C Programming Language》的第1版进行了修订，所以我们也根据ANSI标准和K&R的《The C Programming Language》（第2版）对有关习题解答进行了修订。

K&R所著的《The C Programming Language》（第2版）是C语言方面的经典教材，而这本与之配套的习题解答将帮助您更加深入地理解C语言并掌握良好的C语言编程技巧。您可以通过教材学习C语言，独立地解答书中的练习题，再钻研本书给出的习题答案。有关习题都是用教材中当时已经介绍过的语言结构来解答的，这样做的目的是为了使这本习题解答能够与教材中的教学内容保持同步。在学习到更多的C语言知识之后，相信大家能够给出更好的解决方案。例如，下面这条语句是在教材第15页介绍的：

```
if (表达式)
    语句-1
else
    语句-2
```

所以我们对出现在此之前的习题将不使用这条语句进行解答；但出现在教材第13页上的习题1-8、1-9、和1-10如果使用了这条语句，其解答将得到很大的改进。有时我们在解答中也列出使用了当时尚未介绍到的C语言知识的解决方案。

本书中的习题解答都进行了解释。我们将假设读者都已经读过了教材中有关习题出现之前的内容。我们不打算重复教材已经介绍过的内容，但会把各习题解答的要点指出来。

单凭阅读和学习其语法结构并不能真正掌握一门程序设计语言，必须进行编程实践——亲自编写一些程序并研究一些别人写的程序。我们的目标是：利用C语言良好的特性使程序模块化，充分利用库函数并以格式化的风格编写程序，这些将有助于大家清楚地了解程序的逻辑流程。我们希望这本书能够帮助大家成为C语言的高手。

我们要感谢以下朋友对本书的出版所给予的帮助：Brian Kernighan、Don Kostuch、Bruce Leung、Steve Mackey、Joan Magrabi、Julia Mistrello、Rosemary Morrissey、Andrew Nathanson、Sophie Papanikolaou、Dave Perlin、Carlos Tondo、John Wait和Eden Yount。

Clovis L. Tondo

参与本书翻译工作的人员还有杨晓云、王建桥、胡建平、张玉亭、韩蓝。

目 录

C PROGRAMMING LANGUAGE

第1章 导 言

练习1-1 （《C程序设计语言（第2版·新版）》即教材第3页）

在你自己的系统中运行“hello, world”程序。再有意去掉程序中的部分内容，看看会得到什么出错信息。

```
#include  <stdio.h>

main()
{
      printf("hello, world");
}
```

上面这个例子省略了换行符（\n），这将使光标停留在输出信息的末尾。

```
#include <stdio.h>
main()
{
      printf("hello, world\n")
}
```

第二个例子省略了printf()后面的分号。C程序的语句必须以分号结尾（参见教材第5页）。因此，对于本例，编译器将识别出少了一个分号并给出相应的出错信息。

```
#include <stdio.h>

main()
{
      printf("hello, world\n');
}
```

在第三个例子里，\n后面的双引号"被错写为单引号'。于是，这个单引号及其后面的右括号和分号将被看做是整个输出字符串的一部分。编译器将把这种情况视为一个错误，会报告说缺失了一个双引号；在右花括号前缺失了一个右圆括号；字符串过长；字符串中带有换行符。

练习1-2 （教材第3页）

做个实验，当printf函数的参数字符串中包含\c（其中c是上面的转义字符序列中未曾列出的某一个字符）时，观察一下会出现什么情况？

```
#include <stdio.h>

main()
{
      printf("hello, world\y");
      printf("hello, world\7");
      printf("hello, world\?");
}
```

参考手册（参见教材第169页）中提到："如果\后面紧跟的字符不在以上指定的字符中，则行为是未定义的。"

上面这段代码的执行结果与具体的编译器相关。一种可能出现的结果是：

```
hello, worldyhello, world<BELL>hello, world?
```

其中，<BELL>是ASCII值等于7的字符所产生的一声短蜂鸣。在\的后面，可以用最多3个八进制数字（参见教材第29页）来代表一个字符，而\7在ASCII字符集中代表的是一声短蜂鸣。

练习1-3 （教材第8页）

请修改温度转换程序，使之能在转换表的顶部打印一个标题。

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, . . ., 300; floating-point version */
main()
{
      float fahr, celsius;
      int lower, upper, step;

      lower = 0;   /* lower limit of temperature table   */
      upper = 300; /* upper limit                        */
      step  = 20;  /* step size                          */

      printf("Fahr Celsius\n");
      fahr = lower;
      while (fahr <= upper) {
            celsius = (5.0/9.0) * (fahr-32.0);
            printf("%3.0f   %6.1f\n", fahr, celsius);
            fahr = fahr + step;
```

在循环语句之前增加的printf("Fahr Celsius\n");语句将在温度转换表的顶部产生一个表头。为了让输出内容与这个表头对齐，我们还在%3.0f和%6.1f之间增加了两个空格。上面这个程序中的其余语句与教材第6页中给出的代码完全一致。

练习1-4 （教材第8页）

编写一个程序打印摄氏温度转换为相应华氏温度的转换表。

```
#include <stdio.h>

/* print Celsius-Fahrenheit table
```

```
     for celsius = 0, 20, ..., 300; floating-point version  */
main()
{
     float fahr, celsius;
     int lower, upper, step;

     lower = 0;          /* lower limit of temperature table */
     upper = 300;        /* upper limit                      */
     step  = 20;         /* step size                        */

     printf("Celsius  Fahr\n");
     celsius = lower;
     while (celsius <= upper) {
          fahr = (9.0*celsius) / 5.0 + 32.0;
          printf("%3.0f  %6.1f\n", celsius, fahr);
          celsius = celsius + step;
      }
}
```

本程序将输出一个摄氏温度（0～300）到华氏温度的转换表。华氏温度是用以下语句计算得到的：

```
fahr = (9.0*celsius) / 5.0 + 32.0
```

本题的解题思路与打印华氏温度到摄氏温度的对照表程序（见教材第6页）是相同的。整型变量lower、upper、step分别对应于变量celsius的下限、上限、步长。程序先把变量celsius初始化为它的下限，再在while循环中把对应的华氏温度计算出来。然后，程序打印出这组摄氏温度和华氏温度的值，并按步长递增变量celsius的值。while循环将一直执行直到变量celsius超出其上限为止。

练习1-5　（教材第9页）

修改温度转换程序，要求以逆序（即按照从300度递减到0度的顺序）打印温度转换表。

```
#include <stdio.h>

/* print Fahrenheit-Celsius table in reverse order         */
main()
{
     int fahr;

     for (fahr = 300; fahr >= 0; fahr = fahr - 20)
          printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

惟一的修改之处是：

```
for (fahr = 300; fahr >= 0; fahr = fahr - 20)
```

这条for语句的第一部分：

```
fahr = 300
```

负责把华氏温度变量（fahr）初始化为它的上限；for语句的第二部分（即for循环的控制条件）：

```
fahr >= 0
```

负责检查变量fahr是否大于或等于它的下限——只要这个检查的结果为真，for语句就将继续循环执行；for语句的第三部分（即步长表达式）：

```
fahr = fahr - 20
```

负责对变量fahr按步长进行递减操作。

练习1-6 （教材第11页）

验证布尔表达式getchar() != EOF的取值是0还是1。

```
#include <stdio.h>

main()
{
      int c;

      while (c = getchar() != EOF)
            printf("%d\n", c);
      printf("%d - at EOF\n", c);
}
```

根据教材第11页的论述，表达式

```
c = getchar() != EOF
```

相当于

```
c = (getchar() != EOF)
```

本程序从系统的标准输入读取字符并使用了上面的表达式。当有字符可读时，getchar()不会返回文件结束符（即EOF），所以

```
getchar() != EOF
```

的取值为真，变量c将被赋值为1。当程序遇到文件结束符时，表达式取值为假，此时，变量c将被赋值为0，程序将结束运行。

练习1-7 （教材第11页）

请编写一个打印EOF值的程序。

```
#include <stdio.h>

main()
{
      printf("EOF is %d\n", EOF);
}
```

符号常量EOF是在头文件<stdio.h>中定义的。在上面这个程序中，printf()语句中双引号外的EOF将被替换为头文件<stdio.h>中紧跟在

```
#define EOF
```

之后的文本。在我们的系统中，EOF被定义为-1，但在其他系统中，EOF可能被定义为其他

的值。这正是使用EOF等标准符号常量能够增加程序可移植性的原因所在。

练习1-8 （教材第13页）

编写一个统计空格、制表符和换行符个数的程序。

```
#include <stdio.h>

/* count blanks, tabs, and newlines                                    */
main()
{
      int c, nb, nt, nl;

      nb = 0;                          /* number of blanks        */
      nt = 0;                          /* number of tabs          */
      nl = 0;                          /* number of newlines      */
      while ((c = getchar()) !=EOF) {
            if (c == ' ')
                  ++nb;
            if (c == '\t')
                  ++nt;
            if (c == '\n')
                  ++nl;
      }
      printf("%d %d %d\n", nb, nt, nl);
}
```

整型变量nb nt和nl分别用来统计空格、制表符和换行符的个数。这3个变量的初值都是0。

在while循环的循环体内，出现在输入中的每一个空格、制表符和换行符都将被记录。while循环中的3条if语句在每次循环中都将被执行。如果程序读到的字符不是空格、制表符或换行符，就不执行任何操作。如果程序读到的字符是这三个符号之一，就对相应的计数器加1。当while循环终止（即getchar返回EOF）时，本程序将把空格、制表符和换行符的统计结果打印出来。

对if-else语句的介绍最早出现于教材第14页，下面是使用了这一语法结构的实现方法：

```
#include <stdio.h>

/* count blanks, tabs, and newlines                                    */
main()
{
      int c, nb, nt, nl;

      nb = 0;                          /* number of blanks        */
      nt = 0;                          /* number of tabs          */
      nl = 0;                          /* number of newlines      */
      while ((c = getchar()) !=EOF)
            if (c == ' ')
                  ++nb;
            else if (c == '\t')
                  ++nt;
            else if (c == '\n')
                  ++nl;
```

```
        printf("%d %d %d\n", nb, nt, nl);
}
```

练习1-9 （教材第13页）

编写一个将输入复制到输出的程序，并将其中连续的多个空格用一个空格代替。

```
#include <stdio.h>

#define NONBLANK 'a'

/* replace string of blanks with a single blank                */
main()
{
        int c, lastc;

        lastc = NONBLANK;
        while ((c = getchar()) != EOF) {
                if (c != ' ')
                        putchar(c);
                if (c == ' ')
                        if (lastc != ' ')
                                putchar(c);
                lastc = c;
        }
}
```

整型变量c负责记录当前输入字符的ASCII值，而整型变量lastc则记录着前一个输入字符的ASCII值。符号常量NONBLANK负责把变量lastc初始化为一个任意的非空格字符。

while循环体中的第一条if语句输出非空格字符；第二条if语句处理空格字符，而第三条if语句用于检查当前的空格字符究竟是一个单个的空格符还是一串空格中的第一个空格。最后，对变量lastc进行刷新。以上操作将一直重复到while循环终止（即getchar返回EOF）为止。

对if-else语句的介绍最早出现于教材第14页，下面是使用了这一语法结构的实现方法：

```
#include <stdio.h>

#define  NONBLANK 'a'

/* replace string of blanks with a single blank                */
main()
{
        int c, lastc;

        lastc = NONBLANK;
        while ((c = getchar()) != EOF) {
                if (c != ' ' )
                        putchar(c);
                else if (lastc != ' ')
                        putchar(c);
                lastc = c;
        }
}
```

对逻辑或（OR）操作符||的介绍也最早出现于教材第14页，下面是使用了这一知识的实现方法：

```
#include <stdio.h>

#define  NONBLANK 'a'

/* replace string of blanks with a single blank                    */
main()
{
      int c, lastc;

      lastc = NONBLANK;
      while ((c = getchar()) != EOF) {
            if (c != ' ' || lastc != ' ')
                  putchar(c);
            lastc = c;
      }
}
```

练习1-10 （教材第13页）

编写一个将输入复制到输出的程序，并将其中的制表符替换为\t，把回退符替换为\b，把反斜杠替换为\\，这样可以将制表符和回退符以可见的方式显示出来。

```
#include <stdio.h>

/* replace tabs and backspaces with visible characters      */
main()
{
      int c;

      while ((c = getchar()) != EOF) {
            if (c == '\t')
                  printf("\\t");
            if (c == '\b')
                  printf("\\b");
            if (c == '\\')
                  printf("\\\\");
            if (c != '\b')
                  if (c != '\t')
                        if (c != '\\')
                              putchar(c);
      }
}
```

输入的字符可以是一个制表符、一个回退符、一个反斜杠或者其他任何字符。如果输入是一个制表符，我们就把它替换为\t；如果输入是一个回退符，我们就把它替换为\b；如果输入是一个反斜杠，我们就把它替换为\\；其他字符则按原样输出。

在C语言中，反斜杠是用'\\'来表示的。因此，如果我们想输出两个反斜杠，就必须把字符串"\\\\"传递给printf函数。

对if-else语句的介绍最早出现于教材第14页，下面是使用了这一语法结构的实现方法：

```
#include <stdio.h>

/* replace tabs and backspaces with visible characters     */
main()
{
      int c;

      while ((c = getchar()) !=EOF)
            if (c == '\t')
                  printf("\\t");
            else if (c == '\b')
                  printf("\\b");
            else if (c == '\\')
                  printf("\\\\");
            else
                  putchar(c);
```

练习1-11 （教材第15页）

你准备如何测试单词计数程序？如果程序中存在某种错误，那么什么样的输入最可能发现这类错误呢？

单词计数程序的测试工作首先要从没有任何输入的情况开始。此时，该程序的输出结果应该是“0 0 0”，即零行、零单词、零字符。

接下来测试输入单字符单词的情况。此时，该程序的输出结果应该是“1 1 2”，即一行、一个单词、两个字符（一个字母加上一个换行符）。

再测试一个由两个字符组成的单词。此时，该程序的输出结果应该是“1 1 3”，即一行、一个单词、三个字符（二个字母加上一个换行符）。

然后，再测试两个单字符单词的情况。首先，两个单词出现在同一行，此时的输出结果应该是“1 2 4”；然后，两个单词各占一行，此时的输出结果应该是“2 2 4”。

那些满足边界条件的输入情况最有助于发现单词计数程序中的错误。这些边界条件包括：

——没有输入

——没有单词（只有换行符）

——没有单词（只有空格、制表符和换行符）

——每个单词各占一行的情况（没有空格和制表符）

——单词出现于文本行行首的情况

——单词出现于一串空格之后的情况

练习1-12 （教材第15页）

编写一个程序，以每行一个单词的形式打印其输入。

```
#include <stdio.h>

#define IN  1         /* inside a word                    */
```

```
#define OUT 0          /* outside a word                              */

/* print input one word per line                                      */
main()
{
      int c, state;

      state = OUT;
      while ((c = getchar()) != EOF) {
            if (c == ' ' || c == '\n' || c == '\t') {
                  if (state == IN) {
                        putchar('\n');      /* finish the word   */
                        state = OUT;
                  }
            } else if (state == OUT) {
                  state = IN;               /* beginning of word */
                  putchar(c);
            } else                          /* inside a word     */
                  putchar(c);
      }
}
```

整型变量state是一个布尔量，用于记录程序的处理过程是否正处于某个单词的内部。在程序刚开始运行的时候，变量state将被初始化为OUT，表明尚未处理任何数据。

第一条if语句

```
if (c == ' ' || c == '\n' || c == '\t')
```

判断变量c是否是一个单词分隔符。如果是，则第二条if语句

```
if (state == IN)
```

将判断这个单词分隔符是否表示某个单词结束。如果是，就输出一个换行符并修改变量state的值；如果不是，则不进行任何操作。

如果c不是一个单词分隔符，那么，它将或者是某单词的第一个字符、或者是一个单词中除第一个字符之外的其他字符。对于第一种情况（c是某单词的第一个字符），程序将修改变量state的值并输出这个字符；对于第二种情况（c是某个单词中的其他字符），程序直接输出这个字符。

练习1-13 （教材第17页）

编写一个程序，打印输入中单词长度的直方图。水平方向的直方图比较容易绘制，垂直方向的直方图则要困难些。

```
#include <stdio.h>

#define  MAXHIST 15          /* max length of histogram   */
#define  MAXWORD 11          /* max length of a word      */
#define  IN      1           /* inside a word             */
#define  OUT     0           /* outside a word            */

/* print horizontal histogram                              */
main()
{
```

```
int c, i, nc, state;
int len;                    /* length of each bar          */
int maxvalue;               /* maximum value for wl[]      */
int ovflow;                 /* number of overflow words    */
int wl[MAXWORD];            /* word length counters        */

state = OUT;
nc = 0;                     /* number of chars in a word */
ovflow = 0;                 /* number of words >= MAXWORD*/
for (i = 0; i < MAXWORD; ++i)
     wl[i] = 0;
while ((c = getchar ()) != EOF) {
     if (c == ' ' || c == '\n' || c == '\t') {
          state = OUT;
          if (nc > 0)
               if (nc < MAXWORD)
                    ++wl[nc];
               else
                    ++ovflow;
          nc = 0;
     } else if (state == OUT) {
          state = IN;
          nc = 1;           /* beginning of a new word   */
     } else
          ++nc;             /* inside a word             */
}
maxvalue = 0;
for (i = 1; i < MAXWORD; ++i)
     if (wl[i] > maxvalue)
          maxvalue = wl[i];
for (i = 1; i < MAXWORD; ++i) {
     printf("%5d - %5d : ", i, wl[i]);
     if (wl[i] > 0) {
          if ((len = wl[i] * MAXHIST / maxvalue) <= 0)
               len = 1;
     } else
          len = 0;
     while (len > 0) {
          putchar ('*');
          --len;
     }
     putchar('\n');
}
if (ovflow > 0)
     printf("There are %d words >= %d\n", ovflow, MAXWORD);
}
```

空格、换行符或制表符标志着单词的结束。如果有一个单词（nc>0）且它的长度小于允许的单词最大长度（nc<MAXWORD），这个程序将对相应的单词长度计数器加1（++wl[nc]）。如果单词的长度超出了允许的单词最大长度（nc>=MAXWORD），这个程序将对变量ovflow加1以记录长度大于或等于MAXWORD的单词的个数。

在读入全部单词之后，我们的程序将找出数组wl中的最大值（maxvalue）。

变量len是根据 MAXHIST 和maxvalue的值计算得出的 wl[i]所对应的直方图长度。如果wl[i]大于零，就至少要打印出一个星号。

```
#include <stdio.h>

#define  MAXHIST     15          /* max length of histogram   */
#define  MAXWORD     11          /* max length of a word      */
#define  IN          1           /* inside a word             */
#define  OUT         0           /* outside a word            */

/* print vertical histogram                                   */
main()
{
      int c, i, j, nc, state;
      int maxvalue;              /* maximum value for wl[]    */
      int ovflow;                /* number of overflow words  */
      int wl[MAXWORD];           /* word length counters      */
      state = OUT;
      nc = 0;                 /* number of chars in a word    */
      ovflow = 0;             /* number of words >= MAXWORD   */
      for (i = 0; i < MAXWORD; ++i)
           wl[i] = 0;
      while ((c = getchar()) != EOF) {
           if (c == ' ' || c == '\n' || c == '\t') {
                state = OUT;
                if (nc > 0)
                     if (nc < MAXWORD)
                          ++wl[nc];
                     else
                          ++ovflow;
                nc = 0;
           } else if (state == OUT) {
                state = IN;
                nc = 1;       /* beginning of a new word     */
           } else
                ++nc;         /* inside a word               */
      }
      maxvalue = 0;
      for (i = 1; i < MAXWORD; ++i)
           if (wl[i] > maxvalue)
                maxvalue = wl[i];

      for (i = MAXHIST; i > 0; --i) {
           for (j = 1; j < MAXWORD; ++j)
                if (wl[j] * MAXHIST / maxvalue >= i)
                     printf(" * ");
                else
                     printf("   ");
           putchar('\n');
      }
      for (i = 1; i < MAXWORD; ++i)
           printf("%4d ", i);
      putchar('\n');
      for (i = 1; i < MAXWORD; ++i)
           printf("%4d ", wl[i]);
```

```
        putchar('\n');
        if (ovflow > 0)
            printf("There are %d words >= %d\n", ovflow, MAXWORD);
}
```

这个实现方法将输出一个垂直方向的直方图。这个程序从开始直到求值maxvalue的过程与前一个程序完全相同。然后，这个程序需要计算数组w1中的每一个元素并判断是否需要在数组元素的对应位置上打印一个星号。这一判断过程必不可少，因为垂直方向直方图的所有直方条是同步打印的。最后的两个for循环用来输出数组w1各元素的下标和取值。

练习1-14 （教材第17页）

编写一个程序，打印输入中各个字符出现频度的直方图。

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

#define MAXHIST 15              /* max length of histogram    */
#define MAXCHAR 128             /* max different characters   */

/* print horizontal histogram freq. of different characters */
main()
{
        int c, i;
        int len;                /* length of each bar         */
        int maxvalue;           /* maximum value for cc[]     */
        int cc[MAXCHAR];        /* character counters         */

        for (i = 0; i < MAXCHAR; ++i)
            cc[i] = 0;
        while ((c = getchar()) != EOF)
            if (c < MAXCHAR)
                ++cc[c];
        maxvalue = 0;
        for (i = 1; i < MAXCHAR; ++i)
            if (cc[i] > maxvalue)
                maxvalue = cc[i];

        for (i = 1; i < MAXCHAR; ++i) {
            if (isprint(i))
                printf("%5d - %c - %5d : ", i, i, cc[i]);
            else
                printf("%5d -   - %5d : ", i, cc[i]);
            if (cc[i] > 0) {
                if ((len = cc[i] * MAXHIST / maxvalue) <= 0)
                    len = 1;
            } else
                len = 0;
            while (len > 0) {
                putchar('*');
                --len;
            }
            putchar('\n');
        }
}
```

这个程序与练习1-13中的水平直方图输出程序很相似，但我们现在统计的是各个字符的出现频度。程序中使用了一个元素个数等于MAXCHAR的字符计数器数组，如果我们使用的字符集中存在值大于或等于MAXCHAR的字符，则这些字符将被忽略。另一个区别是我们使用了一个宏来判断某个字符是否是一个可显示字符。关于包含的头文件<ctype.h>的讨论出现于教材第34页，对isprint的介绍则出现于教材第227页（附录B）。

练习1-15 （教材第19页）

重新编写1.2节中的温度转换程序，使用函数实现温度转换计算。

```
#include <stdio.h>

float celsius(float fahr);

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, . . ., 300; floating-point version     */
main()
{
     float fahr;
     int lower, upper, step;

     lower = 0;          /* lower limit of temperature table */
     upper = 300;        /* upper limit                      */
     step  = 20;         /* step size                        */

     fahr = lower;
     while (fahr <= upper) {
          printf("%3.0f %6.1f\n", fahr, celsius(fahr));
          fahr = fahr + step;
     }
}

/* celsius: convert fahr into celsius                        */
float celsius(float fahr)
{
     return (5.0/9.0) * (fahr-32.0);
}
```

我们采用一个函数把华氏温度转换为摄氏温度。这个函数名为celsius，它的输入参数是一个浮点数，返回值也是一个浮点数。这个函数将通过return语句返回表达式的值。有时，表达式是一个简单变量，例如power函数（参见教材第19页）；但有时我们也会使用一个比较复杂的表达式，例如这里的celsius函数，因为这样做可以使所有工作都在return语句中完成。

因为函数celsius的输入参数是一个浮点数，它的返回值也是一个浮点数，所以我们把它声明为如下的形式：

```
float celsius(float fahr);
```

练习1-16 （教材第22页）

修改打印最长文本行的程序的主程序main，使之可以打印任意长度的输入行的长度，并

尽可能多地打印文本。

```
#include <stdio.h>
#define  MAXLINE 1000        /* maximum input line size    */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print longest input line                                 */
main()
{
     int len;                /* current line length        */
     int max;                /* maximum length seen so far */
     char line[MAXLINE];     /* current input line         */
     char longest[MAXLINE]; /* longest line saved here     */

     max = 0;
     while ((len = getline(line, MAXLINE)) > 0) {
          printf("%d, %s", len, line);
          if (len > max) {
               max = len;
               copy(longest, line);
          }
     }
     if (max > 0)            /* there was a line           */
          printf("%s", longest);
     return 0;
}

/* getline: read a line into s, return length              */
int getline(char s[], int lim)
{
     int c, i, j;

     j = 0;
     for (i = 0; (c = getchar()) !=EOF && c != '\n'; ++i)
          if (i < lim-2) {
               s[j] = c;      /* line still in boundaries */
               ++j;
          }
          if (c == '\n') {
               s[j] = c;
               ++j;
               ++i;
          }
          s[j] = '\0';
          return i;
     }

     /* copy: copy 'from' into 'to'; assume to is big enough   */
     void copy (char to[], char from[])
     {
          int i;

          i = 0;
```

```
        while ((to[i] = from[i]) != '\0')
            ++i;
    }
```

我们只对原来的主函数进行了一处修改，如下所示：

```
printf("%d, %s", len, line);
```

这条语句将把输入行的长度（len）和保存在数组line里的字符尽可能多地打印出来。

我们对函数getline也做了几处修改。首先，循环语句

```
for (i = 0; (c = getchar()) != EOF && c != '\n'; ++i)
```

不再检查字符数是否超出数组s的限制——因为这一限制不再是for循环的终止条件之一了，现在的getline能返回任意长的输入行的长度并能存储尽可能多的输入行内容。其次，原来for循环中用于判断字符串数组是否还有空位置的表达式

```
i < lim-1
```

被修改为语句

```
if (i < lim-2)
```

之所以要进行这样的修改，是因为数组s的最后一个下标是

```
lim-1
```

而这又是因为数组s中有lim个元素且我们已经读取了输入字符。所以

```
i < lim-2
```

将在数组s里给换行符留出一个位置，即

```
s[lim-2] = '\n'
```

还要给字符串结束符留出一个位置，即

```
s[lim-1] = '\0'
```

字符串的长度将通过变量i返回；而变量j则记录着被复制到字符串s中的字符的个数。

练习1-17　（教材第22页）

编写一个程序，打印长度大于80个字符的所有输入行。

```
#include <stdio.h>
#define  MAXLINE   1000     /* maximum input line size     */
#define  LONGLINE  80

int getline(char line[], int maxline);

/* print lines longer than LONGLINE                        */
main()
{
     int len;                          /* current line length */
     char line[MAXLINE];               /* current input line  */

     while ((len = getline(line, MAXLINE)) > 0)
          if (len > LONGLINE)
               printf("%s", line);
```

```
        return 0;
}
```

这个程序调用函数getline来读取输入行。函数getline将返回输入行的长度和尽可能多的内容。如果输入行的长度大于80个字符（LONGLINE），我们的程序就将把它打印出来；否则，不进行任何操作。这一过程将一直循环到函数getline返回一个等于零的输入行长度为止。

函数getline与练习1-16中的同名函数相同。

练习1-18 （教材第22页）

编写一个程序，删除每个输入行末尾的空格及制表符，并删除完全是空格的行。

```
#include  <stdio.h>
#define   MAXLINE   1000      /* maximum input line size    */

int getline(char line[], int maxline);
int remove(char s[]);

/* remove trailing blanks and tabs, and delete blank lines */
main()
{
      char line[MAXLINE];     /* current input line         */

      while (getline(line, MAXLINE) > 0)
            if (remove(line) > 0)
                  printf("%s", line);
      return 0;
}

/* remove trailing blanks and tabs from character string s */
int remove (char s[])
{
      int i;

      i = 0;
      while (s[i] != '\n')    /* find newline character     */
            ++i;
      --i;                    /* back off from '\n'         */
      while (i >= 0 && (s[i] == ' ' || s[i] == '\t'))
            --i;
      if (i >= 0) {           /* is it a nonblank line?     */
            ++i;
            s[i] = '\n';      /* put newline character back */
            ++i;
            s[i] = '\0';      /* terminate the string       */
      }
      return i;
}
```

remove函数负责删掉字符串line末尾的空格和制表符并返回它的新长度。如果这个长度大于零，就说明line中有不是空格和制表符的其他字符，程序就会把这一行打印出来；否则，就说明line完全是由空格和制表符构成的，程序就将忽略掉这个输入行。这就保证了完

全是空格的行不会被打印出来。

remove函数首先找到换行符，然后倒退一个位置。随后，这个函数将从后向前检查空格或制表符，直到它找到一个不是空格或制表符的字符或者没有字符可让它继续倒退（即 i< 0）为止。如果i>=0，则说明至少还有一个字符。此后，函数remove将换行符和字符串结束符写回输入行，再返回变量i。

函数getline与练习1-16中的同名函数相同。

练习1-19 （教材第22页）

编写函数reverse(s)将字符串s中的字符顺序颠倒过来。使用该函数编写一个程序，每次颠倒一个输入行中的字符顺序。

```
#include  <stdio.h>
#define   MAXLINE   1000        /* maximum input line size   */

int getline(char line[], int maxline);
void reverse(char s[]);

/* reverse input lines, a line at a time                     */
main()
{
     char line[MAXLINE];        /* current input line        */

     while (getline(line, MAXLINE) > 0) {
          reverse(line);
          printf("%s", line);
     }
}

/* reverse: reverse string s                                 */
void reverse(char s[])
{
     int i, j;
     char temp;

     i = 0;
     while (s[i] != '\0')       /* find the end of string s  */
          ++i;
     --i;                       /* back off from '\0'        */
     if (s[i] == '\n')
          --i;                  /* leave newline in place    */
     j = 0;                     /* beginning of new string s */
     while (j < i) {
          temp = s[j];
          s[j] = s[i];          /* swap the characters       */
          s[i] = temp;
          --i;
          ++j;
     }
}
```

reverse函数先要找到字符串s的末尾，然后从'\0'倒退一个位置，这样可以保证颠倒后

得到的字符串的第一个字符不会成为一个字符串结束符。如果从'\0'倒退一个位置后遇到的是一个换行符'\n'，那就再倒退一个位置，因为换行符也像'\0'一样必须出现在行的末尾。

变量j先被设置为字符串的第一个字符的下标，变量i则被设置为字符串最后一个字符的下标。在交换字符的过程中，程序将对变量j进行递增（即从字符串的第一个字符向字符串的尾部方向移动），对变量i进行递减（即从字符串的最后一个字符向字符串的头部方向移动）。整个过程将一直进行到变量j大于或等于变量i时停止。

主程序每次读取一个输入行，颠倒之，然后把颠倒后的文本行打印出来。

函数getline与练习1-16中的同名函数相同。

练习1-20 （教材第25页）

请编写程序detab，将输入中的制表符替换成适当数目的空格，使空格充满到下一个制表符终止的地方。假设制表符终止位的位置是固定的，比如每隔*n*列就会出现一个制表符终止位。*n*应该作为变量还是符号常量呢？

```
#include <stdio.h>

#define   TABINC   8      /* tab increment size                    */

/* replace tabs with the proper number of blanks                    */
main()
{
     int c, nb, pos;

     nb = 0;              /* number of blanks necessary            */
     pos = 1;             /* position of character in line         */
     while ((c = getchar()) != EOF) {
          if (c == '\t') {          /* tab character                */
               nb = TABINC - (pos - 1) % TABINC;
               while (nb > 0) {
                    putchar(' ');
                    ++pos;
                    --nb;
               }
          } else if (c == '\n' { /* newline character              */
               putchar(c);
               pos = 1;
          } else {                  /* all other characters        */
               putchar(c);
               ++pos;
          }
     }
}
```

我们假设每隔TABINC个位置就会出现一个制表位。在这个程序中，我们把TABINC定义为8。变量pos是程序在文本行中的当前位置。

当遇到制表符的时候，程序将计算出要到达下一个制表位需要的空格数nb。这一数值是用下面这条语句计算出来的：

```
nb = TABINC - (pos - 1) % TABINC
```

如果遇到的是一个换行符，程序将把它打印出来并把变量pos重新初始化为输入行的第一个字符位置（pos=1）。如果遇到的是其他字符，程序将把它打印出来并递增变量pos的值（++pos）。

我们把TABINC定义为一个符号常数。在第5章中，我们将学习到如何向主程序传递参数，利用那一知识，你就能让用户自行设定连续两个制表位之间的间隔了。到那时，你也许会安排一个变量来保存TABINC的值。

程序detab将在练习5-11和练习5-12中进行扩展。

练习1-21　（教材第25页）

编写程序entab，将空格串替换为最少数量的制表符和空格，但要保持单词之间的间隔不变。假设制表符终止位的位置与练习1-20的detab程序的情况相同。当使用一个制表符或者一个空格都可以到达下一个制表符终止位时，选用哪种替换字符比较好？

```
#include  <stdio.h>

#define   TABINC   8                        /* tab increment size     */

/* replace strings of blanks with tabs and blanks                      */
main()
{
     int c, nb, nt, pos;

     nb = 0;                                /* # of blanks necessary */
     nt = 0;                                /* # of tabs necessary   */
     for (pos = 1; (c = getchar()) != EOF; ++pos)
          if (c == ' ') {
               if (pos % TABINC != 0)
                    ++nb;                   /* increment # of blanks */
               else {
                    nb = 0;                 /* reset # of blanks     */
                    ++nt;                   /* one more tab          */
               }
          } else {
               for ( ; nt > 0; --nt)
                    putchar('\t');          /* output tab(s)         */
               if (c == '\t')               /* forget the blank(s)   */
                    nb = 0;
               else                         /* output blank(s)       */
                    for ( ; nb > 0; --nb)
                         putchar(' ');
               putchar(c);
               if (c == '\n')
                    pos = 0;
               else if (c == '\t')
                    pos = pos + (TABINC - (pos-1) % TABINC) - 1;
          }
}
```

整型变量nb和nt分别是用来替换空格串的空格和制表符的最少个数。变量pos是程序在文本行中的当前位置。

程序的主要想法是找出全部空格。变量pos每递增到TABINC的一个倍数时，我们就要把空格串替换为一个制表符。

当遇到一个非空格符时，程序将先把遇到这个字符之前积累起来的制表符和空格打印出来，再把这个字符打印出来。然后，程序将把变量nb和nt重新设置为零，如果当前字符是一个换行符，还要把变量pos重新设置为输入行的开始。

当遇到一个制表符时，程序将只把此前积累的制表符和当前遇到这个制表符打印出来。

如果只需一个空格就能到达下一个制表位，我们的选择是把它替换为一个制表符，因为这有助于避免特殊情况。

程序entab将在练习5-11和练习5-12中进行扩展。

练习1-22 （教材第25页）

编写一个程序，把较长的输入行“折”成短一些的两行或多行，折行的位置在输入行的第n列之前的最后一个非空格符之后。要保证程序能够智能地处理输入行很长以及在指定的列前没有空格或制表符的情况。

```
#include <stdio.h>

#define   MAXCOL    10          /* maximum column of input     */
#define   TABINC    8           /* tab increment size          */

char line[MAXCOL];              /* input line                  */

int exptab(int pos);
int findblnk(int pos);
int newpos(int pos);
void printl(int pos);

/* fold long input lines into two or more shorter lines       */
main()
{
     int c, pos;

     pos = 0;                   /* position in the line        */
     while ((c = getchar()) != EOF) {
          line[pos] = c;        /* store current character     */
          if (c == '\t')        /* expand tab character        */
               pos = exptab(pos);
          else if (c == '\n') {
               printl(pos);     /* print current input line    */
               pos = 0;
          } else if (++pos >= MAXCOL) {
               pos = findblnk(pos);
               printl(pos);
               pos = newpos(pos);
          }
     }
}

/* printl: print line until pos column                        */
void printl(int pos)
```

```
{
     int i;
     for (i = 0; i < pos; ++i)
          putchar(line[i]);
     if (pos > 0)                    /* any chars printed ?       */
          putchar('\n');
}

/* exptab: expand tab into blanks                                 */
int exptab(int pos)
{
     line[pos] = ' ';                /* tab is at least one blank */
     for (++pos; pos < MAXCOL && pos % TABINC != 0; ++pos)
          line[pos] = ' ';
     if (pos < MAXCOL)               /* room left in current line */
          return pos;
     else {                          /* current line is full      */
          printl(pos);
          return 0;                  /* reset current position    */
     }
}

/* findblnk: find blank's position                                */
int findblnk(int pos)
{
     while (pos > 0 && line[pos] != ' ')
          --pos;
     if (pos == 0)                   /* no blanks in the line ?   */
          return MAXCOL;
     else                            /* at least one blank        */
          return pos+1;              /* position after the blank  */
}

/* newpos: rearrange line with new position                       */
int newpos(int pos)
{
     int i, j;

     if (pos <= 0 || pos >= MAXCOL)
          return 0;                  /* nothing to rearrange      */
     else {
          i = 0;
          for (j = pos; j < MAXCOL; ++j) {
               line[i] = line[j];
               ++i;
          }
          return i;                  /* new position in line      */
     }
}
```

MAXCOL是一个符号常量，它给出了输入行的折行位置，即输入行的第*n*列。整型变量pos是程序在文本行中的当前位置。程序将在输入行的每一处第*n*列之前对该输入行折行。

这个程序将把制表符扩展为空格；每遇到一个换行符就把此前的输入文本打印出来；每当变量pos的值达到MAXCOL时，就会对输入行进行“折叠”。

函数findblnk从输入行的pos处开始倒退着寻找一个空格（目的是为了保持折行位置的单词的完整）。如果找到了一个空格符，它就返回紧跟在该空格符后面的那个位置的下标；如果没有找到空格，它就返回 MAXCOL。

函数print1打印输出从位置零到位置pos－1之间的字符。

函数newpos调整输入行，它将把从位置pos开始的字符复制到下一个输出行的开始，然后再返回变量pos的新值。

练习1-23 （教材第25页）

编写一个删除C语言程序中所有的注释语句。要正确处理带引号的字符串与字符常量。在C语言程序中，注释不允许嵌套。

```
#include <stdio.h>

void rcomment(int c);
void in_comment(void);
void echo_quote(int c);

/* remove all comments from a valid C program              */
main()
{
     int c, d;

     while ((c = getchar()) != EOF)
          rcomment(c);
     return 0;
}

/* rcomment: read each character, remove the comments      */
void rcomment(int c)
{
     int d;

     if (c == '/')
          if ((d = getchar()) == '*')
               in_comment();           /*beginning comment*/
          else if (d == '/') {
               putchar(c);             /*another slash    */
               rcomment(d);
          } else {
               putchar(c);             /* not a comment   */
               putchar(d);
          }
     else if (c == '\'' || c == '"')
          echo_quote(c);               /* quote begins    */
     else
          putchar(c);                  /* not a comment   */
}
/* in_comment: inside of a valid comment                   */
void in_comment(void)
{
     int c, d;
```

```
        c = getchar();                          /* prev character  */
        d = getchar();                          /* curr character  */
        while (c != '*' || d != '/') {          /* search for end  */
                c = d;
                d = getchar();
        }
}

/* echo_quote: echo characters within quotes                    */
void echo_quote(int c)
{
        int d;

        putchar(c);
        while ((d = getchar()) != c) {          /* search for end  */
                putchar(d);
                if (d == '\\')
                        putchar(getchar());     /* ignore escape seq*/
        }
        putchar(d);
}
```

这个程序假设输入是一个合法的C程序。函数rcomment搜索注释语句的起始标志（/*）；在找到这个标志时，它将调用另一个函数in_comment搜索注释语句的结束标志（*/），从而确保C程序中的注释语句都能够被删除。

函数rcomment还将搜索单引号和双引号；在找到它们时，它将调用另一个函数echo_quote。函数echo_quote的参数将指明找到的字符是一个单引号还是一个双引号。echo_quote确保引号中的内容能够按原样输出，不会被误认为是注释。函数echo_quote不会把跟在一个反斜杠后面的引号看做是结束引号（参见教材第13页和练习1-2中关于转义字符序列的讨论）。其他任何字符都将按原样输出。

本程序将在getchar返回文件结束符时结束运行。

练习1-24　（教材第25页）

编写一个程序，查找C语言程序中的基本语法错误，如圆括号、方括号以及花括号不配对等。要正确处理引号（包括单引号、双引号）、转义字符序列与注释。（如果读者想把该程序编写成完全通用的程序，难度会比较大。）

```
#include <stdio.h>

int brace, brack, paren;

void in_quote(int c);
void in_comment(void);
void search(int c);

/* rudimentary syntax checker for C programs                    */
main()
{
        int c;
        extern int brace, brack, paren;
```

```
while ((c = getchar()) != EOF) {
      if (c == '/') {
            if ((c = getchar()) == '*')
                  in_comment();          /* inside comment    */
            else
                  search(c);
      } else if (c == '\'' || c == '"')
            in_quote(c);                 /* inside quote      */
      else
            search(c);

      if (brace < 0) {                   /* output errors     */
            printf("Unbalanced braces\n");
            brace = 0;
      } else if (brack < 0) {
            printf("Unbalanced brackets\n");
            brack = 0;
      } else if (paren < 0) {
            printf("Unbalanced parentheses\n");
            paren = 0;
      }
}
if (brace > 0)                           /* output errors     */
      printf("Unbalanced braces\n");
if (brack > 0)
      printf("Unbalanced brackets\n");
if (paren > 0)
      printf("Unbalanced parentheses\n");
}

/* search: search for rudimentary syntax errors                    */
void search(int c)
{
      extern int brace, brack, paren;

      if (c == '{')
            ++brace;
      else if (c == '}')
            --brace;
      else if (c == '[')
            ++brack;
      else if (c == ']')
            --brack;
      else if (c == '(')
            ++paren;
      else if (c == ')')
            --paren;
}

/* in_comment: inside of a valid comment                           */
void in_comment(void)
{
      int c, d;

      c = getchar();                     /* prev character    */
```

```
        d = getchar();                          /* curr character    */
        while (c != '*' || d != '/') {          /* search for end    */
              c = d;
              d = getchar();
        }
}
/* in_quote: inside quote                                          */
void in_quote(int c)
{
        int d;

        while ((d = getchar()) != c)            /* search end quote  */
              if (d == '\\')
                    getchar();                  /* ignore escape seq */
}
```

上面这个程序只对基本语法错误中的一部分进行了检查。

这个程序检查3种语法错误：不配对的圆括号、方括号或花括号；其他语法问题都不在本程序的检查范围之内。

函数search每遇到一个左花括号（'{'）就对变量brace进行递增，每遇到一个右花括号（'}'）就对变量brace进行递减。对变量brack（对应于方括号）和paren（对应于圆括号）的处理也与此类似。

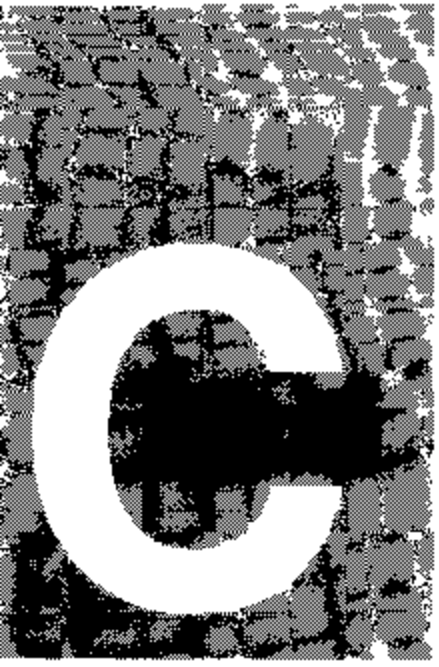
在搜索过程中，变量brace、brack和paren的取值是正数或等于零的情况都是合理的；但如果变量brace、brack和paren的取值变成了负数，就表示有语法错误，程序将打印出一条相应的出错信息。举例来说，连续出现3个左方括号[[[的情况（它将使变量brack等于3）是合法的，因为可能会在后面的搜索中找到与之配对的3个右方括号。可要是连续出现的3个右方括号]]]使变量brack等于-3，就不合法了，因为这表明前面没有与这3个右方括号配对的左方括号。如果有与之配对的3个左方括号的话，变量brack的值应该等于0。因此，语句

```
if (brace < 0) {
        printf("Unbalanced braces\n");
        brace = 0;
} else if (brack < 0) {
        printf("Unbalanced brackets\n");
        brack = 0;
} else if (paren < 0) {
        printf("Unbalanced parentheses\n");
        paren = 0;
}
```

是必不可少的。如果没有这条语句，诸如)(、]]][[[或}}{{之类的情况就都会被认为是配对的。

主函数 main还将搜索并检查引号（单引号、双引号）和注释标志的配对情况，但对配对的标志之间的字符不做检查——因为注释或引号中的圆括号、方括号或花括号不要求配对出现。

当遇到EOF标记的时候，程序将对是否存在尚未配对的圆括号、方括号或花括号进行最后的检查。如果存在有括号不配对的情况，程序就会输出相应的出错信息。



类型、运算符与表达式

练习2-1 （教材第28页）

编写一个程序以确定分别由signed及unsigned限定的char、short、int及long类型变量的取值范围。采用打印标准头文件中的相应值以及直接计算两种方式实现。通过直接计算来确定浮点类型的取值范围是一项难度很大的任务。

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

/* determine ranges of types                                    */
main()
{
     /* signed types                                            */
     printf("signed char min   = %d\n", SCHAR_MIN);
     printf("signed char max   = %d\n", SCHAR_MAX);
     printf("signed short min  = %d\n", SHRT_MIN);
     printf("signed short max  = %d\n", SHRT_MAX);
     printf("signed int min    = %d\n", INT_MIN);
     printf("signed int max    = %d\n", INT_MAX);
     printf("signed long min   = %ld\n", LONG_MIN);
     printf("signed long max   = %ld\n", LONG_MAX);
     /* unsigned types                                          */
     printf("unsigned char max  = %u\n", UCHAR_MAX);
     printf("unsigned short max = %u\n", USHRT_MAX);
     printf("unsigned int max   = %u\n", UINT_MAX);
     printf("unsigned long max  = %lu\n", ULONG_MAX);
}
```

ANSI C标准规定：各种类型的取值范围必须在头文件<limits.h>中定义。short、int和long类型在不同的硬件上有不同的长度，所以它们在不同机器上的取值范围也往往会不同。上面是利用标准头文件来确定类型取值范围的解决方案。

```
#include  <stdio.h>

/* determine ranges of types                                    */
main()
{
     /* signed types                                            */
     printf("signed char min   = %d\n",
                          -(char)((unsigned char) ~0 >> 1));
     printf("signed char max   = %d\n",
                          (char)((unsigned char) ~0 >> 1));
```

```
        printf("signed short min = %d\n",
                        -(short)((unsigned short) ~0 >> 1));
        printf("signed short max = %d\n",
                        (short)((unsigned short) ~0 >> 1));
        printf("signed int min   = %d\n",
                        -(int)((unsigned int) ~0 >> 1));
        printf("signed int max   = %d\n",
                        (int)((unsigned int) ~0 >> 1));
        printf("signed long min  = %ld\n",
                        -(long)((unsigned long) ~0 >> 1));
        printf("signed long max  = %ld\n",
                        (long)((unsigned long) ~0 >> 1));
        /* unsigned types                                          */
        printf("unsigned char max  = %u\n",
                        (unsigned char) ~0);
        printf("unsigned short max = %u\n",
                        (unsigned short) ~0);
        printf("unsigned int max   = %u\n",
                        (unsigned int) ~0);
        printf("unsigned long max  = %lu\n",
                        (unsigned long) ~0);
}
```

另一解决方案是利用按位运算符（参见教材第39页）进行计算。表达式

```
(char)((unsigned char) ~0 >> 1)
```

先把数字0的各个二进制位全部转换为1：

```
~0
```

然后，将结果值转换为unsigned char类型：

```
(unsigned char) ~0
```

再把这个unsigned char类型值右移一位以清除符号位：

```
(unsigned char) ~0 >> 1
```

最后，把它转换为char类型：

```
(char)((unsigned char) ~0 >> 1)
```

这一系列操作的最终结果就得到了signed类型字符的最大值。

练习2-2 （教材第33页）

在不使用&&或||的条件下编写一个与上面的for循环语句等价的循环语句。

原来的for循环语句：

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
```

与之等价的循环语句：

```
enum loop { NO, YES };
enum loop okloop = YES;

i = 0;
while (okloop == YES)
```

```
if (i >= lim-1)              /* outside of valid range ?    */
     okloop = NO;
else if ((c = getchar()) == '\n')
     okloop = NO;
else if (c == EOF)           /* end of file ?               */
     okloop = NO;
else {
     s[i] = c;
     ++i;
}
```

在不使用&&和||操作符的情况下，我们就只能把原来的for循环分解为一系列if语句。我们还必须修改有关的条件表达式。例如，在原来的for循环语句中，

```
i < lim-1
```

的作用是判断变量i是否仍然在其取值范围之内。在等价的循环语句中，

```
i >= lim-1
```

表明变量i超出其取值范围时循环应当结束。

okloop是一个枚举变量。一旦条件中的某一个得到满足，okloop就将被设置为NO，而循环也就结束了。

练习2-3 （教材第37页）

编写函数htoi(s)，把由十六进制数字组成的字符串（包含可选的前缀0x或0X）转换为与之等价的整型值。字符串中允许包含的数字包括：0~9、a~f以及A~F。

```
#define   YES       1
#define   NO        0

/* htoi: convert hexadecimal string s to integer                    */
int htoi(char s[])
{
     int hexdigit, i, inhex, n;

     i = 0;
     if (s[i] == '0') {          /* skip optional 0x or 0X          */
          ++i;
          if (s[i] == 'x' || s[i] == 'X')
               ++i;
     }
     n = 0;                      /* integer value to be returned    */
     inhex = YES;                /* assume valid hexadecimal digit  */
     for ( ; inhex == YES; ++i) {
          if (s[i] >= '0' && s[i] <= '9')
               hexdigit = s[i] - '0';
          else if (s[i] >= 'a' && s[i] <= 'f')
               hexdigit = s[i] - 'a' + 10;
          else if (s[i] >= 'A' && s[i] <= 'F')
               hexdigit = s[i] - 'A' + 10;
          else
               inhex = NO;     /* not a valid hexadecimal digit   */
          if (inhex == YES)
               n = 16 * n + hexdigit;
```

```
        }
        return n;
}
```

整个函数是由下面这条语句控制的：

```
for ( ; inhex == YES; ++i)
```

其中，整型变量i是数组s的下标。当s[i]是一个合法的十六进制数字时，inhex的取值将保持为YES，而循环也将继续执行。整型变量hexdigit的取值范围是0~15。

语句

```
if (inhex == YES)
```

保证数组元素s[i]是一个合法的十六进制数字字符，其值在hexdigit的范围之内。当循环结束时，函数htoi将返回变量n的值。

这个函数与atoi很相似（参见教材第35页）。

练习2-4 （教材第38页）

重新编写函数squeeze(s1, s2)，将字符串s1中任何与字符串s2中字符匹配的字符都删除。

```
/* squeeze: delete each char in s1 which is in s2                    */
void squeeze(char s1[], char s2[])
{
        int i, j, k;

        for (i = k = 0; s1[i] != '\0'; i++) {
                for (j = 0; s2[j] != '\0' && s2[j] != s1[i]; j++)
                        ;
                if (s2[j] == '\0')          /* end of string - no match */
                        s1[k++] = s1[i];
        }
        s1[k] = '\0';
}
```

这个函数的第一条语句

```
for (i = k = 0; s1[i] != '\0'; i++)
```

对整型变量i和k、字符数组s1的下标以及结果字符串（也就是s1）分别进行了初始化。字符串s1中与字符串s2中的字符相匹配的字符都将被删除。整个循环语句将一直执行到字符串s1结束为止。

第二条for语句将s1[i]与s2中的每个字符相比较。这个循环将执行到字符串s2结束或者找到一个匹配字符为止。如果没有找到匹配的字符，s1[i]就将被复制到结果字符串中；如果找到了匹配的字符，语句

```
if (s2[j] == '\0')
```

中的条件表达式的求值结果将是假，s1[i]就不会被复制到结果字符串中（它将从字符串s1中被剔除出去）。

练习2-5 （教材第38页）

编写函数any（s1,s2），将字符串s2中的任一字符在字符串s1中第一次出现的位置作

为结果返回。如果s1中不包含s2中的字符，则返回－1。（标准库函数strpbrk具有同样的功能，但它返回的是指向该位置的指针。）

```
/* any: return first location in s1 where any char from s2 occurs*/
int any(char s1[], char s2[])
{
      int i, j;

      for (i = 0; s1[i] != '\0'; i++)
            for (j = 0; s2[j] != '\0'; j++)
                  if (s1[i] == s2[j])       /* match found?          */
                        return i;           /* location first match  */
      return -1;                            /* otherwise, no match   */
}
```

整个函数是由下面这条语句控制的：

```
for (i = 0; s1[i] != '\0'; i++)
```

当这个循环正常结束时（即到达字符串s1的末尾时），函数any将返回－1以表明在字符串s1中没有找到字符串s2中的字符。

变量i的取值每变化一次，第二条for语句

```
for (j = 0; s2[j] != '\0'; j++)
```

都会被执行。它将s1[i]与s2中的每个字符相比较。如果字符串s2中的某个字符与s1[i]相匹配，则返回变量i——也就是字符串s1中最早出现字符串s2中的字符的那个位置。

练习2-6 （教材第40页）

编写一个函数setbits(x, p, n, y)，该函数返回对x执行下列操作后的结果值：将x中从第p位开始的n个（二进制）位设置为y中最右边n位的值，x的其余各位保持不变。

```
/* setbits: set n bits of x at position p with bits of y    */
unsigned setbits(unsigned x, int p, int n, unsigned y)
{
      return x & ~(~(~0 << n)  << (p+1-n)) |
            (y &   ~(~0 << n)) << (p+1-n);
}
```

为了把x中的n位设置为y最右边的n位的值（如下所示）

```
xxx...xnnnx...xxx   x
yyy.............ynnn   y
```

我们需要对x中的n位清零；把y中除最右边的n位以外的其他位都清零并左移到第p位处；然后对前面两步的结果值进行OR操作。如下所示：

```
xxx...x000x...xxx   x
000...0nnn0...000   y
--------------------
xxx...xnnnx...xxx   x
```

为了对x中的n位清零，我们需要把x与一个屏蔽码进行AND操作。这个屏蔽码从位置p开始的n位都是0，其他位则全为1。

首先，把一个所有位都为1的屏蔽码左移n位，在它的最右边制造出n位0：

```
~0 << n
```

然后，把屏蔽码最右边的n位设置为1，把其余位全部设置为0：

```
~(~0 << n)
```

接下来，把屏蔽码最右边的n个为1的位左移到第p位处：

```
~(~0 << n) << (p+1-n)
```

再往后，把屏蔽码从第p位开始的n位设置为0，把其余位全部设置为1：

```
~(~(~0 << n) << (p+1-n))
```

用这个屏蔽码和x进行AND操作，就完成了对x从第p位开始的n位清零的工作。具体操作如下：

```
x & ~(~(~0 << n) << (p+1-n))
```

为了把y中除最右端的n位以外的所有位清零，我们需要用最右端的n位全为1，其余位全为0的屏蔽码对y进行AND操作，如下所示：

```
~(~0 << n)
```

用这个屏蔽码和y进行AND操作，我们就选出了y最右端的n位。具体操作如下所示：

```
y & ~(~0 << n)
```

接下来，我们还需要用下面的操作把这n位左移到位置p处：

```
(y & ~(~0 << n)) << (p+1-n)
```

最后，对通过上述步骤得到的两个阶段性结果进行OR操作：

```
 x & ~(~(~0 << n)  << (p+1-n)) |
(y &  ~(~0 << n)) << (p+1-n)
```

就完成了“将x中从第p位开始的n位设置为y中最右边n位的值，x的其余位保持不变”。

练习2-7 （教材第40页）

编写一个函数invert(x, p, n)，该函数返回对x执行下列操作后的结果值：将x中从第p位开始的n个（二进制）位求反（即，1变成0，0变成1），x的其余各位保持不变。

```
/* invert: inverts the n bits of x that begin at position p */
unsigned invert(unsigned x, int p, int n)
{
     return x ^ (~(~0 << n) << (p+1-n));
}
```

首先，把一个所有位都为1的屏蔽码左移n位，在它的最右边制造出n位0：

```
(~0 << n)
```

然后，把屏蔽码最右边的n位设置为1，把其余位全部设置为0：

```
~(~0 << n)
```

接下来，把屏蔽码最右边的n个为1的位左移到第p位处：

```
(~(~0 << n) << (p+1-n))
```

用这个屏蔽码和x进行按位异或（^）操作，就完成了对x从第p位开始的n位进行翻转的工作。

具体操作如下：

```
x ^ (~(~0 << n) << (p+1-n))
```

如果两个二进制位取值不同，它们的异或结果将是1；如果两个二进制位取值相同，它们的异或结果将是0。我们的目标是对x从第p位开始的n位进行翻转，而用从第p位开始具有n位1（其余位为0）的屏蔽码和x进行异或操作恰好能实现这一目标：如果原来的位是0，它与1进行异或后的结果将是1——得到了翻转；如果原来的位是1，它与1进行异或后的结果将是0——也得到了翻转。

这n位以外的其他位将与0进行异或：0^0（两个位相同）的结果是0——保持不变；1^0（两个位不同）的结果是1——也保持不变。总之，只有指定的n位得到了翻转。

练习2-8 （教材第40页）

编写一个函数rightrot(x, n)，该函数返回将x循环右移（即从最右端移出的位将从最左端再移入）n（二进制）位后所得到的值。

```
/* rightrot: rotate x to the right by n positions          */
unsigned rightrot(unsigned x, int n)
{
    int wordlength(void);
    int rbit;                   /* rightmost bit           */

    while (n-- > 0) {
        rbit = (x & 1) << (wordlength() - 1);
        x = x >> 1;             /* shift x 1 position right */
        x = x | rbit;           /* complete one rotation    */
    }
    return x;
}
```

首先，我们利用变量rbit把x最右端的位左移到最左端的位置（wordlength() - 1）。

然后，我们把x右移一位，再对右移后的x与rbit进行OR操作，这就完成了一次对x的循环右移操作。函数rightrot将对x做n次循环右移。

函数wordlength()的作用是计算出运行程序的计算机所使用的字长。

```
/* wordlength: computes word length of the machine         */
int wordlength(void)
{
    int i;
    unsigned v = (unsigned) ~0;

    for (i = 1; (v = v >> 1) > 0; i++)
        ;
    return i;
}
```

下面是这个练习的另一种解法：

```
/* rightrot: rotate x to the right by n positions          */
unsigned rightrot(unsigned x, int n)
{
    int wordlength(void);
```

```
        unsigned rbits;

        if ((n = n % wordlength()) > 0) {
             rbits = ~(~0 << n) & x;  /* n rightmost bits of x       */
                                      /* n rightmost bits to left    */
             rbits = rbits << (wordlength() - n);
             x = x >> n;              /* x shifted n positions right */
             x = x | rbits;           /* rotation completed          */
        }
        return x;
   }
```

如果对x进行循环右移的总位数（n）与一个无符号整数的二进制位数（即这台计算机的字长）相等，完成这些次循环右移后的结果将与x完全一样，因此我们就不必对x进行循环右移了。如果n小于这台计算机的字长，那我们就必须把x循环右移n位。如果n大于这台计算机的字长，那么，我们只需先（利用取模运算符）求出n对这台计算机的字长的余数，再把x循环右移这个余数所代表的次数。基于上述分析，我们将得到一个不需要使用循环语句的解决方案。

`~0 << n`　　把一个全1的屏蔽码左移n位，在它的最右端制造出n位0

`~(~0 << n)`　　把屏蔽码最右端的n位设置为1，其余位则设置为0

当我们用这个屏蔽码和x进行AND运算时，x最右端的n位将被赋值给变量rbits。然后，将rbits中的1左移到它的最左端，把x右移n位，再对右移后的x和rbits进行OR运算，就完成了对无符号整数 x循环右移n位。

练习2-9 （教材第41页）

在求对二的补码时，表达式x &= (x-1)可以删除x中最右边值为1的一个二进制位。请解释这样做的道理。用这一方法重写bitcount函数，以加快其执行速度。

```
/* bitcount: count 1 bits in x - faster version          */
int bitcount(unsigned x)
{
     int b;

     for (b = 0; x != 0; x &= x-1)
          ++b;
     return b;
}
```

我们先随便为x-1选择一个值，比如二进制数1010（即十进制中的10）。我们知道，(x-1) + 1的结果其实就等于x，如下所示：

二进制		十进制
1010	x-1	10
+ 1		+ 1
1011	x	11

我们取(x-1)，对其加1得到x。(x-1)最右端值为0的位在结果x中变为1。也就是说，x最右端为1的位与(x-1)在同一位置上为0的位是对应的。所以，在一个采用对二的补码表示法的系统中，x & (x-1)将清除x最右端值为1的位。

我们以一个4位的无符号数值为例分析一下bitcount函数的编写思路。为了统计出这个数值中值为1的位的个数，原来的bitcount函数需要进行4次移位操作并对最右端的位做4次比较。利用“x & (x-1)将清除x最右端值为1的位”这一结论，我们可以编写出一个执行速度更快的bitcount函数。例如，假设x等于9，于是：

```
  1001  用二进制表示的x——十进制数9
& 1000  用二进制表示的(x-1)——十进制数8
  1000  x & (x-1)
```

可见，x最右端值为1的位被清除为0了。运算结果是二进制数1000（十进制数8）。再以8为x，再重复上述过程，得到：

```
  1000  用二进制表示的x——十进制数8
& 0111  用二进制表示的(x-1)——十进制数7
  0000  x & (x-1)
```

x最右端值为1的位又被清除为0了。运算结果是二进制数0000（十进制数0）。此时，新的x（二进制数0000）中再也没有1的位了，过程也就此结束。

在最坏的情况下——即当x中的所有位都为1时，新方案需要进行的AND操作次数与原bitcount函数需要进行的移位次数一样多。但总的来说，按照新方案编写出来的bitcount函数要执行得快一些。

练习2-10 （教材第42页）

重新编写将大写字母转换为小写字母的函数lower，并用条件表达式替代其中的if-else结构。

```
/* lower: convert c to lower case (ASCII only)                    */
int lower(int c)
{
      return c >= 'A' && c <= 'Z' ?  c + 'a' - 'A' : c;
}
```

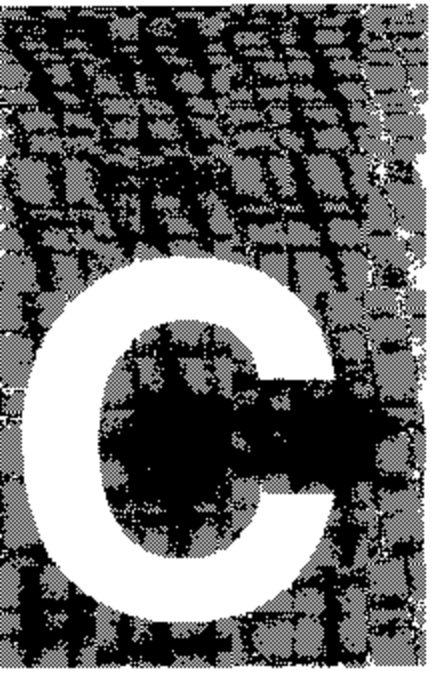
如果c是一个大写字母（这里的讨论只针对ASCII字符），那么条件表达式

```
c >= 'A' && c <= 'Z'
```

就将为真，这个新lower函数就将对下面的表达式求值：

```
c + 'a' - 'A'
```

返回一个小写字母。否则，如果c是一个小写字母，lower函数就会原封不动地返回这个字符。



控 制 流

练习3-1 （教材第47页）

在上面有关折半查找的例子中，while循环语句内共执行了两次测试，其实只要一次就足够（代价是将更多的测试在循环外执行）。重写该函数，使得在循环内部只执行一次测试。比较两种版本函数的运行时间。

```
/* binsearch: find x in v[0] <= v[1] <= . . . <= v[n-1]     */
int binsearch(int x, int v[], int n)
{
     int low, high, mid;

     low = 0;
     high = n - 1;
     mid = (low+high) / 2;
     while (low <= high && x != v[mid]) {
          if (x < v[mid])
               high = mid - 1;
          else
               low = mid + 1;
          mid = (low+high) / 2;
     }
     if (x == v[mid])
          return mid;          /* found match                  */
     else
          return -1;           /* no match                     */
}
```

我们把while语句的循环条件表达式从原先的

```
low <= high
```

修改为

```
low <= high && x != v[mid]
```

这样，while语句的循环部分只用一条if语句即可完成。但这要求我们必须在第一次进入while循环之前以及每次循环当中计算出mid的值。

在while循环结束之后，我们还需要用一条if语句来判断while循环是否是因为在数组v中找到x而终止的。如果在数组里找到了x，函数binsearch将返回mid变量；否则，返回-1。

两种方案的执行时间几乎没有什么差异。我们并没有得到多大的性能改进，反而失掉了代码可读性。教材第48页上原有的代码更容易阅读和理解。

练习3-2 （教材第49页）

编写一个函数escape(s,t)，将字符串t复制到字符串s中，并在复制过程中将换行符、制表符等不可显示字符分别转换为\n、\t等相应的可显示的转义字符序列。要求使用switch语句。再编写一个具有相反功能的函数，在复制过程中将转义字符序列转换为实际字符。

```
/* escape: expand newline and tab into visible sequences     */
/*         while copying the string t to s                   */
void escape(char s[], char t[])
{
     int i, j;

     for (i = j = 0; t[i] != '\0'; i++)
          switch (t[i]) {
          case '\n':                          /* newline       */
               s[j++] = '\\';
               s[j++] = 'n';
               break;
          case '\t':                          /* tab           */
               s[j++] = '\\';
               s[j++] = 't';
               break;
          default:                            /* all other chars  */
               s[j++] = t[i];
               break;
          }
     s[j] = '\0';
}
```

语句

```
for (i = j = 0; t[i] != '\0'; i++)
```

控制着整个循环。变量i是源字符串t的索引，而变量j则是目标字符串s的索引。

switch语句中有3路分支：对应于换行符的'\n'、对应于制表符的'\t'和default。如果字符t[i]与前两种情况不匹配，escape函数就将执行标号为default的部分：把t[i]复制到字符串s中去。

unescape函数与escape函数很相似，如下所示：

```
/* unescape: convert escape sequences into real characters  */
/*           while copying the string t to s                */
void unescape(char s[], char t[])
{
     int i, j;

     for (i = j = 0; t[i] != '\0'; i++)
          if (t[i] != '\\')
               s[j++] = t[i];
          else                              /* it is a backslash*/
               switch(t[++i]) {
               case 'n':                    /* real newline     */
                    s[j++] = '\n';
                    break;
               case 't':                    /* real tab         */
                    s[j++] = '\t';
```

```
                    break;
                default:                        /* all other chars  */
                    s[j++] = '\\';
                    s[j++] = t[i];
                    break;
                }
        s[j] = '\0';
}
```

如果字符t[i]是一个反斜杠，我们就通过一个switch语句把\n转换为换行符，把\t转换为制表符。switch语句的default部分用来处理跟在反斜杠后面的是其他字符的情况——把反斜杠和t[i]复制到字符串s中去。

switch语句是允许嵌套的。下面是此练习的另一种解决方法。

```
/* unescape: convert escape sequences into real characters  */
/*           while copying the string t to s                */
void unescape(char s[], char t[])
{
    int i, j;

    for (i = j = 0; t[i] != '\0'; i++)
        switch (t[i]) {
        case '\\':                          /* backslash        */
            switch(t[++i]) {
            case 'n':                       /* real newline     */
                s[j++] = '\n';
                break;
            case 't':                       /* real tab         */
                s[j++] = '\t';
                break;
            default:                        /* all other chars  */
                s[j++] = '\\';
                s[j++] = t[i];
                break;
            }
            break;
        default:                            /* not a backslash  */
            s[j++] = t[i];
            break;
        }
    s[j] = '\0';
}
```

在上面这段代码中，外层的switch语句负责区分反斜杠和其他字符（default），反斜杠的分支中还嵌套着另一条switch语句。

练习3-3 （教材第52页）

编写函数expand(s1,s2)，将字符串s1中类似于a-z一类的速记符号在字符串s2中扩展为等价的完整列表abc···xyz。该函数可以处理大小写字母和数字，并可以处理a-b-c、a-z0-9与a-z等类似的情况。作为前导和尾随的字符原样复制。

```
/* expand: expand shorthand notation in s1 into string s2   */
void expand(char s1[], char s2[])
```

```
{
      char c;
      int i, j;

      i = j = 0;
      while ((c = s1[i++]) != '\0') /* fetch a char from s1[]*/
           if (s1[i] == '-' && s1[i+1] >= c) {
                i++;
                while (c < s1[i])   /* expand shorthand     */
                     s2[j++] = c++;
           } else
                s2[j++] = c;        /* copy the character    */
      s2[j] = '\0';
}
```

expand函数先把s1中的一个字符读到变量c中，再检查下一个字符。如果下一个字符是-且再下一个字符大于或等于保存在变量c中的字符，expand函数就会对这个速记符号进行扩展。否则，expand函数就将把变量c中的字符直接复制到s2中去。

expand函数只能对ASCII字符进行处理。例如，它将把速记符号a-z扩展为与之等价的列表abc…xyz，把!-~扩展为!"#…ABC…XYZ…abc…xyz…|}~。

这一解决方案是由德国Osnabruck大学的Axel Schreiner提供的。

练习3-4 （教材第53页）

在数的对二的补码表示中，我们编写的itoa函数不能处理最大的负数，即n等于$-(2^{字长-1})$的情况。请解释其原因。修改该函数，使它在任何机器上运行时都能打印出正确的值。

```
#define   abs(x)    ((x) < 0 ?  -(x) : (x))

/* itoa: convert n to characters in s - modified                    */
void itoa(int n, char s[])
{
      int i, sign;
      void reverse(char s[]);

      sign = n;               /* record sign                        */
      i = 0;
      do {                    /* generate digits in reverse order*/
           s[i++] = abs(n % 10) + '0';   /* get next digit     */
      } while ((n /= 10) != 0);          /* delete it          */
      if (sign < 0)
           s[i++] = '-';
      s[i] = '\0';
      reverse(s);
}
```

首先，我们必须了解这样一个事实：

$-(2^{字长-1})$

无法通过语句

$n = -n;$

转换为一个正数。这是因为对二的补码所能表示的最大正数只能是

$(2^{字长-1})-1$

我们解答这道练习的思路是：先按逆序求出各位数字，再用函数reverse（参见练习1-19）对字符串s里的字符做一次颠倒而得到最终的结果。具体地说，在这个itoa函数中，我们用变量sign保存n的初值，用宏abs计算n % 10的绝对值。只需把取模操作的结果转换为正数，这样就能绕过无法把负数

$-(2^{字长-1})$

转换为一个正数的问题。

此外，我们还把do-while语句中的条件表达式从原来的

```
(n /= 10) > 0
```

修改为

```
(n /= 10) != 0
```

这是为了避免因为n是一个负数而使函数itoa陷入无限循环。

练习3-5 （教材第53页）

编写函数itob(n, s, b)，将整数n转换为以b为底的数，并将转换结果以字符的形式保存到字符串s中。例如，itob(n, s, 16)把整数n格式化成十六进制整数保存在s中。

```
/*itob: convert n to characters in s - base b                    */
void itob(int n, char s[], int b)
{
     int i, j, sign;
     void reverse(char s[]);

     if ((sign = n) < 0)                          /* record sign       */
          n = -n;                                 /* make n positive   */
     i = 0;
     do {                    /* generate digits in reverse order      */
          j = n % b;                              /* get next digit    */
          s[i++] = (j <= 9) ? j+'0' : j+'a'-10;
     } while ((n /= b) > 0);                      /* delete it         */
     if (sign < 0)
          s[i++] = '-';
     s[i] = '\0';
     reverse(s);
}
```

我们解答这道练习的思路是：先按逆序生成b进制数的每一位数字，再用函数reverse（参见练习1-19）对字符串s中的字符做一次颠倒而得到最终的结果。因为我们要把整数n转换为一个b进制数，所以我们要用

```
n % b
```

依次返回一个0到b - 1之间的值并把这个值转换为相应的字符保存到字符串s中。然后再用

```
n /= b
```

调整n的值。只要n/b大于零，这一过程就将一直循环下去。

练习3-6 （教材第53页）

修改itoa函数，使得该函数可以接收三个参数。其中，第三个参数为最小字段宽度。为了保证转换后所得的结果至少具有第三个参数指定的最小宽度，在必要时应在所得结果的左边填充一定的空格。

```
#define   abs(x)    ((x) < 0 ?  -(x) : (x))

/* itoa: convert n to characters in s, w characters wide   */
void itoa(int n, char s[], int w)
{
     int i, sign;
     void reverse(char s[]);

     sign = n;              /* record sign                          */
     i = 0;
     do {                   /* generate digits in reverse order*/
          s[i++] = abs(n % 10) + '0';   /* get next digit  */
     } while ((n /= 10) != 0);          /* delete it       */
     if (sign < 0)
          s[i++] = '-';
     while (i < w)                      /* pad with blanks */
          s[i++] = ' ';
     s[i] = '\0';
     reverse(s);
}
```

上面这个函数与练习3-4中的itoa函数很相似。我们对其做了必要的修改：

```
while (i < w)
     s[i++] = ' ';
```

这个while循环的作用是在必要时给字符串s补足空格。

函数与程序结构

练习4-1 （教材第60页）

编写函数strrindex(s,t)，它将返回字符串t在s中最右边出现的位置。如果s中不包含t，则返回－1。

```
/* strrindex: returns rightmost index of t in s, -1 if none */
int strrindex(char s[], char t[])
{
     int i, j, k, pos;

     pos = -1;
     for (i = 0; s[i] != '\0'; i++) {
          for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
               ;
          if (k > 0 && t[k] == '\0')
               pos = i;
     }
     return pos;
}
```

strrindex函数与strindex函数（见教材第59页）很相似。两者的区别是strindex函数只要找到字符串t在字符串s中第一次（最左边）出现的位置就结束了；而strrindex函数在找到字符串t在字符串s中的匹配之后只记录其位置，然后继续搜索，因为它必须返回字符串t在字符串s中最后一次（最右边）出现的位置：

```
if (k > 0 && t[k] == '\0')
     pos = i;
```

下面是本题的另一种解决方法：

```
#include  <string.h>

/* strrindex: returns rightmost index of t in s, -1 if none */
int strrindex(char s[], char t[])
{
     int i, j, k;

     for (i = strlen(s) - strlen(t); i >= 0; i--) {
          for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
               ;
          if (k > 0 && t[k] == '\0')
               return i;
     }
```

```
        return -1;
}
```

就本题而言，第二种方法比前一种方法的执行效率更高。它从字符串s的尾部（最右边）再向串首推进字符串t的长度个字符的位置开始寻找字符串t。如果没有匹配，strrindex函数将（从右向左）后退一个位置并再次进行比较。这样，当strrindex函数在字符串s中找到字符串t时，它将立刻返回变量i，因为变量i此时的取值就是字符串t在字符串s中最右边出现的位置。

练习4-2 （教材第62页）

对atof函数进行扩充，使它可以处理形如

123.45e-6

的科学表示法，其中，浮点数后面可能会紧跟一个e或E以及一个指数（可能有正负号）。

```
#include <ctype.h>

/* atof: convert string s to double */
double atof(char s[])
{
        double val, power;
        int exp, i, sign;

        for (i = 0; isspace(s[i]); i++)     /* skip white space */
                ;
        sign = (s[i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-')
                i++;
        for (val = 0.0; isdigit(s[i]); i++)
                val = 10.0 * val + (s[i] - '0');
        if (s[i] == '.')
                i++;
        for (power = 1.0; isdigit(s[i]); i++) {
                val = 10.0 * val + (s[i] - '0');
                power *= 10.0;
        }
        val = sign * val / power;

        if (s[i] == 'e' || s[i] == 'E') {
                sign = (s[++i] == '-') ? -1 : 1;
                if (s[i] == '+' || s[i] == '-')
                        i++;
                for (exp = 0; isdigit(s[i]); i++)
                        exp = 10 * exp + (s[i] - '0');
                if (sign == 1)
                        while (exp-- > 0)   /* positive exponent     */
                                val *= 10;
                else
                        while (exp-- > 0)   /* negative exponent     */
                                val /= 10;
        }
        return val;
}
```

这个函数的前半部分与atof函数（见教材第61页）完全相同。原atof函数跳过多余的空白符，记录符号位，把字符串转换为相应的数值之后将返回转换结果，但这个扩展版本还必须继续对可能遇到的科学表示法进行处理。

科学表示法中的可选指数部分由新atof函数的后半部分处理。如果浮点数没有可选的指数部分，atof函数将返回保存在变量val中的数值。如果浮点数带有可选的指数部分，指数部分的符号将被保存到变量sign中，而指数部分的值将被计算并保存到变量exp中。

函数最后的操作

```
if (sign == 1)
     while (exp-- > 0)
          val *= 10;
else
     while (exp-- > 0)
          val /= 10;
```

将根据指数部分的值对该浮点数做出调整。如果指数部分是一个正整数，就将该浮点数乘以10的exp次幂；如果指数是一个负整数，就将该浮点数除以10的exp次幂。最终结果将被保存在变量val中返回给atof函数的调用者。

用val去除以10而不是用val去乘以0.1的原因是：0.1无法用二进制数精确地表示出来。在大多数机器上，0.1的二进制表示法都要比0.1稍微小一点，用10.0乘以0.1并不能精确地得到1.0。从结果上看，虽然两种做法都有一定的误差，但连续地“除以10”要比连续地“乘以0.1”更精确。

练习4-3 （教材第67页）

在有了基本框架后，对计算器程序进行扩充就比较简单了。在该程序中加入取模（%）运算符，并注意考虑负数的情况。

```
#include  <stdio.h>
#include  <math.h>          /* for atof()                                  */

#define   MAXOP      100   /* max size of operand or operator */
#define   NUMBER     '0'   /* signal that a number was found  */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator                                               */
main()
{
     int type;
     double op2;
     char s[MAXOP];

     while ((type = getop(s)) != EOF) {
          switch (type) {
          case NUMBER:
               push(atof(s));
               break;
          case '+':
```

```
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '%':
            op2 = pop();
            if (op2 != 0.0)
                push(fmod(pop(), op2));
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
                printf("error: unknown command %s\n", s);
                break;
            }
        }
        return 0;
    }
```

我们对计算器程序的修改集中在它的主程序和函数getop上，不修改函数push和pop（参见教材第65-66页）。

对取模运算符的处理类似于除法运算符（/）。库函数fmod能计算出堆栈顶两个元素的除法余数。op2是栈顶元素。

下面是改进后的getop函数：

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

#define   NUMBER     '0'  /* signal that a number was found  */

int getch(void);
void ungetch(int);

/* getop: get next operator or numeric operand                */
int getop(char s[])
{
    int c, i;
```

```
    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    i = 0;
    if (!isdigit(c) && c != '.' && c != '-')
        return c;                   /* not a number          */
    if (c == '-')
        if (isdigit(c = getch()) || c == '.')
            s[++i] = c;             /* negative number       */
        else {
            if (c != EOF)
                ungetch(c);
            return '-';             /* minus sign            */
        }
    if (isdigit(c))                 /* collect integer part  */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')                   /* collect fraction part */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

getop函数需要检查紧跟在符号 - 后面的那个字符，以判断该符号到底代表一个减号，还是代表一个负号。比如说：

```
- 1 （ - 和1之间有一个空格）
```

是一个减号后面跟一个数字，而

```
-1.23     （ - 和1之间没有空格）
```

则是一个负数。

我们用下面两组运算来验证一下扩展后的计算器能否正确工作：

```
1    -1   +
-10   3   %
```

第一个表达式的运算结果是0: 1 + - 1。第二个表达式的运算结果是 - 1。

练习4-4 （教材第67页）

在栈操作中添加几个命令，分别用于在不弹出元素的情况下打印栈顶元素；复制栈顶元素；交换栈顶两个元素的值。另外增加一个命令用于清空栈。

```
#include  <stdio.h>
#include  <math.h>          /* for atof()                          */

#define   MAXOP     100     /* max size of operand or operator     */
#define   NUMBER    '0'     /* signal that a number was found      */

int getop(char []);
void push(double);
double pop(void);
```

```
void clear(void);

/* reverse Polish calculator                                    */
main()
{
      int type;
      double op1, op2;
      char s[MAXOP];

      while ((type = getop(s)) != EOF) {
            switch (type) {
            case NUMBER:
                  push(atof(s));
                  break;
            case '+':
                  push(pop() + pop());
                  break;
            case '*':
                  push(pop() * pop());
                  break;
            case '-':
                  op2 = pop();
                  push(pop() - op2);
                  break;
            case '/':
                  op2 = pop();
                  if (op2 != 0.0)
                        push(pop() / op2);
                  else
                        printf("error: zero divisor\n");
                  break;
            case '?':      /* print top element of the stack   */
                  op2 = pop();
                  printf("\t%.8g\n", op2);
                  push(op2);
                  break;
            case 'c':      /* clear the stack                  */
                  clear();
                  break;
            case 'd':      /* duplicate top elem. of the stack */
                  op2 = pop();
                  push(op2);
                  push(op2);
                  break;
            case 's':      /* swap the top two elements        */
                  op1 = pop();
                  op2 = pop();
                  push(op1);
                  push(op2);
                  break;
            case '\n':
                  printf("\t%.8g\n", pop());
                  break;
            default:
                  printf("error: unknown command %s\n", s);
```

```
                    break;
                }
            }
            return 0;
    }
```

现有的换行操作符将弹出栈顶元素并打印。我们新增的操作符“?”将弹出栈顶元素并打印，再把它压入堆栈。“?”操作符不会像换行操作符那样把栈顶元素永久地弹出堆栈，而采用“出栈、打印、入栈”这一方案的原因是为了避免主程序直接对堆栈和堆栈指针变量（sp）进行操作。

复制栈顶元素的操作过程是：先弹出栈顶元素，再把它压入两次。

交换栈顶两个元素的操作过程是：先依次弹出两个栈顶元素，再按相反的顺序把它们压入堆栈。

清除堆栈内容的工作很容易完成：把sp设置为零即可。我们增加了一个完成这项工作的函数，并把它与push和pop放置在一起。这样做的目的是只允许这三个函数维护堆栈、访问堆栈及堆栈指针变量。

```
/* clear: clear the stack                                          */
void clear(void)
{
    sp = 0;
}
```

练习4-5 （教材第67页）

给计算器程序增加访问sin、exp与pow等库函数的操作。有关这些库函数的详细信息，请参见附录B.4节（教材第228页）中的头文件<math.h>。

```
#include  <stdio.h>
#include  <string.h>
#include  <math.h>          /* for atof()                            */

#define   MAXOP     100     /* max size of operand or operator */
#define   NUMBER    '0'     /* signal that a number was found  */
#define   NAME      'n'     /* signal that a name was found    */

int getop(char []);
void push(double);
double pop(void);
void mathfnc(char []);

/* reverse Polish calculator                                       */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
```

```
            push(atof(s));
            break;
        case NAME:
            mathfnc(s);
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown commend %s\n", s);
            break;
        }
    }
    return 0;
}

/* mathfnc: check string s for supported meth functions    */
void mathfnc(char s[])
{
    double op2;

    if (strcmp(s, "sin") == 0)
        push(sin(pop()));
    else if (strcmp(s, "cos") == 0)
        push(cos(pop()));
    else if (strcmp(s, "exp") == 0)
        push(exp(pop()));
    else if (strcmp(s, "pow") == 0) {
        op2 = pop();
        push(pow(pop(), op2));
    } else
        printf("error: %s not supported\n", s);
}
```

修改后的getop函数的源代码如下所示：

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

```
#define   NUMBER    '0'  /* signal that a number was found  */
#define   NAME      'n'  /* signal that a name was found    */

int getch(void);
void ungetch(int);
/* getop: get next operator, numeric operand, or math fnc   */
int getop(char s[])
{
     int c, i;

     while ((s[0] = c = getch()) == ' ' || c == '\t')
          ;
     s[1] = '\0';
     i = 0;
     if (islower(c)) {                  /* command or NAME       */
          while (islower(s[++i] = c = getch()))
               ;
          s[i] = '\0';
          if (c != EOF)
               ungetch(c);              /* went one char too far */
          if (strlen(s) > 1)
               return NAME;             /* >1 char; it is NAME   */
          else
               return c;                /* it may be a command   */
     }
     if (!isdigit(c) && c != '.')
          return c;                     /* not a number          */
          if (isdigit(c))               /* collect integer part  */
               while (isdigit(s[++i] = c = getch()))
                    ;
          if (c == '.')                 /* collect fraction part */
               while (isdigit(s[++i] = c = getch()))
                    ;
          s[i] = '\0';
          if (c != EOF)
               ungetch(c);
          return NUMBER;
}
```

改进后的getop函数能够识别出一个由小写字母组成的字符串并把它返回为类型NAME。而主程序将把NAME识别为一个合法的类型并调用函数mathfnc。

函数mathfnc是新增加的。它将通过一系列if语句来寻找与字符串s匹配的函数名，如果没找到匹配，它将报告出错。如果字符串s是一个程序支持的数学函数名，mathfnc函数将从堆栈弹出足够的元素并调用相应的数学函数，然后再把那个数学函数的返回值压入堆栈。

例如，正弦函数sin的输入参数应该是一个弧度值，而π/2的正弦值是1。于是，表达式

```
3.14159265 2 / sin
```

将先把π除以2并压入堆栈，然后再由数学函数sin弹出栈顶元素，计算出它的正弦值，再把该数值压入堆栈。最终的计算结果将是1。而表达式

```
3.141592 2 / sin 0 cos +
```

的计算结果将是2——因为π/2的正弦是1，0的余弦也是1，两者相加等于2。

再来看一个例子，表达式

```
5 2 pow 4 2 pow +
```

将先依次计算出5的2次方和4的2次方，然后再把这两个结果加起来。

getop函数并不知道具体将会调用哪个数学函数，它只负责识别并返回它找到的字符串。这种做法的好处是可以很容易地在mathfnc函数中添加其他的数学函数。

练习4-6 （教材第67页）

给计算器程序增加处理变量的命令（提供26个具有单个英文字母变量名的变量很容易）。增加一个变量存放最近打印的值。

```
#include  <stdio.h>
#include  <math.h>        /* for atof()                                    */

#define   MAXOP     100   /* max size of operand or operator */
#define   NUMBER    '0'   /* signal that a number was found  */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator                                               */
main()
{
      int i, type, var = 0;
      double op2, v;
      char s[MAXOP];
      double variable[26];

for (i = 0; i < 26; i++)
      variable[i] = 0.0;
while ((type = getop(s)) !=EOF) {
      switch (type) {
      case NUMBER:
            push(atof(s));
            break;
      case '+':
            push(pop() + pop());
            break;
      case '*':
            push(pop() * pop());
            break;
      case '-':
            op2 = pop();
            push(pop() - op2);
            break;
      case '/':
            op2 = pop();
            if (op2 != 0.0)
                  push(pop() / op2);
            else
                  printf("error: zero divisor\n");
            break;
```

```
            case '=':
                pop();
                if (var >= 'A' && var <= 'Z')
                    variable[var - 'A'] = pop();
                else
                    printf("error: no variable name\n");
                break;
            case '\n':
                v = pop();
                printf("\t%.8g\n", v);
                break;
            default:
                if (type >= 'A' && type <= 'Z')
                    push(variable[type - 'A']);
                else if (type == 'v')
                    push(v);
                else
                    printf("error: unknown command %s\n", s);
                break;
            }
            var = type;
        }
        return 0;
}
```

我们用大写字母A~Z来代表变量，这些字母作为数组变量的索引。我们再增加一个小写字母变量v存放打印的值。

在遇到一个变量名（即字母A~Z或v）的时候，计算器程序将把该变量的值压入堆栈。我们还新增加了一个操作符“=”，它的作用是把堆栈中的某个元素赋值给这个“=”前面的字母变量。例如：

```
3 A =
```

将把3赋值给变量A。此后，表达式

```
2 A +
```

将把2和3（被赋值给变量A的数值）相加。计算器的换行操作符将输出数值5，同时再把这个5赋值给变量v。如果下一个操作是

```
v 1 +
```

则结果将是6——即5 + 1。

练习4-7 （教材第67页）

编写一个函数ungets(s)，将整个字符串s压回到输入中。ungets函数需要使用buf和bufp吗？它能否仅使用ungetch函数？

```
#include  <string.h>

/* ungets: push string back onto the input                    */
void ungets(char s[])
{
    int len = strlen(s);
    void ungetch(int);
```

```
        while (len > 0)
                ungetch(s[--len]);
}
```

变量len中保存着字符串s的字符个数（但不包括最末尾的'\0'字符），这个数值是由库函数strlen（参见教材第31页）确定的。

函数ungets将调用函数ungetch（参见教材第67页）len次，每次都会把字符串s中的一个字符重新压回输入。ungets函数将按逆序把字符串重新压回输入。

ungets函数不需要直接对buf和bufp进行操作，buf、bufp和出错检查将由函数ungetch处理。

练习4-8 （教材第67页）

假定最多只压回一个字符，请相应地修改getch和ungetch这两个函数。

```
#include  <stdio.h>

char buf = 0;

/* getch: get a (possibly pushed back) character                    */
int getch(void)
{
        int c;

        if (buf != 0)
                c = buf;
        else
                c = getchar();
        buf = 0;
        return c;
}

/* ungetch: push character back onto the input                      */
void ungetch(int c)
{
        if (buf != 0)
                printf("ungetch: too many characters\n");
        else
                buf = c;
}
```

根据题目要求，输入缓冲区buf将不再是一个数组，因为任意时刻该缓冲区最多保存一个字符。

buf的初值是0，而每次getch函数在读取一个字符之后，会再次把buf设置为0。ungetch函数在把一个字符重新压回输入缓冲区之前要检查该缓冲区是否为空。如果缓冲区不为空，ungetch函数将产生一条出错信息。

练习4-9 （教材第68页）

以上介绍的getch和ungetch函数不能正确地处理压回的EOF。考虑压回EOF时应该如何处理？请实现你的设计方案。

```
#include  <stdio.h>

#define   BUFSIZE   100

int buf[BUFSIZE];          /* buffer for ungetch                  */
int bufp = 0;              /* next free position in buf           */

/* getch: get a (possibly pushed back) character                  */
int getch(void)
{
     return (bufp > 0) ? buf[--bufp] : getchar();
}

/* ungetch: push character back onto the input                    */
void ungetch(int c)
{
     if (bufp >= BUFSIZE)
          printf("ungetch: too many characters\n");
     else
          buf[bufp++] = c;
}
```

在原先的getch和ungetch函数（参见教材第67页）中，输入缓冲区buf被声明为一个字符数组：

```
char buf[BUFSIZE];
```

C语言不要求char变量是signed或unsigned类型的（参见教材第34页）；当把一个char类型变量转换为int类型时，转换结果不应该是一个负值。在某些机器上，如果一个char类型变量的最高（左）二进制位是1，那么在把它转换为一个整数的时候，就可能得到一个负数的结果。在另外一些机器上，当需要把一个char类型变量转换为一个整数的时候，系统会在它的最高（左）二进制位处添加一些0——不管将被转换的char类型变量的最高位是1还是0，这种转换的结果将永远是一个正数。

我们知道，十进制的－1将被表示为十六进制的0xFFFF（假设这是一台16位的系统）。当把0xFFFF保存到一个char类型变量去时，实际被保存的数字将是0xFF。当把0xFF转换为一个整数的时候，它可能被转换为0x00FF（十进制的255），也可能被转换为0xFFFF（十进制的－1）。整个过程如下所示：

负数（－1）	→	字符	→	整数
0xFFFF		0xFF		0x00FF（255）
0xFFFF		0xFF		0xFFFF（－1）

如果打算像对待其他字符那样对待EOF（－1），就应该把输入缓冲区buf声明为一个整数数组：

```
int buf[BUFSIZE];
```

这样，就不需要上面提到的那些转换，而EOF（－1）或其他任何负数也都能得到正确的处理。

练习4-10 （教材第68页）

另一种方法是通过getline函数读入整个输入行，这种情况下可以不使用getch与ungetch函数。请运用这一方法修改计算器程序。

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

#define   MAXLINE   100
#define   NUMBER    '0'  /* signal that a number was found  */

int getline(char line[], int limit);

int li = 0;               /* input line index                */
char line [MAXLINE];      /* one input line                  */

/* getop: get next operator or numeric operand               */
int getop(char s[])
{
     int c, i;

     if (line[li] == '\0')
          if (getline(line, MAXLINE) == 0)
               return EOF;
          else
               li = 0;
     while ((s[0] = c = line[li++]) == ' ' || c == '\t')
          ;
     s[1] = '\0';
     if (!isdigit(c) && c != '.')
          return c;                    /* not a number       */
     i = 0;
     if (isdigit(c))                   /* collect integer part  */
          while (isdigit(s[++i] = c = line[li++]))
               ;
     if (c == '.')                     /* collect fraction part */
          while (isdigit(s[++i] = c = line[li++]))
               ;
     s[i] = '\0';
     li--;
     return NUMBER;
}
```

我们将在getop函数中用getline来取代getch和ungetch函数。数组line保存着每次读入的一整行输入行；变量li则是数组line中下一个字符的索引。我们把line和li设置为外部变量而不是局部变量，这样，它们的取值就由不同的调用共同维护。

当getop到达输入行的末尾（或者尚未读入第一个输入行）时，语句

```
if (line[li] == '\0')
```

中的条件表达式将为真，于是getop就将调用getline读入一个输入行。

在原来的getop函数（参见教材第66页）中，getop需要调用getch读入一个字符。在上面这个改进版本中，我们将利用变量li来读取line中的一个字符，然后再对li做递增操作。在getop函数的末尾，我们将不再需要调用ungetch把一个字符重新压回输入缓冲区，只需对变量li减1就能达到后退一个字符的目的。

需要提醒大家的是：在C语言中，任何函数都能使用和改变另外一个函数的外部变量。这就意味着li和line也有被getop以外的其他函数改变的可能。如果想阻止这类情况发生，就

应该把外部变量声明为static变量。但关于static变量的介绍要等到教材第71页才出现，所以我们现在就没有这样做。

练习4-11 （教材第71页）

修改getop函数，使其不必使用ungetch函数。提示：可以使用一个static类型的内部变量解决该问题。

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

#define   NUMBER    '0'  /* signal that a number was found  */

int getch(void);

/* getop: get next operator or numeric operand                */
int getop(char s[])
{
     int c, i;
     static int lastc = 0;

     if (lastc == 0)
          c = getch();
     else {
          c = lastc;
          lastc = 0;
     }
     while ((s[0] = c) == ' ' || c == '\t')
          c = getch();
     s[1] = '\0';
     if (!isdigit(c) && c != '.')
          return c;                    /* not a number          */
     i = 0;
     if (isdigit(c))                   /* collect integer part  */
          while (isdigit(s[++i] = c = getch()))
               ;
     if (c == '.')                     /* collect fraction part */
          while (isdigit(s[++i] = c = getch()))
               ;
     s[i] = '\0';
     if (c != EOF)
          lastc = c;
     return NUMBER;
}
```

我们给函数getop增加了一个内部static变量来存放那个应该被重新压回输入缓冲区的字符（即最后一个被读入的字符）。既然题目要求我们不使用ungetch函数，我们就把那个字符保存到变量lastc中。

当getop函数被调用时，它首先要检查变量lastc中是否保存有前一个字符。如果没有，getop将调用getch读取一个新字符；如果有，getop就将把这个字符复制到变量c并对变量lastc清零。此外，因为getop只有在检查完当前字符c之后才需要读取一个新字符，所以我们对第一个while语句也做了一些改动。

练习4-12 （教材第75页）

运用printd函数的设计思想编写一个递归版本的itoa函数。即通过递归调用把整数转换为字符串。

```
#include  <math.h>

/* itoa: convert n to characters in s; recursive                    */
void itoa(int n, char s[])
{
     static int i;

     if (n / 10)
          itoa(n / 10, s);
     else {
          i = 0;
          if (n < 0)
               s[i++] = '-';
     }
     s[i++] = abs(n) % 10 + '0';
     s[i]   = '\0';
}
```

itoa函数需要用到两个参数：一个是将被转换为字符串的整数n，一个是用来保存转换结果的字符数组s。如果整数除法n/10的结果不为零，itoa就将以整数n/10为参数调用自身，如下所示：

```
if (n / 10)
  itoa(n / 10, s);
```

如果n/10在某次递归调用中结果为零，我们就得到了整数n的最高（左）位数字。我们用一个static变量i作为字符数组s的索引。如果n是一个负数，我们就在字符数组的第一个位置放上一个负号并对变量i进行递增。因为这个itoa函数是递归性的，所以它将按从左至右的顺序计算出整数n的每一位数字。需要注意的是：每次递归调用都将用一个“\0”字符来结束字符数组s，但下一次递归调用——除最后一次外——将覆盖掉这个“\0”字符。

练习4-13 （教材第75页）

编写一个递归版本的reverse(s)函数，以将字符串s倒置。

```
#include  <string.h>

/* reverse: reverse string s in place                               */
void reverse(char s[])
{
     void reverser(char s[], int i, int len);

     reverser(s, 0, strlen(s));
}

/* reverser: reverse string s in place; recursive                   */
void reverser(char s[], int i, int len)
{
```

```
    int c, j;

    j = len - (i + 1);
    if (i < j) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
        reverser(s, ++i, len);
    }
}
```

无论在何种实现中，我们都应保持相同的reverse函数的用户接口。因此，reverse(s)函数的新版本仍将只有一个参数：字符串s。

reverse函数先计算出字符串的长度，然后调用函数reverser原地倒置字符串s。

函数reverser有3个参数：将被倒置的字符串s、字符串从左端算起的索引i以及字符串的长度len（由strlen(s)确定，参见教材第31页）。

i的初值是0。变量j中是字符串从右端算起的索引，它通过以下语句计算得出：

```
j = len - (i + 1);
```

字符串中的字符是按由外而内的顺序依次进行位置交换的。例如，第一组交换位置的两个字符是s[0]和s[len-1]，第二组交换位置的两个字符是s[2]和s[len-2]。每调用一次reverser，字符串从左端算起的索引i就加1，如下所示：

```
reverser(s, ++i, len);
```

这一位置交换过程将一直重复到索引i、j指向同一个字符（i == j）或者索引i所指向的字符出现在索引j所指向的字符的右侧（i > j）为止。

本题并不适合用递归方法来解决。有些问题特别适合用递归技术来解决——比如教材第124页上的treeprint函数。但也有些问题用非递归技术解决会更好，本题就是一个很明显的例子。

练习4-14 （教材第77页）

定义宏swap(t, x, y)，以交换t类型的两个参数。（使用程序块结构会对你有所帮助。）

```
#define   swap(t, x, y)  {   t _z;    \
                             _z = y; \
                             y = x;  \
                             x = _z; }
```

我们利用花括号定义了一个新的程序块。程序块允许我们在它的开头部分对在该语句块内使用的局部变量做出声明。为了交换两个参数的值，我们声明了一个类型为t的局部变量_z。

上面这个swap宏只有在两个参数名都不是_z的前提下才能工作。如果两个参数名之一是_z，如：

```
swap(int, _z, x);
```

那么，这个宏在展开时就会成为

```
{ int _z; _z = _z; _z = x; x = _z; }
```

从而导致两个参数的值无法交换。所以，我们写的这个swap宏是以_z不会被用作变量名为假设前提的。

指针与数组

练习5-1 （教材第83页）

在上面的例子中，如果符号+或－的后面紧跟的不是数字，getint函数将把符号视为数字0的有效表达式。修改该函数，将这种形式的+或-符号重新压回输入缓冲区。

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

int getch(void);
void ungetch(int);

/* getint: get next integer from input into *pn                    */
int getint(int *pn)
{
     int c, d, sign;

     while (isspace(c = getch()))  /* skip white space       */
          ;
     if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
          ungetch(c);              /* it's not a number      */
          return 0;
     }
     sign = (c == '-') ?  -1 : 1;
     if (c == '+' || c == '-') {
          d = c;                   /* remember sign char     */
          if(!isdigit(c = getch())) {
               if (c != EOF)
                    ungetch(c);    /* push back non-digit    */
               ungetch(d);         /* push back sign char    */
               return d;           /* return sign char       */
          }
     }
     for (*pn = 0; isdigit(c); c = getch())
          *pn = 10 * *pn + (c - '0');
     *pn *= sign;
     if (c != EOF)
          ungetch(c);
     return c;
}
```

在遇到符号字符时，修改后的getint函数会把它保存到变量d中并读入下一个字符。如果下一个字符既不是一个数字也不是EOF标记，getint函数就会把新读入的这个字符压回输入缓冲区。然后，函数getint会把保存在变量d中的符号压回输入缓冲区，并返回这个符号

以表明这种情况。

练习5-2 （教材第83页）

模仿函数getint的实现方法，编写一个读取浮点数的getfloat函数。getfloat函数的返回值应该是什么类型？

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

int getch(void);
void ungetch(int);

/* getfloat: get next floating-point number from input      */
int getfloat(float *pn)
{
     int c, sign;
     float power;

     while (isspace(c = getch()))       /* skip white space */
          ;
     if (!isdigit(c) && c != EOF && c != '+' &&
          c != '-' && c != '.') {
          ungetch(c);                   /* it's not a number*/
          return 0;
     }
     sign = (c == '-') ?  -1 : 1;
     if (c == '+' || c == '-')
          c = getch();
     for (*pn = 0.0; isdigit(c); c = getch())
          *pn = 10.0 * *pn + (c - '0'); /* integer part     */
     if (c == '.')
          c = getch();
     for (power = 1.0; isdigit(c); c = getch()) {
          *pn = 10.0 * *pn + (c - '0'); /* fractional part  */
          power *= 10.0;
     }
     *pn *= sign / power;               /* final number     */
     if (c != EOF)
          ungetch(c);
     return c;
}
```

函数getfloat与函数getint（见教材第67页）很相似。getfloat跳过空白符，记录浮点数的符号，再把浮点数的整数部分保存到指针pn指向的地址中。

getfloat还对浮点数的小数部分进行处理（但不涉及科学计数法）。小数部分也将按类似于整数部分的处理方式被添加到*pn上去，如下所示：

```
*pn = 10.0 * *pn + (c - '0');
```

每收集到一个出现在小数点后面的数字，变量power就会被乘以10。例如，如果小数点后面没有数字，变量power就将为1；如果小数点后面有一个数字，变量power就将为10；如果小数点后面有三个数字，变量power就将为1000。

接下来，getfloat把*pn的最终结果乘以sign/power以得到读入的浮点数值。

getfloat函数将返回EOF或者紧跟在浮点数后面的那个字符的ASCII值，因此，这个函数的类型应该是int。

练习5-3 （教材第92页）

用指针方式实现第2章中的函数strcat。函数strcat(s, t)将t指向的字符串复制到s指向的字符串的尾部。

```
/* strcat: concatenate t to the end of s; pointer version  */
void strcat(char *s, char *t)
{
      while (*s)
            s++;
      while (*s++ = *t++)
            ;
}
```

指针s和t的初值分别指向各字符串的开始位置。

第一个while循环将递增指针s，直到它找到字符串结束符（'\0'）为止。在找到字符串结束符之前，语句：

```
while (*s)
```

将一直为真。

第二个while循环负责把字符串t追加到字符串s的末尾。如下所示：

```
while (*s++ = *t++)
      ;
```

上面这条语句的作用是把*t赋值给*s，再递增这两个指针；这一过程将一直循环到指针t指向字符串结束符为止。

练习5-4 （教材第92页）

编写函数strend(s, t)。如果字符串t出现在字符串s的尾部，该函数返回1；否则返回0。

```
/* strend: return 1 if string t occurs at the end of s      */
int strend(char *s, char *t)
{
      char *bs = s;              /* remember beginning of strs */
      char *bt = t;

      for ( ; *s; s++)           /* end of the string s        */
            ;
      for ( ; *t; t++)           /* end of the string t        */
            ;
      for ( ; *s == *t; s--, t--)
            if (t == bt || s == bs)
                  break;         /* at the beginning of a str  */
      if (*s == *t && t == bt && *s != '\0')
            return 1;
      else
            return 0;
}
```

我们把两个字符串的起始地址分别保存在指针bs和bt中，然后像上一练习中的strcat函数那样把它们分别移动到两个字符串的末尾。为了判断字符串t是否出现在字符串s的末尾，我们需要把字符串s的最后一个字符与字符串t的最后一个字符进行比较，然后再向字符串的头部移动。

重复上述过程，如果在字符串s中字符与字符串t中的字符相匹配的同时指针t退回到了字符串t的开始且两个字符串都不是空字符串，strend函数就将返回1（即字符串t出现在了字符串s的末尾），如下所示：

```
if (*s == *t && t == bt && *s != '\0')
     return 1;
```

练习5-5 （教材第92页）

实现库函数strncpy、strncat和strncmp它们最多对参数字符串中的前n个字符进行操作。例如，函数strncpy(s, t, n)将t中最多前n个字符复制到s中。更详细的说明请参见附录B（见教材第227页）。

```
/* strncpy: copy n characters from t to s                    */
void strncpy(char *s, char *t, int n)
{
     while (*t && n-- > 0)
          *s++ = *t++;
     while (n-- > 0)
          *s++ = '\0';
}

/* strncat: concatenate n characters of t to the end of s    */
void strncat(char *s, char *t, int n)
{
     void strncpy(char *s, char *t, int n);
     int strlen(char *);

     strncpy(s+strlen(s), t, n);
}

/* strncmp: compare at most n characters of t with s         */
int strncmp(char *s, char *t, int n)
{
     for ( ; *s == *t; s++, t++)
          if (*s == '\0' || --n <= 0)
               return 0;
     return *s - *t;
}
```

函数strncpy和strncat的类型都是void，与教材第91页上的strcpy函数一样。这几个函数的库函数版本返回的都是一个指向结果字符串开头的指针。

strncpy函数从字符串t最多复制n个字符到字符串s。如果t中的字符少于n个，我们就将在字符串s的末尾填充“\0”字符。

strncat函数将调用strncpy函数把字符串t的前n个字符追加到字符串s的末尾。

strncmp将对字符串t和s的前n个字符进行比较。strncmp与教材第92页上的strcmp

函数很相似，只是这次我们将在到达字符串s或t的末尾或者成功地比较了n个字符后终止比较操作，如下所示：

```
if (*s == '\0' || --n <= 0)
    return 0;
```

练习5-6 （教材第92页）

采用指针而非数组索引方式改写前面章节和练习中的某些程序，例如getline（第1章、第4章），atoi以及它们的变体形式（第2章、第3章、第4章），reverse（第3章），strindex、getop（第4章）等等。

```
#include  <stdio.h>

/* getline: read a line into s, return length                    */
int getline(char *s, int lim)
{
    int c;
    char *t = s;

    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        *s++ = c;
    if (c == '\n')
        *s++ = c;
    *s = '\0';
    return s - t;
}
```

函数getline的输入参数将是一个指向字符数组的指针。这意味着我们要用*s代替s[i]来引用字符数组中的元素，还要用递增指针s的办法来遍历那个字符数组。

语句s[i++] = c;等价于*s++ = c;。

在函数刚开始的时候，指针s将指向字符数组的第一个字符，这个地址将被保存在指针t中：

```
char *t = s;
```

在函数getline读入一个输入行以后，指针s将指向字符串结束符（'\0'），而指针t将指向输入行的第一个字符，所以输入行的长度就将是s - t。

```
#include  <ctype.h>

/* atoi: convert s to integer; pointer version                   */
int atoi(char *s)
{
    int n, sign;

    for ( ; isspace(*s); s++)          /* skip white space */
        ;
    sign = (*s == '-') ?  -1 : 1;
    if (*s == '+' || *s == '-')        /* skip sign        */
        s++;
    for (n = 0; isdigit(*s); s++)
        n = 10 * n + *s - '0';
```

```
    return sign * n;
}
```

s[i]等价于*s, s[i++]等价于*s++。

```
void reverse(char *);

/* itoa: convert n to characters in s; pointer version      */
void itoa(int n, char *s)
{
    int sign;
    char *t = s;                    /* save pointer to s     */

    if ((sign = n) < 0)             /* record sign           */
        n = -n;                     /* make n positive       */
    do {                    /* generate digits in reverse order*/
        *s++ = n % 10 + '0';        /* get next digit        */
    } while ((n /= 10) > 0);        /* delete it             */
    if (sign < 0)
        *s++ = '-';
    *s = '\0';
    reverse(t);
}
```

字符指针t被初始化为指向字符串s的第一个元素：

```
char *t = s;
```

语句*s++ = n % 10 +'0';等价于s[i++] = n % 10 +'0';。

在调用reverse函数时，它的输入参数将是一个指向字符串s第一个字符的指针。

```
#include  <string.h>

/* reverse: reverse string s in place                        */
void reverse(char *s)
{
    int c;
    char *t;

    for (t = s + (strlen(s)-1); s < t; s++, t--) {
        c = *s;
        *s = *t;
        *t = c;
    }
}
```

指针s的初值指向字符串的第一个字符；字符指针t的初值指向字符串的最后一个字符（不包括字符串结束符'\0'），如下所示：

```
t = s + (strlen(s)-1)
```

*s等价于s[i]，*t等价于s[j]。指针版本中for循环的条件表达式s<t相当于索引版本中的条件表达式i<j。

s++与索引i的递增操作（i++）作用相同，t--与索引j的递减操作（j--）作用相同。

```
/* strindex: return index of t in s, -1 if none                    */
int strindex(char *s, char *t)
{
     char *b = s;                        /* beginning of string s */
     char *p, *r;

     for (; *s != '\0'; s++) {
          for (p=s, r=t; *r != '\0' && *p == *r; p++, r++)
               ;
          if (r > t && *r == '\0')
               return s - b;
     }
     return -1;
}
```

s[i]被替换为*s，s[j]被替换为*p，t[k]被替换为*r。b是一个字符指针，它永远指向字符串s的第一个元素（s[0]）。p = s等价于j = i，r = t等价于k = 0。

当if语句if(r > t && *r == '\0')的条件表达式为真时，就找到了一个匹配，strindex函数就将通过return s - b;语句返回字符串t在字符串s中的位置的索引。

```
#include <ctype.h>

/* atof: convert string s to double; pointer version      */
double atof(char *s)
{
     double val, power;
     int sign;

     for ( ; isspace(*s); s++)      /* skip white space    */
          ;
     sign = (*s == '-') ? -1 : 1;
     if (*s == '+' || *s == '-')
          s++;
     for (val = 0.0; isdigit(*s); s++)
          val = 10.0 * val + (*s - '0');
     if (*s == '.')
          s++;
     for (power = 1.0; isdigit(*s); s++) {
          val = 10.0 * val + (*s - '0');
          power *= 10.0;
     }
     return sign * val / power;
}
```

*s++等价于s[i++]。

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

#define   NUMBER    '0'  /* signal that a number was found  */

int getch(void);
void ungetch(int);

/* getop: get next operator or numeric operand; pointer ver */
int getop(char *s)
```

```
{
      int c;

      while ((*s = c = getch()) == ' ' || c == '\t')
            ;
      *(s+1) = '\0';
      if (!isdigit(c) && c != '.')
            return c;                  /* not a number           */
      if (isdigit(c))                  /* collect integer part   */
            while (isdigit(*++s = c = getch()))
                  ;
      if (c == '.')                    /* collect fraction part */
            while (isdigit(*++s = c = getch()))
                  ;
      *s = '\0';
      if (c != EOF)
            ungetch(c);
      return NUMBER;
}
```

数组元素将通过有关指针而不是它的索引来引用。几处修改都很容易看懂。比如说，在索引版本中，我们通过语句s[1] = '\0';在字符数组的第二个位置放上一个字符串结束符；在指针版本中，我们通过语句*(s+1) = '\0';不改变指针取值实现了同样的操作。

练习5-7 （教材第95页）

重写函数readlines，将输入的文本行存储到由main函数提供的一个数组中，而不是存储到调用alloc分配的存储空间中。该函数的运行速度比改写前快多少？

```
#include  <string.h>

#define   MAXLEN    1000 /* maximum length of line          */
#define   MAXSTOR   5000 /* size of available storage space */

int getline(char *, int);

/* readlines: read input lines                                */
int readlines(char *lineptr[], char *linestor, int maxlines)
{
      int len, nlines;
      char line[MAXLEN];
      char *p = linestor;
      char *linestop = linestor + MAXSTOR;

      nlines = 0;
      while ((len = getline(line, MAXLEN)) > 0)
            if (nlines >= maxlines || p+len > linestop)
                  return -1;
            else {
                  line[len-1] = '\0';      /* delete newline   */
                  strcpy(p, line);
                  lineptr[nlines++] = p;
                  p += len;
            }
```

```
        return nlines;
}
```

readlines将把读取的输入行保存在主函数提供的数组linestor中。字符指针p的初值将指向linestor的第一个元素，如下所示：

```
char *p = linestor;
```

原来的readlines函数（见教材第94页）使用了alloc函数（见教材第87页），如下所示：

```
if ((nlines >= maxlines || (p = alloc(len)) == NULL)
```

而这里的readlines函数将把line保存到linestor从位置p开始的地方，语句

```
if (nlines >= maxlines || p+len > linestop)
```

确保linestor中有足够的可用空间。

这个版本的readlines函数要比它的原始版本执行得稍快一些。

练习5-8 （教材第97页）

函数day_of_year和month_day中没有进行错误检查，请解决该问题。

```
static char daytab[2][13] = {
     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
     {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
     int i, leap;

     leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
     if (month < 1 || month > 12)
          return -1;
     if (day < 1 || day > daytab[leap][month])
          return -1;
     for (i = 1; i < month; i++)
          day += daytab[leap][i];
     return day;
}

/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
     int i, leap;

     if (year < 1) {
          *pmonth = -1;
          *pday = -1;
          return;
     }
     leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
     for (i = 1; i <= 12 && yearday > daytab[leap][i]; i++)
          yearday -= daytab[leap][i];
```

```
    if (i > 12 && yearday > daytab[leap][12]) {
        *pmonth = -1;
        *pday = -1;
    } else {
        *pmonth = i;
        *pday = yearday;
    }
}
```

在day_of_year函数中，我们首先要确定变量month和day的取值都是合理的：如果变量month的取值小于0或大于12，函数将返回－1；如果变量day的取值小于1或大于该月份的实际天数，函数也将返回－1。

在month_day函数中，我们必须保证变量year不是一个负数；也许你还打算给day_of_year函数也加上这一检查。接下来，在代表月份的索引i没有超过12的时候，我们将对变量yearday进行递减。当循环结束时，如果i=13且yearday大于该年度最后月份的天数，就说明yearday的初值有错，我们就将把相应的月份和天数都设置为－1以指明这种情况；否则，就说明month_day函数的输入参数是正确的。

练习5-9 （教材第98页）

用指针方式代替数组下标方式，改写函数day_of_year和month_day。

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int leap;
    char *p;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    p = daytab[leap];
    while (--month)
        day += *++p;
    return day;
}

/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int leap;
    char *p;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    p = daytab[leap];
    while (yearday > *++p)
        yearday -= *p;
    *pmonth = p - *(daytab + leap);
    *pday = yearday;
}
```

根据变量leap的取值情况，字符指针p将指向daytab的第一行或第二行（即正常年度或闰年）。

```
p = daytab[leap];
```

我们把原day_of_year函数中如下所示的for循环

```
for (i = 1; i < month; i++)
     day += daytab[leap][i];
```

在新day_of_year函数中替换为如下所示的while循环：

```
while (--month)
     day += *++p;
```

把原month_day函数中如下所示的for循环

```
for (i = 1; yearday > daytab[leap][i]; i++)
     yearday -= daytab[leap][i];
```

在新month_day函数中替换为如下所示的while循环：

```
p = daytab[leap];
while (yearday > *++p)
     yearday -= *p;
```

练习5-10 （教材第102页）

编写程序expr，以计算从命令行输入的逆波兰表达式的值，其中每个运算符或操作数用一个单独的参数表示。例如，命令

```
expr 2 3 4 + *
```

将计算表达式2×(3+4)的值。

```
#include <stdio.h>
#include <math.h>          /* for atof()                          */

#define   MAXOP     100    /* max size of operand or operator */
#define   NUMBER    '0'    /* signal that a number was found  */

int getop(char []);
void ungets(char []);
void push(double);
double pop(void);

/* reverse Polish calculator; uses command line                   */
main(int argc, char *argv[])
{
     char s[MAXOP];
     double op2;

     while (--argc > 0) {
          ungets(" ");                  /* push end of argument  */
          ungets(*++argv);              /* push an argument      */
          switch (getop(s)) {
          case NUMBER:
               push(atof(s));
```

```
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if (op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;
            default:
                printf("error: unknown command %s\n", s);
                argc = 1;
                break;
            }
        }
        printf("\t%.8g\n", pop());
        return 0;
}
```

这里给出的解决方案是在教材第65页上的逆波兰计算器的基础得到的。它使用了push和pop函数（见教材第66页）。

我们先利用ungets函数把一个参数结束标记（' '，一个空格字符）和一个参数依次压入输入缓冲区。这样，我们就可以不加修改地使用getop函数了。getop将调用getch读取字符并分离出下一个运算符或操作数。

如果在读取参数的过程中遇到了错误，argc将被设置为1；主函数中的while循环

```
while (--argc > 0)
```

就将因其条件表达式的求值结果为“假”而使程序终止运行。

如果来自命令行的是一个合法的表达式，它的计算结果就将被放在堆栈的最顶部，这个结果将在我们把输入参数全部处理完毕后被打印。

练习5-11 （教材第102页）

修改程序entab和detab（第1章练习中编写的函数），使它们接受一组作为参数的制表符停止位。如果启动程序时不带参数，则使用默认的制表符停止位设置。

```
#include  <stdio.h>

#define   MAXLINE   100         /* maximum line size           */
#define   TABINC    8           /* default tab increment size */
#define   YES       1
#define   NO        0
```

```
void settab(int argc, char *argv[], char *tab);
void entab(char *tab);
int tabpos(int pos, char *tab);

/* replace strings of blanks with tabs                          */
main(int argc, char *argv[])
{
     char tab[MAXLINE+1];

     settab(argc, argv, tab);       /* initialize tab stops  */
     entab(tab);                    /* replace blanks w/ tab */
     return 0;
}

/* entab: replace strings of blanks with tabs and blanks        */
void entab(char *tab)
{
     int c, pos;
     int nb = 0;                    /* # of blanks necessary */
     int nt = 0;                    /* # of tabs necessary   */

     for (pos = 1; (c=getchar()) != EOF; pos++)
          if (c == ' ') {
               if (tabpos(pos, tab) == NO)
                    ++nb;           /* increment # of blanks */
               else {
                    nb = 0;         /* reset # of blanks     */
                    ++nt;           /* one more tab          */
               }
          } else {
               for ( ; nt > 0; nt--)
                    putchar('\t'); /* output tab(s)          */
               if (c == '\t')       /* forget the blank(s)   */
                    nb = 0;
               else                 /* output blank(s)       */
                    for ( ; nb > 0; nb--)
                         putchar(' ');
               putchar(c);
               if (c == '\n')
                    pos = 0;
               else if (c == '\t')
                    while (tabpos(pos, tab) != YES)
                         ++pos;
          }
}
```

下面是源文件settab.c的内容：

```
#include  <stdlib.h>

#define   MAXLINE   100         /* maximum line size          */
#define   TABINC    8           /* default tab increment size */
#define   YES       1
#define   NO        0
```

```
/* settab: set tab stops in array tab                              */
void settab(int argc, char *argv[], char *tab)
{
     int i, pos;

     if (argc <= 1)           /* default tab stops                 */
          for (i = 1; i <= MAXLINE; i++)
               if (i % TABINC == 0)
                    tab[i] = YES;
               else
                    tab[i] = NO;
     else {                    /* user provided tab stops          */
          for (i = 1; i <= MAXLINE; i++)
               tab[i] = NO;    /* turn off all tab stops           */
          while (--argc > 0) {/* walk through argument list        */
               pos = atoi(*++argv);
               if (pos > 0 && pos <= MAXLINE)
                    tab[pos] = YES;
          }
     }
}
```

下面是源文件tabpos.c的内容：

```
#define   MAXLINE   100        /* maximum line size               */
#define   YES       1

/* tabpos: determine if pos is at a tab stop                       */
int tabpos(int pos, char *tab)
{
     if (pos > MAXLINE)
          return YES;
     else
          return tab[pos];
}
```

这个解决方案的总体框架来源于Kernighan和Plauger合著的《*Software Tools*》（Addison-Wesley，1976）一书中的entab程序。

数组tab中的每一个元素对应着文本行的一个位置，即，tab[1]对应着文本行的第一个位置（pos等于1）。如果文本行的某个位置处有一个制表符停止位，与之对应的tab[i]元素就将取值为YES；否则，与之对应的tab[i]元素就将取值为NO。

函数settab对制表符停止位进行了初始化。如果没有输入参数（即argc等于1），我们就将在文本行上每隔TABINC个位置设置一个制表符停止位。

如果有输入参数，我们就将根据那些输入参数来设置制表符停止位。

函数entab类似于练习1-21中的entab函数。

函数tabpos判断某个位置是否是一个制表符停止位：如果变量pos的取值大于MAXLINE，它将返回YES；否则，它将返回tab[pos]。

```
#include  <stdio.h>

#define   MAXLINE   100        /* maximum line size               */
#define   TABINC    8          /* default tab increment size      */
```

```
#define   YES       1
#define   NO        0

void settab(int argc, char *argv[], char *tab);
void detab(char *tab);
int tabpos(int pos, char *tab);

/* replace tabs with blanks                                    */
main(int argc, char *argv[])
{
     char tab[MAXLINE+1];

     settab(argc, argv, tab);       /* initialize tab stops  */
     detab(tab);                    /* replace tab w/ blanks */
     return 0;
}
/* detab: replace tab with blanks                              */
void detab(char *tab)
{
     int c, pos = 1;

     while ((c = getchar()) != EOF)
          if (c == '\t') {          /* tab character         */
               do
                    putchar(' ');
               while (tabpos(pos++, tab) != YES);
          } else if (c == '\n') {   /* newline character     */
               putchar(c);
               pos = 1;
          } else {                  /* all other characters  */
               putchar(c);
               ++pos;
          }
}
```

这个解决方案的总体框架来源于Kernighan和Plauger合著的《*Software Tools*》(Addison-Wesley，1976)一书中的detab程序。

函数tabpos和settab与本练习前半部分中的一样。

函数detab类似于练习1-20中的detab函数。

练习5-12 （教材第102页）

对程序entab和detab做一些扩充，以接受下列缩写的命令：

```
entab -m +n
```

表示制表符从第*m*列开始，每隔*n*列停止。选择（对使用者而言）比较方便的默认行为。

```
#include  <stdio.h>

#define   MAXLINE   100         /* maximum line size           */
#define   TABINC    8           /* default tab increment size */
#define   YES       1
#define   NO        0

void esettab(int argc, char *argv[], char *tab);
```

```
void entab(char *tab);

/* replace strings of blanks with tabs                              */
main(int argc, char *argv[])
{
     char tab[MAXLINE+1];

     esettab(argc, argv, tab);      /* initialize tab stops  */
     entab(tab);                    /* replace blanks w/ tab */
     return 0;
}
```

下面是源文件esettab.c的内容：

```
#include  <stdlib.h>

#define   MAXLINE   100        /* maximum line size           */
#define   TABINC    8          /* default tab increment size */
#define   YES       1
#define   NO        0

/* esettab: set tab stops in array tab                          */
void esettab(int argc, char *argv[], char *tab)
{
     int i, inc, pos;

     if (argc <= 1)             /* default tab stops           */
          for (i = 1; i <= MAXLINE; i++)
               if (i % TABINC == 0)
                    tab[i] = YES;
               else
                    tab[i] = NO;
     else if (argc == 3 &&      /* user provided range         */
               *argv[1] == '-' && *argv[2] == '+') {
          pos = atoi(&(*++argv)[1]);
          inc = atoi(&(*++argv)[1]);
          for (i = 1; i <= MAXLINE; i++)
               if (i != pos)
                    tab[i] = NO;
               else {
                    tab[i] = YES;
                    pos += inc;
               }
     } else {                   /* user provided tab stops     */
          for (i = 1; i <= MAXLINE; i++)
               tab[i] = NO;     /* turn off all tab stops      */
          while (--argc > 0) {/* walk through argument list */
               pos = atoi(*++argv);
               if (pos > 0 && pos <= MAXLINE)
                    tab[pos] = YES;
          }
     }
}
```

这个解决方案的总体框架来源于Kernighan和Plauger合著的《*Software Tools*》（Addison-Wesley，1976）一书中的entab程序。

这个解决方案与练习5-11中的entab程序很相似，惟一的修改是把settab函数替换为esettab（意思是“extended settab”——扩展了的settab）。

esettab能够接收简写形式 -*m* +*n*做为其输入参数。语句

```
pos = atoi(&(*++argv)[1]);
inc = atoi(&(*++argv)[1]);
```

把变量 pos设置为第一个制表符停止位的位置，把inc设置为制表符停止的位置递增值。这样，我们就能让制表符停止位从位置pos开始每隔inc个位置出现一次。

```
#include  <stdio.h>

#define   MAXLINE   100         /* maximum line size           */
#define   TABINC    8           /* default tab increment size */
#define   YES       1
#define   NO        0

void esettab(int argc, char *argv[], char *tab);
void detab(char *tab);

/* replace tabs with blanks                                    */
main(int argc, char *argv[])
{
     char tab[MAXLINE+1];

     esettab(argc, argv, tab);      /* initialize tab stops  */
     detab(tab);                    /* replace tab w/ blanks */
     return 0;
}
```

这个解决方案的总体框架来源于Kernighan和Plauger合著的《*Software Tools*》（Addison-Wesley，1976）一书中的detab程序。

这个解决方案与练习5-11中的detab程序很相似，并使用了本练习前半部分中的esettab函数。

练习5-13 （教材第102页）

编写程序tail，将其输入中的最后*n*行打印出来。默认情况下，*n*的值为10，但可通过一个可选参数改变*n*的值，因此，命令

```
tail -n
```

将打印其输入的最后*n*行。无论输入或*n*的值是否合理，该程序都应该能正常运行。编写的程序要充分地利用存储空间；输入行的存储方式应该同5.6节中排序程序的存储方式一样，而不采用固定长度的二维数组。

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

#define   DEFLINES  10    /* default # of lines to print      */
#define   LINES     100   /* max # of lines to print          */
#define   MAXLEN    100   /* max length of an input line     */
```

```
void error(char *);
int getline(char *, int);

/* print last n lines of the input                                    */
main(int argc, char *argv[])
{
     char *p;
     char *buf;                 /* pointer to large buffer     */
     char *bufend;              /* end of the buffer           */
     char line[MAXLEN];         /* current input line          */
     char *lineptr[LINES];      /* pointers to lines read      */
     int first, i, last, len, n, nlines;

     if (argc == 1)             /* no argument present         */
          n = DEFLINES;         /* use default # of lines      */
     else if (argc == 2 && (*++argv)[0] == '-')
          n = atoi(argv[0]+1);
     else
          error("usage:  tail [-n]");
     if (n < 1 || n > LINES)    /* unreasonable value for n?   */
          n = LINES;
     for (i = 0; i < LINES; i++)
          lineptr[i] = NULL;
     if ((p = buf = malloc(LINES * MAXLEN)) == NULL)
          error("tail: cannot allocate buf");
     bufend = buf + LINES * MAXLEN;
     last = 0;                  /* index of last line read     */
     nlines = 0;                /* number of lines read        */
     while ((len = getline(line, MAXLEN)) > 0) {
          if (p + len + 1 >= bufend)
               p = buf;         /* buffer wrap around          */
          lineptr[last] = p;
          strcpy(lineptr[last], line);
          if (++last >= LINES)
               last = 0;        /* ptrs to buffer wrap around */
          p += len + 1;
          nlines++;
     }
     if (n > nlines)            /* req. lines more than rec.? */
          n = nlines;
     first = last - n;
     if (first < 0)             /* it did wrap around the list*/
          first += LINES;
     for (i = first; n-- > 0; i = (i + 1) % LINES)
          printf("%s", lineptr[i]);
     return 0;
}

/* error: print error message and exit                                */
void error(char *s)
{
     printf("%s\n", s);
     exit(1);
}
```

这个程序将把其输入中的最后*n*行打印出来（但最多打印LINES行）。如果argc等于1（即没有命令行参数），*n*将默认地取值为DEFLINES；如果argc等于2（即有一个命令行参数），*n*的取值就将来自那个命令行参数；如果argc大于2，则程序出错。

循环

```
while ((len = getline(line, MAXLEN)) > 0)
```

每次读入一个输入行，直到函数getline（参见练习1-16）读到最后一个输入行为止。这个程序将使用预先分配好的缓冲区空间来存放每次读入的输入行。

当缓冲区没有足够的空间来存放当前输入行时，语句

```
if (p + len + 1 >= bufend)
    p = buf;
```

将把指针p重新设置为指向缓冲区的开头。

指针数组lineptr中的各个元素将依次指向最后读入的LINES个输入行各自的第一个字符。我们把这个数组的索引值保存在变量last中。

变量last的初值是0，当last等于LINES（即已经读入了LINES个输入行）时，就需要将其重新设置为0；也就是说，我们将开始循环使用数组lineptr的元素及其缓冲区。

变量nlines保存着当前已被读入的输入行总数。既然这个程序只是用来打印最后*n*个输入行的，那么*n*无论如何也不应该大于这个程序的输入行总数。因此，

```
if (n > nlines)
    n = nlines;
```

如果输入行的总数超过了LINES，last将被重新设置为0，还需要调整起始索引：

```
if (first < 0)
    first += LINES;
```

最后，这个程序将通过以下语句把它的最后*n*个输入行打印出来：

```
for (i = first; n-- > 0; i = (i + 1) % LINES)
    printf("%s", lineptr[i]);
```

因为i从初值first开始递增*n*次，那么，当i大于LINES时，就会出现某些输入行被反复打印多次的情况。为了避免出现这一局面，我们使用了取模运算符（%）来确保i的取值范围能够落在0到LINES－1之间，如下所示：

```
i = (i + 1) % LINES
```

标准库函数exit（参见教材第143页）将在tail程序执行出错时终止它的运行，此时，exit将返回1。

练习5-14 （教材第105页）

修改排序程序，使它能处理－r标记。该标记表明，以逆序（递减）方式排序。要保证－r和－n能够组合在一起使用。

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

#define   NUMERIC   1     /* numeric sort                          */
#define   DECR      2     /* sort in decreasing order              */
```

```
#define   LINES     100  /* max # of lines to be sorted     */

int numcmp(char *, char *);
int readlines(char *lineptr[], int maxlines);
void qsort(char *v[], int left, int right,
          int (*comp)(void *, void *));
void writelines(char *lineptr[], int nlines, int decr);

static char option = 0;

/* sort input lines  */
main(int argc, char *argv[])
{
     char *lineptr[LINES];     /* pointers to text lines     */
     int nlines;               /* number of input lines read */
     int c, rc = 0;

     while (--argc > 0 && (++argv)[0] == '-')
          while (c = *++argv[0])
               switch (c) {
               case 'n' :      /* numeric sort               */
                    option |= NUMERIC;
                    break;
               case 'r':       /* sort in decreasing order   */
                    option |= DECR;
                    break;
               default:
                    printf("sort: illegal option %c\n", c);
                    argc = 1;
                    rc = -1;
                    break;
               }
     if (argc)
          printf("Usage: sort -nr \n");
     else
          if ((nlines = readlines(lineptr, LINES)) > 0) {
               if (option & NUMERIC)
                    qsort((void **) lineptr, 0, nlines-1,
                         (int (*) (void *, void *)) numcmp);
               else
                    qsort((void **) lineptr, 0, nlines-1,
                         (int (*) (void *, void *)) strcmp);
               writelines(lineptr, nlines, option & DECR);
          } else {
               printf("input too big to sort \n");
               rc = -1;
          }
     return rc;
}

/* writelines: write output lines                              */
void writelines(char *lineptr[], int nlines, int decr)
{
     int i;

     if (decr)                     /* print in decreasing order  */
          for (i = nlines-1; i >= 0; i--)
```

```
                printf("%s\n", lineptr[i]);
        else                            /* print in increasing order  */
                for (i = 0; i < nlines; i++)
                printf("%s\n", lineptr[i]);
}
```

排序程序的具体功能由静态字符变量option的二进制位取值决定：

第0位 =0 对字符串排序

=1 对数字排序（-n）

第1位 =0 按升序排序

=1 对降序排序（-r）

如果用户给出了某个命令行选项，按位OR运算符（|）就将把变量option中与之对应的位设置为1。语句

```
option |= DECR;
```

等价于

```
option = option | 2;
```

我们知道，十进制数字2的二进制表示形式是00000010。因为“1 OR 0”和“1 OR 1”的运算结果都将等于1，所以上面这条C语句将把字符变量option的第1位设置为1。（二进制位按从左至右的顺序被编号为第0、1、2、3…位。）

为了判断用户是否给出了某个命令行选项，我们使用了按位AND（&）运算符。

如果用户给出了命令行选项-r，表达式

```
option & DECR
```

就将被求值为“真”；如果用户没有给出命令行选项-r，这个表达式就将被求值为“假”。

我们对函数writelines进行了修改，使它能够对新增加的decr参数进行处理。变量decr是表达式option & DECR的求值结果，排序程序将根据这个变量来决定有关数据在排序后是按升序输出还是按降序输出。

函数strcmp、numcmp、swap、qsort和readlines与排序程序原始版本中的同名函数（参见教材第103页）完全一样。

练习5-15 （教材第105页）

增加选项-f，使得排序过程不考虑字母大小写之间的区别。例如，比较a和A时认为它们相等。

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

#define   NUMERIC   1     /* numeric sort                          */
#define   DECR      2     /* sort in decreasing order              */
#define   FOLD      4     /* fold upper and lower cases            */
#define   LINES     100   /* max # of lines to be sorted           */

int charcmp(char *, char *);
int numcmp(char *, char *);
int readlines(char *lineptr[], int maxlines);
void qsort(char *v[], int left, int right,
           int (*comp)(void *, void *));
void writelines(char *lineptr[], int nlines, int order);
```

```
static char option = 0;

/* sort input lines                                              */
main(int argc, char *argv[])
{
     char *lineptr[LINES];     /* pointers to text lines      */
     int nlines;               /* number of input lines read */
     int c, rc = 0;

     while (--argc > 0 && (*++argv)[0] == '-')
          while (c = *++argv[0])
               switch (c) {
               case 'f':       /* fold upper and lower cases */
                    option |= FOLD;
                    break;
               case 'n':       /* numeric sort               */
                    option |= NUMERIC;
                    break;
               case 'r':       /* sort in decreasing order   */
                    option |= DECR;
                    break;
               default:
                    printf("sort: illegal option %c\n", c);
                    argc = 1;
                    rc = -1;
                    break;
               }

      if (argc)
          printf("Usage: sort -fnr \n");
      else {
          if ((nlines = readlines(lineptr, LINES)) > 0) {
               if (option & NUMERIC)
                    qsort((void **) lineptr, 0, nlines-1,
                         (int (*)(void *, void *)) numcmp);
               else if (option & FOLD)
                    qsort((void **) lineptr, 0, nlines-1,
                         (int (*)(void *, void *)) charcmp);
               else
                    qsort((void **) lineptr, 0, nlines-1,
                         (int (*)(void *, void *)) strcmp);
               writelines(lineptr, nlines, option & DECR);
          } else {
               printf("input too big to sort \n");
               rc = -1;
          }
      }
      return rc;
}

/* charcmp: return <0 if s<t, 0 if s==t, >0 if s>t      */
int charcmp(char *s, char *t)
{
     for ( ; tolower(*s) == tolower(*t); s++, t++)
          if (*s == '\0')
               return 0;
     return tolower(*s) - tolower(*t);
}
```

本习题的解答思路来源于练习5-14。

第2位 = 0　区分字母的大小写

= 1　不区分字母的大小写（-f）

如果用户在命令行上给出了-f选项，变量option的第2位就将被下面这条语句设置为1：

```
option |= FOLD;
```

FOLD（十进制数字4）的二进制表示形式是00000100（二进制位按从左至右的顺序被编号为第0、1、2、3…位。）

函数charcmp与strcmp（教材第92页）很相似，它们都是用来对字符串进行比较的。但为了支持FOLD选项，charcmp会先把字符转换为小写形式后再对它们进行比较。

函数numcmp、swap、qsort、readlines和writelines与我们在解答练习5-14时所使用的同名函数完全一样。

练习5-16 （教材第105页）

增加选项-d（代表目录顺序）。该选项表明，只对字母、数字和空格进行比较。要保证该选项可以和-f组合在一起使用。

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

#define   NUMERIC   1     /* numeric sort                    */
#define   DECR      2     /* sort in decreasing order        */
#define   FOLD      4     /* fold upper and lower cases      */
#define   DIR       8     /* directory order                 */
#define   LINES     100   /* max # of lines to be sorted     */

int charcmp(char *, char *);
int numcmp(char *, char *);
int readlines(char *lineptr[], int maxlines);
void qsort(char *v[], int left, int right,
          int (*comp) (void *, void *));
void writelines(char *lineptr[], int nlines, int order);

static char option = 0;

/* sort input lines                                        */
main(int argc, char *argv[])
{
     char *lineptr[LINES];    /* pointers to text lines     */
     int nlines;              /* number of input lines read */
     int c, rc = 0;

     while (--argc > 0 && (*++argv)[0] == '-')
          while (c = *++argv[0])
               switch (c) {
               case 'd':      /* directory order            */
                    option |= DIR;
                    break;
               case 'f':      /* fold upper and lower cases */
                    option |= FOLD;
                    break;
```

```
            case 'n':       /* numeric sort                    */
                option |= NUMERIC;
                break;
            case 'r':       /* sort in decreasing order        */
                option |= DECR;
                break;
            default:
                printf("sort: illegal option %c\n", c);
                argc = 1;
                rc = -1;
                break;
            }
    if (argc)
        printf("Usage: sort -dfnr \n");
    else {
        if ((nlines = readlines(lineptr, LINES)) > 0) {
            if (option & NUMERIC)
                qsort((void **) lineptr, 0, nlines-1,
                    (int (*)(void *, void *)) numcmp);
            else
                qsort((void **) lineptr, 0, nlines-1,
                    (int (*)(void *, void *)) charcmp);
            writelines(lineptr, nlines, option & DECR);
        } else {
            printf("input too big to sort \n");
            rc = -1;
        }
        return rc;
    }

    /* charcmp: return <0 if s<t, 0 if s==t, >0 if s>t    */
    int charcmp(char *s, char *t)
    {
        char a, b;
        int fold = (option & FOLD) ?  1 : 0;
        int dir = (option & DIR) ?  1 : 0;

        do {
            if (dir) {
                while (!isalnum(*s) && *s != ' ' && *s != '\0')
                    s++;
                while (!isalnum(*t) && *t != ' ' && *t != '\0')
                    t++;
            }
            a = fold ?  tolower(*s) : *s;
            s++;
            b = fold ?  tolower(*t) : *t;
            t++;
            if (a == b && a == '\0')
                return 0;
        } while (a == b);
        return a - b;
    }
```

本题的解答思路来源于练习5-14和练习5-15。

第3位 = 0 不按目录顺序排序

= 1 按目录顺序排序（-d）

如果用户在命令行上给出了-d选项，变量option的第3位就将被下面这条语句设置为1：

```
option |= DIR;
```

DIR（十进制数字8）的二进制表示形式是00001000（二进制位按从左至右的顺序被编号为第0、1、2、3…位。）

我们对练习5-15中的函数charcmp进行了修改，使它能够处理-f选项和-d选项。

如果用户给出了命令行选项-d，程序就会执行到下面这个while循环：

```
while (!isalnum(*s) && *s != ' ' && *s != '\0')
    s++;
```

这个循环将依次检查字符串s的各个字符，并跳过不是字母、数字和空格的字符。宏isalnum是在头文件<ctype.h>中定义的，它的作用是检查某个字符是不是一个字母（a~z，A~Z）或数字（0~9）。如果*s是一个字母或者是一个数字，isalnum(*s)就是一个非负值；否则，isalnum(*s)将等于0。

紧接其后的下一个while循环：

```
while (!isalnum(*t) && *t != ' ' && *t != '\0')
    t++;
```

将依次检查字符串t的各个字符，并跳过不是字母、数字和空格的字符。

接下来，charcmp函数将把“在字符串s中找到的字母、数字或空格”与“在字符串t中找到的字母、数字或空格”逐个进行比较。

本题的另一解答思路是用3个专用的函数：foldcmp、dircmp和folddircmp来代替charcmp。其中，foldcmp用来实现-f选项（不区分字母大小写）的字符比较功能，相当于练习5-15中的charcmp函数；dircmp用来实现-d选项（按目录顺序排序）的字符比较功能；folddircmp用来实现-f与-d选项联合使用时的（不区分字母大小写、按目录顺序排序）的字符比较功能。这3个专用函数的执行速度要高于charcmp函数。我们的选择是逐步增加charcmp函数的复杂程度，而不是编写那么多个专用的函数。

函数numcmp、swap、qsort、readlines和writelines与我们在解答练习5-14时所使用的同名函数完全一样。

练习5-17 （教材第105页）

增加字段处理功能，以使得排序程序可以根据行内的不同字段进行排序，每个字段按照一个单独的选项集合进行排序。（英文原书索引进行排序时，索引条目使用了-df选项，而对页码排序时使用了-n选项。）

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

#define   NUMERIC   1     /* numeric sort                          */
#define   DECR      2     /* sort in decreasing order              */
#define   FOLD      4     /* fold upper and lower cases            */
#define   DIR       8     /* directory order                       */
```

```
#define   LINES     100  /* max # of lines to be sorted      */

int charcmp(char *, char *);
void error(char *);
int numcmp(char *, char *);
void readargs(int argc, char *argv[]);
int readlines(char *lineptr[], int maxlines);
void qsort(char *v[], int left, int right,
          int (*comp)(void *, void *));
void writelines(char *lineptr[], int nlines, int order);

char option = 0;
int pos1 = 0;                    /* field beginning with pos1   */
int pos2 = 0;                    /* ending just before pos2     */

/* sort input lines                                             */
main(int argc, char *argv[])
{
     char *lineptr[LINES];     /* pointers to text lines       */
     int nlines;               /* number of input lines read   */
     int rc = 0;

     readargs(argc, argv);
     if ((nlines = readlines(lineptr, LINES)) > 0) {
          if (option & NUMERIC)
               qsort((void **) lineptr, 0, nlines-1,
                    (int (*)(void *, void *)) numcmp);
          else
               qsort((void **) lineptr, 0, nlines-1,
                    (int (*)(void *, void *)) charcmp);
          writelines(lineptr, nlines, option & DECR);
     } else {
          printf("input too big to sort \n");
          rc = -1;
     }
     return rc;
}

/* readargs: read program arguments                             */
void readargs(int argc, char *argv[])
{
     int c;
     int atoi(char *);

     while (--argc > 0 && (c = (++argv)[0]) == '-' || c == '+') {
          if (c == '-' && !isdigit(*(argv[0]+1)))
               while (c = ++argv[0])
                    switch (c) {
                    case 'd':      /* directory order         */
                         option |= DIR;
                         break;
                    case 'f':      /* fold upper and lower    */
                         option |= FOLD;
                         break;
                    case 'n':      /* numeric sort            */
```

```
                        option |= NUMERIC;
                        break;
                    case 'r':       /* sort in decr. order     */
                        option |= DECR;
                        break;
                    default:
                        printf("sort: illegal option %c\n", c);
                        error("Usage: sort -dfnr [+pos1] [-pos2]"
                        break;
                    }
            else if (c == '-')
                pos2 = atoi(argv[0]+1);
            else if ((pos1 = atoi(argv[0]+1)) < 0)
                error("Usage: sort -dfnr [+pos1] [-pos2]");
        }
        if (argc || pos1 > pos2)
            error("Usage: sort -dfnr [+pos1] [-pos2]");
}
```

下面是源文件numcmp.c的内容：

```
#include  <math.h>
#include  <ctype.h>
#include  <string.h>
#define   MAXSTR    100

void substr(char *s, char *t, int maxstr);
/* numcmp: compare s1 and s2 numerically                    */
int numcmp(char *s1, char *s2)
{
    double v1, v2;
    char str[MAXSTR];

    substr(s1, str, MAXSTR);
    v1 = atof(str);
    substr(s2, str, MAXSTR);
    v2 = atof(str);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}

#define   FOLD      4     /* fold upper and lower cases       */
#define   DIR       8     /* directory order                  */

/* charcmp: return <0 if s<t, 0 if s==t, >0 if s>t            */
int charcmp(char *s, char *t)
{
    char a, b;
    int i, j, endpos;
    extern char option;
    extern int pos1, pos2;
```

```
int fold = (option & FOLD) ?  1 : 0;
int dir  = (option & DIR) ?  1 : 0;

i = j = pos1;
if (pos2 > 0)
     endpos = pos2;
else if ((endpos = strlen(s)) > strlen(t))
     endpos = strlen(t);
do {
     if (dir) {
          while (i < endpos && !isalnum(s[i]) &&
                  s[i] != ' ' && s[i] != '\0')
               i++;
          while (j < endpos && !isalnum(t[j]) &&
                  t[j] != ' ' && t[j] != '\0')
               j++;
     }
     if (i < endpos && j < endpos) {
          a = fold ?  tolower(s[i]) : s[i];
          i++;
          b = fold ?  tolower(t[j]) : t[j];
          j++;
          if (a == b && a == '\0')
               return 0;
     }
} while (a == b && i < endpos && j < endpos);
return a - b;
}
```

下面是源文件substr.c的内容：

```
#include  <string.h>

void error(char *);

/* substr: get a substring of s and put in str                    */
void substr(char *s, char *str)
{
     int i, j, len;
     extern int pos1, pos2;

     len = strlen(s);
     if (pos2 > 0 && len > pos2)
          len = pos2;
     else if (pos2 > 0 && len < pos2)
          error("substr: string too short");
     for (j = 0, i = pos1; i < len; i++, j++)
          str[j] = s[i];
     str[j] = '\0';
}
```

本题的解答思路来源于练习5-14、练习5-15和练习5-16。

排序命令的语法是：

```
sort -dfnr [+pos1] [-pos2]
```

如果你想按文本行中的某个字段进行排序，就需要给出pos1和pos2；排序程序将根据

pos1和pos2之间的字符内容进行排序。如果pos1和pos2等于0，就表示以整个文本行作为排序码。

函数readargs负责分析排序程序的命令行选项。当命令行带有选项且选项前带有减号 - 字符时，readargs函数中的while循环的条件表达式将一直为“真”。

当命令行参数是一个以减号 - 开始的非数字字符时，第一个if语句

```
if (c == '-' && !isdigit(*(argv[0]+1)))
```

将为“真”。由readargs函数识别出来的命令行参数将交给switch语句去处理，具体实现方法与练习5-14、练习5-15和练习5-16中的相同。

只有在用户给出了可选参数 -pos2时，紧随其后的else-if语句

```
else if (t == '-')
```

才会为“真”。

最后的else-if语句负责处理+pos1参数，并检查它是否大于0。

函数charcmp是前面练习中的同名函数的改进版本，我们给它增加了字段处理功能。

函数numcmp对数字进行比较，它与前面练习中的同名函数差不多，但使用了一个新的substr函数——因为排序字段的起始位置和长度不能用作atof的输入参数。我们不想改变atof函数的调用接口，因为修改一个像atof这样的常用函数的调用接口往往会带来很多问题，而增加一个新函数substr要比修改接口更安全。

函数swap、qsort、readlines和writelines与我们在解答练习5-14时所使用的同名函数完全一样；error函数取自于练习5-13。

练习5-18 （教材第109页）

修改dcl程序，使它能够处理输入中的错误。

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

enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

void dcl(void);
void dirdcl(void);
void errmsg(char *);
int gettoken(void);

extern int tokentype;     /* type of last token              */
extern char token[];      /* last token string               */
extern char name[];       /* identifier name                 */
extern char out[];
extern int prevtoken;

/* dcl: parse a declarator                                    */
void dcl(void)
{
     int ns;

     for (ns = 0; gettoken() == '*'; )  /* count *'s          */
```

```
            ns++;
        dirdcl();
        while (ns-- > 0)
            strcat(out, " pointer to");
}

/* dirdcl: parse a direct declaration                              */
void dirdcl(void)
{
        int type;

        if (tokentype == '(') {              /* ( dcl )            */
            dcl();
            if (tokentype != ')')
                errmsg("error: missing )\n");
        } else if (tokentype == NAME)        /* variable name      */
            strcpy(name, token);
        else
            errmsg("error: expected name or (dcl)\n");
        while ((type = gettoken()) == PARENS || type == BRACKETS)
            if (type == PARENS)
                strcat(out, " function returning");
            else {
                strcat(out, " array");
                strcat(out, token);
                strcat(out, " of");
            }
}

/* errmsg: print error message and indicate avail.  token          */
void errmsg(char *msg)
{
        printf("%s", msg);
        prevtoken = YES;
}
```

下面是源文件gettoken.c的内容：

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

enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

extern int  tokentype;    /* type of last token                    */
extern char token[];      /* last token string                     */
int prevtoken = NO;       /* there is no previous token            */

/* gettoken: return next token                                     */
int gettoken(void)
{
        int c, getch(void);
        void ungetch(int);
        char *p = token;

        if (prevtoken == YES) {
```

```
            prevtoken = NO;
            return tokentype;
        }
        while ((c = getch()) == ' ' || c == '\t')
            ;
        if (c == '(') {
            if ((c = getch()) == ')') {
                strcpy(token, "()");
                return tokentype = PARENS;
            } else {
                ungetch(c);
                return tokentype = '(';
            }
        } else if (c == '[') {
            for (*p++ = c; (*p++ = getch()) != ']'; )
                ;
            *p = '\0';
            return tokentype = BRACKETS;
        } else if (isalpha(c)) {
            for (*p++ = c; isalnum(c = getch()); )
                *p++ = c;
            *p = '\0';
            ungetch(c);
            return tokentype = NAME;
        } else
            return tokentype = c;
}
```

我们对函数dirdcl做了一些修改，它现在能够分析出两种记号——跟在dcl调用后的一个右括号（')'）或一个变量名。如果不是这两种记号，我们将调用函数errmsg而不是printf。errmsg会先打印一条出错信息，然后把变量prevtoken设置为YES以通知gettoken说已经读入了一个记号。gettoken开头部分有一个新的if语句，如下所示：

```
if (prevtoken == YES) {
    prevtoken = NO;
    return tokentype;
}
```

这条语句的意思是：如果已经有了一个记号，就不要再读入一个新记号了。

我们的改进版本并不是十全十美的，但它已经具备一定的出错处理能力了。

练习5-19 （教材第109页）

修改undcl程序，使它在把文字描述转换为声明的过程中不会生成多余的圆括号。

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

#define   MAXTOKEN  100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
```

```
void dirdcl(void);
int  gettoken(void);
int  nexttoken(void);

int  tokentype;            /* type of last token                    */
char token[MAXTOKEN];      /* last token string                     */
char out[1000];

/* undcl: convert word description to declaration                   */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                if ((type = nexttoken()) == PARENS ||
                    type == BRACKETS)
                    sprintf(temp, "(*%s)", out);
                else
                    sprintf(temp, "*%s", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
enum { NO, YES };

int gettoken(void);

/* nexttoken: get the next token and push it back                   */
int nexttoken(void)
{
    int type;
    extern int prevtoken;

    type = gettoken();
    prevtoken = YES;
    return type;
}
```

如果“x是一个指向char的指针”，undcl程序的输入将是：

```
x * char
```

改进前的undcl程序的输出结果是：

```
char (*x)
```

但这个输出结果中的括号是多余的。事实上，只有当下一个记号是( )或[ ]时，undcl程序才有必要在自己的输出结果中使用括号。

例如，如果“daytab是一个指针，它指向一个有[13]个int元素的数组”，undcl程序的输入就将是：

```
daytab * [13] int
```

改进前的undcl程序的输出结果

```
int (*daytab)[13]
```

就是正确的。但是，如果“daytab是一个有[13]个元素的指针数组，数组中的每个元素分别指向一个int”，undcl程序的输入就将是：

```
daytab [13] * int
```

改进前的undcl程序的输出结果

```
int (*daytab[13])
```

中就会有多余的圆括号。

我们对undcl进行了修改，让它检查下一个记号是不是()或[]。如果下一个记号是()或[]，undcl程序就必须给它加上括号；否则，输出结果中的括号就将是多余的。也就是说，我们必须根据undcl程序输入中的下一个记号来决定是否需要添加括号。

我们编写了一个简单的nexttoken函数，它将调用gettoken，记录已经读入一个记号的事实并返回该记号的类型。gettoken是我们在解答练习5-18时编写的一个函数，它在读入下一个记号前会先检查是否已经有一个可用的记号了。

改进后的undcl程序将不再产生多余的括号。例如，如果它的输入是：

```
x * char
```

改进后的undcl程序的输出结果就将是：

```
char *x
```

如果输入是：

```
daytab * [13] int
```

改进后的undcl程序的输出结果就将是：

```
int (*daytab)[13]
```

而如果输入是：

```
daytab [13] * int
```

改进后的undcl程序的输出结果就将是：

```
int *daytab[13]
```

练习5-20 （教材第109页）

扩展dcl程序的功能，使它能够处理包含其他成分的声明，例如带有函数参数类型的声明、带有类似于const限定符的声明等。

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

```
enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

void dcl(void);
void dirdcl(void);
void errmsg(char *);
int gettoken(void);

extern int tokentype;     /* type of last token                    */
extern char token[];      /* last token string                     */
extern char name[];       /* identifier name                       */
extern char datatype[];   /* data type = char, int, etc.           */
extern char out[];
extern int prevtoken;

/* dcl: parse a declarator                                         */
void dcl(void)
{
     int ns;

     for (ns = 0; gettoken() == '*'; )  /* count *'s               */
          ns++;
     dirdcl();
     while (ns-- > 0)
          strcat(out, " pointer to");
}

/* dirdcl: parse a direct declaration                              */
void dirdcl(void)
{
     int type;
     void parmdcl(void);

     if (tokentype == '(') {              /* ( dcl )               */
          dcl();
          if (tokentype != ')')
               errmsg("error: missing)\n");
     } else if (tokentype == NAME) {      /* variable name         */
          if (name[0] == '\0')
               strcpy(name, token);
     } else
          prevtoken = YES;
     while ((type = gettoken()) == PARENS || type == BRACKETS ||
                                             type == '(')
          if (type == PARENS)
               strcat(out, "function returning");
          else if (type == '(') {
               strcat(out, " function expecting");
               parmdcl();
               strcat(out, " and returning");
          } else {
               strcat(out, " array");
               strcat(out, token);
               strcat(out, " of");
          }
```

```
}

/* errmsg: print error message and indicate avail.  token    */
void errmsg(char *msg)
{
     printf("%s", msg);
     prevtoken = YES;
}
```

下面是源文件parmdcl.c的内容：

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

#define   MAXTOKEN  100

enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

void dcl(void);
void errmsg(char *);
void dclspec(void);
int  typespec(void);
int  typequal(void);
int  compare(char **, char **);
int  gettoken(void);
extern int tokentype;     /* type of last token                */
extern char token[];      /* last token string                 */
extern char name[];       /* identifier name                   */
extern char datatype[];   /* data type = char, int, etc.       */
extern char out[];
extern int prevtoken;

/* parmdcl: parse a parameter declarator                       */
void parmdcl(void)
{
     do {
          dclspec();
     } while (tokentype == ',');
     if (tokentype != ')')
          errmsg("missing ) in parameter declaration\n");
}

/* dclspec: declaration specification                          */
void dclspec(void)
{
     char temp[MAXTOKEN];

     temp[0] = '\0';
     gettoken();
     do {
          if (tokentype != NAME) {
               prevtoken = YES;
               dcl();
```

```
            } else if (typespec() == YES) {
                strcat(temp, " ");
                strcat(temp, token);
                gettoken();
            } else if (typequal() == YES) {
                strcat(temp, " ");
                strcat(temp, token);
                gettoken();
            } else
                errmsg("unknown type in parameter list\n");
        } while (tokentype != ',' && tokentype != ')');
        strcat(out, temp);
        if (tokentype == ',')
            strcat(out, ",");
}
/* typespec: return YES if token is a type-specifier          */
int typespec(void)
{
        static char *types[] = {
                "char",
                "int",
                "void"
        };
        char *pt = token;

        if (bsearch(&pt, types, sizeof(types)/sizeof(char *),
            sizeof(char *), compare) == NULL)
                return NO;
        else
                return YES;
}

/* typequal: return YES if token is a type-qualifier          */
int typequal(void)
{
        static char *typeq[] = {
                "const",
                "volatile"
        };
        char *pt = token;

        if (bsearch(&pt, typeq, sizeof(typeq)/sizeof(char *),
            sizeof(char *), compare) == NULL)
                return NO;
        else
                return YES;
}

/* compare: compare two strings for bsearch                   */
int compare(char **s, char **t)
{
        return strcmp(*s, *t);
}
```

我们对教材第106页上的语法进行了扩充，增加了对参数声明符的处理能力，如下所示：

dcl: *optional *'s direct-dcl*

direct-dcl: *name*
(dcl)
direct-dcl (optional parm-dcl)
direct-dcl [optional size]

parm-dcl: *parm-dcl, dcl-spec dcl*

dcl-spec: *type-spec dcl-spec*
type-qual dcl-spec

这是C语言语法中描述声明符部分的一个缩略版本。扩展后的dcl程序能够把教材第188页上介绍的一部分类型说明符识别出来。比如说，如果dcl程序的输入是：

```
void *(*comp)(int *, char *, int (*fnc)())
```

它将产生如下所示的输出结果：

```
comp: pointer to function expecting pointer to int, pointer
      to char, pointer to function returning int and returning
      pointer to void
```

我们对dirdcl函数进行了修改，并增加了parmdcl和dclspec函数。

这里使用了我们在解答练习5-18时采用的提前分析下一记号的思路。有时候，我们需要根据下一个记号来决定执行哪些操作。有时候，我们分析出了下一个记号，但还用不上它，那就要把它重新压回输入缓冲区；然后，等我们在词法分析器中的其他地方再次调用到gettoken函数的时候，再把这个记号取出来并进行相应的处理。

bsearch是一个用来完成折半查找操作的标准库函数。

第6章 结 构

练习6-1 （教材第119页）

上述getword函数还不能正确处理下划线、字符串常数、注释及预编译器控制指令。请编写一个更完善的getword函数。

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

/* getword: get next word or character from input            */
int getword(char *word, int lim)
{
     int c, d, comment(void), getch(void);
     void ungetch(int);
     char *w = word;

     while (isspace(c = getch()))
          ;
     if (c != EOF)
          *w++ = c;
     if (isalpha(c) || c == '_' || c == '#') {
          for ( ; --lim > 0; w++)
               if (!isalnum(*w = getch()) && *w != '_') {
                    ungetch(*w);
                    break;
               }
     } else if (c == '\'' || c == '"') {
          for ( ; --lim > 0; w++)
               if ((*w = getch()) == '\\')
                    *++w = getch();
               else if (*w == c) {
                    w++;
                    break;
               } else if (*w == EOF)
                    break;
      } else if (c == '/')
          if ((d = getch()) == '*')
               c = comment();
          else
               ungetch(d);
      *w = '\0';
      return c;
}

/* comment: skip over comment and return a character        */
```

```
int comment(void)
{
      int c;
      while ((c = getch()) != EOF)
            if (c == '*')
                  if ((c = getch()) == '/')
                        break;
                  else
                        ungetch(c);
      return c;
}
```

为了处理下划线和预编译器指令，我们把原来的

```
if (!alpha(c))
```

语句修改为

```
if (isalpha(c) || c == '_' || c == '#')
```

跟在_和#后面的字母和数字字符将被看作是单词的一部分。

字符串常量可能出现在单引号或双引号中。一旦检测到左引号，我们就将把该左引号到与之邻近的右引号或EOF标记之间的全部字符收集起来，并把它们当作一个字符串常量来对待。

如果遇到注释，我们就跳过其内容并返回它的结尾斜线字符，这部分代码类似于练习1-24。

练习6-2 （教材第125页）

编写一个程序，用以读入一个C语言程序，并按字母表顺序分组打印变量名，要求每一组内各变量名的前6个字符相同，其余字符不同。字符串和注释中的单词不予考虑，请将6作为一个可在命令行中设定的参数。

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

struct tnode {                          /* the tree node:       */
      char *word;                       /* points to the text   */
      int match;                        /* match found          */
      struct tnode *left;               /* left child           */
      struct tnode *right;              /* right child          */
};

#define   MAXWORD   100
#define   YES       1
#define   NO        0

struct tnode *addtreex(struct tnode *, char *, int, int *);
void treexprint(struct tnode *);
int getword(char *, int);

/* print in alphabetic order each group of variable names   */
/* identical in the first num cherecters (defeult 6)        */
main(int argc, char *argv[])
{
      struct tnode *root;
```

```
        char word[MAXWORD];
        int found = NO;       /* YES if match was found          */
        int num;              /* number of the first ident. chars*/

        num = (--argc && (++argv)[0] == '-') ?  atoi(argv[0]+1) : 6;
        root = NULL;
        while (getword(word, MAXWORD) != EOF) {
             if (isalpha(word[0]) && strlen(word) >= num)
                  root = addtreex(root, word, num, &found);
             found = NO;
        }
        treexprint(root);
        return 0;
}
struct tnode *talloc(void);
int compare(char *, struct tnode *, int, int *);

/* addtreex: add a node with w, at or below p                    */
struct tnode *addtreex(struct tnode *p, char *w,
                       int num, int *found)
{
        int cond;

        if (p == NULL) {          /* a new word has arrived      */
             p = talloc();        /* make a new node             */
             p->word = strdup(w);
             p->match = *found;
             p->left = p->right = NULL;
        } else if ((cond = compare(w, p, num, found)) < 0)
             p->left = addtreex(p->left, w, num, found);
        else if (cond > 0)
             p->right = addtreex(p->right, w, num, found);
        return p;
}

/* compare: compare words and update p->match                    */
int compare(char *s, struct tnode *p, int num, int *found)
{
        int i;
        char *t = p->word;

        for (i = 0; *s == *t; i++, s++, t++)
             if (*s == '\0')
                  return 0;
        if (i >= num) {       /* identical in first num chars ?  */
             *found = YES;
             p->match = YES;
        }
        return *s - *t;
}

/* treexprint: in-order print of tree p if p->match == YES       */
void treexprint(struct tnode *p)
{
        if (p != NULL) {
```

```
            treeprint(p->left);
            if (p->match)
                printf("%s\n", p->word);
            treeprint(p->right);
        }
}
```

这个程序将把前num个字符相同的变量名打印出来。如果用户没有在命令行指定num的值，我们就把变量num设置为6，如下所示：

```
num = (--argc && (*++argv)[0] == '-') ?  atoi(argv[0]+1) : 6;
```

变量found是一个布尔量。如果新识别出来的单词与变量名树上的某个单词前num个字符相同，我们就把变量found设置为YES；否则，我们就把变量found设置为NO。

如果某单词的第一个字符是一个字母且它的长度大于或等于num，这个程序就会把它添加到变量名树中。函数getword来自练习6-1。函数addtreex是addtree函数（见教材第124页）的改进版本，它负责把单词添加到变量名树中。

函数compare把将被添加到变量名树中的单词与已经在变量树中的单词进行比较。如果在前num个字符中找到了一个匹配，那么，与变量树中的单词相对应的*found以及匹配成员（即p->match）都将被设置为YES，如下所示：

```
if (i >= num) {
        *found = YES;
        p->match = YES;
}
```

treeprint函数把变量名树中的单词打印出来，前num个字符相同的那些单词将被集中打印在一起。

练习6-3 （教材第125页）

编写一个交叉引用程序，打印文档中所有单词的列表，并且每个单词还有一个列表，记录出现过该单词的行号。对the、and 等非实义单词不予考虑。

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

#define   MAXWORD   100

struct linklist {          /* linked list of line numbers    */
        int lnum;
        struct linklist *ptr;
};

struct tnode {                          /* the tree node:       */
        char *word;                     /* points to the text   */
        struct linklist *lines;         /* line numbers         */
        struct tnode *left;             /* left child           */
        struct tnode *right;            /* right child          */
};
```

```
struct tnode *addtreex(struct tnode *, char *, int);
int getword(char *, int);
int noiseword(char *);
void treexprint(struct tnode *);

/* cross-referencer                                          */
main()
{
     struct tnode *root;
     char word[MAXWORD];
     int linenum = 1;

     root = NULL;
     while (getword(word, MAXWORD) != EOF)
          if (isalpha(word[0]) && noiseword(word) == -1)
               root = addtreex(root, word, linenum);
          else if (word[0] == '\n')
               linenum++;
     treexprint(root);
     return 0;
}
struct tnode *talloc(void);
struct linklist *lalloc(void);
void addln(struct tnode *, int);

/* addtreex: add a node with w, at or below p                */
struct tnode *addtreex(struct tnode *p, char *w, int linenum)
{
     int cond;

     if (p == NULL) {          /* a new word has arrived     */
          p = talloc();        /* make a new word            */
          p->word = strdup(w);
          p->lines = lalloc();
          p->lines->lnum = linenum;
          p->lines->ptr = NULL;
          p->left = p->right = NULL;
     } else if ((cond = strcmp(w, p->word)) == 0)
          addln(p, linenum);
     else if (cond < 0)
          p->left = addtreex(p->left, w, linenum);
     else
          p->right = addtreex(p->right, w, linenum);
     return p;
}

/* addln: add a line number to the linked list               */
void addln(struct tnode *p, int linenum)
{
     struct linklist *temp;

     temp = p->lines;
     while (temp->ptr != NULL && temp->lnum != linenum)
          temp = temp->ptr;
     if (temp->lnum != linenum) {
```

```
            temp->ptr = lalloc();
            temp->ptr->lnum = linenum;
            temp->ptr->ptr = NULL;
      }
}

/* treexprint: in-order print of tree p                              */
void treexprint(struct tnode *p)
{
      struct linklist *temp;
      if (p != NULL) {
            treexprint(p->left);
            printf("%10s: ", p->word);
            for (temp = p->lines; temp != NULL; temp = temp->ptr)
                  printf("%4d ", temp->lnum);
            printf("\n");
            treexprint(p->right);
      }
}

/* lalloc: make a linklist node                                      */
struct linklist *lalloc(void)
{
      return (struct linklist *) malloc(sizeof(struct linklist));
}

/* noiseword: identify word as a noise word                          */
int noiseword(char *w)
{
      static char *nw[] = {
            "a",
            "an",
            "and",
            "are",
            "in",
            "is",
            "of",
            "or",
            "that",
            "the",
            "this",
            "to"
      };
      int cond, mid;
      int low = 0;
      int high = sizeof(nw) / sizeof(char *) - 1;

      while (low <= high) {
            mid = (low + high) / 2;
            if ((cond = strcmp(w, nw[mid])) < 0)
                  high = mid - 1;
            else if (cond > 0)
                  low = mid + 1;
            else
                  return mid;
```

```
        }
        return -1;
}
```

树中的每个节点都对应着一个不同的单词。树节点的结构如下所示：

word　　一个指针，指向有关单词的文本
lines　　一个指针，指向一个由行号构成的链表
left　　一个指针，指向本节点的左子节点
right　　一个指针，指向本节点的右子节点

行号链表的每个元素又是一个类型名为linklist的结构。每个结构包含一个行号和一个指针，指针指向链表中的下一个元素（链表最后一个元素的这个指针将指向NULL）。

函数addtreex是addtree函数（见教材123页）的改进版本，该函数把单词添加到变量名树中并把行号添加到相应的链表中。如果是一个新单词，就把保存在变量linenum中的当前行号赋值给链表中的第一个元素，如下所示：

```
p->lines->lnum = linenum;
```

如果单词已经在树中（即已经在文档中出现过），则：

```
((cond = strcmp(w, p->word)) == 0)
```

成立，我们就用函数addln把行号追加到链表的尾部。

addln函数首先检查链表中是否已经存在着相同的行号，如果没有，它将到达链表尾并遇到NULL，如下所示：

```
while (temp->ptr != NULL && temp->lnum != linenum)
        temp = temp->ptr;
```

如果这个行号没有在链表中出现过，addln函数就会把它追加到链表的尾部，如下所示：

```
if (temp->lnum != linenum) {
        temp->ptr = lalloc();
        temp->ptr->lnum = linenum;
        temp->ptr->ptr = NULL;
}
```

函数treexprint是treeprint函数（见教材第124页）的改进版本，它把树中的单词节点按字母表顺序打印出来。对于树中的每一个单词，这个函数将把这个单词和它出现在文档中的所有行号全部打印出来。

noiseword函数用来剔除文档中诸如“the”、“and”之类不需要统计的单词，这些单词都列在一个static数组中。对于需要进行统计的单词，这个函数将返回－1。你可以在数组nw[]中任意添加各种你不打算统计的单词，但前提是必须保持这个数组中的单词按ASCII字符升序排列。

为了记录行号，我们还修改了getword函数，使它能够返回换行符（'\n'），如下所示：

```
while (isspace(c = getch()) && c != '\n')
        ;
```

练习6-4　（教材第125页）

编写一个程序，根据单词的出现频率按降序打印输入的各个不同单词，并在每个单词的

前面标上它的出现次数。

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

#define   MAXWORD   100
#define   NDISTINCT 1000

struct tnode {                          /* the tree node:          */
     char *word;                        /* points to the text      */
     int count;                         /* number of occurrences */
     struct tnode *left;                /* left child              */
     struct tnode *right;               /* right child             */
};

struct tnode *addtree(struct tnode *, char *);
int getword(char *, int);
void sortlist(void);
void treestore(struct tnode *);

struct tnode *list[NDISTINCT];          /* pointers to tree nodes*/
int ntn = 0;                            /* number of tree nodes  */

/* print distinct words sorted in decreasing order of freq. */
main()
{
     struct tnode *root;
     char word[MAXWORD];
     int i;

     root = NULL;
     while (getword(word, MAXWORD) != EOF)
          if (isalpha(word[0]))
               root = addtree(root, word);
     treestore(root);
     sortlist();
     for (i = 0; i < ntn; i++)
          printf("%2d:%20s\n", list[i]->count, list[i]->word);
     return 0;
}

/* treestore: store in list[] pointers to tree nodes        */
void treestore(struct tnode *p)
{
     if (p != NULL) {
          treestore(p->left);
          if (ntn < NDISTINCT)
               list[ntn++] = p;
          treestore(p->right);
     }
}

/* sortlist: sort list of pointers to tree nodes            */
void sortlist()
{
     int gap, i, j;
     struct tnode *temp;
```

```
    for (gap = ntn/2; gap > 0; gap /= 2)
        for (i = gap; i < ntn; i++)
            for (j = i-gap; j >= 0; j -= gap) {
                if ((list[j]->count) >= (list[j+gap]->count))
                    break;
                temp = list[j];
                list[j] = list[j+gap];
                list[j+gap] = temp;
            }
}
```

常数NDISTINCT对不同单词的最大数目设置了限制。tnode与教材第123页上使用的同名结构完全一样。list是一个指针数组，其中的每个指针都指向一个tnode类型的结构。变量ntn中保存着树节点的个数。

程序读入每个单词并把它放到树中。函数treestore把那些指向tnode结构的指针保存到数组list中。函数sortlist是shellsort函数（见教材第52页）的改进版本，它按单词出现次数由高到低的顺序对数组list进行排序。

练习6-5 （教材第127页）

编写一个函数，它将从由lookup和install维护的表中删除一个变量名及其定义。

```
unsigned hash(char *);

/* undef: remove a name and definition from the table        */
void undef(char *s)
{
    int h;
    struct nlist *prev, *np;

    prev = NULL;
    h = hash(s);                /* hash value of string s     */
    for (np = hashtab[h]; np != NULL; np = np->next) {
        if (strcmp(s, np->name) == 0)
            break;
        prev = np;              /* remember previous entry    */
    }
    if (np != NULL) {           /* found name                 */
        if (prev == NULL)       /* first in the hash list ?   */
            hashtab[h] = np->next;
        else                    /* elsewhere in the hash list */
            prev->next = np->next;
        free((void *) np->name);
        free((void *) np->defn);
        free((void *) np);      /* free allocated structure   */
    }
}
```

函数undef将在表中查找字符串s。当undef找到字符串s时，它将跳出for循环，如下所示：

```
if (strcmp(s, np->name) == 0)
    break;
```

如果字符串s不在表中，这个for循环将在指针np变成NULL时终止。

指针数组hashtab中的各个元素分别指向一个链表的开头。如果指针np不为NULL，就说明其所指向的那个表中存在一组符合清除要求的变量名和定义；此时，指针np指向将被清除的那个数据项，而指针prev则指向出现在np位置之前的那个数据项。如果prev是NULL，就说明np是以hashtab[h]开头的那个链表的第一个元素，如下所示：

```
if (prev == NULL)
     hashtab[h] = np->next;
else
     prev->next = np->next;
```

在清除了np所指的数据项之后，我们还要通过free函数（参见教材第147页）把该数据项的名字、定义及其存储结构都释放，如下所示：

```
free((void *) np->name);
free((void *) np->defn);
free((void *) np);
```

练习6-6 （教材第127页）

以本节介绍的函数为基础，编写一个适合C语言程序使用的#define处理器的简单版本（即无参数的情况）。你会发现getch和ungetch函数非常有用。

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

#define   MAXWORD   100

struct nlist {                  /* table entry:                 */
     struct nlist *next;        /* next entry in the chain      */
     char *name;                /* defined name                 */
     char *defn;                /* replacement text             */
};

void error(int, char*);
int getch(void);
void getdef(void);
int getword(char *, int);
struct nlist *install(char *, char*);
struct nlist *lookup(char *);
void skipblanks(void);
void undef(char *);
void ungetch(int);
void ungets(char *);

/* simple version of #define processor                          */
main()
{
     char w[MAXWORD];
     struct nlist *p;

     while (getword(w, MAXWORD) != EOF)
```

```
            if (strcmp(w, "#") == 0) /* beginning of direct.  */
                getdef();
            else if (!isalpha(w[0]))
                printf("%s", w);    /* cannot be defined     */
            else if ((p = lookup(w)) == NULL)
                printf("%s", w);    /* not defined           */
            else
                ungets(p->defn);    /* push definition       */
        return 0;
}

/* getdef: get definition and install it                    */
void getdef(void)
{
        int c, i;
        char def[MAXWORD], dir[MAXWORD], name[MAXWORD];

        skipblanks();
        if (!isalpha(getword(dir, MAXWORD)))
            error(dir[0],
                "getdef: expecting a directive after #");
        else if (strcmp(dir, "define") == 0) {
            skipblanks();
            if (!isalpha(getword(name, MAXWORD)))
                error(name[0],
                    "getdef: non-alpha - name expected");
            else {
                skipblanks();
                for (i = 0; i < MAXWORD-1; i++)
                    if ((def[i] = getch()) == EOF ||
                                        def[i] == '\n')
                        break;      /* end of definition     */
                def[i] = '\0';
                if (i <= 0)         /* no definition ?       */
                    error('\n', "getdef; incomplete define");
                else                /* install definition    */
                    install(name, def);
            }
        } else if (strcmp(dir, "undef") == 0) {
            skipblanks();
            if (!isalpha(getword(name, MAXWORD)))
                error(name[0], "getdef: non-alpha in undef");
            else
                undef(name);
        } else
            error(dir[0],
                "getdef: expecting a directive after #");
}

/* error: print error message and skip the rest of the line */
void error(int c, char *s)
{
        printf("error: %s\n", s);
        while (c != EOF && c != '\n')
            c = getch();
}
```

```
/* skipblanks: skip blank and tab characters                    */
void skipblanks(void)
{
     int c;

     while ((c = getch()) == ' ' || c == '\t')
          ;
     ungetch(c);
}
```

主函数控制着全局。define和undef指令必须跟在一个#后面，面函数getdef也正是利用这一点来解析它们的。如果getword函数返回的字符不是一个字母，就不可能对那个单词做出定义，程序将报告出错并把那个单词打印出来；否则，程序将开始搜索可能与该单词配对的定义（它可能存在，也可能不存在）。如果该单词确实有一个配对的定义，函数unget（参见练习4-7）将把它们按逆序重新压回输入流。

函数getdef能够处理下面这两种指令：

```
#define    name       definition
#undef     name
```

其中，name是一个由字母或数字字符构成的变量名，definition是这个变量名的定义。

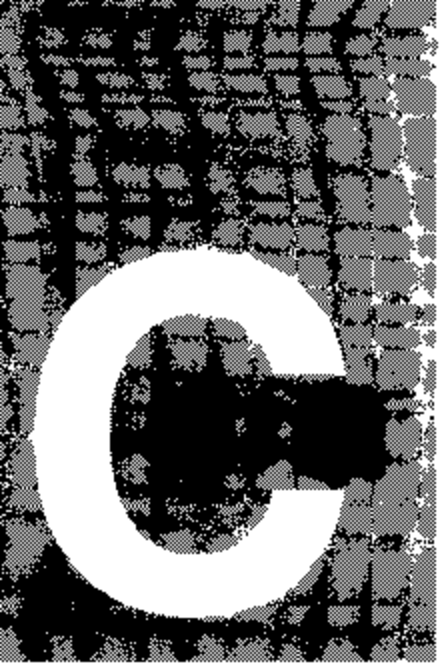
如果遇到的是define指令，下面这个循环：

```
for (i = 0; i < MAXWORD-1; i++)
     if ((def[i] = getch()) == EOF ||
                              def[i] == '\n')
          break;
```

将逐个字符地把变量名的定义收集到一起，直到到达行尾（'\n'）或文件尾（EOF）为止。如果变量名有一个配对的定义，getdef函数将会调用install函数（参见教材127页）把它添加到表中。

如果遇到的是undef指令，程序将从表中删除指定的变量名（参见练习6-5）。

为了让程序输出与输入数据相似，我们还修改了getword函数，使它能够返回空格。



输入与输出

练习7-1 （教材第135页）

编写一个程序，根据它自身被调用时存放在argv[0]中的名字，实现将大写字母转换为小写字母或将小写字母转换为大写字母的功能。

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

/* lower: converts upper case to lower case                    */
/* upper: converts lower case to upper case                    */
main(int argc, char *argv[])
{
     int c;

     if (strcmp(argv[0], "lower") == 0)
          while ((c = getchar()) != EOF)
               putchar(tolower(c));
     else
          while ((c = getchar()) != EOF)
               putchar(toupper(c));
     return 0;
}
```

如果用名字lower来调用这个程序，它将执行把大写字母转换为小写字母的功能；如果用其他名字来调用这个程序，它将执行把小写字母转换为大写字母的功能。

如果argv[0]是字符串lower，strcmp函数的返回值就将是0。

在UNIX系统中，下面这条语句：

```
if (strcmp(argv[0], "lower") == 0)
```

中的argv[0]就是用户在命令行中输入的程序名。在其他一些操作系统中，argv[0]是程序的完整路径（而不是仅限于用户亲自输入的程序名）。

这个程序使用了来自头文件<ctype.h>的tolower和toupper函数。

练习7-2 （教材第136页）

编写一个程序，以合理的方式打印任何输入。该程序至少能根据用户的习惯以八进制或十六进制打印非显示字符，并截断长文本行。

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

```
#define   MAXLINE  100    /* max number of chars in one line */
#define   OCTLEN     6    /* length of an octal value        */

/* print arbitrary input in a sensible way                    */
main()
{
     int c, pos;
     int inc(int pos, int n);

     pos = 0;                    /* position in the line      */
     while ((c = getchar()) != EOF)
          if (iscntrl(c) || c == ' ') {
                                 /* non-graphic or blank      */
               pos = inc(pos, OCTLEN);
               printf(" \\%03o ", c);
                                 /* newline character ?       */
               if (c == '\n') {
                    pos = 0;
                    putchar('\n');
               }
          } else {               /* graphic character         */
               pos = inc(pos, 1);
               putchar(c);
          }
     return 0;
}

/* inc: increment position counter for output                 */
int inc(int pos, int n)
{
     if (pos + n < MAXLINE)
          return pos+n;
     else {
          putchar('\n');
          return n;
     }
}
```

每个输出行最多能容纳MAXLINE个字符。宏iscntrl来自头文件<ctype.h>，它的用途是寻找非显示字符——删除控制符（八进制0177）和普通控制字符（小于八进制040）。空格也被看作是一个非显示字符。非显示字符将打印为长度是OCTLEN个字符的八进制数字（数字前有一个空格和一个反斜杠\，数字后又有一个空格）。在遇到换行符时，变量pos重新设置为0，如下所示：

```
if (c == '\n') {
     pos = 0;
     putchar('\n');
}
```

inc函数用来对输出行上的写位置（pos+n）进行计算，如果输出行上的剩余空间不足n个字符，就在当前位置插入一个换行符。

练习7-3 （教材第137页）

改写minprintf函数，使它能完成printf函数的更多功能。

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

#define   LOCALFMT  100

/* minprintf: minimal printf with variable argument list    */
void minprintf(char *fmt, ...)
{
     va_list ap;          /* points to each unnamed arg      */
     char *p, *sval;
     char localfmt[LOCALFMT];
     int i, ival;
     unsigned uval;
     double dval;

     va_start(ap, fmt);  /* make ap point to 1st unnamed arg*/
     for (p = fmt; *p; p++) {
          if (*p != '%') {
               putchar(*p);
               continue;
          }
          i = 0;
          localfmt[i++] = '%';              /* start local fmt  */
          while (*(p+1) && !isalpha(*(p+1)))
               localfmt[i++] = *++p;        /* collect chars    */
          localfmt[i++] = *(p+1);           /* format letter    */
          localfmt[i]   = '\0';
          switch(*++p) {                    /* format letter    */
          case 'd':
          case 'i':
               ival = va_arg(ap, int);
               printf(localfmt, ival);
               break;
          case 'x':
          case 'X':
          case 'u':
          case 'o':
               uval = va_arg(ap, unsigned);
               printf(localfmt, uval);
               break;
          case 'f':
               dval = va_arg(ap, double);
               printf(localfmt, dval);
               break;
          case 's':
               sval = va_arg(ap, char *);
               printf(localfmt, sval);
               break;
          default:
               printf(localfmt);
               break;
          }
     }
     va_end(ap);                            /* clean up         */
}
```

minprintf函数对自己的输入参数表加以分析，然后调用printf来完成具体的打印输出功能。

为了更多地实现一些printf函数的功能．我们使用了一个字符数组localfmt来保存从minprintf函数的输入参数表中分析出来的%字符和其他字符，直到遇到一个字母字符（即输出格式控制字符）为止，最终得到的localfmt将被用作printf的格式参数。

练习7-4 （教材第140页）

类似于上一节中的函数minprintf，编写scanf函数的一个简化版本。

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

#define   LOCALFMT  100

/* minscanf: minimal scanf with variable argument list      */
void minscanf(char *fmt, ...)
{
     va_list ap;          /* points to each unnamed arg      */
     char *p, *sval;
     char localfmt[LOCALFMT];
     int c, i, *ival;
     unsigned *uval;
     double *dval;

i = 0;                    /* index for localfmt array        */
va_start(ap, fmt);  /* make ap point to 1st unnamed arg*/
for (p = fmt; *p; p++) {
     if (*p != '%') {
          localfmt[i++] = *p;       /* collect chars     */
          continue;
     }
     localfmt[i++] = '%';           /* start format      */
     while (*(p+1) && !isalpha(*(p+1)))
          localfmt[i++] = *++p;     /* collect chars     */
     localfmt[i++] = *(p+1);        /* format letter     */
     localfmt[i]   = '\0';
     switch(*++p) {                 /* format letter     */
     case 'd':
     case 'i':
          ival = va_arg(ap, int *);
          scanf(localfmt, ival);
          break;
     case 'x':
     case 'X':
     case 'u':
     case 'o':
          uval = va_arg(ap, unsigned *);
          scanf(localfmt, uval);
          break;
    case 'f':
         dval = va_arg(ap, double *);
```

```
                scanf(localfmt, dval);
                break;
            case 's':
                sval = va_arg(ap, char *);
                scanf(localfmt, sval);
                break;
            default:
                scanf(localfmt);
                break;
            }
            i = 0;                              /* reset index      */
        }
        va_end(ap);                             /* clean up         */
}
```

minscanf函数在很多地方模仿了minprintf函数。它先对输入的格式字符串中的字符进行收集直到在%找到一个字母字符。然后再把字符数组localfmt和有关指针传递给scanf函数。

scanf函数的参数必须是指针：一个指针指向格式字符串，另一个指针则指向用于接收scanf输入值的变量。我们先通过va_arg函数得到指针的值，再把它复制为一个局部指针，最后通过调用scanf把输入值读到用户指定的变量中去。

练习7-5 （教材第140页）

改写第4章中的后缀计算器程序，用scanf函数和（或）sscanf函数实现输入以及数的转换。

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

#define   NUMBER    '0'         /* signal that a number was found  */

/* getop: get next operator or numeric operand                      */
int getop(char s[])
{
    int c, i, rc;
    static char lastc[] = " ";

    sscanf(lastc, "%c", &c);
    lastc[0] = ' ';                         /* clear last character     */
    while ((s[0] = c) == ' ' || c == '\t')
        if (scanf("%c", &c) == EOF)
            c = EOF;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;                           /* not a number             */
    i = 0;
    if (isdigit(c))                         /* collect integer part     */
        do {
            rc = scanf("%c", &c);
            if (!isdigit(s[++i] = c))
                break;
        } while (rc != EOF);
    if (c == '.')                           /* collect fraction part    */
        do {
```

```
                rc = scanf("%c", &c);
                if (!isdigit(s[++i] = c))
                    break;
            } while (rc != EOF);
        s[i] = '\0';
        if (rc != EOF)
            lastc[0] = c;
        return NUMBER;
}
```

我们只需对函数getop（参见教材第67页）做一些修改。

在调用getop函数时，字符总是跟在一个数字的后面出现。lastc是具有两个元素的静态数组，用于记录最后读入的字符（sscanf读入一个字符串）。

函数调用

```
sscanf(lastc, "%c", &c)
```

把lastc[0]中的字符读到变量c中。你可以用赋值语句

```
c = lastc[0]
```

来代替之。

scanf的返回值是成功地得到匹配和赋值的输入项的个数（参见教材第138页），它在读到文件尾时将返回EOF。

此外，因为题目要求我们必须使用scanf来处理输入，所以我们要把表达式

```
isdigit(s[++i] = c = getch())
```

替换为：

```
rc = scanf("%c", &c);
if (!isdigit(s[++i] = c))
    break;
```

即调用scanf来读取字符，把字符赋值给字符串s，再测试它是不是一个数字。

注意，当scanf读到EOF时，它将不会改变变量c的值，这正是我们增加下面这条测试的原因：

```
rc != EOF
```

如果只是利用scanf来每次读入一个字符，那么将不会给getop带来多大的性能改善。

下面是本题的另一个解答方法：

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

#define   NUMBER    '0'        /* signal that a number was found  */

/* getop: get next operator or numeric operand                    */
int getop(char s[])
{
    int  c, rc;
    float f;

    while ((rc = scanf("%c", &c)) != EOF)
        if ((s[0] = c) != ' ' && c != '\t')
```

```
            break;
        s[1] = '\0';
        if (rc == EOF)
            return EOF;
        else if (!isdigit(c) && c != '.')
            return c;
        ungetc(c, stdin);
        scanf("%f", &f);
        sprintf(s, "%f", f);
        return NUMBER;
}
```

我们先通过一个while循环以每次读一个字符的方法跳过空格和制表符。这个while循环在读到文件尾标记EOF时也将终止。

如果读到的是一个数字字符或者是一个小数点，我们就用库函数ungetc把它重新压回输入缓冲区，然后再完整地读入整个数字。因为getop返回的是一个浮点数值，所以我们还要用sprintf把变量f中的浮点数值转换为变量s中的一个字符串。

练习7-6 （教材第145页）

编写一个程序，比较两个文件并打印它们第一个不相同的行。

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

#define   MAXLINE   100

/* comp: compare two files, printing first different line          */
main(int argc, char *argv[])
{
    FILE *fp1, *fp2;
    void filecomp(FILE *fp1, FILE *fp2);

    if (argc != 3) {    /* incorrect number of arguments ?        */
        fprintf(stderr, "comp: need two file names\n");
        exit(1);
    } else {
        if ((fp1 = fopen(*++argv, "r")) == NULL) {
            fprintf(stderr, "comp: can't open %s\n", *argv);
            exit(1);
        } else if ((fp2 = fopen(*++argv, "r")) == NULL) {
            fprintf(stderr, "comp: can't open %s\n", *argv);
            exit(1);
        } else {        /* found and opened files to be compared*/
            filecomp(fp1, fp2);
            fclose(fp1);
            fclose(fp2);
            exit(0);
        }
    }
}

/* filecomp: compare two files - a line at a time                  */
void filecomp(FILE *fp1, FILE *fp2)
```

```
{
      char line1[MAXLINE], line2[MAXLINE];
      char *lp1, *lp2;

      do {
            lp1 = fgets(line1, MAXLINE, fp1);
            lp2 = fgets(line2, MAXLINE, fp2);
            if (lp1 == line1 && lp2 == line2) {
                  if (strcmp(line1, line2) != 0) {
                        printf("first difference in line\n%s\n", line1);
                        lp1 = lp2 = NULL;
                  }
            } else if (lp1 != line1 && lp2 == line2)
                  printf("end of first file at line\n%s\n", line2);
            else if (lp1 == line1 && lp2 != line2)
                  printf("end of second file at line\n%s\n", line1);
      } while (lp1 == line1 && lp2 == line2);
}
```

参数个数应该等于3：一个程序名和两个文件名。程序打开两个文件，然后调用函数filecomp以每次一行的方式对它们进行比较。

函数filecomp先从两个文件中分别读入一行。函数fgets的返回值是一个指针，它指向刚读入的那一行，如果到达文件尾，fgets就返回NULL。如果lp1和lp2指向两个文件相应的行且两个文件都没有结束，filecomp将对这两行进行比较。若这两行内容不匹配，filecomp就把它们出现差异的这一行内容打印出来。

如果lp1或lp2不再指向相应的行，就说明有一个文件结束了（读到了EOF），也就是说，这两个文件从此位置开始出现了差异。

如果lp1和lp2保持同步前进直到两个文件同时结束，就说明这两个文件完全相同。

练习7-7 （教材第145页）

修改第5章的模式查找程序，使它从一个命名文件的集合中读取输入（有文件名参数时），如果没有文件名参数，则从标准输入中读取输入。当发现一个匹配行时，是否应该将相应的文件名打印出来？

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

#define   MAXLINE   1000      /* maximum input line length  */

/* find: print lines that match pattern from 1st argument   */
main(int argc, char *argv[])
{
      char pattern[MAXLINE];
      int c, except = 0, number = 0;
      FILE *fp;
      void fpat(FILE *fp, char *fname, char *pattern,
            int except, int number);

      while (--argc > 0 && (*++argv)[0] == '-')
            while (c = *++argv[0])
```

```
            switch (c) {
            case 'x':
                 except = 1;
                 break;
            case 'n':
                 number = 1;
                 break;
            default:
                 printf("find: illegal option %c\n", c);
                 argc = 0;
                 break;
            }
if (argc >= 1)
     strcpy(pattern, *argv);
else {
     printf("Usage: find [-x] [-n] pattern [file ...]\n");
     exit(1);
}
if (argc == 1)              /* read standard input          */
     fpat(stdin, "", pattern, except, number);
else
     while (--argc > 0)  /* get a named file              */
          if ((fp = fopen(*++argv, "r")) == NULL) {
               fprintf(stderr, "find: can't open %s\n",
                       *argv);
               exit(1);
          } else {         /* named file has been opened */
               fpat(fp, *argv, pattern, except, number);
               fclose(fp);
          }
     return 0;
}

/* fpat: find pattern                                          */
void fpat(FILE *fp, char *fname, char *pattern,
          int except, int number)
{
     char line[MAXLINE];
     long lineno = 0;

     while (fgets(line, MAXLINE, fp) != NULL) {
          ++lineno;
          if ((strstr(line, pattern) != NULL) != except) {
               if (*fname)                    /* have a file name */
                    printf("%s - ", fname);
               if (number)                    /* print line number*/
                    printf("%ld: ", lineno);
               printf("%s", line);
          }
     }
}
```

主程序对可选参数的处理方法源于第5章（参见教材第101页）。然后，它至少还需要一个参数——模式。如果在模式后面没有给出文件名，它将使用系统的标准输入做为其输入源；否则，它就会打开一个指定文件。无论怎样，它都将调用函数fpat。

fpat函数的大部分代码与第5章中的原始程序很相似。它每次读入一行，直到fgets函数（参见教材第144页）返回NULL为止。fpat会根据有关参数在每一行中寻找有无给定的匹配模式，下面是各种可能出现的情况：

(strstr (line, pattern) != NULL)	!=	except	结果
0 （没有找到匹配模式）	!=	0（未设定）	false（假）
1 （找到匹配模式）	!=	0（未设定）	true（真）
0 （没有找到匹配模式）	!=	1（设定）	true（真）
1 （找到匹配模式）	!=	1（设定）	false（假）

当这个表达式的结果为“真”时，fpat将把文件名（如果不是标准输入的话）、行号（如果有命令行参数-n）以及这一行的内容依次打印出来。

练习7-8 （教材第145页）

编写一个程序，以打印一个文件集合，每个文件从新的一页开始打印，并且打印每个文件相应的标题和页数。

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

#define   MAXBOT    3    /* maximum # lines at bottom page   */
#define   MAXHDR    5    /* maximum # lines at head of page  */
#define   MAXLINE   100  /* maximum size of one line         */
#define   MAXPAGE   66   /* maximum # lines on one page      */

/* print: print files - each new one on a new page           */
main(int argc, char *argv[])
{
     FILE *fp;
     void fileprint(FILE *fp, char *fname);

     if (argc == 1)       /* no args; print standard input   */
          fileprint(stdin, " ");
     else                 /* print file(s)                   */
          while (--argc > 0)
               if ((fp = fopen(*++argv, "r")) == NULL) {
                    fprintf(stderr,
                         "print: can't open %s\n", *argv);
                    exit(1);
               } else {
                    fileprint(fp, *argv);
                    fclose(fp);
               }
     return 0;
}

/* fileprint: print file fname                               */
void fileprint(FILE *fp, char *fname)
{
     int lineno, pageno = 1;
     char line[MAXLINE];
     int heading(char *fname, int pageno);
```

```
        lineno = heading(fname, pageno++);
        while (fgets(line, MAXLINE, fp) != NULL) {
                if (lineno == 1) {
                        fprintf(stdout, "\f");
                        lineno = heading(fname, pageno++);
                }
                fputs(line, stdout);
                if (++lineno > MAXPAGE - MAXBOT)
                        lineno = 1;
        }
        fprintf(stdout, "\f");   /* page eject after the file  */
}

/*heading: put heading and enough blank lines                    */
int heading(char *fname, int pageno)
{
        int ln = 3;

        fprintf(stdout, "\n\n");
        fprintf(stdout, "%s   page %d\n", fname, pageno);
        while (ln++ < MAXHDR)
                fprintf(stdout, "\n");
        return ln;
}
```

这个程序的功能类似于cat程序（见教材第143页）。

函数fileprint需要两个指针做为输入参数：其中一个指向一个打开的文件，另一个则指向该文件的文件名字符串（若是以标准输入为输入源，则指向一个空字符串）。函数fileprint读入文本行并打印输出。

字符\f是一个换页符。

变量lineno记录着已经在某页纸上打印了多少行。页面最大长度是MAXPAGE行。每当lineno变为1时，fileprint就将输出一个换页符，在新页面上打印标题，然后重置lineno。此外，在每个文件的最后一页，我们也要输出一个换页符。

函数heading先打印文件名和页码，然后用足够的换行符在每页的开头留出总共MAXHDR行。

MAXBOT是为每页末尾保留的空白行数。

练习7-9 （教材第148页）

类似于isupper这样的函数可以通过某种方式实现以达到节省空间或时间的目的。考虑节省空间或时间的实现方式。

```
/* isupper: return 1 (true) if c is an upper case letter    */
int isupper(char c)
{
        if (c >= 'A' && c <= 'Z')
                return 1;
        else
                return 0;
}
```

这个版本的isupper函数用了一个简单的if-else结构来检查字符的大小写情况。如果被测字符落在ASCII大写字母的取值范围内，isupper函数返回1（真）；否则，返回0（假）。这个版本的isupper函数空间利用率较高。

```
#define   isupper(c)     ((c) >= 'A' && (c) <= 'Z') ? 1 : 0
```

这个版本的isupper函数时间效率较高，但要使用较多的空间。

说它节约时间，是因为它没有函数调用方面的开销；说它要使用较多的空间，是因为它是一个宏，每次执行都要进行展开。

使用宏的isupper版本需要注意由于参数可能会被求值两次而带来的潜在问题。例如：

```
char *p = "This is a string";

if (isupper(*p++))
    ...
```

这个宏将被展开为：

```
((*p++) >= 'A' && (*p++) <= 'Z') ? 1 : 0
```

根据*p的取值情况，这个表达式可能对指针p做两次递增操作。但是，如果isupper是一个函数，就不存在对指针p做两次递增操作的隐患——因为函数的参数只会被求值一次。

指针p的第二次递增操作并不是我们所期望的，只会导致不正确的结果。下面是一种值得参考的解决方案：

```
char *p = "This is a string";

if (isupper(*p))
    ...
p++;
```

对那些有可能对参数进行多次求值的宏，一定要提高警惕。头文件<ctype.h>中的两个宏toupper和tolower是很好的学习例子。

UNIX系统接口

练习8-1 （教材第153页）

用read、write、open和close系统调用代替标准库中功能等价的函数，重写第7章的cat程序，并通过实验比较两个版本的相对执行速度。

```
#include  <stdio.h>
#include  <fcntl.h>
#include  "syscalls.h"

void error(char *fmt, ...);

/* cat: concatenate files - read / write / open / close      */
main(int argc, char *argv[])
{
     int fd;
     void filecopy(int ifd, int ofd);
     if (argc == 1)       /* no args; copy standard input     */
          filecopy(0, 1);
     else
          while (--argc > 0)
               if ((fd = open(*++argv, O_RDONLY)) == -1)
                    error("cat: can't open %s", *argv);
               else {
                    filecopy(fd, 1);
                    close(fd);
               }
     return 0;
}
/* filecopy: copy file ifd to file ofd                        */
void filecopy(int ifd, int ofd)
{
     int n;
     char buf[BUFSIZ];

     while ((n = read(ifd, buf, BUFSIZ)) > 0)
          if (write(ofd, buf, n) != n)
               error("cat: write error");
}
```

语句

```
if ((fd = open(*++argv, O_RDONLY)) == -1)
```

以读方式打开一个文件并返回一个文件描述符（一个整数）；如果发生错误，则返回-1。

filecopy函数的功能是使用文件描述符ifd读入BUFSIZ个字符。read函数的返回值是它实际读入的字符的字节数：这个字节计数值大于0，通常表示没有出错；如果它等于0，表示读到文件尾；如果它等于-1，就表明读操作出错。write函数的功能是写出n个字节，如果实际写出的字节（write的返回值）与要求它写出的字节（这里是n）不符，就说明写操作出现了错误。

error是教材第153页上的函数。

这个版本要比教材第7章中的原始版本快大约两倍。

练习8-2 （教材第157页）

用字段代替显式的按位操作，重写函数fopen和_fillbuf。比较相应代码的长度和执行速度。

```
#include  <fcntl.h>
#include  "syscalls.h"
#define   PERMS     0666      /* RW for owner, group, others    */

/* fopen: open file, return file ptr                            */
FILE *fopen(char *name, char *mode)
{
     int fd;
     FILE *fp;

     if (*mode != 'r' && *mode != 'w' && *mode != 'a')
          return NULL;
     for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
          if (fp->flag.is_read == 0 && fp->flag.is_write == 0)
               break;               /* found free slot          */
     if (fp >= _iob + OPEN_MAX)
          return NULL;              /* no free slots            */

     if (*mode == 'w')              /* create file              */
          fd = creat(name, PERMS);
     else if (*mode == 'a') {
          if ((fd = open(name, O_WRONLY, 0)) == -1)
               fd = creat(name, PERMS);
          lseek(fd, 0L, 2);
     } else
          fd = open(name, O_RDONLY, 0);
     if (fd == -1)                  /* couldn't access name     */
          return NULL;
     fp->fd = fd;
     fp->cnt = 0;
     fp->base = NULL;
     fp->flag.is_unbuf = 0;
     fp->flag.is_buf = 1;
     fp->flag.is_eof = 0;
     fp->flag.is_err = 0;
     if (*mode == 'r') {            /* read                     */
          fp->flag.is_read = 1;
          fp->flag.is_write = 0;
     } else {                       /* write                    */
          fp->flag.is_read = 0;
```

```
            fp->flag.is_write = 1;
        }
        return fp;
}

/* _fillbuf: allocate and fill input buffer                    */
int _fillbuf(FILE *fp)
{
        int bufsize;

        if (fp->flag.is_read == 0 ||
            fp->flag.is_eof  == 1 ||
            fp->flag.is_err  == 1 )
             return EOF;
        bufsize = (fp->flag.is_unbuf == 1) ?  1 : BUFSIZ;
        if (fp->base == NULL)                /* no buffer yet      */
             if ((fp->base = (char *) malloc(bufsize)) == NULL)
                  return EOF;                /* can't get buffer   */
        fp->ptr = fp->base;
        fp->cnt = read(fp->fd, fp->ptr, bufsize);
        if (--fp->cnt < 0) {
             if (fp->cnt == -1)
                  fp->flag.is_eof = 1;
             else
                  fp->flag.is_err = 1;
             fp->cnt = 0;
             return EOF;
        }
        return (unsigned char) *fp->ptr++;
}
```

结构struct _iobuf的类型定义（即typedef）见教材第155页；_iobuf有一个成员变量是

```
int  flag;
```

按照题目要求，我们对变量flag按位字段方式重新进行了定义，如下所示：

```
struct flag_field {
        unsigned is_read  : 1;
        unsigned is_write : 1;
        unsigned is_unbuf : 1;
        unsigned is_buf   : 1;
        unsigned is_eof   : 1;
        unsigned is_err   : 1;
};
```

在原来的fopen函数中有下面的if语句：

```
if ((fp->flag & (_READ | _WRITE)) == 0)
        break;
```

这条语句对_READ和_WRITE标志进行了**OR**操作，如下所示：

```
(_READ          |          _WRITE)
        01      |           02          (八进制)
        01      |           10          (二进制)
                11                      (运算结果)
```

这意味着：只有当flag的两个最低位全都为0（不读也不写）时，这条if语句才会成真。这条语句将核实这样一个情况：_iob中的某个元素没有被用来完成读或写操作。

在采用位字段的方案中，我们必须明确地对这一情况加以测试，如下所示：

```
if (fp->flag.is_read == 0 && fp->flag.is_write == 0)
    break;
```

接下来的修改之处是对位字段进行初始化部分，如下所示：

```
fp->flag.is_unbuf = 0;
fp->flag.is_buf = 1;
fp->flag.is_eof = 0;
fp->flag.is_err = 0;
```

再往后，原来的fopen函数中有下面语句：

```
fp->flag = (*mode == 'r') ?  _READ : _WRITE;
```

其作用是根据mode的取值情况来设置flag：如果mode取值为“r”，就将flag设置为_READ；否则，将flag设置为_WRITE。

在采用位字段的方案中，如果mode取值为“r”，我们需要把is_read位设置为1；否则，就需要把is_write位设置为1，如下所示：

```
if (*mode == 'r') {
    fp->flag.is_read = 1;
    fp->flag.is_write = 0;
} else {
    fp->flag.is_read = 0;
    fp->flag.is_write = 1;
}
```

我们对_fillbuf函数也做了类似的修改。

首先，_fillbuf函数在遇到以下几种情况时将返回EOF：（1）文件不是为了进行读操作而打开；（2）已经到达文件尾；（3）检测到一个执行错误。下面是原来的_fillbuf函数中用于检查这些条件的if语句：

```
if ((fp->flag & (_READ|_EOF|_ERR)) != _READ)
```

我们把它修改为使用位字段的条件判断语句：

```
if (fp->flag.is_read == 0 ||
    fp->flag.is_eof  == 1 ||
    fp->flag.is_err  == 1)
```

接着，我们还需要把原来的_fillbuf函数中的语句

```
bufsize = (fp->flag & _UNBUF) ?  1 : BUFSIZ;
```

修改为

```
bufsize = (fp->flag.is_unbuf == 1) ?  1 : BUFSIZ;
```

把原来的_fillbuf函数中的语句

```
    fp->flag |= _EOF;
else
    fp->flag |= _ERR;
```

修改为

```
        fp->flag.is_eof = 1;
    else
        fp->flag.is_err = 1;
```

新方案的代码规模增加了不少，函数的执行速度也变慢了。位字段操作不仅要依赖于计算机硬件，还会降低执行速度。

练习8-3 （教材第157页）

请设计并编写函数_flushbuf、fflush和fclose。

```
#include  "syscalls.h"

/* _flushbuf: allocate and flush output buffer                          */
int _flushbuf(int x, FILE *fp)
{
    unsigned nc;                        /* # of chars to flush    */
    int bufsize;                        /* size of buffer alloc.  */

    if (fp < _iob || fp >= _iob + OPEN_MAX)
        return EOF;                     /* error: invalid pointer*/
    if ((fp->flag & (_WRITE | _ERR)) != _WRITE)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL) {             /* no buffer yet          */
        if ((fp->base = (char *) malloc(bufsize)) == NULL) {
            fp->flag |= _ERR;
            return EOF;                 /* can't get buffer       */
        }
    } else {                            /* buffer already exists */
        nc = fp->ptr - fp->base;
        if (write(fp->fd, fp->base, nc) != nc) {
            fp->flag |= _ERR;
            return EOF;                 /* error: return EOF      */
        }
    }
    fp->ptr = fp->base;                 /* beginning of buffer    */
    *fp->ptr++ = (char) x;              /* save current char      */
    fp->cnt = bufsize - 1;
    return x;
}

/* fclose: close file                                                   */
int fclose(FILE *fp)
{
    int rc;                             /* return code            */

    if ((rc = fflush(fp)) != EOF) {     /* anything to flush?     */
        free(fp->base);                 /* free allocated buffer */
        fp->ptr = NULL;
        fp->cnt = 0;
        fp->base = NULL;
        fp->flag &= ~(_READ | _WRITE);
    }
```

```
        return rc;
}
/* fflush: flush buffer associated with file fp                    */
int fflush(FILE *fp)
{
        int rc = 0;

        if (fp < _iob || fp >= _iob + OPEN_MAX)
                return EOF;                     /* error: invalid pointer*/
        if (fp->flag & _WRITE)
                rc = _flushbuf(0, fp);
        fp->ptr = fp->base;
        fp->cnt = (fp->flag & _UNBUF) ?  1 : BUFSIZ;
        return rc;
}
```

当文件不是为了写操作而打开的或者发生错误时，_flushbuf函数将返回一个EOF，如下所示：

```
if ((fp->flag & (_WRITE | _ERR)) != _WRITE)
        return EOF;
```

如果此时还没有缓冲区，我们将像在_fillbuf函数里那样（参见教材第157页）分配一个缓冲区；如果缓冲区已经存在，我们就把它里面的字符全部清除（并写入相应文件）。

下一步是把输入参数保存到这个缓冲区中：

```
*fp->ptr++ = (char) x;
```

因为要给刚刚保存的字符(char) x留一个位置，所以缓冲区能够容纳的字符个数（fp->cnt）将等于缓冲区的长度值减去1。

函数fclose需要调用fflush。如果文件是为了写操作而打开的，就需要把缓冲区中的数据写入到有关文件中去。flose将对_iobuf结构的各成员变量进行重置，这是为了保证今后的fopen调用所分配到的_iobuf结构中不会有无意义的值。如果操作成功，fclose的返回代码将是0。

fflush先检查文件指针是否合法。然后，如果该文件是为了写操作而打开的，就调用_flushbuf把缓冲区里的数据全部写入文件。接下来，fflush重置ptr和cnt，再返回rc。

练习8-4 （教材第157页）

标准库函数

```
int fseek(FILE *fp, long offset, int origin)
```

类似于函数lseek，所不同的是，该函数中的fp是一个文件指针而不是文件描述符，且返回值是一个int类型的状态而非位置值。编写函数fseek，并确保该函数与库中其他函数使用的缓冲能够协同工作。

```
#include "syscalls.h"

/* fseek: seek with a file pointer                                 */
int fseek(FILE *fp, long offset, int origin)
{
        unsigned nc;                    /* # of chars to flush     */
```

```
    long rc = 0;                        /* return code                  */

    if (fp->flag & _READ) {
        if (origin == 1)                /* from current position ?      */
            offset -= fp->cnt;          /* remember chars in buffer     */
        rc = lseek(fp->fd, offset, origin);
        fp->cnt = 0;                    /* no characters buffered       */
    } else if (fp->flag & _WRITE) {
        if ((nc = fp->ptr - fp->base) > 0)
            if (write(fp->fd, fp->base, nc) != nc)
                rc = -1;
        if (rc != -1)                   /* no errors yet ?              */
            rc = lseek(fp->fd, offset, origin);
    }
    return (rc == -1) ? -1 : 0;
}
```

变量rc中保存着返回代码；如果执行出错，就把它设置为-1。

fseek需要对两种情况做出处理：一是文件为读操作而打开，二是文件为写操作而打开。

如果文件是为读操作而打开的且origin等于1，偏移量offset将从当前位置开始计算（其他情况是：如果origin等于0，偏移量offset将从文件头位置开始计算；如果origin等于2，偏移量offset将从文件尾开始计算）。为了计算出从当前位置算起的偏移量，fseek必须把已经被读入到缓冲区中的那些字符也考虑进去，如下所示：

```
if (origin == 1)
    offset -= fp->cnt;
```

然后，fseek调用lseek完成相应的功能，并丢弃缓冲区中的字符：

```
rc = lseek(fp->fd, offset, origin);
fp->cnt = 0;
```

如果文件是为写操作而打开的，fseek必须先把缓冲区中的字符全都写入文件：

```
if ((nc = fp->ptr - fp->base) > 0)
    if (write(fp->fd, fp->base, nc) != nc)
        rc = -1;
```

如果直到现在还没有出错，fseek就调用lseek，如下所示：

```
if (rc != -1)
    rc = lseek(fp->fd, offset, origin);
```

如果执行成功，函数fseek将返回0。

练习8-5 （教材第162页）

修改fsize程序，打印i结点项中包含的其他信息。

```
#include  <stdio.h>
#include  <string.h>
#include  <fcntl.h>             /* flags for read and write   */
#include  <sys/types.h>         /* typedefs                   */
#include  <sys/stat.h>          /* structure returned by stat */
#include  "dirent.h"
```

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: print inode #, mode, links, size of file "name"   */
void fsize(char *name)
{
     struct stat stbuf;

     if (stat(name, &stbuf) == -1) {
          fprintf(stderr, "fsize: can't access %s\n", name);
          return;
     }
     if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
          dirwalk(name, fsize);
     printf("%5u %6o %3u %8ld %s\n", stbuf.st_ino,
          stbuf.st_mode, stbuf.st_nlink, stbuf.st_size, name);
}
```

我们修改后的fsize程序将打印以下信息：i结点编号、以八进制表示的文件模式、文件的链接个数、文件长度以及文件名。你还可以选择打印更多的信息——这完全取决于不同的信息对你的重要程度。

其中，函数dirwalk是教材第160页上介绍的函数。

练习8-6 （教材第166页）

标准库函数calloc(n, size)返回一个指针，它指向n个长度为size的对象，且所有分配的存储空间都被初始化为0。通过调用或修改malloc函数来实现calloc函数。

```
#include   "syscalls.h"

/* calloc: allocate n objects of size size          */
void *calloc(unsigned n, unsigned size)
{
     unsigned i, nb;
     char *p, *q;

     nb = n * size;
     if ((p = q = malloc(nb)) != NULL)
          for (i = 0; i < nb; i++)
               *p++ = 0;
     return q;
}
```

函数calloc为长度是size的n个对象分配存储空间。我们把需要分配的字节总数计算并保存在变量nb中，如下所示：

```
nb = n * size;
```

malloc将返回一个指针，该指针指向一个长度为nb个字节的内存块。这个内存块的起始位置将被保存在指针p和q中。如果内存分配操作执行成功，我们就用下面的语句把分配到的nb个字节全部初始化为0：

```
for (i = 0; i < nb; i++)
     *p++ = 0;
```

calloc函数将返回一个指针，指针指向它申请到的内存块的起始位置，同时把这块内存区域全部初始化为0。

练习8-7 （教材第166页）

malloc接收对存储空间的请求时，并不检查请求长度的合理性；而free则认为被释放的块包含一个有效的长度字段。改进这些函数，使它们具有错误检查的功能。

```
#include  "syscalls.h"

#define   MAXBYTES  (unsigned) 10240

static unsigned maxalloc;/* max number of units allocated          */
static Header   base;    /* empty list to get started             */
static Header   *freep = NULL;     /* start of free list           */

/* malloc: general-purpose storage allocator                       */
void *malloc(unsigned nbytes)
{
     Header *p, *prevp;
     static Header *morecore(unsigned);
     unsigned nunits;

     if (nbytes > MAXBYTES) {       /* not more than MAXBYTES      */
          fprintf(stderr,
                  "alloc: can't allocate more than %u bytes\n",
                  MAXBYTES);
          return NULL;
     }
     nunits = (nbytes + sizeof(Header) - 1) / sizeof(Header) + 1;

     /* . . . */                    /* as on page 187 K&R          */
}

#define   NALLOC    1024           /* minimum #units to request   */

/* morecore: ask system for more memory                            */
static Header *morecore(unsigned nu)
{
     char *cp, *sbrk(int);
     Header *up;

     if (nu < NALLOC)
          nu = NALLOC;
     cp = sbrk(nu * sizeof(Header));
     if (cp == (char *) -1)         /* no space at all             */
          return NULL;
     up = (Header *) cp;
     up->s.size = nu;
     maxalloc = (up->s.size > maxalloc) ? up->s.size : maxalloc;
     free((void *)(up+1));
     return freep;
}
```

```
/* free: put block ap in free list                                    */
void free(void *ap)
{
     Header *bp, *p;

     bp = (Header *)ap - 1;          /* point to block header         */
     if (bp->s.size == 0 || bp->s.size > maxalloc) {
          fprintf(stderr, "free: can't free %u units\n",
                  bp->s.size);
          return;
     }
     for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)

          /* . . . */    ;          /* as on page 189 K&R            */
```

malloc函数先检查用户申请分配的字节数是否大于常数 MAXBYTES。请根据系统的具体情况预先为MAXBYTES设置一个最合适的值。

函数morecore每分配一个新的内存块，都会把截止到当时的最大内存块的长度记录在静态变量maxalloc里。这样，free函数就能对释放的内存块的长度进行验证：这个长度值既不能等于0，也不能大于当时已经分配的最大内存块的长度（maxalloc）。

练习8-8 （教材第166页）

编写函数bfree(p,n)，释放一个包含n个字符的任意块p，并将它放入由malloc和free维护的空闲块链表中。通过使用bfree，用户可以在任意时刻向空闲块链表中添加一个静态或外部数组。

```
#include  "syscalls.h"

/* bfree: free an arbitrary block p of n chars                        */
unsigned bfree(char *p, unsigned n)
{
     Header *hp;

     if (n < sizeof(Header))
          return 0;                  /* too small to be useful       */
     hp = (Header *) p;
     hp->s.size = n / sizeof(Header);
     free((void *)(hp+1));
     return hp->s.size;
}
```

bfree函数有两个参数：一个指向起始位置的指针p和一个给出字符个数的数值n。如果想调用bfree把某个内存块释放到空闲区列表中，则这个内存块的长度至少要等于sizeof(Header)，否则，返回值为0。

先把指针p转换为Header类型，再把它赋值给hp：

```
hp = (Header *) p;
```

以sizeof(Header)为单位计算的内存块长度值是：

```
hp->s.size = n / sizeof(Header);
```

最后，调用函数free来释放那个内存块。free函数的输入参数是一个指针，而它指向的位置要求刚好越过内存块的头部区域，所以我们必须使用(hp+1)——就像我们在练习8-7中的morecore函数中做的那样，并且还要把它转换为void *类型。

如果内存块的长度太小，bfree函数将返回0；否则，它将返回一个以sizeof(Header)为单位计算的内存块长度值。

本书是对Brian W. Kernighan和Dennis M. Ritchie所著的《C程序设计语言（第2版·新版）》所有练习题的解答，是极佳的编程实战辅导书。K&R的著作是C语言方面的经典教材，而这本与之配套的习题解答将帮助您更加深入地理解C语言并掌握良好的C语言编程技能。

单凭阅读和学习语法结构并不能真正掌握一门程序设计语言，必须进行编程实践——亲自编写一些程序并研究别人写的程序。您可以通过K&R教材学习C语言，独立地解答书中的练习题，再钻研本书给出的习题解答。

本书特点

- 有关练习题都用K&R教材介绍的语言结构进行解答，与K&R教材中的教学内容保持同步。读者在学习到更多的C语言知识之后，可以给出更好的解决方案
- 不重复K&R教材中的内容，但对每道练习题的答案要点都给出了清晰的解释
- 利用C语言良好的特性使程序模块化，充分利用库函数并以格式化的风格编写程序，有助于读者清楚地了解程序的逻辑流程

《C程序设计语言（第2版·新版）》

2003年11月隆重上市

ISBN 7-111-12943-1
9 787111 129431

封面设计 陈子平

HZ BOOKS
华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com

ISBN 7-111-12943-1/TP·2900
定价：15.00元