

第1章 基本概念

本章首先对C语言做简要介绍。目的是通过实际的程序向读者介绍C语言的本质要素，而不是一下子就陷入到具体细节、规则及例外情况中去。因此，在这里我们并不想完整地或很精确地对C语言进行介绍（但所举例子都是正确的）。我们想尽可能快地让读者学会编写有用的程序，因此，重点介绍其基本概念：变量与常量、算术运算、控制流、函数、基本输入输出。本章并不讨论那些编写较大的程序所需要的重要特性，包括指针、结构、大多数运算符、部分控制流语句以及标准库。

这样做也有缺陷，其中最大的不足之处是在这里找不到对任何特定语言特性的完整描述，并且，由于太简略，也可能会使读者产生误解。而且，由于所举的例子没有用到C语言的所有特性，故这些例子可能并未达到简明优美的程度。我们已尽力缩小这种差异。另一个不足之处是，本章所讲过的某些内容在后续有关章节还必须重复介绍。我们希望这种重复带给读者的帮助会胜过烦恼。

无论如何，经验丰富的程序员应能从本章所介绍的有关材料中推断他们在程序设计中需要的东西。初学者则应编写类似的小程序来充实它。这两种人都可以把本章当作了解后续各章的详细内容的框架。

1.1 入门

学习新的程序设计语言的最佳途径是编写程序。对于所有语言，编写的第一个程序都是相同的：

打印如下单词：

```
hello, world
```

在初学语言时这是一个很大的障碍，要越过这个障碍，首先必须建立程序文本，然后成功地对它进行编译，并装入、运行，最后再看看所产生的输出。只要把这些操作细节掌握了，其他内容就比较容易了。

在C语言中，用如下程序打印“ hello, world ”：

```
#include <stdio.h>

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

至于如何运行这个程序取决于使用的系统。作为一个特殊的例子，在操作系统中，必须首先在某个以“.”c作为扩展名的文件中建立起这个程序，如 hello.c，然后再用如下命令编译它：

```
cc hello.c
```

如果在输入上述程序时没有出现错误（例如没有漏掉字符或错拼字符），那么编译程序将往下执行并产生一个可执行文件 a.out。如果输入命令

```
a. out
```

运行a.out程序，则系统将打印

```
hello, world
```

在其他操作系统上操作步骤会有所不同，读者可向身边的专家请教。

```
#include <stdio.h>                包含有关标准库的信息

main()                            定义名为main的函数，它不接收变元值
{                                 main的语句括在花括号中
    printf("hello,  world\n");    main函数调用库函数printf可打印字符序列，\n代表换行符
}
```

下面对这个程序本身做一些解释说明。每一个 C程序，不论大小如何，都由函数和变量组成。函数中包含若干用于指定所要做的计算操作的语句，而变量则用于在计算过程中存储有关值。 C中的函数类似于 FORTRAN语言中的子程序与函数或 Pascal语言中的过程与函数。在本例中，函数的名字为main。一般而言，可以给函数任意命名，但 main是一个特殊的函数名，每一个程序都从名为main的函数的起点开始执行。这意味着每一个程序都必须包含一个 main函数。

main函数通常要调用其他函数来协助其完成某些工作，调用的函数有些是程序人员自己编写的，有些则由系统函数库提供。上述程序的第一行

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

用于告诉编译程序在本程序中包含标准输入输出库的有关信息。许多 C源程序的开始处都包含这一行。我们将在第7章和附录中对标准库进行详细介绍。

在函数之间进行数据通信的一种方法是让调用函数向被调用函数提供一串叫做变元的值。函数名后面的一对圆括号用于把这一串变元（变元表）括起来。在本例子中，所定义的 main函数不要求任何变元，故用空变元表（）表示。

函数中的语句用一对花括号 {}括起来。本例中的main函数只包含一个语句：

```
printf("hello,  world\n");
```

当要调用一个函数时，先要给出这个函数的名字，再紧跟用一对圆括号括住的变元表。上面这个语句就是用变元 "hello, world\n" 来调用函数printf printf是一个用于打印输出的库函数，在本例中，它用于打印用引号括住的字符串。

用双引号括住的字符序列叫做字符串或字符串常量，如 "hello, world\n"就是一个字符串。目前仅使用字符串作为 printf及其他函数的变元。

在C语言中，字符序列 \n表示换行符，在打印时它用于指示从下一行的左边换行打印。如果在字符串中遗漏了\n（一个值得做的试验），那么输出打印完后没有换行。在 printf函数的变元中必须用\n引入换行符，如果用程序中的换行来代替 \n如：

```
printf("hello,  world
```

```
");
```

那么C编译器将会产生一个错误信息。

printf函数永远不会自动换行，我们可以多次调用这个函数来分阶段打印一输出行。上面给出的第一个程序也可以写成如下形式：

```
#include <stdio.h>

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

它所产生的输出与前面一样。

请注意，\n只表示一个字符。诸如\n等换码序列为表示不能打印或不可见字符提供了一种通用可扩充机制。除此之外， C语言提供的换码序列还有：表示制表符的 \t，表示回退符的\b，表示双引号的 \"，表示反斜杠符本身的 \\。2.3节将给出换码序列的完整列表。

练习1-1　请读者在自己的系统上运行“ hello, world ”程序。再做个实验，让程序中遗漏一些部分，看看会出现什么错误信息。

练习1-2　做个实验，观察一下当printf函数的变元字符串中包含\c（其中c是上面未列出的某个字符）时会出现什么情况。

1.2　变量与算术表达式

下面的程序用公式

$$^{\circ}C = (5/9)\,(^{\circ}F - 32)$$

打印华氏温度与摄氏温度对照表：

```
0      -17
20      -6
40       4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220    104
240    115
260    126
280    137
```

```
300    148
```

这个程序本身仍只由一个名为main的函数的定义组成，它要比前面用于打印“ hello, world ”的程序长，但并不复杂。这个程序中引入了一些新的概念，包括注解、说明、变量、算术表达式、循环以及格式输出。该程序如下：

```
#include <stdio.h>

        /* 对fahr = 0, 20, ..., 300
        打印华氏温度与摄氏温度对照表 */
        main()
        {
            int  fahr, celsius;
            int  lower, upper, step;

            lower = 0;      /* 温度表的下限 */
            upper = 300 ;   /* 温度表的上限 */
            step  = 20;     /* 步长 */

            fahr = lower;
            while  (fahr <= upper)  {
                celsius = 5 * (fahr-32) / 9;
                printf("%d\t%d\n", fahr, celsius);
                fahr = fahr + step;
            }
        }
```

其中的两行

```
/* 对fahr=0, 20, ..., 300
打印华氏温度与摄氏温度对照表 */
```

叫做注解，用于解释该程序是做什么的。夹在 /*与*/之间的字符序列在编译时被忽略掉，它们可以在程序中自由地使用，目的是为了使程序更易于理解。注解可以出现在任何空格、制表符或换行符可以出现的地方。

在C语言中，所有变量都必须先说明后使用，说明通常放在函数开始处的可执行语句之前。说明用于声明变量的性质，它由一个类型名与若干所要说明的变量组成，例如

```
int  fahr, celsius;
int  lower, upper, step;
```

其中，类型int表示所列变量为整数变量，与之相对， float表示所列变量为浮点变量（浮点数可以有小数部分）。int与float类型的取值范围取决于所使用的机器。对于 int类型，通常为16位（取值在 - 32768 ~ +32767之间），也有用32位表示的。 float类型一般都是32位，它至少有6位有效数字，取值范围一般在10^{-38}～10^{+38}之间。

除int与float之外，C语言还提供了其他一些基本数据类型，包括：

char	字符 —— 单字节
short	短整数

long　　长整数

double　　双精度浮点数

这些数据类型对象的大小也取决于机器。另外，还有由这些基本类型组成的数组、结构与联合类型、指向这些类型的指针类型以及返回这些类型的函数，我们将在后面适当的章节再分别介绍它们。

上面温度转换程序计算以4个赋值语句

```
lower = 0;
upper = 300 ;
step  = 20;
fahr = lower;
```

开始，用于为变量设置初值。各个语句均以分号结束。

温度转换表中的每一行均以相同的方式计算，故可以用循环语句来重复产生各行输出，每行重复一次。这就是while循环语句的用途：

```
while  (fahr <= upper)  {
   ...
}
```

while循环语句的执行步骤如下：首先测试圆括号中的条件。如果条件为真（ fahr小于等于upper），则执行循环体（括在花括号中的三个语句）。然后再重新测试该条件，如果为真，则再次执行该循环体。当该条件测试为假（ fahr大于upper）时，循环结束，继续执行跟在该循环语句之后的下一个语句。在本程序中，循环语句后再没有其他语句，因此整个程序终止执行。

while语句的循环体可以是用花括号括住的一个或多个语句（如上面的温度转换程序），也可以是不用花括号括住的单个语句，例如：

```
while  (i < j)
   i = 2 * i;
```

在这两种情况下，我们总是把由while控制的语句向里缩入一个制表位（在书中以四个空格表示），这样就可以很容易地看出循环语句中包含那些语句。这种缩进方式强化了程序的逻辑结构。尽管C编译程序并不关心程序的具体形式，但使程序在适当位置采用缩进空格的风格对于使程序更易于为人们阅读是很重要的。我们建议每行只写一个语句，并在运算符两边各放一个空格字符以使运算组合更清楚。花括号的位置不太重要，尽管每个人都有他所喜爱的风格。我们从一些比较流行的风格中选择了一种。读者可以选择自己所合适的风格并一直使用它。

绝大多数任务都是在循环体中做的。循环体中的赋值语句

```
celsius = 5 * (fahr-32) / 9;
```

用于求与指定华氏温度所对应的摄氏温度值并将值赋给变量 celsius。在该语句中，之所以把表达式写成先乘5然后再除以9而不直接写成5/9，是因为在C语言及其他许多语言中，整数除法要进行截取：结果中的小数部分被丢弃。由于5和9都是整数，5/9相除后所截取得的结果为0，故这样所求得的所有摄氏温度都变成0。

这个例子也对printf函数的工作功能做了更多的介绍。 printf是一个通用输出格式化函数，第7章将对此做详细介绍。该函数的第一个变元是要打印的字符串，其中百分号（ %）指示用其他

变元（第2、第3个…变元）之一对其进行替换，以及打印变元的格式。例如， %d指定一个整数变元，语句

```
printf("%d\t%d\n", fahr, celsius);
```

用于打印两个整数fahr与celsius值并在两者之间空一个制表位（\t）。

printf函数第1个变元中的各个%分别对应于第2个、第3个… 第n个变元，它们在数目和类型上都必须匹配，否则将出现错误。

顺便指出， printf函数并不是C语言本身的一部分， C语言本身没有定义输入输出功能。printf是标准库函数中一个有用的函数，标准库函数一般在 C程序中都可以使用。ANSI标准中定义了printf函数的行为，从而其性质在使用每一个符合标准的编译程序与库中都是相同的。

为了集中讨论C语言本身，在第7章之前的各章中不再对输入输出做更多的介绍，特别是把格式输入延后到第7章。如果读者想要了解数据输入，请先阅读 7.4节对scanf函数的讨论。scanf函数类似于printf函数，只不过它是用于读输入数据而不是写输出数据。

上面这个温度转换程序存在着两个问题。比较简单的一个问题是，由于所输出的数不是右对齐的，输出显得不是特别好看。这个问题比较容易解决：只要在 printf语句的第1个变元的%d中指明打印长度，则打印的数字会在打印区域内右对齐。例如，可以用

```
printf("%3d %6d\n", fahr, celsius);
```

打印fahr与celsius的值，使得fahr的值占3个数字宽、 celsius的值占6个数字宽，如下所示：

```
0        -17
20        -6
40         4
60        15
80        26
100       37
…
```

另一个较为严重的问题是，由于使用的是整数算术运算，故所求得的摄氏温度不很精确，例如，与0℉ 对应的精确的摄氏温度为-17.8℃，而不是-17℃。为了得到更精确的答案，应该用浮点算术运算来代替上面的整数算术运算。这就要求对程序做适当修改。下面给出这个程序的第2个版本：

```
#include <stdio.h>

/* 对fahr = 0, 20, ..., 300打印华氏温度与摄氏温度对照表；
   浮点数版本 */
main()
{
      float  fahr, celsius;
      int  lower, upper, step;

      lower = 0;       /* 温度表的下限 */
      upper = 300 ;    /* 温度表的上限 */
      step  = 20;      /* 步长 */
```

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

这个版本与前一个版本基本相同，只是把 fahr与celsius说明成float浮点类型，转换公式的表达也更自然。在前一个版本中，之所以不用5/9是因为按整数除法它们相除截取的结果为 0。然而，在此版本中5.0/9.0是两个浮点数相除，不需要截取。

如果某个算术运算符的运算分量均为整数类型，那么就执行整数运算。然而，如果某个算术运算符有一个浮点运算分量和一个整数运算分量，那么这个整数运算分量在开始运算之前会被转换成浮点类型。例如，对于表达式 fahr -32，32在运算过程中将被自动转换成浮点数再参与运算。不过，在写浮点常量时最好还是把它写成带小数点，即使该浮点常量取的是整数值，因为这样可以强调其浮点性质，便于人们阅读。

第2章将详细介绍把整型数转换成浮点数的规则。现在请注意，赋值语句

```
fahr = lower;
```

与条件测试

```
while ( fahr <= upper )
```

也都是以自然的方式执行 —— 在运算之前先把int转换成float。

printf中的转换说明%3.0f表明要打印的浮点数（即 fahr）至少占3个字符宽，不带小数点与小数部分。%6.1f表示另一个要打印的数（ celsius）至少有6个字符宽，包括小数点和小数点后 1 位数字。输出类似于如下形式：

```
  0   -17.8
 20    -6.7
 40     4.4
...
```

在格式说明中可以省去宽度（ %与小数点之间的数）与精度（小数点与字母 f之间的数）。例如，%6f的意思是要打印的数至少有 6个字符宽；%.2f说明要打印的数在小数点后有两位小数，但整个数的宽度不受限制；%f的意思仅仅是要打印的数为浮点数。

%d	打印十进制整数
%6d	打印十进制整数，至少6个字符宽
%f	打印浮点数
%6f	打印浮点数，至少6个字符宽
%.2f	打印浮点数，小数点后有两位小数
%6.2f	打印浮点数，至少6个字符宽，小数点后有两位小数

此外，printf函数还可以识别如下格式说明：表示八进制数的 %o、表示十六进制数的%x、表示字符的%c、表示字符串的%s以及表示百分号%本身的%%。

练习1-3 修改温度转换程序，使之在转换表之上打印一个标题。

练习1-4 编写一个用于打印摄氏与华氏温度对照表的程序。

1.3 for语句

对于一个特定任务，可以用多种方法来编写程序。下面是前面讲述的温度转换程序的一个变种：

```
# include <stdio.h>

/* 打印华氏与摄氏温度对照表 */
main( )
{
      int  fahr;

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

这个版本与前一个版本执行的结果相同，但看起来有些不同。一个主要的变化是它删去了大部分变量，只留下了一个 fahr，其类型为 int。本来用变量表示的下限、上限与步长都在新引入的 for语句中作为常量出现，用于求摄氏温度的表达式现在已变成了 printf函数的第3个变元，而不再是一个独立的赋值语句。

这最后一点变化说明了一个通用规则：在所有可以使用某个类型的变量的值的地方，都可以使用该类型的更复杂的表达式。由于 printf函数的第3个变元必须为与 %6.1f匹配的浮点值，则可以在这里使用任何浮点表达式。

for语句是一种循环语句，是 while语句的推广。如果将其与前面介绍的 while语句比较，就会发现其操作要更清楚一些。在圆括号内共包含三个部分，它们之间用分号隔开。第一部分

```
fahr = 0
```

是初始化部分，仅在进入循环前执行一次。第二部分是用于控制循环的条件测试部分：

```
fahr <= 300
```

这个条件要进行求值。如果所求得的值为真，那么就执行循环体（本例循环体中只包含一个 printf函数调用语句）。然后再执行第三部分

```
fahr = fahr + 20
```

加步长，并再次对条件求值。一旦求得的条件值为假，那么就终止循环的执行。像 while语句一样，for循环语句的体可以是单个语句，也可以是用花括号括住的一组语句。初始化部分（第一部分）、条件部分（第二部分）与加步长部分（第三部分）均可以是任何表达式。

至于在while与for这两个循环语句中使用哪一个，这是随意的，主要看使用哪一个更能清楚地描述问题。 for语句比较适合描述这样的循环：初值和增量都是单个语句并且是逻辑相关的，因为for语句把循环控制语句放在一起，比 while语句更紧凑。

练习1-5　修改温度转换程序，要求以逆序打印温度转换表，即从 300度到0度。

1.4　符号常量

在结束对温度转换程序的讨论之前，再来看看符号常量。把 300、20等“幻数”埋在程序中并不是一种好的习惯，这些数几乎没有向以后可能要阅读该程序的人提供什么信息，而且使程序的修改变得困难。处理这种幻数的一种方法是赋予它们有意义的名字。 #define指令就用于把符号名字（或称为符号常量）定义为一特定的字符串：

```
#define  名字  替换文本
```

此后，所有在程序中出现的在 #define中定义的名字，该名字既没有用引号括起来，也不是其他名字的一部分，都用所对应的替换文本替换。这里的名字与普通变量名有相同的形式：它们都是以字母打头的字母或数字序列。替换文本可以是任何字符序列，而不仅限于数。

```
#include  <stdio.h>

#define  LOWER  0      /* 表的下限 */
#define  UPPER   300   /* 表的上限 */
#define  STEP    20    /* 步长 */

/* 打印华氏-摄氏温度对照表 */
main ( )
{
    int  fahr;

    for  ( fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP )
         printf ( "%3d   %6.1f\n", fahr,  (5.0 / 9.0) * (fahr - 32 ) );
}
```

LOWER、UPPER与STEP等几个量是符号常量，而不是变量，故不需要出现在说明中。符号常量名通常用大写字母拼写，这样就可以很容易与用小写字母拼写的变量名相区别。注意，#define指令行的末尾没有分号。

1.5　字符输入输出

接下来讨论一些与字符数据处理有关的程序。读者将会发现，许多程序只不过是这里所讨论的程序原型的扩充版本。

由标准库提供的输入输出模型非常简单。文本的输入输出都是作为字符流处理的，不管它从何处输入、输出到何处。文本流是由一行行字符组成的字符序列，而每一行字符则由 0个或多个字符组成，并后跟一个换行符。标准库有责任使每一输入输出流符合这一模型，使用标准库的C程序员不必担心各字符行在程序外面怎么表示。

标准库中有几个函数用于控制一次读写一个字符，其中最简单的是 getchar和putchar这两个函数。getchar函数在被调用时从文本流中读入下一个输入字符并将其作为结果值返回。即，在执行

```
c = getchar ()
```

之后，变量c中包含了输入流中的下一个字符。这种输入字符通常是从键盘输入的。关于从文件

输入字符的方法将在第7章讨论。

putchar函数在调用时将打印一个字符。例如，函数

```
putchar ( c )
```

用于把整数变量c的内容作为一个字符打印，它通常是显示在屏幕上。putchar与printf这两个函数可以交替调用，输出的次序即调用的次序。

1.5.1 文件复制

借助getchar与putchar函数，可以在不掌握其他输入输出知识的情况下编写出许多有用的代码。最简单的程序是一次一个字符地把输入复制到输出，其基本思想如下：

```
读一个字符
while（该字符不是文件结束指示符）
    输出刚读进的字符
    读下一个字符
```

下面是其C程序：

```
#include  <stdio.h>

/* 用于将输入复制到输出的程序；第1个版本 */
main ( )
{
      int  c;

      c = getchar ( );
while ( c != EOF ) {
      putchar ( c );
      c = getchar ( );
  }
}
```

其中的关系运算符!=的意思是“不等于”。

像其他许多东西一样，一个字符不论在键盘或屏幕上以什么形式出现，在机器内部都是以位模式存储的。char类型就是专门用于存储这种字符数据的类型，当然任何整数类型也可以用于存储字符数据。由于某种微妙却很重要的理由，此处使用了int类型。

需要解决的问题是如何将文件中的有效数据与文件结束标记区分开来。C语言采取的解决方法是，getchar函数在没有输入时返回一个特殊值，这个特殊值不能与任何实际字符相混淆。这个值叫做EOF（end of file，文件结束）。必须把c说明成一个大到足以存放getchar函数可能返回的各种值的类型。之所以不把c说明成char类型，是因为c必须大到除了能存储任何可能的字符外还要能存储文件结束符EOF。因此，把c说明成int类型的。

EOF是一个在<stdio.h>库中定义的整数，但其具体的数值是什么并不重要，只要知道它与char类型的所有值都不相同就行了。可以通过使用符号常量来保证EOF在程序中不依赖于特定的数值。

对于经验比较丰富的C程序员，可以把字符复制程序编写得更精致些。在C语言中，诸如

```
c = getchar ( )
```

之类的赋值操作是一个表达式，因而就有一个值，即赋值后位于 =左边变量的值。换言之，赋值可以作为更大的表达式的一部分出现。可以把将字符赋给 c的赋值操作放在while循环语句的测试部分中，即可以将上面的字符复制程序改写成如下形式：

```
#include  <stdio.h>

/* 用于将输入复制到输出的程序；第个版本  */
main ( )
{
    int  c;

    while ( (c = getchar ( ) ) != EOF )
          putchar ( c );
}
```

在这一程序中，while循环语句先读一个字符并将其赋给 c，然后测试该字符是否为文件结束标记。如果该字符不是文件结束标记，那么就执行 while语句体，把该字符打印出来。再重复执行该while语句。当最后到达输入结束位置时， while循环语句终止执行，从而整个 main程序执行结束。

这个版本的特点是将输入集中处理 —— 只调用了一次 getchar函数 —— 这样使整个程序的规模有所缩短，所得到的程序更紧凑，从风格上讲，程序更易阅读。读者将会不断地看到这种风格。（然而，如果再往前走，所编写出的程序可能很难理解，我们将对这种趋势进行遏制。）

在while条件中用于括住赋值表达式的圆括号不能省略。不等运算符 !=的优先级要比赋值运算符的优先级高，这就是说，在不使用圆括号时关系测试 !=将在赋值之前执行。故语句

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

等价于

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

这个语句的作用是把 c的值置为 0或1（取决于 getchar函数在调用执行时所读的数据是否为文件结束标记），这并不是我们所希望的结果。（有关这方面的更多的内容将在第 2章介绍。）

练习1-6　验证表达式getchar ( ) != EOF的值是0还是1。

练习1-7　编写一个用于打印EOF值的程序。

1.5.2　字符计数

下面这个程序用于对字符计数，与上面的文件复制程序类似：

```
#include  <stdio.h>

/* 统计输入的字符数； 第个版本  */
main ( )
{
    long  nc;
```

```
    nc = 0;
    while ( getchar ( ) != EOF )
    ++nc;
    printf("%ld\n", nc);
}
```

其中的语句

```
++nc;
```

引入了一个新的运算符++，其功能是加1。可以用

```
nc = nc + 1;
```

来代替它，但++nc比之要更精致，通常效率也更高。与该运算符相对应的还有一个减一运算符--。++与--这两个运算符既可以作为前缀运算符（如 ++nc），也可以作为后缀运算符（如nc++）。正如第2章将要指出的，这两种形式在表达式中有不同的值，但 ++nc与nc++都使nc的值加1。我们暂时只使用前缀形式。

这个字符计数程序没有用 int类型的变量而是用 long类型的变量来存放计数值。 long整数(长整数)至少要占用32位存储单元。尽管在某些机器上 int与long类型的值具有同样大小，但在其他机器上int类型的值可能只有16位存储单元（最大取值32 767），相当小的输入都可能使int类型的计数变量溢出。转换说明%ld用来告诉printf函数对应的变元是long整数类型。

如果使用double(双精度浮点数)类型，那么可以统计更多的字符。下面不再用 while循环语句而用for循环语句来说明编写循环的另一种方法：

```
#include  <stdio.h>

/* 统计输入的字符数； 第2个版本  */
main ( )
{
    double  nc;

    for ( nc = 0; getchar ( ) != EOF; ++nc )
        ;
    printf("%.0f\n", nc);
}
```

可用于 float类型，double %.0f用于控制不打印小数点和小数部分，因此小数部分为 0。

这个for循环语句的体是空的，这是因为它的所有工作都在测试（条件）部分与加步长部分做了。但C语言的语法规则要求 for循环语句必须有一个体，因此用单独的分号代替。单个分号叫做空语句，它正好能满足语句的这一要求。把它单独放在一行是为了使它显目一点。

在结束讨论字符计数程序之前，请观察一下以下情况：如果输入中不包含字符，那么，while语句或for语句中的条件从一开始就为假， getchar函数一次也不会调用，程序的执行结果为0，这个结果也是正确的。这一点很重要。 while语句与for语句的优点之一就是在执行循环体之前就对条件进行测试。如果没有什么事要做，那么就不去做，即使它意味着不执行循环体。程序在出现0长度的输入时应表现得机灵一点。 while语句与for语句有助于保证在出现边界条件时做合理的事情。

1.5.3 行计数

下一个程序用于统计输入的行数。正如上文提到的，标准库保证输入文本流是以行序列的形式出现的，每一行均以换行符结束。因此，统计输入的行数就等价于统计换行符的个数。

```
#include  <stdio.h>

/* 统计输入的行数 */
main ( )
{
    long  c, nl;

    nl = 0;
    while ( (c = getchar ( ) ) != EOF )
          if ( c =='\n' )
          ++nl;
    printf("%d\n", nl);
}
```

这个程序中循环语句的体是一个 if语句，该if语句用于控制增值 ++nl。if语句执行时首先测试圆括号中的条件，如果该条件为真，那么就执行内嵌在其中的语句（或括在花括号中的一组语句）。这里再次用缩进方式指示哪个语句被哪个语句控制。

双等于号==是C语言中表示“等于”的运算符（类似于 Pascal中的单等于号=及FORTRN中的.EQ.）。由于C已用单等于号=作为赋值运算符，故为区别用双等于号 = =表示相等测试。注意，C语言初学者常常会用 =来表示==的意思。正如第 2章所述，即使这样误用了，得到的通常仍是合法的表达式，故系统不会给出警告信息。

夹在单引号中的字符表示一个整数值，这个值等于该字符在机器字符集中的数值。它叫做字符常量，尽管它只不过是比较小的整数的另一种写法。例如， 'A' 即字符常量；在ASCII字符集中其值为65（即字符的内部表示值为 65）。当然，用 'A'要比用65优越，'A'的意义清楚，并独立于特定的字符集。

字符串常量中使用的换码序列也是合法的字符常量，故 '\n' 表示换行符的值，在ASCII字符集中其值为10。我们应该仔细注意到，'\n'是单个字符，在表达式中它只不过是一个整数；而另一方面，"\n"是一个只包含一个字符的字符串常量。有关字符串与字符之间的关系将在第 2章做进一步讨论。

练习1-8　编写一个用于统计空格、制表符与换行符个数的程序。

练习1-9　编写一个程序，把它的输入复制到输出，并在此过程中将相连的多个空格用一个空格代替。

练习1-10　编写一个程序，把它的输入复制到输出，并在此过程中把制表符换成 \t、把回退符换成\b、把反斜杠换成\\。这样可以使得制表符与回退符能以无歧义的方式可见。

1.5.4 单词计数

我们将要介绍的第四个实用程序用于统计行数、单词数与字符数，这里对单词的定义放得

比较宽，它是任何其中不包含空格、制表符或换行符的字符序列。下面这个比较简单的版本是在UNIX系统上实现的完成这一功能的程序 wc：

```
#include  <stdio.h>

#define IN   1  /* 在单词内 */
#define OUT  0  /* 在单词外 */

/* 统计输入的行数、单词数与字符数 */
main ( )
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ( (c = getchar ( ) ) != EOF ) {
        ++nc;
        if ( c =='\n' )
            ++nl;
        if ( c == ' ' ||c == '\n' || c == '\t' )
            state = OUT;
        else if ( state == OUT ) {
                 state = IN;
                 ++nw;
                      }
            }
            printf("%d  %d  %d\n", nl,  nw, nc);
}
```

程序在执行时，每当遇到单词的第一个字符，它就作为一个新单词加以统计。 state变量用于记录程序是否正在处理一个单词（是否在一个单词中），它的初值是“ 不在单词中 ”，即被赋初值为OUT。我们在这里使用了符号常量 IN与OUT而没有使用其字面值 1与0，主要是因为这可以使程序更可读。在比较小的程序中，这样做也许看不出有什么区别，但在比较大的程序中，如果从一开始就这样做，那么所增加的一点工作量与所提高的程序的明晰性相比是很值得的。读者也会发现，在程序中，如果幻数仅仅出现在符号常量中，那么对程序做大量修改就显得比较容易。

语句行

```
nl = nw = nc = 0;
```

用于把其中的三个变量nl nw 与都置为 0。这种情况并不特殊，但要注意这样一个事实，在兼有值与赋值两种功能的表达式中，赋值结合次序是由右至左。所以上面这个语句也可以写成：

```
nl = ( nw = ( nc = 0 ));
```

运算符 || 的意思是OR(或) ，所以程序行

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

的意思是“ 如果c是空格或c是换行符或c是制表符 ”（回忆一下，换码序列 \t 是制表符的可见表

示)。与之对应的一个运算符是&&，其含义是AND（与），其优先级只比 || 高一级。经由&&或 || 连接的表达式由左至右求值，并保证在求值过程中只要已得知真或假，求值就停止。如果 c是一个空格，那么就没有必要再测试它是否为换行符或制表符，故后两个测试无需再进行。在这里这倒不特别重要，但在某些更复杂的情况下这样做就显得很重要，不久我们将会看到这种例子。

这个例子中还给出了 else部分，它指定当 if语句中的条件部分为假时所要采取的动作。其一般形式为：

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

在if-else的两个语句中有一个并且只有一个被执行。如果表达式的值为真，那么就执行语句 $_1$，否则，执行语句$_2$。这两个语句均可以或者是单个语句或者是括在花括号内的语句序列。在单词计数程序中，else之后的语句仍是一个 if语句，这个 if语句用于控制括在花括号中的两个语句。

练习1-11　你准备怎样测试单词计数程序？如果程序中出现任何错误，那么什么样的输入最有利于发现这些错误？

练习1-12　编写一个程序，以每行一个单词的形式打印输入。

1.6　数组

下面编写一个用来统计各个数字、空白符（空格符、制表符及换行符）以及所有其他字符出现次数的程序。这个程序听起来有点矫揉造作，但有助于在一个程序中对 C语言的几个方面加以讨论。

由于所有输入的字符可以分成 12个范畴，因此用一个数组比用十个独立的变量来存放各个数字的出现次数要方便一些。下面是这个程序的第 1个版本：

```
#include  <stdio.h>
/* 统计各个数字、空白符及其他字符分别出现的次数 */
main ( )
{
    int  c, i, nwhite, nother;
    int  ndigit[10];

    nwhite = nother = 0;
    for ( i =0;  i < 10; ++i )
        ndigit[i] = 0;

    while ( ( c = getchar() ) != EOF )
        if ( c >= '0' && c <= '9' )
            ++ndigit[c - '0'];
        else if ( c == ' ' ||c == '\n' || c == '\t' )
            ++nwhite;
        else
```

```
            ++nother;

    printf( "digits =" );
    for ( i =0; i < 10; ++i )
        printf( "  %d", ndigit[i] );
        printf( ", white space = %d, other = %d/n", nwhite, nother);
}
```

当把程序自身作为输入时，输出为：

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

程序中的说明语句

```
int  ndigit[10];
```

用于把ndigit**说明为由** 10**个整数组成的数组。在** C**语言中，数组下标总是从** 0**开始，故这个数组的** 10**个元素是** ndigit[0]**、** ndigit[1]**、…、** ndigit[9]**。这一点在分别用于初始化和打印数组的两个** for **循环语句中得到反映。**

下标可以是任何整数表达式，包括整数变量（如 i**）与整数常量。**

这个程序的执行取决于数字的字符表示的性质。例如，测试

```
if ( c >= '0' && c <= '9' ) ...
```

用于判断 c**中的字符是否为数字。如果它是数字，那么该数字的数值是：**

```
c - '0'
```

这种做法只有在'0'**、**'1'**、…、**'9'**具有连续增加的值时才有效。所幸所有字符集都是这样做的。**

根据定义， char**类型的字符是小整数，故** char**类型的变量和常量等价于算术表达式中** int**类型的变量和常量。这样做既自然又方便，例如，** c - '0'**是一个整数表达式，对应于存储在** c**中的字符** '0'**至**'9'**，其值为** 0**至** 9**，因此可以充当数组** ndigit**的合法下标。**

关于一个字符是数字、空白符还是其他字符的判定是由如下语句序列完成的：

```
if ( c >= '0' && c <= '9' )
    ++ndigit[c - '0'];
else if ( c == ' ' ||c == '\n' || c == '\t' )
    ++nwhite;
else
    ++nother;
```

在程序中经常会用如下模式来表示多路判定：

```
if (条件1)
    语句1
else if (条件2)
    语句2
…
    …
else
    语句n
```

在这个模式中，各个条件从前往后依次求值，直到满足某个条件，这时执行对应的语句部

分，语句执行完成后，整个if构造完结。（其中的任何语句都可以是括在花括号中的若干个语句。）如果其中没有一个条件满足，那么就执行位于最后一个 else之后的语句（如果有这个语句）。如果没有最后一个else及对应的语句，那么这个 if构造就不执行任何动作，如同前面的单词计数程序一样。在第一个if与最后一个else之间可以有0个或多个

```
else if (条件)
    语句
```

就风格而言，我们建议读者采用缩进格式。如果每一个 if都比前一个else向里缩进一点，那么对一个比较长的判定序列就有可能越出页面的右边界。

第3章将要讨论的switch语句提供了编写多路分支的另一种手段，它特别适合于表示数个整数或字符表达式是否与一常量集中的某个元素匹配的情况。为便于对比，我们将在 3.4节给出用switch语句编写的这个程序的另一个版本。

练习1-13　编写一个程序，打印其输入的文件中单词长度的直方图。横条的直方图比较容易绘制，竖条直方图则要困难些。

练习1-14　编写一个程序，打印其输入的文件中各个字符出现频率的直方图。

1.7　函数

C语言中的函数类似于 FORTRAN语言中的子程序或函数，或者 Pascal语言中的过程或函数。函数为计算的封装提供了一种简便的方法，在其他地方使用函数时不需要考虑它是如何实现的。在使用正确设计的函数时不需要考虑它是怎么做的，只需要知道它是做什么的就够了。 C语言使用了简单、方便、有效的函数，我们将会经常看到一些只定义和调用了一次的短函数，这样使用函数使某些代码段更易于理解。

到目前为此，我们所使用的函数（如 printf、getchar与putchar等）都是函数库为我们提供的。现在是我们自己编写一些函数的时候了。由于 C语言没有像FORTRAN语言那样提供诸如 **之类的乘幂运算符，我们可以通过编写一个求乘幂的函数　power(m, n) 来说明定义函数的方法。power(m, n)函数用于计算整数 m的正整数次幂 n，即power(2,5)的值为32。这个函数不是一个实用的乘幂函数，它只能用于处理比较小的整数的正整数次幂，但它对于说明问题已足够了。（在标准库中包含了一个用于计算 xy的函数pow(x, y)。）

下面给出函数power(m, n)的定义及调用它的主程序，由此可以看到整个结构。

```
#include  <stdio.h>

int  power(int m, int n);

/* 测试power函数 */
main ( )
{
    int  i;

    for ( i =0; i < 10; ++i )
```

```
        printf("%d  %d  %d\n", i, power(2, i), power(-3, i));
    return 0;
}

/* power: 求底的n次幂； n >=0  */
int  power(int base, int n)
{
    int  i, p;

    p = 1;
    for ( i = 1; i <= n; ++i )
        p = p * base;
    return p;
}
```

函数定义的一般形式为：

```
返回值类型函数名（可能有的参数说明）
{
    说明序列
    语句序列
}
```

不同函数的定义可以以任意次序出现在一个源文件或多个源文件中，但同一函数不能分开存放在几个文件中。如果源程序出现在几个文件中，那么对它的编译和装入比将整个源程序放在同一文件时要做更多说明，但这是操作系统的任务，而不是语言属性。我们暂且假定两个函数放在同一文件中，从而前面所学的有关运行 C程序的知识在目前仍然有用。

main主程序在如下命令行中对power函数进行了两次调用：

```
printf("%d  %d  %d\n", i, power(2, i), power(-3, i));
```

每一次调用均向power函数传送两个变元，而 power函数则在每次调用执行完时返回一个要按一定格式打印的整数。在表达式中， power(2, i)就像２和i一样是一个整数。（并不是所有函数都产生一个整数值，第4章将对此进行讨论。）

power函数本身的第一行

```
int  power(int base, int n)
```

说明参数的类型与名字以及该函数返回的结果的类型。 power的参数名只能在 power内部使用，在其他函数中不可见：在其他函数中可以使用与之相同的参数名而不会发生冲突。对变量 i与p亦如此：power函数中的i与main函数中的i无关。

一般而言，把在函数定义中用圆括号括住的表中命名的变量叫做参数，而把函数调用中与参数对应的值叫做变元。为了表示两者的区别，有时也用形式变元与实际变元这两个术语。

power函数计算得的值由return语句返回给main函数。关键词return可以后跟任何表达式：

```
return 表达式；
```

函数不一定都返回一个值。不含表达式的 return语句用于使控制返回调用者（但不返回有用的值），如同在达到函数的终结右花括号时“脱离函数”一样。调用函数也可以忽略（不用）一

个函数所返回的值。

读者可能已经注意到，在main函数末尾有一个return语句。由于main本身也是一个函数，它也就可以向其调用者返回一个值，这个调用者实际上就是程序的执行环境。一般而言，返回值为0表示正常返回，返回值非 0则引发异常或错误终止条件。从简明性角度考虑，在这之前的main函数中都省去了 return语句，但在以后的 main函数中将包含 return语句，以提醒程序要向环境返回状态。

main函数前的说明语句

```
int  power(int m, int n);
```

表明power是一个有两个int类型的变元并返回一个 int类型的值的函数。这个说明叫做函数原型，要与power函数的定义和使用相一致。如果该函数的定义和使用与这一函数原型不一致，那么就是错误的。

函数原型与函数说明中参数的名字不要求相同。更确切地说，函数原型中的参数名是可有可无的。故上面这个函数原型也可以写成：

```
int  power(int, int);
```

但是，选择一个合适的参数名是一种良好的文档编写风格，我们在使用函数原型时仍将指明参数名。

历史回顾：ANSI C和C的较早版本之间的最大区别在于函数的说明与定义方法。在 C语言的最初定义中，power函数要写成如下形式：

```
/* power: 求底的n次幂；n >=0  */
/*  (老方式版本)  */
power(base, n)
int base, n;
{
    int  i, p;

    p = 1;
    for ( i = 1; i <= n; ++i )
        p = p * base;
    return p;
}
```

参数的名字在圆括号中指定，但参数类型则在左花括号之前说明，如果一个参数未在这一位置加以说明，那么缺省为 int类型。（函数的体与ANSI C相同。）

在程序的开始处，也可以将power说明成如下形式：

```
int  power( );
```

在函数说明中不包含参数，这样编译程序就不能马上对调用 power的合法性进行检查。实际上，由于在缺省情况下假定power要返回int类型的值，这个函数说明可以全部省去。

在新定义的函数原型的语法中，编译程序可以很容易检测函数调用在变元数目和类型方面的错误。在ANSI C中仍可以使用老方式的函数说明与定义，至少它可以作为一个过渡阶段。但我们还是强烈建议读者：在可以使用支持新方式的编译程序时，最好使用新形式的函数原型。

练习1-15　重写1.2节的温度转换程序，使用函数来实现温度转换。

1.8　变元——按值调用

使用其他语言（特别是FORTRAN语言）的程序员可能对C语言有关参数的这样一个方面不太熟悉。在C语言中，所有函数变元都是“按值”传递的。这意味着，被调用函数所得到的变元值放在临时变量中而不是放在原来的变量中。这样它的性质就与诸如FORTRAN等采用“按引用调用”的语言或诸如Pascal等采用var参数的语言有所不同，在这些语言中，被调用函数必须访问原来的变元，而不是采用局部复制的方法。

最主要的区别在于，在C语言中，被调用函数不能直接更改调用函数中变量的值，它只能更改其私有临时拷贝的值。

按值调用有益处而非弊端。由于在采用按值调用时在被调用函数中参数可以像通常的局部变量一样处理，这样可以使函数中只使用少量的外部变量，从而使程序更简洁。例如，下面是利用这一性质的power版本：

```
/* power: 求底的n次幂； n >=0；第2个版本  */
int  power(int base, int n)
{
    int  p;

    for ( n = 1; n > 0; --n )
        p = p * base;
    return p;
}
```

参数n被用做临时变量，（通过一个向后执行的for循环语句）向下计数，一直到其值变成0，这样就不再需要引入变量i。在power函数内部对n的操作不会影响到调用函数在调用power时所使用的变元的值。

在必要时，也可以在对函数改写，使之可以修改调用例程中的变量。此时调用者要向被调用函数提供所要改变值的变量的地址（从技术角度看，地址就是指向变量的指针），而被调用函数则要把对应的参数说明成指针类型，并通过它间接访问变量。我们将在第5章讨论指针。

对数组的情况有所不同。当把数组名用做变元时，传递给函数的值是数组开始处的位置或地址——不是数组元素的副本。在被调用函数中可以通过数组下标来访问或改变数组元素的值。这是下一节所要讨论的问题。

1.9　字符数组

C语言中最常用的数组类型是字符数组。为了说明字符数组以及用于处理字符数组的函数的用法，我们来编写一个程序，它用于读入一组文本行并把最长的文本行打印出来。对其算法描述相当简单：

```
while  (还有没有处理的行)
   if  (该行比已处理的最长的行还要长)
       保存该行
```

保存该行的长度
打印最长的行

这一算法描述很清楚，很自然地把所要编写的程序分成了若干部分，分别用于读入新行、测试读入的行、保存该行及控制这一过程。

由于分割得比较好，故可以像这样来编写程序。首先编写一个独立的函数 getline来读取输入的下一行。我们想使这个函数在其他地方也能使用。 getline函数至少在读到文件末尾时要返回一个信号，而更有用的设计是它能在读入文本行时返回该行的长度，而在遇到文件结束符时返回 0。由于0不是有效的行长度，因此是一个可以接受的标记文件结束的返回值。每一行至少要有一个字符，只包含换行符的行的长度为 1。

当发现某一个新读入的行比以前读入的最长的行还要长时，就要把该新行保存起来。这意味着需要用第二个函数copy来把新行复制到一个安全的位置。

最后，需要用主函数main来控制对getline和copy这两个函数的调用。整个程序如下：

```
#include  <stdio.h>
#define  MAXLINE  1000    /* 最大输入行的大小 */

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

/* 打印最长的输入行 */
main ( )
{
    int  len;                /* 当前行长度 */
    int  max;                /* 至目前为止所发现的最长行的长度 */
    char  line[MAXLINE];     /* 当前输入的行 */
    char  longest[MAXLINE];  /* 用于保存最长的行 */

    max = 0;
    while  ( ( len = getline (line, MAXLINE) ) > 0 )
        if  (len > max)  {
            max = len;
            copy ( longest, line );
        }
    if  (max > 0)    /* 有一行 */
        printf ("%s" , longest ) ;
    return 0 ;
}

/* getline：将一行读入s中并返回其长度 */
int getline (char s [ ], int lim)
{
    int  c, i;

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

```
    if   (c == '\n' )  {
         s[i] = c;
         ++i;
    }
    s[i] =  '\0';
    return i;
}

/* copy：从from拷贝到to; 假定to足够大 */
void copy ( char to [ ], char from [ ])
{
    int i;

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

在程序的一开始就对getline和copy这两个函数进行了说明，假定它们都放在同一文件中。

main与getline这两个函数通过一对变元及一个返回值进行交换。在getline函数中，两个变元是通过程序行

```
int getline (char s [ ], int lim)
```

说明的，它把第一个变元s说明成数组，把第二个变元lim说明成整数。在说明中提供数组大小的目的是留出存储空间。在getline函数中没有必要说明数组s的长度，因为该数组的大小是在main函数中设置的。如同power函数一样，getline函数使用了一个return语句把值回送给其调用者。这一程序行也说明了getline函数的返回值类型为int，由于int为缺省返回值类型，故可以省略。

有些函数要返回一个有用的值，而另外一些函数（如 copy）则仅用于执行一些动作，并不返回值。copy函数的返回类型为void，它用于显式指明该函数不返回任何值。

getline函数把字符 '\0'（即空字符，其值为0）放到它所建立的数组的末尾，以标记字符串的结束。这一约定也已被C语言采用，当在C程序中出现诸如

```
"hello\n"
```

的字符串常量时，它被作为字符数组存储，该数组中包含这个字符串中各个字符并以 '\0' 来标记字符串结束：

h	e	l	l	o	\n	\0

在printf库函数中，格式说明%s要求所对应的变元是以这种形式表示的字符串。 copy函数也基于这样的事实，其输入变元值以 '\0' 结束，并将这个字符拷贝到输出变元中。（所有这一切都意味着空字符 '\0' 不是通常文本的一部分。）

值得一提的是，在传递参数（变元）时，即使是像本例这样很小的程序也会遇到某些麻烦的设计问题。例如，当所遇到的行比所允许的最大值还要大时， main函数应该怎么处理？getline函数的执行是安全的，当数组满了时它就停止读字符，即使它还没有遇到换行符。 main函数可以通过测试行长度以及所返回的最后一个字符来判定该行是否太长，然后再按其需要进

行处理。为了使程序简洁一些，我们在这里将不处理这个问题。

getline函数的调用程序没有办法预先知道一个输入行可能有多长，故 getline函数采用了检查溢出的方法。另一方面，copy函数的调用程序则已经知道（或可以找出）所处理字符串的长度，故其中没有错误检查处理。

练习1-16 对用于打印最长行的程序的主程序 main进行修改，使之可以打印任意长度的输入行的长度以及文本行中尽可能多的字符。

练习1-17 编写一个程序，把所有长度大于 80个字符的输入行都打印出来。

练习1-18 编写一个程序，把每个输入行中的尾部空格及制表符都删除掉，并删除空格行。

练习1-19 编写函数reverse (s)，把字符串s颠倒过来。用它编写一个程序，一次把一个输入行字符串颠倒过来。

1.10 外部变量与作用域

main函数中的变量（如line、longest等）是main函数私有的或称局部于 main函数的。由于它们是在main函数中说明的，其他函数不能直接访问它们。在其他函数中说明的变量也同样如此，例如，getline函数中说明的变量i与copy函数中说明的变量i没有关系。函数中的每一个局部变量只在该函数被调用时存在，在该函数执行完退出时消失。这也是仿照其他语言通常把这类变量称为自动变量的原因。以后我们将用自动变量来指局部变量。（第4章将讨论静态存储类，属于静态存储类的局部变量在函数调用之间保持其值不变。）

由于自动变量只在函数调用执行期间存在，故在函数的两次调用期间自动变量不保留在前次调用时所赋的值，且在函数的每次调用执行时都要显式给其赋初值。如果没有给自动变量赋初值，那么其中所存放的是无用数据。

作为对自动变量的替补，可以定义适用于所有函数的外部变量，即可以被所有函数通过变量名访问的变量。（这一机制非常类似于 FORTRAN语言的COMMON变量或Pascal语言在最外层分程序中说明的变量。）由于外部变量可以全局访问，因此可以用外部变量代替变元表用于在函数间交换数据。而且，外部变量在程序执行期间一直存在，而不是在函数调用时产生、在函数执行完时消失，即使从为其赋值的函数返回后仍保留原来的值不变。

外部变量必须在所有函数之外定义，且只能定义一次，定义的目的是为之分配存储单元。在每一个函数中都要对所要访问的外部变量进行说明，说明所使用外部变量的类型。在说明时可以用extern语句显式指明，也可以通过上下文隐式说明。为了更具体地讨论外部变量，我们重写上面用于打印最长文本行的程序，把 line、longest与max说明成外部变量。这需要修改所有这三个函数的调用、说明与函数体。

```
#include  <stdio.h>

#define  MAXLINE  1000    /* 最大输入行的大小 */

int  max; /* 至目前为止所发现的最长行的长度 */
char  line[MAXLINE];      /* 当前输入的行 */
```

```
char  longest[MAXLINE];   /* 用于保存最长的行 */

int  getline (void );
void  copy ( void );

/* 打印最长的输入行； 特别版本 */
main ( )
{
    int  len;
    extern int  max;
    extern char  longest[ ];

    max = 0;
    while  ( ( len = getline ( ) ) > 0 )
        if  (len > max)  {
            max = len;
            copy ( );
        }
    if  (max > 0)    /* 有一行 */
        printf ("%s" , longest ) ;
    return 0 ;
}

/* getline：特别版本 */
int getline (void )
{
    int  c, i;
    extern char  line[ ];

    for  (i = 0; i < MAXLINE -1 && (c = getchar ( ) ) != EOF && c != '\n'; ++i )
         line[i] = c;
    if  (c == '\n' )  {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy：特别版本 */
void copy ( void )
{
    int i;
    extern char  line[ ], longest[ ];

    i = 0;
    while  ( ( longest[ i ] = line [ i ]) != '\0')
```



```
        ++i;
}
```

在这个例子中，前几行定义了在main、getline与copy函数中使用的几个外部变量，指明各外部变量的类型并使系统为之分配存储单元。从语法角度看，外部变量定义就像局部变量的定义一样，但由于它们出现在各个函数的外部，这些变量就成了外部变量。一个函数在使用外部变量之前必须使变量的名字在该函数中可见，一种方法是在该函数中编写一个 extern说明，说明除了在前面加了一个关键词extern外，其他地方均与普通说明相同。

在某些情况下，extern说明可以省略。如果外部变量的定义在源文件中出现在使用它的函数之前，那么在该函数中就没有必要使用 extern说明。于是，main、getline及copy中的几个extern说明都是多余的。事实上，比较常用的做法是把所有外部变量的定义放在源文件的开始处，这样就可以省略extern说明。

如果程序包含几个源文件，某个变量在 file1文件中定义、在file2与file3文件中使用，那么在file2与file3文件中就需要使用 extern说明来连接该变量的出现。人们通常把变量的 extern说明与函数放在一个单独的文件中（历史上叫做头文件），在每一个源文件的前面用 #include语句把所要用的头文件包含进来。后缀 .h 被约定为头文件名的扩展名。例如，标准库中的函数就是在诸如<stdio.h>的头文件中说明的。这一问题将在第 4章详细讨论，而库本身在第 7章及附录B中讨论。

由于getline与copy函数的特别版本中不带有变元，从道理上讲，在源文件开始处它们的原型应该是getline( )与copy( )。但为了与较老的C程序相兼容，C标准把空变元表作为老方式说明，并关闭所有对变元表的检查。如果变元表本身是空的，那么要使用关键词 void，第4章将对此做进一步讨论。

读者应该注意到，这一节我们在说到外部变量时很小心谨慎地使用着两个词定义与说明。“定义”指变量建立或分配存储单元的位置，而“说明”则指指明变量性质的位置，但并不分配存储单元。

顺便指出，现在有一种把所有看得见的东西都作为外部变量的趋势，因为这样似乎可以简化通信 —— 变元表变短了，且变量在需要时总是存在。但外部变量即使在不需要时也还是存在的。过分依赖于外部变量充满了危险，因为这将会使程序中的数据联系变得很不明显 —— 外部变量的值可能会被意外地或不经意地改变，程序也变得难以修改。上面打印最长行的程序的第 2个版本就不如第1个版本，之所以如此，部分是由于这个原因，部分是由于它把两个有用的函数所操纵的变量的名字绑到函数中，使这两个函数失去了一般性。

到目前为此，我们已经对C语言传统的核心部分进行了介绍。借助于这少量的构件，我们已经能编写出相当规模的程序，因此建议读者花上较长的时间来编写一些程序。下面给出的练习比本章前面的程序复杂一些。

练习1-20　编写程序detab，将输入中的制表符替换成适当数目的空白符（使空白充满到下一制表符停止位）。假定制表符停止位的位置是固定的，比如在每个 n列的位置上。n应为变量或符号参数吗？

练习1-21　编写程序entab，将空白符串用可达到相同空白的最小数目的制表符和空白符来

替换。使用与detab程序相同的制表停止位。请问，当一个制表符与一个空白符都可以到达制表符停止位时，选用哪一个比较好？

练习1-22　编写一个程序，用于把较长的输入行“折”成短一些的两行或多行，折行的位置在输入的第n列之前的最后一个非空白符之后。要保证程序具备一定的智能，能应付输入行很长以及在指定的列前没有空白符或制表符时的情况。

练习1-23　编写一个用于把C程序中所有注解都删除掉的程序。不要忘记处理好带引号的字符串与字符常量。在C程序中注解不允许嵌套。

练习1-24　编写一个程序，查找C程序中的基本语法错误，如圆括号、方括号、花括号不配对等。不要忘记引号（包括单引号和双引号）、换码序列与注解。（如果读者想把该程序编写成完全通用性的，那么难度比较大。）

第2章 类型、运算符与表达式

变量与常量是程序中所要处理的两种基本数据对象。说明语句中列出了所要使用的变量的名字及该变量的类型，可能还要给出该变量的初值。运算符用于指定要对变量与常量进行的操作。表达式则用于把变量与常量组合起来产生新的值。一个对象的类型决定着该对象可取值的集合以及可以对该对象施行的运算。本章将要对这些构件进行详细讨论。

ANSI C语言标准对语言的基本类型与表达式做了许多小的修改与增补。所有整数类型现在都有signed（有符号）与unsigned（无符号）两种形式，且可以表示无符号常量与十六进制字符常量。浮点运算可以以单精度进行，另外还可以使用更高精度的long double类型。字符串常量可以在编译时连接。枚举现在也成了语言的一部分，这是经过长期努力才形成的语言特征。对象可以说明成 const（常量），这种对象的值不能进行修改。语言还对算术类型之间的自动强制转换规则做了扩充，使这一规则可以适合更多的数据类型。

2.1 变量名

对变量与符号常量的名字存在着一些限制，这一点在第 1章中没有指出来。名字由字母与数字组成，但其第一个字符必须为字母。下划线_也被看做是字母，它有时可用于命名比较长的变量名以提高可读性。由于库函数通常使用以下划线开头的名字，因此不要将这类名字用做变量名。大写字母与小写字母是有区别的， x与X是两个不同的名字，一般把由大写字母组成的名字用做符号常量。

在内部名字中至少前31个字符是有效的。对于函数名与外部变量名，其中所包含的字符的数目可以小于31个，这是因为它们可能会被语言无法控制的汇编程序和装配程序使用。对于外部名，ANSI C标准保证了唯一性仅对前6个字符而言并且不区分大小写。诸如if、else、int、float等关键词是保留的，不能把它们用做变量名。所有关键词中的字符都必须小写。

在选择变量名时比较明智的方法是使所选名字的含义能表达变量的用途。我们倾向于局部变量使用比较短的名字（尤其是循环控制变量，亦叫循环位标），外部变量使用比较长的名字。

2.2 数据类型与大小

在C语言中只有如下几个基本数据类型：

char	单字节，可以存放字符集中一个字符。
int	整数，一般反映了宿主机上整数的自然大小。
float	单精度浮点数。
double	双精度浮点数。

此外，还有一些可用于限定这些基本类型的限定符。其中 short与long这两个限定符用于限定整数类型：

```
short int  sh;
long int  counter;
```

在这种说明中，int可以省去，一般情况下许多人也是这么做的。

引入这两个类型限定符的目的是为了使short与long提供各种满足实际要求的不同长度的整数。int通常反映特定机器的自然大小，一般为16位或32位，short对象一般为16位，long对象一般为32位。各个编译程序可以根据其硬件自由选择适当的大小，唯一的限制是， short与int对象至少要有16位，而long对象至少要有32位。short对象不得长于int对象，而int对象则不得长于long对象。

类型限定符signed与unsigned可用于限定char类型或任何整数类型。经unsigned限定符限定的数总是正的或0，并服从算术模2^n定律，其中n是该类型机器表示的位数。例如，如果 char对象占用8位，那么unsigned char变量的取值范围为0~255，而signed char变量的取值范围则为- 128~127（在采用补码的机器上）。普通char对象是有符号的还是无符号的则取决于具体机器，但可打印字符总是正的。

long double类型用于指定高精度的浮点数。如同整数一样，浮点对象的大小也是由实现定义的，float、double与long double类型的对象可以具有同样大小，也可以表示两种或三种不同的大小。

在标准头文件<limits.h>与<float.h>中包含了有关所有这些类型的符号常量以及机器与编译程序的其他性质。这些内容将在附录 B中讨论。

练习2-1　编写一个程序来确定 signed及unsigned的char、short、int与long变量的取值范围，可以通过打印标准头文件中的相应值来完成，也可以直接计算来做。后一种方法较困难一些，因为要确定各种浮点类型的取值范围。

2.3　常量

诸如1234一类的整数常量是 int常量。 long常量要以字母 l或L结尾，如123456789L。一个整数常量如果大到在int类型中放不下，那么也被当做 long常量处理。无符号常量以字母 u或U结尾，后缀ul或UL用于表示unsigned long常量。

浮点常量中必须包含一个小数点（如 123.4）或指数（如1e-2）或两者都包含，在没有后缀时类型为double。后缀f与F用于指定float常量，而后缀l或L则用于指定long double常量。

整数值除了用十进制表示外，还可以用八进制或十六进制表示。如果一个整数常量的第一个数字为0，那么这个数就是八进制数；如果第一个数字为0x或0X，那么这个数就是十六进制数。例如，十进制数31可以写成八进制数037，也可以写成十六进制数0x1f或0X1F。在八进制与十六进制常量中也可以带有后缀 l或L（long，表示长八进制或十六进制常量）以及后缀 u或U（unsigned，表示无符号八进制或十六进制常量），例如，0XFUL是一个unsigned long 常量（无符号长整数常量），其值等于十进制数15。

字符常量是一个整数，写成用单引号括住单个字符的的形式，如 'x'。字符常量的值是该字符在机器字符集中的数值。例如，在ASCII字符集中，字符 '0' 的值为48，与数值0没有关系。如果用字符 '0' 来代替像48一类的依赖于字符集的数值，那么程序会因独立于特定的值而更易于阅读。虽然字符常量一般用来与其他字符进行比较，但字符常量也可以像整数一样参与数值运算。

有些字符用字符常量表示，这种字符常量是诸如 \n （换行符）的换码序列，换码序列看起来像两个字符，但只用来表示一个字符。此外，我们可以用

```
'\ooo'
```

来指定字节大小的位模式，*ooo*是1~3个八进制数字（0…7）。位模式还可以用

```
'\xhh'
```

来指定，*hh*是一个或多个十六进制数字（0…9，a…f，A…F）。因此，可以如下写：

```
#define  VTAB  '\013'     /* ASCII纵向制表符 */
#define  BELL  '\007'     /* ASCII响铃符 */
```

也可以用十六进制写

```
#define  VTAB  '\xb'     /* ASCII纵向制表符 */
#define  BELL  '\x7'     /* ASCII响铃符 */
```

下面是所有的换码序列：

\a	响铃符	\\	反斜杠
\b	回退符	\?	问号
\f	换页符	\'	单引号
\n	换行符	\"	双引号
\r	回车符	*ooo*	八进制数
\t	横向制表符	*xhh*	十六进制数
\v	纵向制表符		

字符常量 '\0' 表示其值为0的字符，即空字符。我们用 '\0' 来代替0，以在某些表达式中强调字符的性质，但其数字值就是0。

常量表达式是其中只涉及到常量的表达式。这种表达式可以在编译时计算而不必推迟到运行时，因而可以用在常量可以出现的任何位置，例如：

```
#define  MAXLINE  1000
char  line[MAXLINE+1];
```

或：

```
#define  LEAP  1    /* 闰年 */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

字符串常量也叫字符串字面值，是用双引号括住的由 0个或多个字符组成的字符序列。例如：

```
"I am a string"
```

或：

```
""   /* 空字符串 */
```

双引号不是字符串的一部分，它只用于限定字符串。在字符常量中使用的换码序列同样也可以用在字符串中，在字符串中用 \" 表示双引号字符。编译时可以将多个字符串常量连接起来：

```
"hello," " world"
```

等价于

```
"hello, world"
```

这种表示方法可用于将比较长的字符串分成若干源文件行。

从技术角度看，字符串常量就是字符数组。在内部表示字符串时要用一个空字符 '\0' 来结尾，故用于存储字符串的物理存储单元数比括在双引号中的字符数多一个。这种表示方法意味着， C 语言对字符串的长度没有限制，但程序必须扫描完整个字符串才能决定这个字符串的长度。标准库函数strlen(s) 用于返回其字符串变元s的长度，不包括末尾的 '\0'。下面是我们设计的一个版本：

```
    /* strlen：返回s的长度 */
    int strlen(char s[])
    {
    int i;

    i = 0;
    while  (s[i] != '\0')
        ++i;
    return i;
}
```

strlen等字符串函数均说明在标准头文件 <string.h>中。

请仔细区分字符常量与只包含一个字符的字符串的区别： 'x'与 "x"不相同。前者是一个整数，用于产生字母x在机器字符集中的数值（内部表示值）；后者是一个只包含一个字符（即字母 x）与一个 '\0'的字符数组。

另外还有一种常量，叫做枚举常量。枚举是常量整数值的列表，如同下面一样：

```
enum boolean { NO, YES };
```

在enum说明中第一个枚举名的值为 0，第二个为 1，如此等等，除非指定了显式值。如果不是所有值都指定了，那么未指定名字的值依着最后一个指定值向后递增，如同下面两个说明中的第二个说明：

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                /* FEB的值为2，MAR的值为3，等等。*/
```

不同的枚举中的名字必须各不相同，同一枚举中各个名字的值不要求不同。

枚举是使常量值与名字相关联的又一种方便的方法，其相对于 #define语句的优势是常量值可以由自己控制。虽然可以说明 enum类型的变量，但编译程序不必检查在这个变量中存储的值是否为该枚举的有效值。不过枚举变量仍然提供了做这种检查的机会，故其比 #define更具优势。此外，调试程序能以符号形式打印出枚举变量的值。

2.4 说明

除了某些可以通过上下文做的隐式说明外，所有变量都必须先说明后使用。说明中不仅要

指定类型，还要包含由一个或多个该类型的变量组成的变量表。例如：

```
int  lower, upper, step;
char  c, line [1000];
```

同一类型的变量可以以任何方式分散在多个说明中，上面两个说明也可以等价地写成如下五个说明：

```
int  lower;
int  upper;
int  step;
char  c;
char  line [1000];
```

后一种形式需要占用较多的空间，但这样不仅便于向各个说明中增添注解，也便于以后的修改。

变量在说明时可以同时初始化。如果所说明的变量名后跟一个等号与一个表达式，那么这个表达式被作为初始化符。例如：

```
char  esc = '\\';
int  i = 0;
int  limit =MAXLINE+1;
float eps = 1.0e-5;
```

如果所涉及的变量不是自动变量，那么只初始化一次，而且从概念上讲应该在程序开始执行之前进行，此时要求初始化符必须为常量表达式。显式初始化的自动变量每当进入其所在的函数或分程序时就进行一次初始化，其初始化符可以是任何表达式。外部变量与静态变量的缺省初值为0。未经显式初始化的自动变量的值为未定义值（即为垃圾）。

在变量说明中可以用const限定符限定，该限定符用于指定该变量的值不能改变。对于数组，const限定符使数组所有元素的值都不能改变：

```
const double e = 2.71828182845905;
const char msg [ ] = "warning:";
```

const说明也可用于数组变元，表明函数不能改变数组的值：

```
int  strlen(const char[]);
```

如果试图修改const限定的值，那么所产生的后果取决于具体实现。

2.5 算术运算符

二元算术运算符包括+、-、*、/以及取模运算符%。整数除法要截取掉结果的小数部分。表达式

```
x % y
```

的结果是x除以y的余数，当y能整除x时，x % y的结果为0。例如，如果某一年的年份能被4整除但不能被100整除，那么这一年就是闰年，此外，能被400整除的年份也是闰年。因此，有

```
if ( (year % 4 == 0 && year % 100 != 0) || year % 400 == 0 )
    printf( "%d is a leap year\n", year );
else
    printf("%d is not a leap year\n", year );
```

取模运算符%不能作用于float或double对象。在有负的运算分量时，整数除法截取的方向以及取模运算结果的符号取决于具体机器，在出现上溢或下溢时所要采取的动作也取决于具体机器。

二元＋和－运算符的优先级相同，但它们的优先级比 *、/和%的优先级低，而后者又比一元＋和－运算符低。算术运算符采用从左至右的结合规则。

本章末尾的表2-1总结了所有运算符的优先级和结合律。

2.6 关系运算符与逻辑运算符

关系运算符有如下几个：

```
>   >=   <   <=
```

所有关系运算符具有相同的优先级。优先级正好比它们低一级的是等于运算符：

```
==   !=
```

关系运算符的优先级比算术运算符低。因而表达式

```
i < lim -1
```

等于

```
i < (lim - 1)
```

更有趣的是逻辑运算符 &&与||。由&&与||连接的表达式从左至右计算，并且一旦知道结果的真假值就立即停止计算。绝大多数 C程序利用了这些性质。例如，下面这个循环语句取自第１章的输入函数getline：

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

在读一个新字符之前必须先检查一下在数组 s中是否还有空间存放这个字符，因此首先必须测试是否i < lim -1。而且，如果这一测试失败（即i < lim -1不成立），那么就没有必要继续读下一字符。

类似地，如果在调用 getchar函数之前就对c是否为EOF进行测试，那么也是令人遗憾的，因此，函数调用与赋值都必须在对 c中的字符进行测试之前完成。

&&运算符的优先级比 ||运算符的优先级高，但两者都比关系运算符和等于运算符的优先级低。从而，像

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

之类的表达式就不需要另外加圆括号了。但是，由于 !=运算符的优先级高于赋值运算符 =的优先级，在子表达式

```
(c = getchar()) != '\n'
```

中，圆括号还是需要的，这样才能达到我们所希望的先把函数值赋给 c再与 '\n' 进行比较的效果。

按照定义，如果关系表达式与逻辑表达式的计算结果为真，那么它们的值为 1；如果为假，那么它们的值为0。

一元求反运算符!用于将非0运算分量转换成0，把0运算分量转换成1。该运算符通常用在诸如

```
if ( !valid )
```

一类的构造中，一般不用

```
if ( valid == 0 )
```

来代替。要想笼统地说哪个更好比较难。诸如 !valid一类的构造读起来好听一点（“如果不是有效的”），但这种形式在比较复杂的情况下可能难于理解。

练习2-2　不使用&&或||运算符编写一个与上面的for循环语句等价的循环语句。

2.7　类型转换

当一个运算符的几个运算分量的类型不相同时，要根据一些规则把它们转换成某个共同的类型。一般而言，只能把“比较窄的”运算分量自动转换成“比较宽的”运算分量，这样才能不丢失信息，例如，在诸如

```
f + i
```

一类的表达式的计算中要把整数变量i的值自动转换成浮点类型。不允许使用没有意义的表达式，例如，不允许把float表达式用作下标。可能丢失信息的表达式可能会招来警告信息，如把较长整数类型的值赋给较短整数类型的变量，把浮点类型赋给整数类型，等等，但不是非法表达式。

由于char类型就是小整数类型，在算术表达式中可以自由地使用 char类型的变量或常量。这就使得在某些字符转换中有了很大的灵活性。一个例子是用于将数字字符串转换成对应的数值的函数atoi：

```
/* atoi：将字符串s转换成整数  */
int atoi( char s[])
{
     int  i, n;

     n = 0;
     for ( i = 0; s[i] >= '0' && s[i] <= '9'; ++i )
         n = 10 * n + (s[i] - '0');
     return n;
}
```

正如第１章所述，表达式

```
s[i] - '0'
```

用于求s[i]中存储的字符所对应的数字值，因为 '0'、'1'、'2' 等的值形成了一个连续的递增序列。

将字符转换成整数的另一个例子是函数 lower，它把ASCII字符集中的字符映射成对应的小写字母。如果所要转换的字符不是大写字母，那么 lower函数返回原来的值。

```
/* lower：把字符c转换成小写字母；仅对ASCII字符集 */
int lower (int c)
{
        if (c >= 'A' && c<= 'Z' )
            return  c + 'a' - 'A';
    else
```

```
        return c;
}
```

这个函数是为ASCII字符集设计的。在ASCII字符集中，大写字母与对应的小写字母像数字值一样有固定的距离，并且每一个字母都是连续的——在A至Z之间只有字母。然而，后一个结论对于EBCDIC字符集不成立，故这一函数在EBCDIC字符集不只是转换了字母。

附录B中介绍的标准头文件 <ctype.h>定义了一组用于进行独立于字符集的测试和转换的函数。例如，tolower(c)函数用于在c为大写字母时将之转换成小写字母，故 tolower是上述lower函数的替代函数。同样条件，

```
c >= '0' && c<= '9'
```

可以用

```
isdigit(c)
```

代替。

我们从现在起要使用<ctype.h>中定义的函数。

在将字符转换成整数时有一点比较微妙。 C语言没有指定char类型变量是无符号量还是有符号量。当把一个 char类型的值转换成 int类型的值时，其结果是不是为负整数？结果视机器的不同而有所变化，反映了不同机器结构之间的区别。在某些机器上，如果字符的最左一位为 1，那么就被转换成负整数（称做“ 符号扩展 ”）。在另一些机器上，采取的是提升的方法，通过在最左边加上0把字符提升为整数，这样转换的结果总是正的。

C语言的定义保证了机器的标准可打印字符集中的字符不会是负的，故在表达式中这些字符总是正的。但是，字符变量存储的位模式在某些机器上可能是负的，而在另一些机器上却是正的。为了保证程序的可移植性，如果要在 char变量中存储非字符数据，那么最好指定 signed或unsigned限定符。

关系表达式（如i > j）和由&&与||连接的逻辑表达式的值在其结果为真时为 1，在其结果为假时为0。因此，赋值语句

```
d = c >= '0' && c <= '9'
```

在c的值为数字时将d置为1，否则将d置为0。然而，诸如isdigit一类的函数在变元为真时返回的可能是任意非0值。在if、while、for等语句的测试部分，“ 真 ”的意思是“ 非 0 ”，从这个意义上看，它们没有什么区别。

我们很希望能进行隐式算术类型转换。一般而言，如果诸如 + 或 * 等二元运算符的两个运算分量具有不同的类型，那么在进行运算之前先要把“ 低 ”的类型提升为“ 高 ”的类型。附录A.6节严格地给出了转换规则。然而如果没有无符号类型的运算分量，那么只要使用如下一组非正式的规则就够了：

如果某个运算分量的类型为 long double，那么将另一个运算分量也转换成 long double类型；
否则，如果某个运算分量的类型为 double，那么将另一个运算分量也转换成 double类型；
否则，如果某个运算分量的类型为 float，那么将另一个运算分量也转换成 float类型；
否则，将char与short类型的运算分量转换成int类型。
然后，如果某个运算分量的类型为 long，那么将另一个运算分量也转换成 long类型。

注意，在表达式中float类型的运算分量不自动转换成double类型，这与原来的定义不同。一般而言，数学函数（如定义在标准头文件 <math.h>中的函数）要使用双精度。使用 float类型的主要原因是为了在使用较大的数组时节省存储单元，有时也为了节省机器执行时间（双精度算术运算特别费时）。

当表达式中包含unsigned类型的运算分量时，转换规则要复杂一些。主要问题是，在有符号值与无符号值之间的比较运算取决于机器，因为它们取决于各个整数类型的大小。例如，假定int对象占16位，long对象占32位，那么，-1L < 1U，这是因为int类型的-1U被提升为signed long类型；但-1L > 1UL，这是因为-1L被提升为unsigned long类型，因此它是一个比较大的正数。

在进行赋值时也要进行类型转换，= 右边的值要转换成左边变量的类型，后者即赋值表达式结果的类型。

如前所述，不管是否要进行符号扩展，字符值都要转换成整数值。

当把较长的整型数转换成较短的整型数或字符时，要把超出的高位部分丢掉。于是，当程序

```
int  i;
char c;

i = c;
c = i;
```

执行后，c的值保持不变，无论是否要进行符号扩展。然而，如果把两个赋值语句的次序颠倒一下，那么执行后可能会丢失信息。

如果x是float类型且i是int类型，那么

```
x = i
```

与

```
i = x
```

这两个赋值表达式在执行时都要引起类型转换，当把float类型转换成int类型时要把小数部分截取掉；当把double类型转换成float类型时，是四舍五入还是截取取决于具体实现。

由于函数调用的变元是表达式，当把变元传递给函数时也可能引起类型转换。在没有函数原型的情况下，char与short类型转换成int类型，float类型转换成double类型，这就是即使在函数是用char与float类型的变元表达式调用时仍把参数说明成 int与double类型的原因。

最后，在任何表达式中都可以进行显式类型转换（即所谓的“强制转换”），这时要使用一个叫做强制转换的一元运算符。在如下构造中，表达式被按上述转换规则转换成由类型名所指名的类型：

（类型名）表达式

强制转换的精确含义是，表达式首先被赋给类型名指定类型的某个变量，然后再将其用在整个构造所在的位置。例如，库函数 sqrt需要一个double类型的变元，但如果在其他地方作了不适当的处理，那么就会产生无意义的结果（sqrt是在<math.h>中说明的一个函数）。因而，如果n是整数，那么可以用

```
sqrt ( (double) n)
```

使得在把n传递给sqrt函数之前先把n的值转换成double类型。注意，强制转换只是以指明的类型产生n的值，n本身的值没有改变。强制转换运算符与其他一元运算符具有相同的优先级，如同本章末尾的表中所总结的那样。

如果变元是通过函数原型说明的，那么在通常情况下，当该函数被调用时，系统对变元自动进行强制转换。于是，对于sqrt的函数原型

```
double sqrt(double);
```

调用

```
root = sqrt(2);
```

不需要强制转换运算符就自动将把整数2强制转换成double类型的值2.0。

在标准库中包含了一个用于实现伪随机数发生器的函数rand与一个用于初始化种子的函数srand。在前一个函数中使用了强制转换：

```
unsigned long int next = 1;

/* rand：返回取值在0~32767之间的伪随机数 */
int rand(void)
{
    next = next *1103515245 +12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand：为rand()函数设置种子 */
void srand(unsigned int seed)
{
    next = seed;
}
```

练习2-3 编写函数htoi(s)，把由十六进制数字组成的字符串（前面可能包含0x或0X）转换成等价的整数值。字符串中允许的数字为：0~9，a~f、以及A~F。

2.8 加一与减一运算符

C语言为变量增加与减少提供了两个奇特的运算符。加一运算符++用于使其运算分量加1，减一运算符--用于使其运算分量减1。我们常常用++运算符来使变量的值加1，如在下述语句中一样：

```
if  (c == '\n')
    ++nl;
```

++与--这两个运算符奇特的方面在于，它们既可以用作前缀运算符（用在变量前面，如++n），也可以用作后缀运算符（用在变量后面，如n++）。在这两种情况下，效果都是使n加1。但是，它们之间仍存在一点区别，表达式

```
++n
```

在n的值被使用之前先使n加1，而表达式

```
n++
```

则是在n的值被使用之后再使n加1。这意味着，在该值被使用的上下文中，++n和n++的效果是不同的。如果n的值是５，那么

```
x = n++;
```

将x的值置为5，而

```
x = ++n;
```

则将x的值置为６。在这两个语句执行完后，n的值都是６。加一与减一运算符只能作用于变量，诸如

```
(i + j)++
```

一类的表达式是非法的。

在除了加1的运算效果，不需要任何具体值的地方，如表达式

```
if  (c == '\n')
    nl++;
```

中，++作为前缀与后缀效果是一样的。在有些情况下需要特别指定。例如，考虑下面的函数squeeze(s,c)，它用于从字符串s中把所有出现的字符c都删除掉：

```
/* squeeze：从s中删除掉c  */

void squeeze(char s[], int c)
{
    int  i, j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

每当出现一个不等于c的字符时，就把它拷贝到j的当前值所指向的位置，并将j的值加1，以准备处理下一个字符。其中的if语句完全等价于

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

具有类似构造的另一个例子是第１章的getline函数。我们可以将这个函数中的if语句

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

用更为精致的if语句

```
if (c == '\n')
    s[i++] = c;
```

代替。

作为第三个例子，再看一下标准函数 strcat(s,t)，它用于把字符串 t连接到字符串 s的后面。strcat函数假定在s中有足够的空间来保存这两个字符串连接的结果。下面所编写的这个函数没有返回任何值（在标准库中，这个函数要返回一个指向新字符串的指针）：

```
/* strcat：把字符串t连接到字符串s的后面；s必须有足够大的空间  */
void strcat(char s[], char t[])
{
     int i, j;

     i = j = 0;
     while (s[i] != '\0')    /*  找到s的末尾  */
         i++;
     while ( (s[i++] = t[j++]) != '\0')    /*  拷贝t  */
         ;
}
```

在将t中的字符逐个拷贝到s后面时，用后缀运算符 ++ 作用于i与j，以保证在循环过程中i与j均指向下一个位置。

练习2-4　重写squeeze(s1,s2)函数，把字符串s1中与字符串s2中字符匹配的各个字符都删除掉。

练习2-5　编写函数any(s1,s2)，它把字符串s2中任一字符在字符串s1中的第一次出现的位置作为结果返回。如果s1中没有包含s2中的字符，那么返回-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。）

2.9　按位运算符

C语言提供了六个用于位操作的运算符，这些运算符只能作用于整数分量，即只能作用于有符号或无符号的char、short、int与long类型：

```
&       按位与（AND）
|       按位或（OR）
^       按位异或（XOR）
<<      左移
>>      右移
~       求反码（一元运算符）
```

按位与运算符&经常用于屏蔽某些位，例如：

```
n = n & 0177;
```

用于将n除7个低位外的各位置成0。

按位或运算符 | 用于打开某些位，例如：

```
x = x | SET_ON;
```

用于将x中与SET_ON中为1的位对应的那些位也置为1。

按位异或运算符^用于在其两个运算分量的对应位不相同时置该位为 1，否则，置该位为0。

我们必须将按位运算符 &和 | 同逻辑运算符&&和 || 区分开来，后者用于从左至右求表达式

的真值。例如，如果x的值为1，y的值为2，那么，x & y的结果是0，而x && y的值则为1。

移位运算符<<与>>分别用于将左运算分量左移与右移由右运算分量所指定的位数（右运算分量的值必须是正的）。于是，表达式x << 2用于将x的值左移2位，右边空出的2位用0填空，这个表达式的结果等于左运算分量乘以 4。当右移无符号量时，左边空出的部分用 0填空；当右移有符号的量时，在某些机器上对左边空出的部分用符号位填空（即“算术移位”），而在另一些机器上对左边空出的部分则用0填空（即“逻辑移位”）。

一元~运算符用于求整数的反码，即它分别将运算分量各位上的 1转换成0，0转换成1。例如

```
x = x & ~077
```

用于将x的最后六位置为0。注意，表达式x &~077是独立于字长的，它要比诸如 x & 0177700一类的表达式好，后者假定x是16位的量。这种可移植的形式并没有增加额外开销，因为 ~ 077是常量表达式，可以在编译时求值。

为了对某些按位运算符做进一步说明，考虑函数 getbits(x, p, n)，它用于返回x从p位置开始的（右对齐的） n位的值。假定第 0位是最右边的一位， n与p都是符合情理的正值。例如，getbits(x, 4, 3)返回右对齐的第4、3、2共三位：

```
/* getbits：取从第p位开始的n位 */
unsigned getbits(unsigned x, int p, int n)
{
    return  (x >> (p+1-n) ) & ~(~ 0 << n);
}
```

其中的表达式 x >> (p+1-n) 将所希望的位段移到字的右边。~ 0 将所有位都置为1，~ 0 << n将 ~ 0 左移n位，将最右边的n位用0填空。再对这个表达式求反，将最右边 n位置为1，其余各位置为0。

练习2-6　编写一个函数setbits(x, p, n, y)，返回对x做如下处理得到的值：x从第p位开始的n位被置为y的最右边n位的值，其余各位保持不变。

练习2-7　编写一个函数invert(x, p, n)，返回对x做如下处理得到的值：x从第p位开始的n位被求反（即，1变成0，0变成1），其余各位保持不变。

练习2-8　编写一个函数rightrot (x, n)，返回将x向右循环移动n位所得到的值。

2.10　赋值运算符与赋值表达式

在一个赋值表达式[⊖]中，如果赋值运算符左边的变量在右边紧接着又要重复一次，如：

```
i = i +2
```

那么可以将这种表达式改写成更精简的形式：

```
i += 2
```

⊖ 作者这里所说的赋值运算符实际上专指复合赋值运算符（ +=、-=、*=、/=、%=、>>=、<<=、&=、^=与!=），没有包含简单赋值运算符（ =）。严格来讲，赋值运算符应包含简单赋值运算符与复合赋值运算符两类。——译者注

其中的运算符 += 叫做赋值运算符。

大多数二元运算符（即有左右两个运算分量的运算符）都有一个对应的赋值运算符 *op*=，*op* 是下面这些运算符中的一个：

```
+  -  *  /  %  <<  >>  &  ^  |
```

且表达式

表达式$_1$ *op*= 表达式$_2$

等价于

表达式$_1$ = (表达式$_1$) *op* (表达式$_2$)

区别在于，在前一种形式中表达式$_1$只计算一次。注意，表达式$_1$与表达式$_2$两边的圆括号，它们是不可少的，如：

```
x *= y + 1
```

的意思是：

```
x = x * (y + 1)
```

而不是：

```
x = x * y + 1
```

例如，下面的函数bitcount用于统计其整数变元中值为1的位的个数：

```
/* bitcount：统计x中值为1的位数 */
int bitcount (unsigned x )
{
     int  b;

     for  ( b = 0; x != 0; x >>= 1)
          if  ( x & 01 )
          b++;
   return b;
}
```

将x说明为无符号整数是为了保证：当将 x右移时，不管该函数运行于什么机器上，左边空出的各位能用0（而不是符号位）填满。

除了简明外，这类赋值运算符还有一个其表示方式与人们的思维习惯比较接近的优点。我们通常会说“把2加到i上”或“i加上2”，而不会说“取i，加上2，再把结果放回到i中”，因此，表达式i += 2比i = i + 2好。此外，对于诸如

```
yyval [ yypv [ p3 + p4 ] + yypv [ p1 + p2 ] ] += 2
```

等更复杂的表达式，这种赋值运算符使程序代码更易于理解，读者不需要煞费苦心地去检查两个长表达式是否完全一样，也无需为两者为什么不一样而感到疑惑不解。而且，这种赋值运算符还有助于编译程序产生高效的目标代码。

我们已经看到，赋值语句[⊖]有一个值，而且可以用在表达式中。最常见的例子是：

⊖ ANSI C中没有使用赋值语句这一术语，这里似乎应叫做赋值表达式。而且语句以分号结束，作为语句不能出现在表达式中。 —— 译者注

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

其他赋值运算符（即复合赋值运算符+=、-=等）也可以用在表达式中，尽管这种用法比较少。

在所有这类表达式中，赋值表达式的类型就是左运算分量的类型，值也是在赋值后左运算分量的值。

练习2-9　在求反码时，表达式x &= (x-1)用于把x最右边的值为1的位删除掉。请解释一下这样做的道理。用这一方法重写bitcount函数，使之执行得更快一点。

2.11　条件表达式

语句

```
if  (a > b)
    z = a;
else
    z = b;
```

用于求a与b中的最大值并将之放到z中。作为另一种方法，可以用条件表达式（使用三元运算符?:）来写这段程序及类似的代码段。在表达式

表达式$_1$? 表达式$_2$: 表达式$_3$

中，首先计算表达式$_1$，如果其值不等于0（即为真），则计算表达式$_2$的值，并以该值作为本条件表达式的值；否则计算表达式$_3$的值，并以该值作为本条件表达式的值。在表达式$_2$与表达式$_3$中，只有一个会被计算到。因此，以上语句可以改写成：

```
z = (a > b) ? a : b;       /* z = max(a, b) */
```

应该注意到，条件表达式就是一种表达式，它可以用在其他表达式能用的所有地方。如果表达式$_2$与表达式$_3$具有不同的类型，那么结果的类型由本章前面讨论的转换规则决定。例如，如果f为float类型，n为int类型，那么表达式

```
(n > 0) ? f : n
```

的类型为float，无论n是不是正的。

条件表达式中用于括住第一个表达式的圆括号并不是必需的，这是因为条件运算符 ?: 的优先级非常低，仅高于赋值运算符。但我们还是建议使用圆括号，因为这可以使表达式的条件部分更易于阅读。

条件表达式可用于编写简洁的代码。例如，下面的循环语句用于打印一个数组的 n个元素，每行打印10个元素，每一列之间用一个空格隔开，每行用一个换行符结束（包括最后一行）：

```
for (i = 0; i<n; i++)
    printf("%6d%c", a[i], (i % 10 == 9 || i == n - 1) ? '\n' : ' ');
```

在每10个元素之后以及在第n个元素之后都要打印一个换行符，所有其他元素后都要跟一个空格，这看起来有点麻烦，但要比相应的if-else结构紧凑。下面是另一个使用条件运算符的好例子：

```
printf("you have %d item%s.\n", n, n == 1 ? "" : "s");
```

练习2-10　重写用于将大写字母转换成小写字母的函数lower，用条件表达式替代其中的if-

else结构。

2.12 运算符优先级与表达式求值次序

表2-1总结了所有运算符的优先级与结合律规则，包括尚未讨论的一些规则。同一行的各个运算符具有相同的优先级，纵向看越往下优先级越低。例如，*、/与%三者具有相同的优先级，它们的优先级都比二元+与-运算符高。运算符( )指函数调用。运算符->与.用于访问结构成员，第6章将讨论这两个运算符以及sizeof（对象大小）运算符。第5章将讨论运算符*（用指针间接访问）与&（对象的地址），第3章将讨论逗号（,）运算符。

表2-1 运算符优先级与结合律

运 算 符	结 合 律
( ) [ ] -> .	从左至右
! ~ ++ -- + - * & (类型) sizeof	从右至左
* / %	从左至右
+ -	从左至右
<< >>	从左至右
< <= > >=	从左至右
== !=	从左至右
&	从左至右
^	从左至右
|	从左至右
&&	从左至右
||	从左至右
?:	从右至左
= += -= *= /= %= &= ^= |= <<= >>=	从右至左
,	从左至右

注：一元+、-与*运算符的优先级比相应二元运算符高。

注意，按位运算符 &、^ 与 | 的优先级比等于运算符 == 与 != 低。这意味着，在诸如

```
if  ( (x & MASK) == 0)
   ……
```

中，位测试表达式必须用圆括号括起来，才能得到正确的结果。

如同大多数语言一样，C语言没有指定同一运算符的几个运算分量的计算次序（&&、|、?:与，除外）。例如，在诸如

```
x = f( ) + g( );
```

一类的语句中，f( )可以在g( )之前计算，也可以在g( )之后计算。因此，如果函数f或g中改变了另一个函数所要使用的变量的值，那么x的结果值可能依赖于这两个函数的计算次序。为了保证特定的计算次序，可以把中间结果保存到临时变量中。

同样，在函数调用中各个变元的求值次序也是未指定的。因而，函数调用语句

```
printf("%d %d\n", ++n, power(2,n) );    /* 错 */
```

对不同的编译程序可能会产生不同的结果（视 n加一运算是在power调用之前还是之后而定）。为了解决这一问题，可把该语句改写成：

```
++n;
    printf("%d %d\n", n, power(2,n) );
```

函数调用、嵌套的赋值语句、加一与减一运算符都有可能引起“副作用”——作为表达式求值的副产品，改变了某些变量的值。在涉及到副作用的表达式中，对作为表达式一部分的本来的求值次序存在着微妙的依赖关系。下面的表达式语句是这种使人讨厌的情况的一个典型例子：

```
a[i] = i++;
```

问题是，数组下标的值i是旧值还是新值。编译程序对之可以有不同的解释，并视不同的解释产生不同的结果。C语言标准故意留下了许多诸如此类的问题未作具体规定。何时处理表达式中的副作用（对变量赋值）是各个编译程序的事情，因为最好的求值次序取决于机器结构。（标准明确规定了所有变元的副作用都必须在该函数被调用之前生效，但这对上面对 printf函数的调用没有什么好处。）

从风格角度看，在用任何语言编写程序时，编写依赖于求值次序的代码不是一种好的程序设计习惯。很自然地，我们需要知道哪些事情需要避免，但如果不知道它们在各种机器上是如何执行的，那么不要试图去利用特定的实现。

第3章 控 制 流

一个语言的控制流语句用于指定各个计算执行的次序。在前面的例子中我们已经见到了一些最常用的控制流结构。本章将全面讨论控制流语句，更精确、更全面地对它们进行介绍。

3.1 语句与分程序

在诸如x =0、i++或printf (...)之类的表达式之后加上一个分号（;），就使它们变成了语句⊖：

```
x = 0;
i++;
printf(…);
```

在C语言中，分号是语句终结符，而不是像 Pascal等语言那样把分号用做语句之间的分隔符。

可以用一对花括号{与}把一组说明和语句括在一起构成一个复合语句（也叫分程序），复合语句在语法上等价于单个语句，即可以用在单个语句可以出现的所有地方。一个明显的例子是在函数说明中用花括号括住的语句，其他的例子有在 if、else、while与for之后用花括号括住的多个语句。（在任何分程序中都可以说明变量，第 4章将对此进行讨论。）在用于终止分程序的右花括号之后不能有分号。

3.2 if-else语句

if-else语句用于表示判定。其语法形式如下：

```
if （表达式）
    语句1
else
    语句2
```

其中else部分是任选的。在if语句执行时，首先计算表达式的值，如果其值为真（即，如果表达式的值非0），那么就执行语句$_1$；如果其值为假（即，如果表达式的值为0），并且包含else部分，那么就执行语句$_2$。

由于if语句只是测试表达式的数值，故表达式可以采用比较简捷的形式。最明显

⊖ 在表达式后加上分号构成的语句叫做表达式语句。—— 译者注

的例子是用

```
if  (表达式)
```

代替

```
if  (表达式 != 0 )
```

有时这样既自然又清楚，但有时又显得比较隐秘。

由于if-else语句的else部分是任选的，当在嵌套的 if语句序列中缺省某个 else部分时会引起歧义。这个问题可以通过使每一个 else与最近的还无else匹配的if匹配。例如，在如下语句中：

```
if  ( n > 0 )
    if  ( a > b )
        z = a;
    else
        z = b;
```

else部分与嵌套在里面的 if匹配，正如缩进结构所表示的那样。如果这不是我们所希望的，那么可以用花括号来使该 else部分与所希望的if强制结合：

```
if  ( n > 0 )  {
    if  ( a > b )
        z = a;
}
else
    z = b;
```

歧义性在有些情况下特别有害，例如，在如下程序段中：

```
if  ( n >= 0 )
    for ( i = 0; i < n; i++ )
        if  ( s[i] > 0)  {
            printf ( "…" );
            return i;
        }
else              /*  错  */
    printf ( "error -- n is negative\n" );
```

其中的缩进结构明确地给出了我们所希望的结果，但编译程序无法得到这一信息，它会使 else部分与嵌套在里面的 if匹配。这种错误很难发现，因此我们建议在 if语句嵌套的情况下尽可能使用花括号。

顺便请读者注意，在语句

```
if  ( a > b )
    z = a;
else
    z = b;
```

中，在z = a后有一个分号。这是因为，从语法上讲，跟在 if后面的语句总是以一个分号终结，诸如z = a之类的表达式语句也不例外。

3.3 else-if 语句

在C程序经常使用如下结构：

```
if  ( 表达式 )
    语句
else if  ( 表达式 )
    语句
else if  ( 表达式 )
    语句
else if  ( 表达式 )
    语句
else
    语句
```

由于这种结构经常要用到，值得单独对之进行简要讨论。这种嵌套的 if语句构成的序列是编写多路判定的最一般的方法。各个表达式依次求值，一旦某个表达式为真，那么就执行与之相关的语句，从而终止整个语句序列的执行。每一个语句可以是单个语句，也可以是用花括号括住的一组语句。

最后一个else部分用于处理“上述条件均不成立”的情况或缺省情况，此时，上面的各个条件均不满足。有时对缺省情况不需要采取明显的动作，在这种情况下，可以把该结构末尾的

```
else
    语句
```

省略掉，也可以用它来检查错误，捕获“不可能”的条件。

可以通过一个二分查找函数来说明三路判定的用法。这个函数用于判定在数组 v中是否有某个特定的值x。数组v的元素必须以升序排列。如果在 v中包含x，那么该函数返回 x在v中的位置（介于0~n-1之间的一个整数）；否则，该函数返回- 1。

在二分查找时，首先将输入值 x与数组v的中间元素进行比较。如果 x小于中间元素的值，那么在该数组的前半部查找；否则，在该数组的后半部查找。在这两种情况下，下一步都是将 x与所选一半的中间元素进行比较。这一二分过程一直进行下去，直到找到指定的值，或查找范围为空。

```
/*  binsearch：在v[0]<=v[1]<=v[2]<=……<=v[n-1]中查找x  */
int binsearch ( int x, int v[ ], int n )
{
    int  low, high, mid;

    low = 0;
    high = n - 1;
    while  ( low <= high )  {
        mid = ( low + high ) / 2;
        if  ( x < v[mid] )
            high = mid - 1;
    else if  ( x > v[mid] )
        low = mid + 1;
    else       /*  找到了匹配的值  */
        return  mid;
    }
    return  -1;     /* 没有查到 */
}
```

这个函数的基本判定是，在每一步 x是小于、大于还是等于中间元素 v[mid]，这自然就用到了else-if结构。

练习3-1 在上面有关二分查找的例子中，在 while循环语句内共作了两次测试，其实只要一次就够了（以把更多的测试放在外面为代价）。重写这个函数，使得在循环内部只进行一次测试，并比较两者运行时间的区别。

3.4 switch语句

switch语句是一种多路判定语句，它根据表达式是否与若干常量整数值中的某一个匹配来相应地执行有关的分支动作。

```
switch  ( 表达式 )  {
case  常量表达式:  语句序列
case  常量表达式:  语句序列
default:  语句序列
}
```

每一种情形都由一个或多个整数值常量或常量表达式标记。如果某一种情形与表达式的值匹配，那么就从这个情形开始执行。各个情形中的表达式必须各不相同。如果没有一个情形能满足，那么执行标记为default的情形。 default情形是任选的。如果没有 default情形并且没有一个情形与表达式的值匹配，那么该switch语句不执行任何动作。各个情形及 default情形的出现次序是任意的。

第1章曾用if…else if…else结构编写过一个程序来统计各个数字、空白符及所有其他字符出现的次数。下面是用 switch语句改写的程序：

```
#include  <stdio.h>

main ( )    /* 统计数字、空白及其他字符 */
{
      int  c, i, nwhite, nother, ndigit[10];

      nwhite = nother = 0;
      for  ( i = 0; i < 10, i++ )
          ndigit[i] = 0;
      while  ( ( c = getchar ( ) ) != EOF )  {
          switch  ( c ) {
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
              ndigit[c - '0']++;
              break;
          case ' ':
          case '\n':
          case '\t':
              nwhite++;
              break;
          default:
              nother++;
              break;
          }
```

```
        }
        printf ( "digits = " );
        for  ( i = 0; i < 10, i++ )
            printf ( "  %d", ndigit[i] );
    printf ( ",  white space = %d,  other = %d\n",  nwhite,  nother );
        return  0;
    }
```

break语句用于从switch语句中退出。由于在switch语句中case情形的作用就像标号一样，在某个case情形之后的代码执行完后，就进入下一个 case情形执行，除非显式控制转出。转出switch语句最常用的方法是使用break语句与return语句。break语句还可用于从while、for与do循环语句中立即强制性退出，对于这一点，稍后还将做进一步讨论。

对于依次执行各种情形这种做法毁誉参半，好的一方面，它可以把若干个情形组合在一起完成某个任务，如上例中对数字的处理。但是，为了防止直接进入下一个情形执行，它要求在每一个情形后以一个break语句结尾。从一个情形直接进入下一个情形执行这种做法不是一种健全的做法，在程序修改时很容易出现错误。除了将多个标号用于表示同一计算的情况外，应尽量少从从一个情形直接进入下一个情形执行并在不得不使用时加上适当的注解。

作为一种好的风格，可以在switch语句最后一个情形（即default情形）后加上一个break语句，虽然这样做在逻辑上没有必要，但当以后需要在该 switch语句后再添加一种情形时，这种防范型程序设计会使我们少犯错误。

练习3-2　编写函数escape (s, t)，将字符串t拷贝到字符串s中，并在拷贝过程中将诸如换行符与制表符等等字符转换成诸如 \n与\t等换码序列。使用 switch语句。再编写一个具有相反功能的函数，在拷贝过程中将换码序列转换成实际字符。

3.5　while与for循环语句

我们在前面已经遇到过while与for循环语句。在while循环语句

```
while  ( 表达式 )
    语句
```

执行中，首先求表达式的值。如果其值不等于0，那么执行语句并再次求该表达式的值。这一周期性过程一直进行下去，直到该表达式的值变为假，此时从语句的下一个语句接着执行。

for循环语句

```
for  ( 表达式1; 表达式2; 表达式3 )
    语句
```

等价于

```
表达式1;
while  ( 表达式2 )  {
    语句
    表达式3;
}
```

但包含contine语句时的行为除外，该语句将在3.7节中介绍。

从语法上看，for循环语句的三个组成部分都是表达式。最常见的情况是，表达式$_1$与表达式$_3$是赋值表达式或函数调用，表达式$_2$是关系表达式。这三个表达式中任何一个都可以省略，但分号必须保留。如果表达式$_1$与表达式$_3$被省略了，那么它退化成了while循环语句。如果用于测试的表达式$_2$不存在，那么就认为表达式$_2$的值永远是真的，从而，for循环语句

```
for  ( ; ; )  {
    …
}
```

就是一个“无限”循环语句，这种语句要用其他手段（如break语句或return语句）才能终止执行。

在这两种循环语句中到底选用while语句还是for语句主要取决于程序人员的个人爱好。例如，在如下语句中:

```
while  ( ( c = getchar ( ) ) == ' ' || c == '\n' || c == '\t' )
    ;       /* 跳过空白符 */
```

不包含初始化或重新初始化部分，所以使用while循环语句最为自然。

如果要做简单地初始化与增量处理，那么最好还是使用 for语句，因为它可以使循环控制的语句更密切，而且它把控制循环的信息放在循环语句的顶部，易于程序理解。这在如下语句中表现得更为明显：

```
for  ( i = 0;  i < n;  i++ )
    …
```

这是C语言在处理一个数组的前n个元素时的一种习惯性用法，类似于 FORTRAN语言的DO循环语句与Pascal语言的for循环语句。但是，这种类比不够恰当，因为 C语言循环语句的位标值和终值在循环语句体内可以改变，在循环因某种原因终止时位标变量 i的值仍然保留。由于for语句的各个组成部分可以是任何表达式，故 for语句并不限于以算术值用于循环控制。然而，强制性地把一些无关的计算放到for语句的初始化或增量部分是一种很坏的程序设计风格，它们最好用做循环控制的操作。

作为一个更大的例子，再次考虑用于将字符串转换成对应数值的函数 atoi。下面这个版本要比第2章介绍的那个版本更为通用一些，它可以处理任何前导空白符与加减号。（第4章将介绍另一个类似的函数atof，它用于对浮点数作同样的转换。）

程序的结构反映了输入的形式：

跳过可能有的空白符
取可能有的符号
取整数部分并转换它

每一步都处理其输入，并给下一步留下一个清楚的状态。整个处理过程持续到不是数的一部分的第一个字符为止。

```
#include  <ctype.h>

/* atoi：将s转换成整数；版本2*/
int atoi ( char  s[ ])
{
   int  i, n , sign;
```

```
for ( i = 0;  isspace ( s[i] );  i++ )   /* 跳过空白符 */
       ;
   sign =  ( s[i] == '-' ) ? -1 : 1;
   if  (s[i] == '+' || s[i] == '-' )    /* 跳过符号 */
      i++;
   for  ( n = 0;  isdigit ( s[i] );  i++)
       n = 10 * n + (s[i] - '0' );
   return  sign * n;
}
```

标准库中提供了一个更精巧的函数strtol，它用于把字符串转换成长整数，参见附录B.5节。

当使用嵌套循环语句时，把循环控制集中到一起的优点更为明显。下面的函数用于实现对整数数组进行排序的Shell排序法。这个排序算法是由D. L. Shell于1959年发明的，其基本思想是，先对隔得比较远的元素进行比较，而不是像简单交换排序算法中那样比较相邻的元素。这样可以快速地减少大量的无序情况，以后就可以少做些工作。各个被比较的元素之间的距离在逐步减少，一直减少到1，此时排序变成了相邻元素的互换。

```
/* shellsort：以递增顺序对v[0]、v[1]、……、v[n-1] 进行排序*/
void shellsort ( int v[ ], int n )
{
   int  gap, i, j, temp;

   for  ( gap = n/2;  gap > 0;  gap /= 2 )
        for  ( i = gap;  i < n;  i++ )
               for  ( j = i-gap;  j >= 0 && v[j] > v[j+gap];  j -= gap )  {
                      temp = v[j];
                      v[j] = v[j+gap];
                      v[j+gap] = temp;
        }
}
```

这个函数中包含三个嵌套的for循环语句。最外层的for语句用于控制两个被比较元素之间的距离，从n/2开始对折，一直到0。中间的for语句用于控制每一个元素。最内层的for语句用于比较各对相距gap个位置的元素，并在这两个元素的大小位置颠倒时把它们互换过来。由于gap的值最终要减到1，所有元素最终都会在正确的排序位置上。注意即使在非算术值的情况下，for语句的通用性也使得外层循环能够适应。

C语言还有一个运算符，叫做逗号运算符“ , ”，在for循环语句中经常要使用到它。由逗号分隔的各个表达式从左至右进行求值，结果的类型和值是右运算分量的类型和值。因此，在for循环语句中，可以把多个表达式放在不同的部分，例如，可以同时处理两个位标（控制变量）。这可以通过函数reverse (s)来说明，该函数用于把字符串s中各个字符的位置颠倒一下。

```
#include  <string.h>

/* reverse：颠倒字符串s中各个字符的位置 */
void reverse ( char s[ ] )
{
   int  c, i, j;
```

```
    for  ( i = 0, j =strlen(s) - 1;  i < j;  i++, j-- )  {
         c = s[i];
         s[i] = s[j];
         s[j] = c;
    }
}
```

用于分隔函数变元、说明中的变量等的逗号不是逗号运算符，对逗号运算符也不保证要从左至右求值。

应谨慎使用逗号运算符。逗号运算符最适合用于描述彼此密切相关的构造，如上面 reverse 函数内的for语句中的逗号运算符以及需要在单个表达式中表示多步计算的宏。逗号表达式也适合用在reverse函数的元素交换过程中，该交换过程被当做单步操作。

```
for  ( i = 0, j =strlen(s) - 1;  i < j;  i++, j-- )
c = s[i],  s[i] = s[j],  s[j] = c;
```

练习3-3　编写函数expand(s1, s2)，将字符串s1中诸如a-z一类的速记符号在字符串s2中扩展成等价的完整列表abc……xyz。允许处理大小写字母和数字，并可以处理诸如 a-b-c与a-z0-9与-a-z等情况。正确安排好前导与尾随的 -。

3.6　do-while循环语句

正如第1章所述，while与for这两个循环语句在循环体执行前对终止条件进行测试。与之相对应的，C语言中的第三种循环语句 —— do-while循环语句 —— 则是在循环体执行完后再测试终止条件，循环体至少要执行一次。

do-while循环语句的语法是：

```
do
    语句
while  ( 表达式 )
```

在do-while循环语句执行时，先要执行语句，然后再求表达式的值。如果表达式的值为真，那么就再次执行语句，如此等等。当表达式的值变成假的的时候，就终止循环的执行。除了条件测试的语义外，do-while循环语句与Pascal语言的repeat-until语句等价。

经验表明，使用do-while语句的场合要比使用while语句和for语句的场合少得多。然而，do-while循环语句有时还是很有价值的，如下面的函数 itoa。itoa函数是atoi函数的逆函数，用于把数字转换成字符串。这一工作要比想象的复杂，因为如果以产生数字的方法来产生字符串，所产生的字符串的次序正好颠倒了。故先生成颠倒的字符串，然后再把它颠倒过来。

```
/* itoa：将数字n转换成字符存到s中 */
void itoa ( int n,  char s[ ] );
{
   int  i, sign;

   if  ( ( sign = n ) < 0 )      /* 记录符号 */
       n = -n;                   /* 使n成为正数 */
   i = 0;
```

```
    do  {                           /* 以反序生成数字 */
        s[i++] = n % 10 + '0';      /* 取下一个数字 */
    } while ( (n /= 10) > 0);       /* 删除该数字 */
    if  (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

因为即使n为0也要至少把一个字符放到数组 s中，所以在这里有必要使用 do-while语句，至少使用do-while语句要方便一些。我们也用花括号来括住作为 do-while语句体的单个语句，即使没有必要这样做，但这样可以使那些比较轻率的读者在使用 while语句时少犯些错误。

练习3-4　在数的反码表示中，上述itoa函数不能处理最大的负数，即n为- $(2^{字长-1})$ 时的情况。解释其原因。对该函数进行修改，使之不管在什么机器上运行都能打印出正确的值。

练习3-5　编写函数itob(n, s, b)，用于把整数n转换成以b为基的字符串并存到字符串 c中。特别地，itob(n, s, 16)用于把n格式化成十六进制整数字符串并存在 s中。

练习3-6　修改itoa函数使之改为接收三个变元。第三个变元是最小域宽。为了保证转换得的数（即字符串表示的数）有足够的宽度，在必要时应在数的左边补上一定的空格。

3.7　break语句与continue语句

在循环语句执行过程中，除了通过测试从循环语句的顶部或底部正常退出外，有时从循环中直接退出来要显得更为方便一些。 break语句可用于从for、while与do-while语句中提前退出来，正如它可用于从 switch语句中提前退出来一样。 break语句可以用于立即从最内层的循环语句或switch语句中退出。

下面的函数trim用于删除一个字符串尾部的空格符、制表符与换行符。它用了一个 break语句在找到最右边的非空格符、非制表符、非换行符时从循环中退出。

```
/* trim：删除字符串尾部的空格符、制表符与换行符 */
int trim(char s[]);
{
    int  n;

    for  (n = strlen(s)-1; n >= 0; n--)
         if  (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
             break;
    s[n+1] = '\n';
return  n;
}
```

strlen用于返回字符串的长度。 for循环语句用于从字符串的末尾反过来向前寻找第一个既不是空格符、制表符，也不是换行符的字符。循环在找到这样一个字符时中止执行，或在循环控制变量n变成负数时（即整个字符串都被扫描完时）终止执行。读者可以验证，即使是在字符串

为空或仅包含空白符时，该函数也是正确的。

continue语句与break语句相关，但较少用到。continue语句用于使其所在的for、while或do-while语句开始下一次循环。在while与do-while语句中，continue语句的执行意味着立即执行测试部分；在for循环语句中，continue语句的执行则意味着使控制传递到增量部分。continue语句只能用于循环语句，不能用于switch语句。如果某个continue语句位于switch语句中，而后者又位于循环语句中，那么该continue语句用于控制下一次循环。

例如，下面这个程序段用于处理数组a中的非负元素。如果某个元素的值为负，那么跳过不处理。

```
for  (i =0; i < n; i++)  {
     if  (a[i] < 0)   /* 跳过负元素 */
         continue;
     …      /* 处理正元素 */
}
```

在循环的某些部分比较复杂时常常要使用continue语句。如果不使用continue语句，那么就可能要把测试反过来，或嵌入另一层循环，而这又会使程序的嵌套更深。

3.8 goto语句与标号

C语言提供了可以毫无节制使用的goto语句以及标记goto语句所要转向的位置的标号。从理论上讲，goto语句是没有必要的，实际上，不用它也能很容易地写出代码。本书即未使用goto语句。

然而，在有些情况下使用goto语句可能比较合适。最常见的用法是在某些深度嵌套的结构中放弃处理，例如一次中止两层或多层循环。break语句不能直接用于这一目的，它只能用于从最内层循环退出。下面是使用goto语句的一个例子：

```
for  ( … )
      for  ( … )  {
           …
           if  (disaster)
           goto error;
      }
   …
error:
   清理操作
```

如果错误处理比较重要并且在好几个地方都会出现错误，那么使用这种组织就比较灵活方便。

标号的形式与变量名字相同，其后要跟一个冒号。标号可以用在任何语句的前面，但要与相应的goto语句位于同一函数中。标号的作用域是整个函数。

再看一个例子，考虑判定在两个数组a与b中是否具有相同元素的问题。一种可能的解决方法是：

```
    for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
          if (a[i] == b[j])
```

```
            goto  found;
   /* 没有找到相同元素 */
   …
found:
   /* 取一个满足a[i] ==b[j]的元素 */
   …
```

所有带有goto语句的程序代码都可以改写成不包含goto语句的程序，但这可能需要以增加一些额外的重复测试或变量为代价。例如，可将这个判定数组元素是否相同的程序段改写成如下形式：

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
           found = 1;
if (found)
   /* 取一个满足a[i-1] ==b[j-1]的元素 */
 …
else
   /* 没有找到相同元素 */
 …
```

除了以上介绍的几个程序段外，依赖于goto语句的程序段一般都比不使用goto语句的程序段难以理解与维护。虽然不特别强调这一点，但我们还是建议尽可能减少goto语句的使用。

第4章 函数与程序结构

函数用于把较大的计算任务分解成若干个较小的任务，使程序人员可以在其他函数的基础上构造程序，而不需要从头做起。一个设计得当的函数可以把具体操作细节对程序中不需要知道它们的那些部分隐藏掉，从而使整个程序结构清楚，减轻了因修改程序所带来的麻烦。

C语言在设计函数时考虑了效率与易于使用这两个方面。一个 C程序一般都由许多较小的函数组成，而不是只由几个比较大的函数组成。一个程序可以驻留在一个文件中，也可以存放在多个文件中。各个文件可以单独编译并与库中已经编译过的函数装配在一起。但我们不打算详细讨论这一编译装配过程，因为具体编译与装配细节在各个编译系统中各不相同。

ANSI C标准对C语言所做的最显著的修改是在函数说明与定义这两个方面。正如第1章所述，C语言现在已经允许在说明函数时说明变元的类型。为了使函数说明与定义匹配，ANSI C标准对函数定义的语法也做了修改。故编译程序可以查出比以前更多的错误。而且，如果变元说明得当，那么程序可以自动地进行适当的类型强制转换。

ANSI C标准进一步明确了名字的作用域规则，尤其是它要求每一个外部变量只能有一个定义。初始化做得更一般化了：现在自动数组与结构都可以初始化。

C的预处理程序的功能也得到了增强。新的预处理程序所包含的条件编译指令（一种用于从宏变元建立带引号字符串的方法）更为完整，对宏扩展过程的控制更严格。

4.1 函数的基本知识

下面首先设计并编写一个程序，用于把输入中包含特定的“模式”或字符串的各行打印出来（这是UNIX程序grep的特殊情况）。例如，对如下一组文本行查找包含字母字符串“ould”的行：

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

可以产生如下输出：

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

这个程序段可以清楚地分成三部分：

```
while  ( 还有未处理的行 )
    if  ( 该行包含指定的模式 )
        打印该行
```

虽然可以把所有这些代码都放在主程序main中，但一个更好的方法是把每一部分设计成一个独立的函数。分别处理三个较小的部分要比处理一个大的整体容易，因为这样可以把不相关的细节隐藏在函数中，从而减少了不必要的相互影响的机会。而且这些函数也可以在其他程序中使用。

我们用函数getline来实现“还有未处理的行”，这个函数已在第1章介绍过；用printf函数来实现“打印该行”，这是一个别人早就为我们提供的函数，这意味着我们只需编写一个判定“该行包含指定的模式”的函数。

我们可以通过编写一个函数strindex(s, t)来解决这个问题，该函数返回字符串t在字符串s中出现的开始位置或位标，但当s中不包含t时，返回值为-1。由于C语言数组的下标从0开始，下标的值为0或正数，故用-1之类的负数作为失败信号是比较方便的。若以后需要更复杂的模式匹配，只需替换掉strindex函数即可，程序的其余部分可保持不动。（标准库中提供的库函数strstr的功能类似于strindex函数，只是该库函数返回的是指针而不是下标值。）

在做了这样的设计后，填写程序的细节就比较简单了。下面即整个程序，读者可以看看各个部分是怎样组合在一起的。我们现在所要查找的模式是字面值字符串，它不是一种最通用的机制。我们将简单讨论一下字符数组的初始化方法，第5章将介绍如何在模式中加入可在程序运行时设置的参数。getline函数的版本也稍有不同，读者可将其与第1章所介绍的版本进行比较。

```
#include  <stdio.h>
#define  MAXLINE  100   /*最大输入行长度 */

int getline (char line[ ], int max);
int strindex(char source[ ], char searchfor[ ]);

char pattern[] = "ould";     /*要查找的模式 */

/* 找出所有与模式匹配的行 */
main ( )
{
   char  line[MAXLINE];
   int  found = 0;

   while  ( getline(line, MAXLINE) > 0 )
      if  ( strindex(line, pattern) >= 0 )  {
            printf( "%s", line);
            found++;
      }
   return  found;
}
/* getline：取一行放到s中，并返回该行的长度 */
int getline(char s[ ], int lim)
```

```
{
   int  c, i;

   i = 0;
   while  ( -- lim > 0&& ( c = getchar() ) != EOF && c != '\n' )
       s[i++] = c;
   if  (c == '\n' )
       s[i++] = c;
   s[i] = '\0';
   return  i;
}

/* strindex：返回t在s中的位置，若未找到则返回-1 */
int strindex(char s[], char t[] )
{
   int  i, j, k;
   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' )
         return  i;
   }
   return  -1;
}
```

每一个函数定义均具有如下形式：

```
返回类型 函数名（变元说明表）
{
    说明序列与语句序列
}
```

函数定义的各个部分都可以缺省。最简单的函数结构如下：

```
dummy( ) { }
```

这个函数什么也不做、什么也不返回。像这种什么也不做的函数有时很有用，它可以在程序开发期间用做占位符。如果在函数定义中省略了返回类型，则缺省为 int。

程序是变量定义和函数定义的集合。函数之间的通信可以通过变元、函数返回值以及外部变量进行。函数可以以任意次序出现在源文件中。源程序可以分成多个文件，只要不把一个函数分在几个文件中就行。

return**语句用于从被调用函数向调用者返回值，** return**之后可以跟任何表达式：**

```
return 表达式；
```

在必要时要把表达式**转换成函数的返回类型（结果类型）。**表达式**两边往往要加一对圆括号，但不是必需的，而是可选的。**

调用函数可以随意忽略掉返回值。而且， return**之后也不一定要跟一个表达式。在** return**之后没有表达式的情况下，不向调用者返回值。当被调用函数因执行到最后的右花括号而完成执行时，控制同样返回调用者（不返回值）。如果一个函数在从一个地方返回时有返回值而从另一**

个地方返回时没有返回值，那么这个函数不一定非法，但可能存在问题。在任何情况下，如果一个函数不能返回值，那么它的“值”肯定是没有用的。

上面的模式查找程序从主程序 main中返回一个状态，即所匹配的字符串的数目。这个值可以在调用该程序的环境中使用。

在不同系统上对驻留在多个源文件中的 C程序的编译与载入机制有很大的区别。例如，在UNIX系统上是用在第 1章中已提到过的 cc命令来完成这一任务的。假定有三个函数分别存放在名为main.c、getline.c与strindex.c的三个文件中，那么命令

```
cc  main.c getline.c strindex.c
```

用于编译这三个文件，并把目标代码分别存放在文件 main.o、getline.o与strindex.o中，然后再把这三个文件一起载入到可执行文件 a.out中。如果源程序中出现了错误（比如文件 main.c中出现了错误），那么可以用命令

```
cc  main.c getline.o strindex.o
```

对main.c文件重新编译，并将编译的结果与以前已编译过的目标文件 getline.o和strindex.o一起载入。cc命令用 .c 与 .o 这两种扩展名来区分源文件与目标文件。

练习4-1　编写一个函数strrindex(s, t)，用于返回字符串t在s中最右出现的位置，如果 s中不包含t，那么返回-1。

4.2　返回非整数值的函数

到目前为止，我们所讨论的函数均是不返回任何值（ void）或只返回int类型的值。假如一个函数必须返回其他类型的值，那么该怎么办呢？许多数值函数（如 sqrt、sin与cos等函数）返回的是double类型的值，另一些专用函数则返回其他类型的值。

为了说明让函数返回非整数值的方法，编写并使用函数 atof(s)，它用于把字符串 s转换成相应的双精度浮点数。 atof函数是atoi函数的扩充，第 2章与第3章已讨论了 atoi函数的几个版本。atof函数要处理可选的符号与小数点以及整数部分与小数部分。我们这个版本并不是一个高质量的输入转换函数，它所占用的空间比我们可以使用的要多。标准库中包含了具有类似功能的 atof函数，它在头文件<stdlib.h>中说明。

首先，由于atof函数返回值的类型不是 int，因此在该函数中必须说明它所返回值的类型。返回值类型的名字要放在函数名字之前：

```
#include <ctype.h>

/* 把字符串s转换成相应的双精度浮点数 */
double atof( char s[ ])
{
    double  val, power;
    int  i, sign;

    for  ( i = 0; isspace(s[i]); i++ )     /* 跳过空白 */
        ;
    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;
     }
     return  sign * val / power;
}
```

其次，也是比较重要的，调用函数必须知道 atof**函数返回的是非整数值。为了保证这一点，一种方法是在调用函数中显式说明** atof**函数。下面所示的基本计算器程序（仅适用于支票簿计算）中给出了这个说明，程序一次读入一行数（一行只放一个数，数的前面可能有一个正负号），并把它们加在一起，在每一次输入后把这些数的连续和打印出来：**

```
#include  <stdio.h>

#define  MAXLINE  100

/* 基本计算器程序 */
main ( )
{
    double  sum, atof ( char [ ] );
    char  line[MAXLINE];
    int  getline(char line[], int max);

    sum =0;
    while  ( getline(line, MAXLINE) > 0 )
        printf( "\t%g\n", sum += atof(line) );
    return  0;
}
```

其中，说明语句

```
double  sum, atof ( char [ ] );
```

表明sum**是一个**double**类型的变量，**atof**是一个具有**char[ ]**类型的变元且返回值类型为**double**的函数。**

函数atof**的说明与定义必须一致。如果** atof**函数与调用它的主函数** main**放在同一源文件中，并且具有不一致的类型，那么编译程序将会检测出这个错误。但是，如果** atof**函数是独立编译的（这是一种更可能的情况），那么这种不匹配的错误就不会被检测出来，** atof**函数将返回**double**类型的值，而**main**函数则将之处理为**int**类型，从而这样所求得的结果毫无意义。**

按照上述说明与定义匹配的讨论，这似乎很令人吃惊。发生不匹配现象的一个原因是，如果没有函数原型，则该函数在第一次出现的表达式中隐式说明，例如下面的表达式：

```
sum += atof(line)
```

如果在前面已经说明过的某个名字出现在某个表达式中并且左边跟一个左圆括号，那么就根据上下文认为该名字是函数名字，该函数的返回值类型为 int**，但对变元没有给出上面信息。而且，**

如果一个函数说明中不包含变元，比如：

```
double atof ( );
```

那么也认为没有给出atof函数的变元信息，所有参数检查都被关闭。对空变元表做这种特殊的解释是为了使新的编译程序能编译比较老的C程序。但是，在新程序中也如此做是不明智的。如果一个函数有变元，那么说明它们；如果没有变元，那么使用 void。

借助恰当说明的atof函数，可以编写出函数atoi（将字符串转换成整数）：

```
/* atoi：利用atof函数把字符串s转换成整数 */
int atoi( char s[ ] )
{
   double  atof(char s[ ]);

   return  (int) atof ( s );
}
```

请注意其中说明和return语句的结构。在return语句：

```
return  表达式；
```

中的表达式的值在返回之前被转换成所在函数的类型。因此，如果对 atof函数的调用直接出现在atoi函数中的return语句中，如

```
return  atof ( s );
```

那么，由于函数atoi的返回值类型为int，系统要把atof函数的double类型的结果返回值自动转换成int类型。然而，这种操作可能会丢失信息，有些编译程序可能会为此给出警告信息。在此函数中由于采用了强制转换的方法显式地表明了所要做的转换操作，可以屏蔽有关警告信息。

练习4-2　对atof函数进行扩充，使之可以处理形如

```
123.45e-6
```

一类的科学表示法，即在浮点数后跟e或E与一个（可能有正负号的）指数。

4.3　外部变量

C程序由一组外部对象（外部变量或函数）组成。形容词 external与internal相对，internal用于描述定义在函数内部的函数变元以及变量。外部变量在函数外面定义，故可以在许多函数中使用。由于C语言不允许在一个函数中定义其他函数，因此函数本身是外部的。在缺省情况下，外部变量与函数具有如下性质：所有通过名字对外部变量与函数的引用（即使这种引用来自独立编译的函数）都是引用的同一对象（标准中把这一性质叫做外部连接）。在这个意义上，外部变量类似于FORTRAN语言的COMMON块或Pascal语言中在最外层分程序中说明的变量。后面将介绍如何定义只能在某个源文件使用的外部变量与函数。

由于外部变量是可以全局访问的，这就为在函数之间交换数据提供了一种可以代替函数变元与返回值的方法。任何函数都可以用名字来访问外部变量，只要这个名字已在某个地方做了说明。

如果要在函数之间共享大量的变量，那么使用外部变量要比使用一个长长的变元表更方便、

有效。然而，正如在第1章所指出的，这样使用必须充分小心，因为这样可能对程序结构产生不好的影响，而且可能会使程序在各个函数之间产生太多的数据联系。

外部变量的用途还表现在它们比内部变量有更大的作用域和更长的生存期。自动变量只能在函数内部使用，当其所在函数被调用时开始存在，当函数退出时消失。而外部变量是永久存在的，它们的值在从一次函数调用到下一次函数调用之间保持不变。因此，如果两个函数必须共享某些数据，而这两个函数都互不调用对方，那么最为方便的是，把这些共享数据作成外部变量，而不是作为变元来传递。

下面通过一个更大的例子来说明这个问题。问题是要编写一个具有加（+）、减（-）、乘（*）、除（/）四则运算功能的计算器程序。为了更易于实现，在计算器中使用逆波兰表示法来代替普通的中辍表示法（逆波兰表示法用在某些袖珍计算器中，诸如Forth与Postscript等语言也使用了逆波兰表示法）。

在使用逆波兰表示法时，所有运算符都跟在其运算分量的后面。诸如

```
(1 - 2) * (4 + 5)
```

一类的中辍可用逆波兰表示法表示成：

```
1 2 - 4 5 + *
```

在使用逆波兰表示法时不再需要圆括号，只需知道每一个运算符需要几个运算分量。

计算器程序的实现很简单。每一个运算分量都被依次下推到栈中；当一个运算符到达时，从栈中弹出相应数目的运算分量（对二元运算符是两个运算分量），把该运算符作用于所弹出的运算分量，并把运算结果再下推回栈中。例如，对上面所述逆波兰表达式，首先把1与2下推到栈中，再用两者之差-1来取代它们；然后，把4与5下推到栈中，再用两者之和9来取代它们。最后，从栈中取出栈顶的-1与9，把它们的积-9下推到栈顶。当到达输入行的末尾时，把栈顶的值弹出并打印出来。

这样，该程序的结构是一个循环，每一次循环对一个运算符及相应的运算分量执行一次操作：

```
while  ( 下一个运算符或运算分量不是文件结束指示符 )
    if  ( 数 )
        将该数下推到栈中
    else if  ( 运算符 )
        弹出所需数目的运算分量
        执行运算
        将结果下推到栈中
    else if  ( 换行符 )
        弹出并打印栈顶的值
    else
        错误
```

栈的下推与弹出操作比较简单，但是，如果把错误检测与恢复操作都加进去，那么它们就会显得很长，最好把它们设计成独立的函数，而不要把它们作为在这个程序中重复的代码段。另外还需要一个单独的函数来取下一个输入运算符或运算分量。

到目前为此还没有讨论的主要设计决策是，把栈放在哪里？即哪些函数可以直接访问它？

一种可能是把它放在主函数 main中，把栈及其当前位置作为传递给要对它进行下推或弹出操作的函数。但是， main函数不需要知道控制该栈的变量信息，它只进行下推与弹出操作。因此，可以把栈及其相关信息放在外部变量中，并只供 push与pop函数访问，而不能为 main函数则所访问。

把上面这段话翻译成代码很容易。如果把这个程序放在一个源文件中，那么它为如下形式：

```
#include
#define

用于main的函数说明

main() { … }

用于push与pop的外部变量

void push ( double f ) { … }
double pop(void) { … }

int getop(char s[ ]) { … }

被getop调用的函数
```

我们在以后将讨论怎样把这个程序分割成两个或多个源文件。

main函数主要由一个循环组成，该循环中包含了一个对运算符与运算分量进行分情形操作的switch语句，这里对switch语句的使用要比3.4节所示的例子更为典型。

```
#include  <stdio.h>
#include  <stdlib.h>       /* 供atof()函数使用 */

#define  MAXOP   100       /* 运算分量或运算符的最大大小 */
#define  NUMBER  '0'       /* 表示找到数的信号 */

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

/* 逆波兰计算器 */
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 )
               push ( 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;
}
```

由于 + 与 * 是两个满足交换律的运算符，因此弹出的两个运算分量的次序无关紧要，但是，- 与 / 的左右运算分量的次序则是必需的。在如下所示的函数调用中：

```
push ( pop() - pop() );     /* 错 */
```

对pop函数的两次调用的次序没有定义。为了保证正确的次序，必须像在 main函数中一样把其第一个值弹出到一个临时变量中。

```
#define  MAXVAL  100      /* 栈val的最大深度  */

int  sp = 0;              /* 下一个自由栈元素位置 */
double  val[MAXVAL];      /* 值栈 */

/* push：把f下推到值栈中 */
void push ( double f )
{
   if  ( sp < MAXVAL )
       val[sp++] = f;
   else
       printf ( "error: stack full, can't push %g\n", f);
}
```

```
/* pop：弹出并返回栈顶的值 */
double pop(void);
{
   if ( sp > 0 )
       return val[--sp];
   else  {
       printf ( "error: stack empty\n" );
       return  0.0;
   }
}
```

一个变量如果在函数的外面定义，那么它就是外部变量。因此，我们把必须为 push和pop函数共享的栈和栈顶指针定义在这两个函数的外面。但 main函数本身并没有引用该栈或栈顶指针，因此将它们对它隐藏。

下面讨论getop函数的实现，它用于取下一个运算符或运算分量。这一任务比较容易。跳过空格与制表符。如果下一个字符不是数字或小数点，那么返回；否则，把这些数字字符串收集起来（其中可能包含小数点），并返回NUMBER，用这个信号表示数已经收集起来了。

```
#include  <ctype.h>

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

/* getop：取下一个运算符或数值运算分量 */
int getop(char s[ ] )
{
   int  i, c;

   while  ( (s[0] = c = getch() ) == ' ' || c == '\t' )
    ;
   s[1] = '\0';
   if  ( !isdigit( c ) && c != '.' )
       return c;     /* 不是数 */
       i = 0;
   if  ( isdigit( c ) )  /* 收集整数部分*/
       while  ( isdigit(s[++i] = c = getch( ) ) )
           ;
   if  (c == '.' )     /* 收集小数部分 */
       while  ( isdigit(s[++i] = c = getch( ) ) )
           ;
   s[i] = '\0';
   if  (c != EOF )
       ungetch( c );
   return  NUMBER;
}
```

这段程序中的getch与ungetch是两个什么样的函数呢？在程序中经常会出现这样的情况，一个程序在读进过多的输入之前不能确定它已经读入的输入是否足够。一个例子是在读进用于组

成数的字符的时候：在看到第一个非数字字符之前，所读入数的完整性是不能确定的。由于程序要超前读入一个字符，最后有一个字符不属于当前所要读入的数。

如果能“反读”不需要的字符，那么这个问题就能得到解决。每当程序多读进一个字符时，就可以把它推回到输入中，对代码其余部分而言就像这个字符并没有读过一样。我们可以通过编写一对相互配合的函数来比较方便地模拟反取字符操作。 getch函数用于读入下一个待处理的字符，而ungetch函数则用于把字符放回到输入中，使得此后对 getch函数的调用将在读新的输入之前先返回经ungetch函数放回的那些字符。

把这两个函数放在一起配合使用很简单。 ungetch函数把要推回的字符放到一个共享缓冲区（字符数组）中，而getch函数在该缓冲区不空时就从中读取字符，在缓冲区为空时调用 getchar函数直接从输入中读字符。为了记住缓冲区中当前字符的位置，还需要一个下标变量。

由于缓冲区与下标变量是供 getch与ungetch函数共享的，在两次调用之间必须保持值不变，它们必须为这两个函数的外部变量。这样，可以如下编写 getch与ungetch函数及其共享变量：

```
#define  BUFSIZE  100

char  buf[BUFSIZE];      /* 用于unget函数的缓冲区 */
int  bufp = 0;           /* buf中下一个自由位置 */

int getch(void)          /* 取一个字符（可能是推回的字符） */
{
return  ( bufp > 0 ) ? buf[--bufp] : getchar( );
}

void ungetch(int c)       /* 把字符推回到输入中 */
{
   if  ( bufp >= BUFSIZE )
       printf ( "ungetch: too many characters\n" );
   else
       buf[bufp++] = c;
}
```

标准库中提供了函数 ungetc，用于推回一个字符，第 7章将对它进行讨论。为了说明更一般的方法，我们这里使用了一个数组而不是一个字符用于推回字符。

练习4-3　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（%）运算符并注意负数的情况。

练习4-4　在栈操作中添加几个命令分别用于在不弹出时打印栈顶元素、复制栈顶元素以及交换栈顶两个元素的值。再增加一个命令用于清空栈。

练习4-5　增加对诸如sin、exp与pow等库函数的访问操作。有关这些库函数参见附录 B.4节中的头文件<math.h>。

练习4-6　增加处理变量的命令（提供26个由单字母变量很容易）。增加一个变量存放用于最近打印的值。

练习4-7　编写一个函数ungets(s)，用于把整个字符串推回到输入中。ungets函数要使用buf与bufp吗？它可否仅使用ungetch函数？

练习4-8　假定最多只要推回一个字符。请相应地修改getch与ungetch这两个函数。

练习4-9　上面所介绍的getch与ungetch函数不能正确地处理推回的EOF。决定当推回EOF时应具有什么性质，然后再设计实现。

练习4-10　另一种组织方法是用getline函数读入整个输入行，这样便无需使用getch与ungets函数。运用这一方法修改计算器程序。

4.4　作用域规则

用以构成C程序的函数与外部变量完全没有必要同时编译，一个程序可以放在几个文件中，可以从库中调入已编译过的函数。我们比较感兴趣的问题主要有：

- **怎样编写说明才能使所说明的变量在编译时被认为是正确的？**
- **怎样安排说明才能保证在程序载入时各部分能正确相连？**
- **怎样组织说明才能使得只需一份拷贝？**
- **怎样初始化外部变量？**

为了便于讨论这些问题，我们把计算器程序组织在若干个文件中。从实用角度看，计算器程序比较小，不值得分几个文件存放，但通过它可以很好地说明在较大的程序中所遇到的有关问题。

一个名字的作用域指程序中可以使用该名字的部分。对于在函数开头说明的自动变量，其作用域是说明该变量名字的函数。在不同函数中说明的具有相同名字的各个局部变量毫不相关。对于函数的参数也如此，函数参数实际上可以看作是局部变量。

外部变量或函数的作用域从其说明处开始一直到其所在的被编译的文件的末尾。例如，如果main、sp、val、push与pop是五个依次定义在某个文件中的函数与外部变量，即：

```
main( )  { … }

int  sp = 0;
double  val[MAXVAL];

void push( double f )  { … }

double pop( void )  { … }
```

那么，在push与pop这两个函数中不需做任何说明就可以通过名字来访问变量sp与val，但是，这两个变量名字不能用在main函数中，push与pop函数也不能用在main函数中。

另一方面，如果一个外部变量在定义之前就要使用到，或者这个外部变量定义在与所要使用它的源文件不相同的源文件中，那么要在相应的变量说明中强制性地使用关键词extern。

将对外部变量的说明与定义严格区分开来很重要。变量说明用于通报变量的性质（主要是变量的类型），而变量定义则除此以外还引起存储分配。如果在函数的外部包含如下说明：

```
int  sp;
double  val[MAXVAL];
```

那么这两个说明定义了外部变量 sp与val，并为之分配存储单元，同时也用作供源文件其余部分使用的说明。另一方面，如下两行:

```
extern int  sp;
extern double  val[MAXVAL];
```

为源文件剩余部分说明了 sp是一个int类型的外部变量， val是一个double数组类型的外部变量(该数组的大小在其他地方确定)，但这两个说明并没有建立变量或为它们分配存储单元。

在一个源程序的所有源文件中对一个外部变量只能在某个文件中定义一次，而其他文件可以通过extern说明来访问它（在定义外部变量的源文件中也可以包含对该外部变量的 extern说明)。在外部变量的定义中必须指定数组的大小，但在 extern说明中则不一定要指定数组的大小。

外部变量的初始化只能出现在其定义中。

假定函数push与pop在一个文件中定义，变量val与sp在另一个文件中定义并初始化（虽然一般不可能这样组织程序)。这些定义与说明必须把这些函数和变量捆在一起：

在文件file1中：

```
   extern int  sp;
   extern double  val [ ];

   void push( double f )  { … }

   double pop( void )  { … }
```

在文件file2中：

```
   int  sp = 0;
   double  val[MAXVAL];
```

由于文件 file1中的extern说明不仅放在函数定义的外面而且还放在它们前面，故它们适用于所有函数，这一组说明对文件 file1已足够了。如果 sp与val的定义跟在对它们的使用之后，那么也要这样来组织文件。

4.5 头文件

下面考虑把计算器程序分成若干个源文件。主函数 main单独放在文件main.c中，push与pop函数及它们所使用的外部变量放在第二个文件 stack.c中，getop函数放在第三个文件 getop.c中，getch与ungetch函数放在第四个文件 getch.c中。之所以把它们分开，是因为在实际程序中它们来自于一个独立编译的库。

还有一个问题需要考虑，即这些文件之间的定义与说明的共享问题。我们将尽可能使所要共享的部分集中在一起，以使得只需一个拷贝，当要对程序进行改进时也能保证程序的正确性。我们将把这些公共部分放在头文件 calc.h中，在需要使用该头文件时可以用 #include指令引入(#include指令将在4.11节介绍)。如此得到的程序形式如下所示：

头文件calc.h：

```
#define  NUMBER  '0'

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

文件main.c：

```
#include  <stdio.h>
#include  <stdlib.h>
#include  "calc.h"
#define  MAXOP  100

main( )
{
    …
}
```

文件getop.c：

```
#include  <stdio.h>
#include  <ctype.h>
#include  "calc.h"
getop( )
{
    …
}
```

文件getch.c：

```
#include  <stdio.h>
#define  BUFSIZE  100
char  buf[BUFSIZE];
int  bufp = 0;
int getch( void )
{
    …
}
void ungetch( int)
{
    …
}
```

文件stack.c：

```
#include  <stdio.h>
#include  "calc.h"
#define  MAXVAL  100
int  sp = 0;
```

```
double  val[MAXVAL];

void push( double )
{
    …
}

double pop( void)
{
    …
}
```

我们对如下两个方面做了折衷：一方面是对每一个文件只能访问它完成任务所需要的信息的要求，另一方面是维护较多的头文件比较困难的现实。对于某些中等规模的程序，最好是只使用一个头文件来存放程序中各个部分需要共享的实体，这是我们在这里所做的结论。对于比较大的程序，需要做更精心的组织，使用更多的头文件。

4.6 静态变量

stack.c文件中定义的变量sp与val以及getch.c文件中定义的变量buf 与bufp仅供它们各自所在的源文件中的函数使用，不能被其他函数访问。 static说明适用于外部变量与函数，用于把这些对象的作用域限定为被编译源文件的剩余部分。通过外部 static对象，可以把诸如buf与bufp一类的名字隐藏在getch-ungetch组合中，使得这两个外部变量可以被 getch与ungetch函数共享，但不能被getch与ungetch函数的调用者访问。

可以在通常的说明之前前缀以关键词 static来指定静态存储。如果把上述两个函数与两个变量放在一个文件中编译，如下：

```
static char  buf[BUFSIZE];     /* 供ungetch函数使用的缓冲区 */
static int  bufp = 0;          /* 缓冲区buf的下一个自由位置 */

int getch( void )  { …}

void ungetch( int c)  { … }
```

那么其他函数不能访问变量 buf与bufp，故这两个名字不会和同一程序中其他文件中的同名名字相冲突。基于同样的理由，可以通过把变量 sp与val说明为静态的，使这两个变量只能供进行栈操作的push与pop函数使用，而对其他文件隐藏。

外部static说明最常用于说明变量，当然它也可用于说明函数。通常情况下，函数名字是全局的，在整个程序的各个部分都可见。然而，如果把一个函数说明成静态的，那么该函数名字就不能用在除该函数说明所在的文件之外的其他文件中。

static说明也可用于说明内部变量。内部静态变量就像自动变量一样局部于某一特定函数，只能在该函数中使用，但与自动变量不同的是，不管其所在函数是否被调用，它都是一直存在的，而不像自动变量那样，随着所在函数的调用与退出而存在与消失。换而言之，内部静态变量是一种只能在某一特定函数中使用的但一直占据存储空间的变量。

练习4-11　修改getop函数，使之不再需要使用 ungetch函数。提示：使用一个内部静态变量。

4.7　寄存器变量

register**说明用于提醒编译程序所说明的变量在程序中使用频率较高。其思想是，将寄存器变量放在机器的寄存器中，这样可以使程序更小、执行速度更快。但编译程序可以忽略此选项。**

register**说明如下所示：**

```
register int  x;
register char  c;
```

寄存器说明只适用于自动变量以及函数的形式参数。对于后一种情况，例子如下：

```
f( register unsigned m, register long n )
{
    register int  i;
    …
}
```

在实际使用时，由于硬件环境的实际情况，对寄存器变量会有一些限制。在每一个函数中只有很少的变量可以放在寄存器中，也只有某些类型的变量可以放在寄存器中。然而，过量的寄存器说明并没有什么害处，因为对于过量的或不允许的寄存器变量说明，编译程序可以将之忽略掉。另外，不论一个寄存器变量实际上是不是存放在寄存器中，它的地址都是不能访问的（关于这一问题将在第5章讨论）。对寄存器变量的数目与类型的具体限制视不同的机器而有所不同。

4.8　分程序结构

C**语言不是**Pascal**等语言意义上的分程序结构的语言，因为它不允许在函数中定义函数。但另一方面，变量可以以分程序结构的形式在函数中定义。变量的说明（包括初始化）可以跟在用于引入复合语句的左花括号的后面，而不是只能出现在函数的开始部分。以这种方式说明的变量可以隐藏在该分程序外面说明的同名变量，并在与该左花括号匹配的右花括号出现之前一直存在。例如，在如下程序段中：**

```
if  ( n > 0 )  {
   int  i;     /* 说明一个新的i */

   for  ( i = 0; i < n; i++ )
    …
}
```

变量i**的作用域是**if**语句的“真”分支，这个**i**与在该分程序之外说明的**i**无关。在分程序中说明与初始化的自动变量每当进入这个分程序时就被初始化。静态变量只在第一次进入分程序时初始化一次。**

自动变量（包括形式参数）也隐藏同名的外部变量与函数。对于如下说明：

```
int  x;
```



```
int  y;

f ( double x )
{
    double  y;
    …
}
```

在函数f内，所出现的x引用的是参数，其类型为double，而在函数f之外，引用的是类型为int的外部变量。对变量y也如此。

就风格而言，最好避免出现变量名字隐藏外部作用域中同名名字的情况，否则可能会出现大量混乱与错误。

4.9 初始化

前面已多次提到初始化的概念，但一直没有认真讨论它。这一节在前面讨论了各种存储类的基础上总结一些初始化规则。

在没有显式初始化的情况下，外部变量与静态变量都被初始化为 0，而自动变量与寄存器变量的初值则没有定义（即，其初值是“垃圾”）。

在定义纯量变量时，可以通过在所定义的变量名字后加一个等号与一个表达式来进行初始化：

```
int   x = 1;
char  squote = '\'';
long  day = 1000L * 60L *60L * 24L;   /* 每天的毫秒数 */
```

对于外部变量与静态变量，初始化符必须是常量表达式，初始化只做一次（从概念上讲是在程序开始执行前进行初始化）。对于自动变量与寄存器变量，则在每当进入函数或分程序时进行初始化。

对于自动变量与寄存器变量，初始化符不一定限定为常量：它可以是任何表达式，其中可以包含前面已定义过的值甚至可以包含函数调用。对 3.3节介绍的二分查找程序的初始化可以用如下形式：

```
int binsearch( int x, int v[ ], int n )
{
   int  low = 0;
   int  high = n - 1;
   int  mid;
   …
}
```

来代替原来的形式：

```
int low, high, mid;

low = 0;
high = n - 1;
```

实际上，自动变量的初始化部分就是赋值语句的缩写。到底使用哪一种形式还是一个尚待

尝试的问题，我们一般使用显式的赋值语句，因为说明中的初始化符比较难以为人们发现，并且距使用点比较远。

数组的初始化也是通过说明中的初始化符完成的。数组初始化符是用花括号括住并用逗号分隔的初始化符序列。例如，当要用每一个月的天数来初始化数组 days时，可用如下变量定义：

```
int days[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

当数组的大小缺省时，编译程序就通过统计花括号中初始化符的个数作为数组的长度，本例中数组的大小为12个元素。

如果初始化符序列中初始化符的个数比数组元素数少，那么对于没有得到初始化的数组元素在该数组为外部变量、静态变量与自动变量时被初始化为 0。如果初始化符序列中初始化符的个数比数组元素数多，那么就是错误的。我们既无法一次性地为多个数组元素指定一个初始化符，也不能在没有指定前面数组元素值的情况下初始化后面的数组元素。

字符数组的初始化比较特殊，可以用一个字符串来代替用花括号括住并用逗号分隔的初始化符序列：

```
char  pattern [] = "ould";
```

它是如下虽然长些但却与之等价的定义的缩写：

```
char  pattern [] = { 'o', 'u', 'l', 'd' , '\0' };
```

在此情况下，数组的大小是5（4个字符外加一个字符串结束符 '\0'）。

4.10 递归

C函数可以递归调用，即一个函数可以直接或间接调用自己。考虑把一个数作为字符串打印的情况。如前所述，数字是以相反的次序生成的：低位数字先于高位数字生成，但它们必须以相反的次序打印出来。

对这一问题有两种解决方法。一种方法是将生成的各个数依次存储到一数组中，然后再以相反的次序把它们打印出来，正如3.6节对itoa函数所做的那样。另一种方法是使用递归解法，用于完成这一任务的函数 printd首先调用自身处理前面的（高位）数字，然后再把后面的数字打印出来。这个版本不能处理最大的负数。

```
#include  <stdio.h>

/* printd：以十进制打印数n */
void printd(int n )
{
   if  ( n < 0 )  {
       putchar( '-' );
       n = -n;
   }
   if  ( n / 10 )
       printd( n / 10 );
   putchar( n % 10 + '0' );
}
```

当一个函数递归调用自身时，每一次调用都会得到一个与以前的自动变量集合不同的新的自动变量集合。因此，在调用 printd (123) 时，第一次调用printd的变元n = 123。它把12传递给对printd的第二次调用，后者又把 1传递给对printd的第三次调用。第三次对 printd的调用将先打印1，然后再返回到第二次调用。从第三次调用返回后的第二次调用同样先打印 2，然后再返回到第一次调用。后者打印出3后结束执行。

另一个用于说明递归的一个例子是快速排序。快速排序算法是 C. A. R. Hoare于1962年发明的。对于一个给定的数组，从中选择一个元素（叫做分区元素），并把其余元素划分成两个子集合 —— 一个是由所有小于分区元素的元素组成的子集合，另一个是由所有大于等于分区元素的元素组成的子集合。对这样两个子集合递归应用同一过程。当某个子集合中的元素数小于两个时，这个子集合不需要再排序，故递归停止。

下面这个版本的快速排序函数可能不是最快的一个，但它是最简单的一个。在每一次划分子集合时都选取各个子数组的中间元素。

```
/* qsort：以递增顺序对v[left] … v[right] 进行排序 */
void qsort( int v[], int left, int right )
{
   int  i, last;
   void swap( int v[], int i, int j );

   if  ( left >= right )    /* 若数组所包含的元素数少于两个，则什么也不做 */
       return;
   swap( v, left, (left + right)/2 );    /* 把分区元素移到v[0] */
   last = left;
   for  ( i = left+1; i <= right; i++ )    /* 分区 */
        if  ( v[i] < v[left] )
              swap( v, ++last, i );
   swap( v, left, right );    /* 恢复分区元素 */
   qsort( v, left, last-1 );
   qsort( v, last+1, right );
}
```

之所以把数组元素交换操作作为一个独立的函数 swap，是因为它在qsort函数中要使用三次。

```
/* swap：交换v[i]与v[j]的值 */
void swap( int v[], int i, int j )
{
   int  temp;

   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

标准库中包含了qsort函数的一个版本，它可用于对任何类型的对象排序。

递归的中间结果不需在内存特别保存，因为在某处有一个被处理值的栈。递归的执行速度并不快，但递归代码比较紧致，要比相应的非递归代码易于编写与理解。在描述诸如树等递归

定义的数据结构时使用递归尤其方便，6.5节将给出这方面的一个例子。

练习4-12　运用printd函数的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成字符串。

练习4-13　编写一个递归版本的reverse(s)函数，把字符串s颠倒过来。

4.11　C预处理程序

C语言通过预处理程序提供了一些语言功能，预处理程序从理论上讲是编译过程中单独进行的第一个步骤。其中两个最常用的预处理功能是 #include指令（用于在编译期间把指定文件的内容包含进当前文件中）与 #define指令：（用任意字符序列取代一个标记）。本节还将介绍其他功能，如条件编译与带变元的宏。

4.11.1　文件包含

文件包含指令，即#include指令，使我们比较容易处理一组 #define指令以及说明等。在源程序文件中，任何形如：

```
#include  "文件名"
```

或

```
#include  <文件名>
```

的行都被替换成由文件名所指定的文件的内容。如果文件名用引号括起来，那么就在源程序所在位置查找该文件；如果在这个位置没有找到该文件，或者如果文件名用尖括号<与>括起来，那么就按实现定义的规则来查找该文件。被包含的文件本身也可包含 #include指令。

在源文件的开始处一般都要有一些 #include指令，或包含#define语句与extern说明，或访问诸如<stdio.h>等头文件中库函数的函数原型说明。（严格地说，这些没有必要做成文件。访问头文件的细节依赖于实现。）

对于比较大的程序，#include指令是把各个说明捆在一起的优选方法。它使所有源文件都被提供以相同的定义与变量说明，从而可避免发生一些特别讨厌的错误。自然地，如果一个被包含的文件的内容做了修改，那么所有依赖于这个被包含文件的源文件都必须重新编译。

4.11.2　宏替换

宏定义，即#define指令，具有如下形式：

```
#define  名字  替换文本
```

它是一种最简单的宏替换 —— 出现各个的名字都将被替换文本替换。#define指令中的名字与变量名具有相同的形式，替换文本可以是任意字符串。正常情况下，替换文本是#define指令所在行的剩余部分，但也可以把一个比较长的宏定义分成若干行，这时只需在尚待延续的行后加上一个反斜杠 \ 即可。#define指令所定义的名字的作用域从其定义点开始到被编译的源文件的结束。在宏定义中也可以使用前面的宏定义。替换只对单词进行，对括在引号中的字符串不起作用。例如，如果YES是一个被定义的名字，那么在 printf ("YES") 或YESMAN中不能进行替换。

用替换文本可以定义任何名字，例如：

```
#define  forever  for ( ; ; )    /* 无限循环 */
```

为无限循环定义了一个新的关键词 forever。

在宏定义中也可以带变元，这样可以对不同的宏调用使用不同的替换文本。例如，可通过：

```
#define  max( A, B )  ( ( A ) > ( B ) ? ( A ) : ( B ) )
```

定义一个宏max。对max的使用看起来很像是函数调用，但宏调用是直接将替换文本插入到代码中。形式参数（在此为A与B）的每一次出现都被替换成对应的实在变元。于是，语句

```
x = max( p+q, r+s );
```

将被替换成

```
x = ( ( p+q ) > ( r+s ) ? ( p+q ) : ( r+s ) );
```

只要变元能得到一致的处理，宏定义可以用于任何数据类型。没有必要像函数那样为不同数据类型定义不同的max。

如果仔细检查一下max的展开式，那么你将注意到它存在某些缺陷。其中作为变元的表达式要重复计算两次，当表达式中会带来副作用（如含有加一运算符或输入输出）时，会出现很坏的情况。例如，

```
max( i++, j++ )   /* 错 */
```

将对每一个变元做两次加一操作。同时也必须小心，为了保证计算次序的正确性要适当使用圆括号。请读者考虑一下，对于宏定义

```
#define  square( x )  x * x     /* 错 */
```

当用square(z + 1)调用它时会出现什么情况。

然而，宏还是很有价值的。在 <stdio.h>头文件中有一个很实用的例子， getchar与putchar函数在实际上往往被定义为宏，这样可以避免在处理字符时调用函数所需的运行时开销。在 <ctype.h>头文件中定义的函数也常常用宏来实现。

可以用#undef指令取消对宏名字的定义，这样做通常是为了保证一个调用所调用的是一个实际函数而不是宏：

```
#undef  getchar
int  getchar( void )  { … }
```

形式参数不能用带引号的字符串替换。然而，如果在替换文本中，参数名以 #作为前缀，那么它们将被由实际变元替换的参数扩展成带引号的字符串。例如，可以将其与字符串连接运算结合起来制作调试打印宏：

```
#define  dprint( expr )  printf( #expr " = %g\n", expr )
```

当用诸如

```
dprint( x/y );
```

调用该宏时，该宏就被扩展成

```
printf( "x/y" " = %g\n", x/y );
```

其中的字符串被连接起来，即这个宏调用的效果是：

```
printf( "x/y = %g\n", x/y );
```

在实际变元中，双引号 " 被替换成 \"，反斜杠 \ 被替换成 \\，故替换后的字符串是合法的字符串常量。

预处理运算符##为宏扩展提供了一种连接实际变元的手段。如果替换文本中的参数用 ##相连，那么参数就被实际变元替换， ##与前后的空白符被删除，并对替换后的结果重新扫描。例如，下面定义的宏paste用于连接两个变元：

```
#define  paste( front, back )  front ## back
```

从而宏调用paste(name, 1)的结果是建立单词name1。

关于##嵌套使用的规则比较难以掌握，详细细节请参阅附录 A。

练习4-14　定义宏swap(t, x, y)，用于交换t类型的两个变元（使用分程序结构）。

4.11.3　条件包含

在预处理语句中还有一种条件语句，用于在预处理中进行条件控制。这提供了一种在编译过程中可以根据所求条件的值有选择地包含不同代码的手段。

#if语句中包含一个常量整数表达式（其中不得包含sizeof、类型强制转换运算符或枚举常量），若该表达式的求值结果不等于 0时，则执行其后的各行，直到遇到 #endif、#elif或#else语句为止（预处理语句#elif类似于if语句的else if结构）。在#if语句中可以使用一个特殊的表达式 defined(名字)：当名字已经定义时，其值为 1；否则，其值为0。

例如，为了保证hdr.h文件的内容只被包含一次，可以像下面这样用条件语句把该文件的内容包围起来：

```
#if  !defined( HDR )
#define  HDR

/* hdr.h文件的内容*/

#endif
```

被#if与#endif包含的第一行定义了名字HDR，其后的各行将会发现该名字已有定义并跳到#endif。还可以用类似的样式来避免多次重复包含同一文件。如果连续使用这种，那么每一个头文件中都可以包含它所依赖的其他头文件，而不需要它的用户去处理这种依赖关系。

下面的预处理语句序列用于测试名字 SYSTEM以确定要包含进哪一个版本的头文件：

```
#if SYSTEM == SYSV
    #define HDR  "sysv.h"
#elif SYSTEM == BSD
    #define HDR  "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR  "msdos.h"
#else
    #define HDR  "default.h"
```

```
#endif
# include  HDR
```

当需要测试一个名字是否已经定义时，可以使用两个特殊的预处理语句： #ifdef**与**#ifndef**。可以使用**#ifdef**将上面第一个关于**#if**的例子改写如下：**

```
#ifdef  HDR
#define  HDR

/* hdr.h文件的内容 */

#endif
```


第5章 指针与数组

指针是一种用于存放另一个变量的地址的变量。指针在 C语言中使用非常广泛，部分原因是指针有时是表达计算的唯一方法，部分原因是较之其他方法指针通常可以生成更高效、更紧凑的代码。指针与数组之间的关系十分密切，本章将对它们之间的关系进行讨论，并探讨如何使用这种关系。

指针和goto语句一样，会产生令人难以理解的程序。当我们不注意使用指针时这种情况尤其明显，而且容易在程序中建立指向未预料到地方的指针。然而，如果小心地使用指针，则可以利用它来产生简单明了的程序。在下文我们将说明这一点。

ANSI C中最重要的变化是使操纵指针的规则更加清楚，这些规则实际上优秀的程序人员和好的编译程序中已采用的指针操作方法。此外， ANSI C中用类型void *(指向void的指针)代替char *作为通用指针的类型。

5.1 指针与地址

我们先用一个简化的图形来看看内存是如何组织的。机器的存储器通常由连续编号（或编址）的存储单元序列组成，这些存储单元可以以单个的或相连成组的方式操纵。通常情况下，一个字节可表示一个字符，一对相连的存储单元可表示一个短整数，而四个相邻的字节则构成一个长整数。指针由能存放一个地址的一组存储单元 (通常是两个或四个字节)构成。因此，如果c的类型是char并且p是指向c的指针，那么可用下图表示它们之间的关系：

图 5-1

一元运算符&用于取一个对象的地址，因而语句

```
p = &c;
```

用于将c的地址赋给变量p，并且说p是指向c的指针。取地址运算符 &只能应用于内存中的对象（即变量与数组元素），它不能对表达式、常量或寄存器变量进行操作。

一元运算符 * 是间接寻址或间接引用运算符，当它应用于指针时，它将访问指针所指向的对象。假定x与y是整数，且ip是指向整数的指针。下面的代码段显示了如何

在程序中说明指针及怎样使用运算符 &和 * :

```
int  x = 1, y = 2, z[10];
int  *ip;          /* ip是指向整数的指针 */

ip = &x;           /* ip现在指向x */
y = *ip;           /* y的值现在为1 */
*ip = 0;           /* x的值现在为0 */
ip = &z[0];        /* ip现在指向z[0] */
```

在前面的有关章节中我们已经对变量 x、y与z进行了说明。这里再对上面所使用的指针变量 ip说明如下：

```
int  *ip;
```

其目的是为了便于记忆，它的意思是表达式 *ip是int类型的。这种变量说明的语法模仿了可能包含该变量的表达式的语法。同样，对函数的说明也可以采用这种方法。例如，说明

```
double  *dp, atof(char *);
```

的意思是在一个表达式中 *dp和atof(s)都具有double类型的值，且atof的参数是一个指向 char类型的指针。

应该注意的是一个指针只能指向一个特定类型的对象：每一个指针对象也有一确定的数据类型(例外情况：指向 void类型的指针可转换成指向任何对象类型的指针，但它不能间接引用它自身。5.11节将再次讨论这个问题)。

如果指针ip指向整数x，那么在x可以出现的任何上下文中 * ip同样可以出现，因此语句

```
*ip = *ip + 10;
```

将 * ip的值增加10。

一元运算符 *和&的优先级比算术运算符的优先级高，因而赋值语句

```
y = * ip + 1
```

把ip所指向的对象的值取出来，与 1相加后将结果赋给 y，而赋值语句

```
* ip += 1
```

则将ip所指向的对象的值加 1，它们之间的区别正如语句

```
++*ip
```

和

```
(*ip) ++
```

之间的区别一样。在后一个语句中必须要用一对圆括号将 *ip括起来，否则，该表达式将对 ip进行加一运算而不是对 ip所指向的对象进行加一运算，这是因为诸如 *和++这样的一元运算符在表达式求值时按从右到左的顺序和运算分量结合。

最后，由于指针也是变量，所以在程序中不必通过间接引用的方法就可以直接使用它们。例如，如果iq是另一个指向整数的指针，那么语句

```
iq = ip
```

就将ip的值赋给iq，因此该语句使得指针 iq指向ip所指向的对象。

5.2 指针与函数变元

由于C语言以按值调用的方式将变元传递给函数，因而被调用函数不能直接更改调用函数中变量的值。例如，排序函数可能用一个名叫 swap的函数来交换两个次序颠倒的元素。但如下语句实现不了这个功能：

```
swap(a, b);
```

这里swap函数定义为：

```
void  swap(int x, int y)  /* 错误定义的函数 /
{
    int  temp;

    temp = x;
    x = y ;
    y = temp;
}
```

由于按值调用的缘故，上述 swap函数将不会影响调用它的子程序中的变元 a和b的值。该函数仅交换临时变量区中与a和b相对应的临时拷贝的值。

调用程序要想得到预期的结果，它应将指向所要交换值的指针传递给被调用函数 swap，即：

```
swap(&a, &b);
```

由于一元运算符&用来取一个变量的地址，这样 &a就是一个指向变量a的指针。swap函数本身将它所有的形式参数都说明为指针类型，并且通过这些指针来间接访问它们所指向的运算分量。

```
void  swap(int * px, int * py)  /* 交换*px和*py */
{
    int  temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

可用图形将之形象化描述成图 5-2所示。

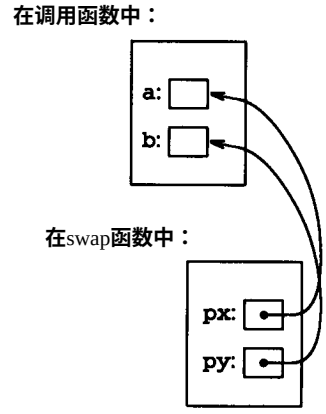

图 5-2

指针类型的变元使得被调用函数能够访问和更改调用函数中对象的值。考虑这样一个例子：函数 getint通过将字符流分解成整数值而完成自由格式输入的转换，它每被调用一次就得到一个整数。getint必须返回转换后的整数，并且在输入结束时也要发出已到达文件尾的信号。这些值必须通过各自的路径返回， EOF(用于表示遇到文件结尾的文件结束指示符)不管用什么值表示都可以，当然也可用一个整数值表示它。

可以这样来设计该函数：让 getint将表示文件结尾的状态作为函数值返回，同时使用一个指针参数存储转换后的整数，从而传递给调用函数。函数 scanf的实现也采用了这个方法，具体细节请参见7.4节。

下面的循环语句调用getint函数来给数组元素赋整数值：


```
int  n, array[SIZE], getint(int *);
for ( n = 0; n< SIZE && getint(&array[n]) !=EOF；n++)
     ;
```

循环语句每调用一次getint就把输入流中的下一个整数赋给数组元素 array[n]并将n加1。注意：必须将array[n]的地址传递给函数getint，否则函数getint就无法将转换后的整数传回给调用者。

这个版本的getint函数当遇到文件结尾时返回 EOF，当下一个输入不是一个数字时返回 0，当在输入中包含一个有意义的数字时返回一个正数。

```
#include <ctype.h>

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

/* getint:  将输入流中的下一个整数赋给*pn  */
int  getint(int *pn)
{
   int  c, sign;

   while (isspace(c = getch( )))   /* 跳过空白符 */
       ;
   if (!isdigit(c) && c != EOF && c != '+' && c !='-') {
       ungetch(c); /* 输入不是一个数字 */
       return  0;
    }
    sign = ( c = = '-' ) ? -1 : 1;
    if (c = =  '+' || c = = '-')
       c = getch( );
    for (*pn = 0; isdigit(c); c = getch( ))
       *pn = 10 * * pn + (c - '0');
    *pn *= sign;
    if (c != EOF )
       ungetch(c);
    return c;
}
```

在getint中，*pn作为一个普通的整数类型变量使用。还使用了函数 getch和ungetch(参见4.3节)，这样，函数getint中必须读出的一个额外的字符就可以写回到输入流中了。

练习5-1　在上面的例子中，函数 getint将后面不跟数字的一个 +或-视为0的有效表达方式。通过将+或-写回输入流的方法来解决这个问题。

练习5-2　模仿函数getint的实现方法，写一个浮点转换函数 getfloat。getfloat函数返回值的类型是什么？

5.3　指针与数组

在C语言中，指针和数组之间的关系十分密切，因此以下将同时讨论指针与数组。数组下标所能完成的任何运算都可以用指针来实现。一般而言，指针运算比数组下标运算的速度快，但

客观上讲，用指针实现的程序稍微难理解一些。

说明语句：

```
int  a[10] ;
```

定义了一个大小为 10的数组 a，即定义了一个由 10个存储在内存相邻区域内的名为 a[0]，a[1]，...，a[9]的对象组成的集合（见图5-3）。

图 5-3

其中，符号a[i]表示该数组的第i个元素。如果pa是一个指向整数类型对象的指针，则其说明为：

```
int  *pa;
```

那么赋值语句

```
pa = &a[0]
```

用于将指针pa指向数组a的第0个元素，也即pa的值为数组元素a[0]的地址（见图5-4）。

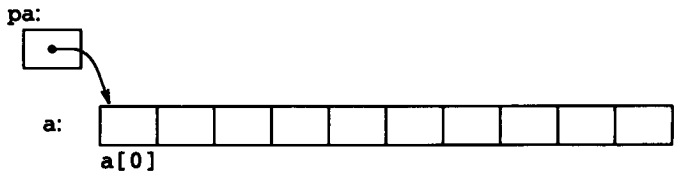

图 5-4

则赋值语句

```
x = *pa;
```

把数组元素a[0]的内容赋给变量x。

如果pa指向一个数组的某一特定元素，那么根据指针运算的定义， pa + 1指向该数组中pa所指向数组元素的下一个元素， pa+i指向pa所指向数组元素之后的第 i个元素，而pa -i指向pa所指向数组元素之前的第i个元素。因此，如果指针 pa指向a[0]，那么

```
*(pa+1)
```

表示数组元素a[1]的内容， pa+i表示数组元素a[i]的地址， *(pa+i) 表示数组元素a[i]的内容（见图5-5）。

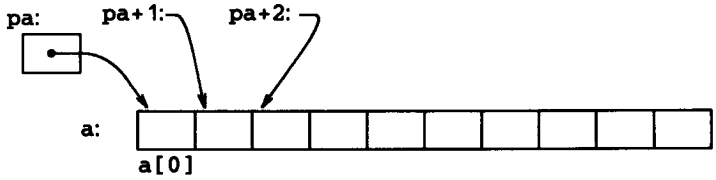

图 5-5

无论数组a的元素具有什么样的类型或大小，上面的结论都正确。“ 指针加1 ”的意思是pa+1指向pa所指对象的下一个对象，相应地， pa+i指向pa所指对象之后的第 i个元素，这可以推广到

所有的指针运算。

下标和指针运算有着很密切的对应关系。按照定义，一个类型为数组的变量或表达式的值是该数组第0个元素的地址。因此在赋值语句

```
pa = &a[0];
```

执行后，pa和a具有相同的值。由于一个数组的名字即该数组第 0个元素的位置，所以赋值语句pa = &a[0]也可以写成如下形式：

```
pa = a;
```

对数组元素a[i]的引用也可以写为*(a+i)这样的形式，这一点至少初看起来很令人吃惊。在求数组元素a[i]的值时，C语言实际上先将其转换成*(a+i)的形式然后再求值，因而在程序中这两种形式等价。当把取地址运算符 &应用于这两种等价的表示形式时，可以知道 &a[i]和a+i的含义也是相同的：a+i是a之后第i个元素的地址。相应的，如果 pa是一个指针，那么表达式中可使用具有下标的指针 pa，pa[i]与*(pa+i)的含义一样。简而言之，一个用数组和下标实现的表达式可等价地用指针和和偏移量来实现。

然而，必须注意到，数组名字和指针之间仍然存在着一点区别。指针是变量，因而在C语言中，语句pa = a和pa++都是合法的。但数组名字不是变量，因而诸如a = pa和a++这样的语句是非法的。

当把一个数组名字传递给一个函数时，实际上传递的是该数组第一个元素的位置。由于一个数组名字参数就是一个指针，所以在被调用函数中，与数组名字参数对应的变元是一个包含地址值的局部变量。可以利用这个事实编写另一个版本的strlen函数，该函数用于计算一个字符串的长度。

```
/* strlen: 返回字符串s的长度 */
int  strlen(char *s)
{
   int  n;
   for (n = 0; *s != '\0'; s++)
        n++;
   return  n;
}
```

由于s是一个指针类型的变量，所以对其进行加一运算是合法的， s++不会影响调用strlen的函数中的字符串，它仅对该指针在 strlen函数中的私有拷贝进行加一运算。因而诸如下面这样的函数调用：

```
strlen("hello, word");   /* 字符串常量 */
strlen(array);           /* array是一个具有100元素的字符数组 */
strlen(ptr);             /* ptr是一个指向字符类型对象的指针 */
```

都能正常运行。

在函数定义中，将

```
char  s[ ];
```

和

```
char  *s;
```

作为函数的形式参数所表示的含义是等价的，我们更喜欢使用后一种形式，因为它比前者更明

白地表示了该参数是一个指针。当数组名字被传递给函数时，函数就根据操作的方便来判定传递给它的是数组还是指针，然后按相应的方式操纵该参数。为了清楚而恰当地描述函数，在函数中甚至可同时使用数组和指针这两种表示法。

也可以通过传递指向子数组的指针的方法把数组的一部分作为参数传递给函数。例如，如果a是一个数组，那么如下两个函数调用：

```
f ( &a[2] )
```

与

```
f ( a+2 )
```

都把起始于a[2]的子数组的地址传递给函数f。在函数f中，参数说明的形式可以为：

```
f ( int arr[ ] ) { ... }
```

或：

```
f ( int *arr ) { ... }
```

因此就函数f本身而言，当其参数是一个指向较大数组的子数组的指针时并不会对它造成什么影响。

如果确信某个元素存在，那么也可以在数组中用下标指向后面的元素：诸如 p[-1]、p[-2]等等这样的表达式在语法上都是合法的，它们用于引用位于 p[0]之前的某个相应元素。当然，引用数组边界之外的对象是非法的。

5.4 地址算术运算

如果p是一个指向数组某个元素的指针，那么 p++对p进行加一运算使它指向下一个元素，而p+=i对p进行增量运算使它指向指针 p当前所指向的元素之后的第 i个元素。这类运算是指针或地址的算术运算中最简单的形式。

C语言中地址的算术运算方法很有规律，将指针、数组和地址的算术运算集成在一起是该语言的一大特色。下面以一个基本的存储分配程序为例来说明这个问题，它由两个函数组成。第一个函数alloc(n)返回一个指向n个连续字符存储单元的指针， alloc函数的调用者可利用该指针来存储字符序列。第二个函数 afree(p)释放已分配的存储空间，因而它们以后可以进行再分配。这两个函数都是“ 基本函数 ”，对afree函数的调用必须以与调用 alloc函数相反的次序进行，即alloc与afree以栈式（即后进先出列表的方式）进行存储空间的管理。标准库中提供了类似的函数malloc和free来管理存储空间，但对它们没有上述限制， 8.7节将说明如何实现这些函数。

最容易的实现方法是让 alloc函数对一个大字符数组 allocbuf中的空间进行分配。该数组是alloc和free的私有数组。由于函数 alloc和free处理的对象是指针而不是数组下标，其他函数无需知道该数组名字，因而可以在包含 alloc和free的源文件中将该数组说明为 static类型，使得它对外不可见。实际实现时，这个数组最好没有名字，它可通过调用 malloc函数或向操作系统申请一个指向无名存储区的指针来得到。

另一个需要知道的信息是 allocbuf中的空间当前已经使用了多少。使用指针 allocp指向allocbuf数组中下一个空闲单元。当向 alloc申请n个字符的空间时， alloc检查allocbuf数组看有没有足够的剩余空间用于分配。如果有足够的空闲空间， alloc就返回allocp的当前值（即自由块的

开始位置)，然后将allocp加n使它指向下一个空闲区域。如果空闲空间不够，则 alloc返回0。如果p在allocbuf的边界之内，afree(p) 仅将allocp的值设置为p（见图5-6)。

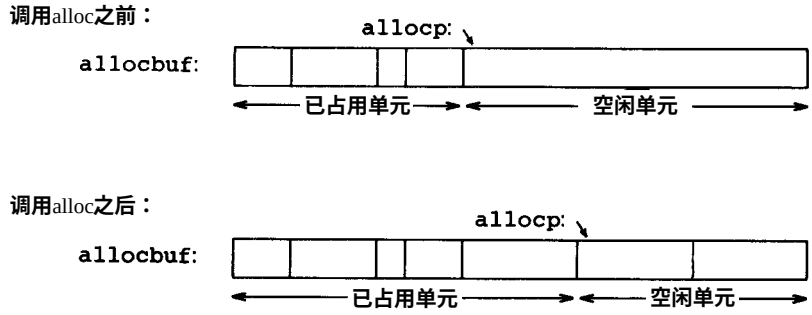

图 5-6

```
#define  ALLOCSIZE  10000              /* 可用空间大小 */

static char  allocbuf[ALLOCSIZE];      /* alloc使用的存储区 */
static char  *allocp = allocbuf;       /* 下一个空闲位置 */

char  *alloc(int n)                    /* 返回指向n个字符的指针 */
{
    if  (allocbuf + ALLOCSIZE - allocp >= n ) { /* 有足够的空闲空间 */
        allocp += n;
        return  allocp - n;  /* 分配前的指针 */
    } else       /* 空闲空间不够  */
        return  0;
}

void  afree(char *p)   /* 释放p指向的存储区 */
{
   if (p >= allocbuf && p <allocbuf + ALLOCSIZE)
       allocp = p;
}
```

一般而言，可以像对其他变量一样对指针进行初始化，尽管在一般情况下，指针的有意义值只能为0或表示已定义的某一类型的数据的地址的表达式。例如，说明语句

```
static char  *allocp = allocbuf;
```

将allocp定义为字符类型指针并将它初始化为 allocbuf的起始地址，该起始地址是程序开始运行时的下一个空闲位置。上述语句也可以写为如下形式：

```
static char  *allocp = &allocbuf[0];
```

这是由于该数组名字实际上是其第 0个元素的地址。

条件语句：

```
if  ( allocbuf + ALLOCSIZE - allocp >= n )  { /* 有足够的空闲空间 */ }
```

中的测试部分检查是否有足够的空闲空间满足 n个字符的存储空间请求。如果空闲空间足够，存储空间分配后的allocp新值至多比allocbuf的尾端地址大1。如果存储空间的申请得到满足， alloc返回一个指向所需大小的字符块首地址的指针（注意函数本身的说明)。否则，alloc必须返回说明没有足够的空闲空间可以分配的信号。 C语言保证0不是数据的有效地址，因而0返回值可用来表示发生了一个异常事件。在本例中， 0返回值表示没有足够的空闲空间可以分配。

指针与整数不能相互转换，但 0例外：常量0可以赋给指针，指针也可以和常量 0进行比较。程序中经常用符号常量 NULL代替常量0，这样有助于更清楚地记住常量 0是指针的一个特殊值。符号常量NULL 在标准头文件<stdio.h>中定义，后面将用到NULL。

诸如

```
if (allocbuf + ALLOCSIZE - allocp >= n)
```

和

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

等条件测试语句表明指针的算术运算有几个重要特点。首先，在某些情况下对指针可以进行比较运算。例如，如果指针p和q指向同一个数组的成员，那么它们之间就可以进行诸如 = =、 !=、 <、>= 等关系比较运算。如果p所指向的数组成员的位置在 q所指向的数组成员位置之前，那么关系

```
p < q
```

成立。任何指针与0进行相等或不等的比较运算都有意义。但对指向不同数组成员的指针之间的算术或比较运算，其执行行为没有定义。 (这里有一个特例：指针的算术运算中可使用一个数组的第一个元素的地址。)

其次，如前面所见，指针可以和整数进行相加或相减运算。例如，表达式

```
p + n
```

表示指针p当前所指对象之后的第 n个对象的地址。无论指针 p所指的对象具有什么样的类型，上述结论都正确。在计算 p+n时，n根据p所指对象的大小按比例缩放，而 p所指对象的大小决定于 p的说明。例如，如果一个整数占四个字节的存储空间，那么 int对应的n就按4的倍数来计算。

指针的减运算也有意义：如果 p和q指向相同数组中的元素且 p<q，那么q-p+1就等于位于p和q所指元素之间的元素的数目。由此可得函数 strlen的另一个版本如下：

```
/* strlen: 返回字符串s的长度 */
   int  strlen(char *s)
{
   char  *p = s;

   while (*p != '\0')
     p++;
     return  p - s;
}
```

在上述说明中，指针 p被初始化为s，即它指向该字符串的第一个字符。在 while循环语句中，依次检查字符串中的每个字符直到遇到空字符 '\0'。由于p是指向字符的指针，所以p++每执行一次就使得p指向下一个字符的地址， p-s则给出已经检查过的字符数，即从字符串第一个字符开始

到p所指字符之间的字符串长度。(字符串中的字符数有可能超过整数类型所能表示的范围。头文件<stddef.h>中定义的类型ptrdiff_t足以表示两个指针的带符号差值。然而为谨慎起见，用 size_t作为函数strlen的返回值类型，以使其和标准库中的版本相匹配。 size_t是由运算符sizeof返回的无符号整数类型。)

指针的算术运算具有一致性：如果处理的是比字符类型占据更多存储空间的浮点类型，并且p是一个指向浮点类型的指针，那么执行 p++后p就指向下一个浮点数的地址。因此只需将 alloc和afree函数中所有的char类型替换为float类型，就可以得到一个针对浮点类型而不是字符类型的内存分配函数版本。所有的指针运算都会自动考虑它所指对象的大小。

有效的指针运算包括：相同类型指针之间的赋值运算；指针值加或减一个整数值的运算；指向相同数组中的元素的指针之间的减或比较运算；将指针赋 0或指针与0之间的比较运算。所有其他形式的指针运算均非法。诸如下列形式的运算就是非法的指针运算：指针间的加法、乘法、除法、移位或屏蔽运算；指针值加单双精度浮点数的运算；除两者之一是 void*类型指针外，不经强制类型转换就将指向一种类型对象的指针赋给指向另一种类型对象的指针的运算。

5.5 字符指针与函数

如下所示的字符串常量：

```
"I am a string"
```

是一个字符数组。在字符串的内部表示中，该数组以空字符 '\0' 结尾，所以程序中可通过检查空字符找到该字符数组的结尾。字符串常量所占的存储单元数也因此比双引号内的字符数大 1。

字符串常量最常见的用处或许就是用做函数变元，正如函数

```
printf("hello, world\n ");
```

中的字符串常量。当诸如这样的一个字符串出现在程序中时，实际上是通过字符指针访问它，上述函数printf接收的是一个指向字符数组的头一个字符的指针。即，字符串常量可通过一个指向其第一个元素的的指针来访问。

字符串常量不一定总是作为函数变元出现。如果 pmessage被说明为

```
char  *pmessage;
```

那么语句

```
pmessage = "now is the time";
```

把一个指向该字符数组的指针赋值给指针变量 pmessage，其中并未进行字符串的复制，只涉及到指针的操作。C语言没有提供将一个完整的字符串作为一个整体处理的运算符。

如下两个定义的差别很大：

```
char  amessage[ ] = "now is the time";  /* 定义一个数组 */
char  *pmessage = "now is the time";    /* 定义一个指针 */
```

上述说明中，amessage是一个足以存放字符串初值和空字符 '/0' 的一维数组。可以更改数组中的单个字符，但amessage是一个不可改变的常量，它总指向同一存储区。另一方面， pmessage是一个指针，其初值指向一个字符串常量，之后它可以被修改指向其他地址，但如果试图修改字

符串的内容，结果将不确定（见图 5-7）。

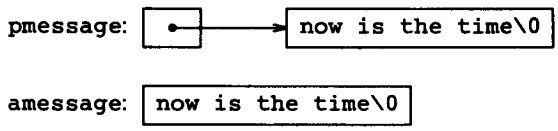

图 5-7

下面通过研究从标准库中改编的两个有用的函数来阐明指针和数组其他方面的问题。第一个函数strcpy(s, t)把指针t所指向的字符串复制到指针 s所指向的位置。看起来好像用语句 s=t就可实现该功能，但这样一来，实际上是复制了指针，而非字符串。为进行字符串的复制，使用了一个循环语句。 strcpy函数的第1个版本通过数组方法来实现：

```
/* strcpy: 将指针t指向的字符串复制到指针s所指的位置；数组下标版本 */
void  strcpy(char *s, char *t)
{
   int  i;

   i = 0;
   while ((s[i] = t[i]) != '\0')
      i++;
}
```

与之相对，下面是用指针方法实现的 strcpy函数：

```
/* strcpy: 将指针t指向的字符串复制到s所指的位置；指针方法版本1 */
void  strcpy(char *s, char *t)
{
   while ((* s = * t) != '\0') {
      s++;
      t++;
   }
}
```

因为所有的变元都是按值传递的，所以在 strcpy函数中可以以任何方式使用参数 s和t。在这里s和t是已初始化过的指针，循环每执行一次，它们就沿着相应的数组前进一个字符，直到将 t的结束符 '\0' 复制到s。

实际上的strcpy函数并不像上面写的那样。经验丰富的程序员更喜欢将它写成如下形式：

```
/* strcpy: 将指针t指向的字符串复制到s所指的位置；指针方法版本2 */
void  strcpy(char *s, char *t)
{
   while ((*s++ = *t++) != '\0')
       ;
}
```

这个版本的函数将 s和t的加一运算放到循环的测试部分。表达式 *t++的值是进行加一运算之前 t所指向的字符，后缀加一运算符在取出该字符之后才改变 t的值。同理，字符在s进行加一运算之前存储到指针s所指向的位置。这个字符同时也被用来和空字符 '/0' 进行比较运算以控制循环的

执行。最后的结果是依次将t所指向的字符复制到s所指的位置，直到遇到t的结束符 '\0' 为止(也复制该结束符)。

从程序简练的角度看，表达式同'\0'的比较是多余的，因为问题仅仅在于表达式是否为 0。因而该函数可进一步写成如下形式：

```
/* strcpy: 将指针t指向的字符串复制到s所指的位置；指针方法版本3 */
void  strcpy(char *s, char *t)
{
    while (*s++ = *t++)
       ;
}
```

这个函数乍一看似乎不好理解，但应该注意这种表示方法所带来的好处并掌握它，因为 C程序中经常采用这种写法。

标准库(<string.h>)中的函数strcpy用于将目标字符串作为函数值返回。

我们要研究的第二个函数是字符串比较函数 strcmp(s, t)，该函数比较字符串s和t并且视s在字典序上小于、等于或大于t而返回负整数、0或正整数。该返回值是s和t由前往后逐字符比较时遇到的第一个不相等处的字符的差值。

```
/* strcmp: 视s在字典序上小于、等于或大于t而返回负整数、0或正整数*/
int  strcmp(char *s, char *t)
{
    int  i;

    for (i = 0; s[i] = = t[i]; i++)
       if (s[i] = = '\0')
           return  0;
    return  s[i] - t[i];
}
```

下面是用指针方法实现的strcmp:

```
/* strcmp: 视s在字典序上小于、等于或大于t而返回负整数、0或正整数 */
int  strcmp(char *s, char *t)
{
    for (;*s = = *t; s++, t++)
       if (*s = = '\0')
          return 0;
    return  *s - *t;
}
```

由于++和--既可以作为前缀运算符也可作为后缀运算符出现，所以尽管 *与运算符++和--的结合不多见但也可能会出现。例如，如下表达式：

```
*--p
```

在取出指针p所指向的字符之前对p进行减一运算。事实上，下面的一对表达式：

```
*p++ = val;     /* 将val压进栈*/
val = *--p;     /* 将栈顶元素弹出到val中*/
```


是进栈和出栈的标准用法，见4.3节。

头文件<string.h>中包含本节中所提到的函数的说明，其中还包括标准库中的许多其他字符串处理函数的说明。

练习5-3　用指针方法实现第2章所描述过的函数strcat：strcat(s, t)将t所指向的字符串添加到s所指向的字符串末尾。

练习5-4　编写函数strend (s, t): 如果字符串t出现在字符串s的尾部，则返回1；否则返回0。

练习5-5　实现库函数strncpy、strncat和strncmp，它们最多对变元字符串的前n个字符进行操作。例如，函数strncpy(s, t, n)将t所指向字符串中最多前n个字符复制到s所指向的字符数组中。详尽的描述请参见附录B。

练习5-6　用指针而不是数组索引改写前面章节和练习中的某些程序，包括：getline(第1、4章)；atoi、itoa以及它们的变种(第2、3、4章)；reverse(第3章)；strindex和getop(第4章)。

5.6　指针数组与指向指针的指针

由于指针本身也是变量，所以它们也可以像其他变量一样存储在数组中。下面通过编写UNIX程序sort的一个简化版本来说明这一点，该程序按字母顺序对由文本行组成的集合进行排序。

第3章中曾描述过一个用于对整数数组中的元素进行排序的Shell排序函数，在第4章用快速排序算法对之做了改进。这些排序算法在本例中同样有效，只不过现在处理的是长度不一的文本行，而且与整数不同的是它们不能在单个运算中完成比较或移动操作。我们需要一个能够高效、方便地处理可变长度文本行的数据表示方法。

很自然地用到了指针数组。如果待排序的文本行首尾相连地存储在一个长字符数组中，那么每一个文本行可通过指向它的第一个字符的指针来访问。这些指针自身可以存储在一个数组中。于是把分别指向两个文本行的指针传递给函数strcmp就可实现对这两个文本行的比较。当交换两个次序颠倒的文本行时，实际上交换的是指针数组中与这两个文本行相对应的指针，而不是这两个文本行自身（见图5-8）。

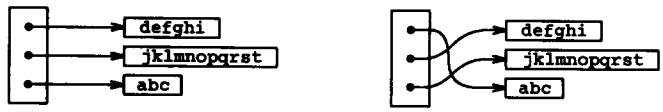

图　5-8

这种实现方法消除了由于移动文本行本身所带来的复杂的存储管理和巨大的开销这两个问题。

排序过程由如下三步组成：

读取输入流中的所有文本行

对文本行进行排序

按次序打印文本行

在通常情况下，最好将程序分成若干个和问题的自然划分相匹配的函数，其中主函数控制

其他函数的执行。关于文本行如何排序这一步稍后再做说明，现在把精力集中在数据结构的确定、输入和输出函数的实现上。

输入函数必须收集和保存每一个文本行中的字符，并建立一个指向这些文本行的指针的数组。它同时也统计已输入的行数，因为在排序和打印时要用到这一信息。由于输入函数只能处理有限数目的输入行，所以在输入行数太多而超过所限定的最大行数时它返回某个用于表示非法行数的数值，例如-1。

输出函数只需按文本行所对应的指针在指针数组中出现的次序打印这些文本行即可。

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

#define  MAXLINES  5000    /* 能够存储的最大文本行数 */

char  *lineptr[MAXLINES];   /* 指向文本行的指针数组*/

int  readlines(char *lineptr[ ], int nlines);
void  writelines(char *lineptr[ ], int nlines);

void  qsort(char *lineptr[ ], int left, int right);

/* 对输入的文本行进行排序*/
main( )
{
    int  nlines;     /* 输入行数目*/

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
       qsort(lineptr, 0, nlines-1);
       writelines(lineptr, nlines);
       return  0;
    } else  {
       printf("error: input too big to sort\n");
       return  1;
    }
}

#define  MAXLEN  1000  /* 能输入的文本行的最大长度*/
int  getline(char *, int);
char  *alloc(int);

/* readlines: 读输入行*/
int  readlines(char *lineptr[ ], int maxlines)
{
   int  len, nlines;
   char  *p, line[MAXLINE];

   nlines = 0;
```

```
   while ((len = getline(line, MAXLEN)) > 0)
      if (nlines >= maxlines | | (p = alloc(len)) = = NULL)
          return  -1;
      else  {
          line[len-1] = '\0';   /* 删除换行符 */
          strcpy(p, line);
          lineptr[nlines++] = p;
       }
    return  nlines;
}

/* writelines: 写输出行 */
void  writelines(char *lineptr[ ], int nlines)
{
    int  i;

    for ( i = 0; i < nlines; i++)
       printf("%s\n", lineptr[i]);
}
```

关于函数getline参见1.9节。

在这个例子中，新出现的且比较重要的是指针数组 lineptr的说明：

```
char  *lineptr[MAXLINES]
```

它表示lineptr是一个具有 MAXLINES个元素的一维数组，其中数组的每个元素是一个指向字符类型对象的指针。即， lineptr[i]是一个字符指针，而 *lineptr[i]是该指针所指向的第 i个文本行的首字符。

由于lineptr本身是一个数组名字，故可采用与前面的例子相同的方法将其作为指针使用，因而writelines可改写为：

```
/* writelines: 写输出行 */
void  writelines(char *lineptr[ ], int nlines)
{
   while (nlines -- > 0)
      printf("%s\n", *lineptr++);
}
```

循环刚开始执行时， *lineptr指向第一行，每执行一次加一运算就使得 *lineptr指向下一行，同时nline进行减一运算。

在掌握了输入和输出函数的实现方法之后，下面接着研究文本行的排序问题。在这里需要对第4章的快速排序函数做一些小改动：首先，要修改该函数的说明部分；其次，要通过调用strcmp函数来完成文本行的比较运算。但排序算法在这里仍然有效，无需改动。

```
/* qsort: 按递增顺序对v[left]...v[right] 进行排序*/
void  qsort(char * v[ ], int left, int right)
{
    int  i, last;
```

```
    void  swap(char *v[ ], int i, int j);

    if (left >= right )      /* 如果数组元素个数小于2，则返回*/
        return;
    swap(v, left, (left + right) / 2);
    last = left;
    for (i = left + 1; i <= right; i++)
       if (strcmp(v[i], v[left] < 0)
           swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last - 1);
    qsort(v, last + 1, right);
}
```

同样，swap函数也只需做些很小的改动：

```
/* swap: 交换v[i] 和v[j] */
void  swap(char *v[ ], int i, int j)
{
    char  *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

由于v的任一元素(别名为lineptr)都是字符指针并且temp也是字符指针，因而其中一个指针指向的字符串可以被复制到另一个指针指向的位置。

练习5-7　改写函数readlines，将输入的文本行存储到由主函数提供的一个数组中，而不是存储到通过调用alloc分配得到的存储空间中。该函数的运行速度比改写前快多少？

5.7　多维数组

C语言也提供了有些类似于矩阵的多维数组，尽管实际上它们并不像指针数组使用得那么广泛。本节将阐述多维数组的特性。

下面研究把某月某日这种日期表示形式转换成某年中第几天的表示形式及相反的问题。例如，3月1日是非闰年的第60天，是闰年的第61天。定义如下两个函数来进行日期的转换：函数day_of_year将某月某日的日期表示形式转换为某一年中第几天的表示形式，函数 month_day则做相反的转换。因为后一个函数要返回两个值，所以在函数 month_day中，月和日这两个变元应使用指针的形式。例如语句

```
month_day(1988, 60, &m, &d);
```

把m的值设置为2而把d的值设置为29(2月29日)。

这些函数都要用到一张用于记录每月有多少天数的表 (如“9月有30天”等)。因为对闰年和非闰年而言每个月的天数不同，所以将它们分别放到一个二维数组的两行中比在计算的过程中

判断2月有多少天数使用起来更容易。该数组及进行日期转换的相应函数如下所示：

```
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: 将某月某日的日期表示形式转换为某年中第几天的表示形式*/
int  day_of_year(int year, int month, int day)
{
   int  i, leap;

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

/* month_day: 将某年中第几天的日期表示形式转换为某月某日的表示形式*/
void  month_day(int year, int yearday, int *pmonth, int *pda)
{
    int  i, leap;

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

前面的章节中曾讲过，逻辑表达式的算术运算值是0(当逻辑值为假时)或者是1(当逻辑值为真时)。在本例中，与leap对应的逻辑表达式的值不是0就是1，所以leap可用做数组daytab的下标。

数组daytab在函数day_of_year和month_day的外面说明，以使这两个函数都可以使用该数组。之所以将daytab的元素说明为字符类型，是为了说明字符类型变量中存放较小的非字符整数也是合法的。

daytab是到目前为止我们所遇到的第一个二维数组。在C语言中，二维数组实际上是一种特殊的一维数组：其元素也是一维数组。因此数组下标应写成如下形式：

```
daytab[i][j]   /* [行][列] */
```

而不能写成：

```
daytab[i, j]   /* 错误的形式*/
```

除符号上的区别外，C语言中二维数组的使用方式和其他语言一样。数组元素按行存储，因此当按存储顺序访问数组时，最右边的数组下标（即列）变化得最快。

数组通过花括号内的初值列表来初始化，二维数组的每一行由相应的子列表初始化。本例将数组daytab的第一列元素置为0，因而月份为1~12而不是0~11。由于在这里空间不是主要问题，

这种处理方式比在程序中调整数组的下标显得更清楚。

如果要将二维数组作为变元传递给函数，那么函数的参数说明中应该指明相应数组的列数。数组的行数不必指定，因为正如前所述，函数调用时传递给它的是一个指向由行向量构成的一维数组的指针，其中每个行向量是具有 13个整数元素的一维数组。在这个例子中，传递给函数的是一个指向由具有13个整数元素的行向量组成的一维数组的指针。因此如果将 daytab作为变元传递给函数f，那么f的说明应写成如下形式：

```
f(int daytab[2][13])  { ... }
```

也可以写为：

```
f(int daytab[ ][13])  { ... }
```

因为参数与数组的行数无关，因而它还能写成：

```
f(int (*daytab)[13])  { ... }
```

它表明参数是一个指向具有13个整数元素的一维数组的指针。因为方括号 [ ]的优先级高于*的优先级，所以上述说明中必须使用圆括号。否则，说明语句

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

说明了一个具有13个指向整数类型对象的指针元素的一维数组。一般而言，除第一维可以不指定大小外，其余各维都必须明确指定大小。

我们将在第5.12节针对复杂的说明做更深入的讨论。

练习5-8　函数day_of_year和month_day中没有进行错误检查，请纠正之。

5.8　指针数组的初始化

下面研究如何实现函数 month_name(n)，它返回一个指向第 n个月名字的字符串的指针。这里采用内部静态数组作为数据结构比较理想。 month_name函数中包含一个私有的字符串数组，并且当它被调用时返回一个指向正确的字符串位置的指针。本节说明如何初始化该名字数组。

指针数组的初始化语法和早先见过的初始化相似：

```
/* month_name: 返回第n个月份的名字*/
char  * month_name(int n)
{
    static  char *name[ ] = {
            "Illegal month",
            "January", "February", "March",
            "April", "May", "June",
            "July", "August", "September",
            "October", "November", "December"
     };

     return  ( n < 1 | | n > 12 ) ? name[0] : name[n];
}
```

其中name的说明与排序例子中 lineptr的说明相同，它是一个元素为字符指针类型的一维数组。

name数组的初始化通过一个字符串列表实现，列表中的每个字符串赋值给数组的相应元素。编译程序在内存中的某个地方为第 i个字符串分配空间，然后将指向该存储空间的指针存放到name[i]中。由于上述说明中没有指明数组 name的大小，因而编译程序编译时将对初始化部分中字符串进行统计并将之作为该数组的大小填写到有关的符号表中。

5.9 指针与多维数组

C语言的初学者有时会混淆二维数组与指针数组之间的区别，例如上面例子中的 name。假如有如下两个定义：

```
int  a[10][20];
int  *b[10];
```

那么a[3][4]与b[3][4]在语法上都是对一个整数的合法引用。但 a是一个真正的二维数组：它分配了200个整数大小的存储空间，并且用常规的矩阵下标计算公式 20×row+col来计算元素a[row][col]的位置。但对b而言，该定义仅仅分配了 10个指针而且没有对它们进行初始化，它们的初始化必须以静态的方式或用代码显式地进行。假定 b的每个元素都指向一个具有 20个元素的数组，那么编译程序就要为其分配 200个整数加上10个指针的存储空间。指针数组的重要优点在于数组的每一行可以有不同的长度，即并不是 b的每个元素都指向一个具有 20个元素的向量，有些元素可以指向具有两个元素的向量，有些可以指向具有50个元素的向量，而有些根本就不指向任何向量。

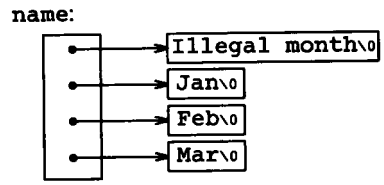

图 5-9

尽管以上面是就整数来讨论这个问题，但到目前为止，指针数组最频繁的用处是存放具有不同长度的字符串，就像在函数month_name中一样。下面是指针数组的说明和图形化描述（见图5-9）：

```
char  *name[ ] = {"Illegal month", "Jan", "Feb", "Mar"};
```

而下面是二维数组的说明和图形化描述（见图 5-10）：

```
char  aname[ ][15] = {" Illegal month", "Jan", "Feb", "Mar" };
```

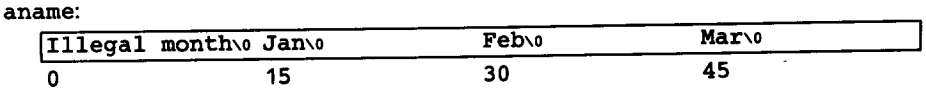

图 5-10

练习5-9 用指针代替数组下标改写函数day_of_year和month_day。

5.10 命令行变元

在支持C语言的环境中，可以在程序开始执行时将命令行变元或参数传递给程序。调用主函数时，它可以带有两个变元。第一个变元 (习惯上称做argc，用于变元计数)的值为程序执行时命令行中变元的数目，第二个变元 (称为argv，用于变元向量)是一个指向字符串数组的指针，其中每个字符串对应一个变元。我们通常用多级指针处理这些字符串。

最简单的例子是回应程序 echo，它将命令行变元回送到屏幕上的一行中，其中命令行中各变元之间用空格隔开。即，命令

```
echo  hello, world
```

将打印如下输出

```
hello, world
```

按照C语言的规定，argv[0]的值为调用相应程序的命令名，因此 argc的值至少为1。如果argc的值为1，那么命令名后面没有命令行变元。在上面的例子中， argc的值为3，argv[0]、argv[1]和argv[2]的值分别为 echo、hello和world。第一个可选变元为 argv[1]，而最后一个可选变元为argv[argc-1]，此外，C语言的标准要求argv[argc]的值在实现时必须为一空指针（见图 5-11）。

回应程序echo的第1个版本将argv看做一个字符串指针数组：

```
#include <stdio.h>

/* 回应程序命令行变元；第1个版本 */
main(int argc, char *argv[ ])
{
    int  i;

    for  (i = 1; i < argc; i++)
          printf("%s%s", argv[i], (i < argc -1) ? " " : " ");
    printf("\n");
    return  0;
}
```

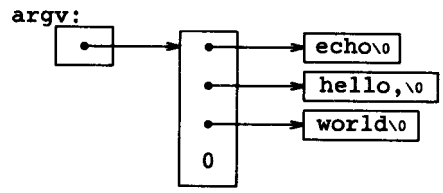

图 5-11

因为argv是一个指向指针数组的指针，所以可以通过指针而不是数组下标来处理命令行变元。回应程序的第 2个版本是在对 argv进行加一运算同时对 argc进行减一运算的基础上实现的，其中argv是一个指向字符指针的指针：

```
#include <stdio.h>

/* 回应程序命令行变元；第2个版本 */
main(int argc, char *argv[ ])
{
    while (--argc > 0)
          printf("%s%s", argv[i], (i < argc -1) ? " " : "");
    printf("\n");
    return  0;
}
```

因为argv是一个指向由变元组成的字符串数组的开始位置的指针，所以加一运算 (++argv)将使它指向原来的argv[1]而不是argv[0]。每执行一次加一运算就使 argv指向下一个变元，* argv即指向那个变元的指针。与此同时， argc执行减一运算，当它变成0时，就完成了所有变元的打印。

也可以将printf语句写成如下形式：

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

该语句表明printf的格式化变元也可以是表达式。

作为第二个例子，我们强化4.1节中模式查找程序的功能。回忆4.1节，我们将待搜索的模式串嵌到了程序中，这种解决方法显然不能令人满意。下面效仿UNIX程序grep的实现方法来改写模式查找程序，使得待匹配的模式串由命令行的第一个变元指定。

```
#include <stdio.h>
#include <string.h>
#define  MAXLINE  1000

int  getline(char *line, int max);

/* find: 打印与第一个变元指定的模式串匹配的行 */
main(int argc, char *argv[ ])
{
    char  line[MAXLINE];
    int  found = 0;

    if (argc != 2)
       printf("Usage: find pattern\n");
    else
       while (getline(line, MAXLINE) > 0)
             if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
             }
    return  found;
}
```

标准库函数strstr(s, t)用于返回子字符串t在字符串s中第一次出现处的指针；如果子字符串t未在字符串s中出现过，那么返回值为空指针。它在头文件<string.h>中说明。

为进一步解释指针结构，下面精心设计模式查找程序。假定允许程序有两个可选变元。其中一个表示“打印除匹配模式串之外的所有文本行”，另一个表示“每个被打印文本行前面都有相应的行号”。

对UNIX系统上的C程序而言，一个公共的约定是以负号开头的变元表示引入了一个可选标志或参数。如果用-x (x代表except)表示打印所有与模式串不匹配的文本行，用-n (n代表number)表示打印行号，那么如下命令：

```
find  -x -n 模式
```

将打印每一个与模式串不匹配的行，其中每个被打印行之前都有相应的行号。

可选变元应能以任意顺序排列，同时程序的其余部分应独立于命令行中所出现的变元的数目。此外，如果可选变元能够组合，那么将会给用户带来方便，正如如下命令所示：

```
find  -nx 模式
```

改写后的模式查找程序如下：

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

```
#include <string.h>
#define  MAXLINE  1000

int  getline(char * line, int max);

/* find: 打印所有与第一个变元指定的模式串相匹配的行 */
main(int argc, char *argv[ ])
{
    char  line[MAXLINE];
    long  lineno = 0;
    int  c, except = 0, number = 0, found = 0;

    while (- -argc > 0 && (* ++argv)[0] = = '-')
          while (c = *++argv[0])
            switch (0)  {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc !=1)
      printf("Usage: find -x -n pattern\n");
    else
       while (getline(line, MAXLINE) > 0) {
             lineno++;
              if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%d:", lineno);
                printf("%s", line);
                found++;
           }
       }
    return  found;
}
```

在处理每一个可选变元之前，argc进行减一运算，而argv则进行加一运算。循环语句结束时，如果不出现错误，argc的值是还没有处理的变元数，而argv指向这些未处理变元中的第一个变元。因此这时argc的值应为1，而*argv应指向模式串。注意 *++argv是一个指向变元字符串的指针，因此(*++argv)[0]是它的第一个字符(另一个有效形式是 **++argv)。因为[ ]与运算分量的结合性比*和++的高，所以在上述表达式中必须使用圆括号，否则编译程序将会把该表达式当做

*++(argv[0])。事实上，内循环中使用了表达式 *++argv[0]，其目的是遍历一个特定的变元串。在内循环中，表达式*++argv[0]对指针argv[0]进行加一运算。

很少见到有人使用比这些更复杂的指针表达式，在遇到这类情况时，将它们分为两步或三步来理解将会更直观。

练习5-10　编写程序expr，计算命令行中逆波兰表达式的值，其中每个运算符或运算分量用一个变元表示。例如命令

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

将计算表达式2×(3 + 4)的值。

练习5-11　修改程序entab与detab(它们是第1章练习中所编写的函数)，使它们用变元来接收一个由制表符停止位组成的列表。如果程序以没有带变元的形式执行，则使用默认的制表符停止位设置。

练习 5-12　对程序entab和detab的功能做一些扩充，使得如下命令：

```
entab  -m +n
```

表示制表符从第m列开始，每隔n列停止。在程序不带变元的情况下，为程序设计合适的 (对用户而言)缺省行为。

练习 5-13　编写程序tail，打印其输入的最后n行。缺省情况下，n的值为10，但可通过一个可选变元改变它，所以命令

```
tail  -n
```

将打印它输入的最后 n行。不管输入或 n的值如何的不合理，该程序都应能正常运行。使该程序能充分利用存储空间，输入行的存储方式应该同 5.6节的排序程序使用的存储方式一样，而不采用固定大小的二维数组。

5.11　指向函数的指针

在C语言中，函数本身不是变量，但可以定义指向函数的指针，这种指针可以被赋值、存放于数组之中、传递给函数及作为函数返回值等等。下面将通过修改本章前面的排序函数，使其在给定可选变元- n的情况下按数值大小而不是字典顺序对输入行进行排序，来说明指向函数的指针的用法。

排序程序通常由如下三部分组成：判断一对对象次序的比较操作、将对象次序颠倒的交换操作及一个用于比较和交换对象直到所有对象都按次序排好的排序算法。由于排序算法独立于比较和交换操作，因此通过在排序算法中调用不同的比较和交换函数，便可以以不同的标准进行排序。这就是新版本排序函数所采用的方法。

如前所述，函数strcmp按字典顺序比较两个输入行，在此需要一个在数值的基础上比较两个输入行并且返回和 strcmp同样的比较结果的函数 numcmp。这些函数在main之前说明，并且将一个指向它们之一的指针传递给 qsort函数。我们将不准备在变元的错误处理上花太多的笔墨，而将重点放在指向函数的指针这个主要问题上。

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

#define  MAXLINES  5000     /* 待排序的最大行数 */
char  *lineptr[MAXLINES];   /* 指向文本行的指针 */

int  readlines(char *lintptr[ ], int nlines);
void  writelines(char *lineptr[ ], int nlines);

void  qsort(void *lineptr[ ], int left, int right,
         int (*comp) (void *, void *));
int  numcmp(char *, char *);

/* 对输入的文本行进行排序 */
main(int argc, char *argv[ ])
{
   int  nlines;         /* 读入的输入行数 */
   int  numeric = 0;    /* 若进行数值排序值为1 */

   if (argc > 1 && strcmp(argv[1], "-n ") = = 0)
       numeric = 1;
   if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
      qsort((void **) lineptr, 0, nlines-1,
        (int (*)(void *, void *)) (numeric ? numcmp : strcmp));
      writelines(lineptr, nlines);
      return  0;
   }else {
       printf("input too big to sort\n");
       return  1;
   }
}
```

在调用函数qsort的语句中，strcmp和numcmp是函数的地址。因为它们是函数，所以它们前面不需要取地址运算符&，同样数组名前也不需要&。

改写后的qsort函数能够对任何数据类型的对象（而不仅仅是对字符串）进行排序。正如函数qsort的原型所示，它的参数表由一个指针数组、两个整数和一个具有两个指针变元的函数组成。通用指针类型void*用做指针变元。由于任何类型的指针都可以转换为 void*类型并且当它又转换回原来的类型时不会丢失信息，所以可以通过将变元转换为 void*类型的方式来调用qsort函数。由于函数变元具有两个void*类型的变元，因而在转换时将把比较函数的变元转换成 void*类型。这种转换通常不会影响实际表达，但要确保编译程序不会出错。

```
/* qsort: 以递增顺序对v[left] ... v[right] 进行排序 */
void  qsort(void *v[ ], int left, int right, int (*comp) (void *, void *))
{
   int  i, last;
   void  swap(void *v[ ], int, int);
```

```
    if (left >= right)    /* 如果数组元素个数小于2，则什么都不做 */
      return;
    swap(v, left, (left + right) / 2);
    last = left;
    for (i = left + 1; i <= right; i++)
       if ((*comp) (v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last - 1, comp);
    qsort(v, last+1, right, comp);
}
```

现在来仔细研究这些说明。qsort函数的第四个参数为：

```
int  (*comp) (void *, void *)
```

它表示comp是一个指向函数的指针，该函数具有两个类型为 void*的变元，其返回值类型为 int。

在if语句

```
if  ((*comp) (v[i], v[left]) < 0)
```

中使用的comp和其对应的说明是一致的：comp是一个指向函数的指针，*comp是一个函数，而

```
(*comp) (v[i], v[left])
```

是对这个函数的调用。这里必须使用圆括号才能使各成份正确结合，否则

```
int  *comp(void *, void *)      /* 错误的写法 */
```

表示comp是一个返回一个指向整数的指针的函数，这显然有很大的区别。

前面已经描述过比较两个字符串的函数 strcmp。下面给出函数 numcmp，它通过调用 atof来计算字符串对应的数值，然后在此基础上比较两个字符串：

```
#include <stdio.h>

/* numcmp: 以数值序比较字符串s1和s2 */
int  numcmp(char *s1, char *s2)
{
    double  v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if  (v1 < v2)
        return  -1 ;
    else if (v1 > v2)
        return  1;
    else
        return  0;
}
```

这个交换两个指针的swap函数和本章前面所描述的swap函数相同，只不过它将变元 v[ ]说明为void *类型。

```
void  swap(void *v [ ], int i, int j)
```

```
{
    void  *temp;

    temp = v[i]'
    v[i] = v[j];
    v[j] = temp;
}
```

许多其他的可选项可增加到该排序程序中，其中的一部分留给读者完成。

练习5-14　修改排序程序，使它能处理 -r 标记，该标记的意思是以逆序(递减)方式进行排序。确保-r能和-n组合在一起使用。

练习5-15　增加可选项-f 使得在排序过程中不考虑字母的大小写的区别，例如，在比较 a和A时认为它们相等。

练习5-16　增加可选项-d（目录顺序），它的意思是只对字母、数字和空格进行比较。确保它能和 -f 组合在一起使用。

练习5-17　增加字段处理能力，以使可以根据行内的字段来排序，每个字段依照一个独立的可选项集合排序。

5.12　复杂说明

C语言有时因说明的语法问题而受到人们的批评，特别是那些涉及到指向函数的指针的语法。C语言的语法试图使说明和使用相一致，对于简单的情况它很有效，但对于复杂的情况它有可能令人混淆，因为说明不能从左往右读，而且使用了太多的圆括号。如下所示的两个说明：

```
int  * f( );        /* f: 返回一个指向整数类型对象的指针的函数 */
```

和：

```
int  (*pf) ( );     /* pf: 返回一个指向函数的指针，该函数返回一个整数 */
```

所表示的含义的差别说明：* 是一个前缀运算符且其优先级低于( )，所以后一个说明中必须使用圆括号以使运算分量能够正确地结合。

尽管真正复杂的说明实际上很少会出现，但对于读者而言，懂得怎么理解、甚至怎么建立使用这些复杂的说明是很重要的。使用 typedef在几步之内就可合成说明，6.7节将讨论这个好方法。作为一种变通的方法，本节将讨论一对用于将一个将正确的 C说明转换为文字描述和完成相反转换的程序。其中，文字描述应该从左往右读。

第一个程序dcl要复杂一些。它将C语言的说明转换为文字描述，正如下例所示：

```
char  **argv
    argv: 指向字符指针的指针
int  (*daytab) [13]
    daytab: 指向由13个整数类型元素组成的一维数组的指针
int  *daytab[13]
     daytab: 由13 个指向整数类型对象的指针组成的一维数组
void  *comp( )
```

comp：返回值为指向通用类型的指针的函数

```
void  (*comp) ( )
```

comp：指向返回值为通用类型的函数的指针

```
char  (*(*x( )) [ ]) ( )
```

x：返回值为指向一维数组的指针的函数，该一维数组由指向返回字符类型的函数的指针组成。

```
char  (*(*x[3])( )) [5]
```

x：由3个指向函数的指针组成的一维数组，该函数返回指向由5个字符组成的一维数组的指针。

dcl程序基于说明符的语法，附录 A.8.5节对该语法做了精确的描述，下面是其简化的语法形式：

说明符：　可选的*序列 直接说明符

直接说明符：名字

(说明符)

直接说明符( )

直接说明符[可选的大小]

简而言之，说明符即前面也许带有符号 *的直接说明符。直接说明符可以是名字、由一对圆括号括住的说明符、后面跟有一对圆括号的直接说明符或后面跟有由方括号括住可选大小的直接说明符。

该语法可用来分析C语言的说明。例如，研究如下说明符：

```
(*pfa[ ]) ( )
```

在dcl程序分析该说明时，pfa将被识别出是一个名字，从而它是一个直接说明符。于是 pfa[ ]也是一个直接说明符。接着 *pfa[ ]被识别出是一个说明符，因而 (*pfa[ ])是一个直接说明符。可以用如图5-12所示的分析树来说明分析的过程。

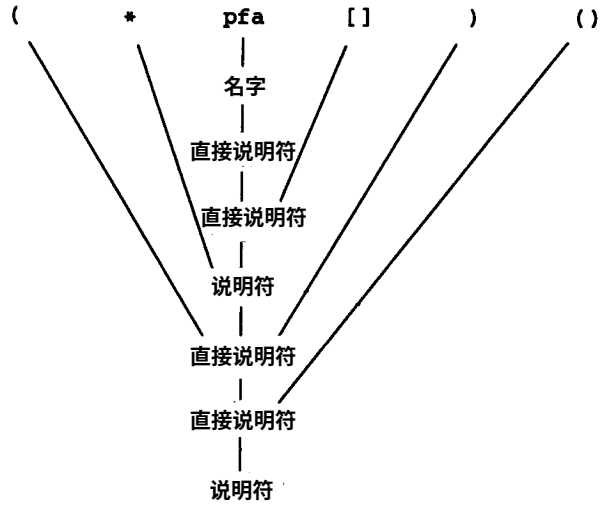

图 5-12

dcl程序的核心是一对函数 dcl与dirdcl，它们依照说明符的语法来分析说明。因为该语法是

递归定义的，所以在识别一个说明的组成部分时这两个函数将相互递归调用，我们称该程序是一个递归下降分析程序。

```
/*dcl:  分析一个说明符*/
void  dcl(void)
{
   int  ns;

   for (ns = 0; gettoken( ) = = '*';  )    /* 统计字符'*'的个数*/
       ns++;
   dirdcl( );
   while (ns- - > 0)
       strcat(out, "pointer to");
}

/* dirdcl: 分析一个直接说明符*/
void  dirdcl(void)
{
int type

   if (tokentype = = '(' ) {   /* 该直接说明符的形式为：(说明符) */
       dcl( );
       if (tokentype != ')' )
           printf("error: missing ) \n");
   } else if (tokentype = = NAME)    /* 变量名*/
           strcpy(name, token);
   else
       printf("error: expected name or (dcl)\n");
   while ((type = gettoken( )) = = PARENS || type = = BRAKCETS)
         if (type = = PARENS)
             strcat(out, "function returning");
         else {
             strcat(out, "  array");
             strcat (out,token);
             strcat(out, " of");
   }
}
```

该程序旨在说明问题，并不追求完美，所以 dcl程序在功能上有许多明显的缺陷。它只处理诸如char或int这样简单的数据类型，而不处理函数中的变元类型或诸如 const这样的限定符。它不能辨认不合逻辑的空白。由于没有进行太多的错误恢复处理，因此它也不能辨认无意义的说明。这些方面的改进留做练习。

下面是相应的全局变量和主程序：

```
#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  tokentype;                    /* 最后一个单词的类型 */
char  token[MAXTOKEN];             /* 最后一个单词字符串 */
char  name[MAXTOKEN];              /* 标识符名 */
char  datatype[MAXTOKEN];          /* 数据类型为char、int等 */
char  out[1000];                   /* 输出字符串  */

main( )    /* 将说明转换为文字描述*/
{
    while (gettoken( ) != EOF)  {     /* 该行的第一个单词是数据类型*/
           strcpy(datatype, token);
           out[0] = '\0';
           dcl( );                    /* 分析该行的其余部分*/
           if (tokentype != '\n')
               printf("syntax error\n");
           printf("%s: %s %s\n", name, out, datatype);
   }
   return  0;
}
```

函数gettoken**跳过空格与制表符，然后查找输入中的下一个单词，“ 单词 ”包括名字、圆括号、可能包含数字的中括号或者是任何其他单个字符。**

```
int  gettoken(void)    /* 返回下一个单词*/
{
    int  c, getch(void);
    void  ungetch(int);
    char  *p = token;

    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;
}
```

函数getch和ungetch曾在第4章讨论过。

另一个方向的转换更容易，尤其是在不考虑生成多余的圆括号的情况下。程序 undcl把诸如“x是一个返回值为指向一维数组的指针的函数，该一维数组由指向返回值为字符类型的函数的指针组成”且用如下形式表示的文字描述：

```
x ( ) * [ ] * ( ) char
```

转换为：

```
char (* (* x( ) ) [ ] ) ( )
```

由于输入的语法做了简化，所以可以重用上面定义的 gettoken函数。undcl和dcl使用了相同的外部变量。

```
/* undcl: 将文字描述转换为说明 */
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 = = '*') {
              sprintf(temp, "(*%s)", out);
              strcpy(out, temp);
          } else if (type = = NAME) {
              sprintf(temp, "%s %s", token, out);
              strcpy(out, tmep);
          } else
              printf("invalid input at %s\n", token);
      printf("%s\n", out);
   }
   return  0;
}
```

练习5-18　改写dcl，使它能够处理输入错误。

练习5-19　修改undcl，使得它在把文字描述转换为说明时不会生多余的圆括号。

练习5-20　扩展程序dcl的功能，使它能处理具有函数变元类型、诸如 const这样的限定符等其他成分的说明。

第6章 结 构

结构是一个或多个变量的集合，这些变量可能属于不同的类型，为了处理方便而组织在一个名字下（在有些语言中结构被称之为“记录”，如Pascal语言）。由于结构用于把一组相关变量看做一个单元而不是各自独立的实体，故通常被用来组织复杂的数据，特别是在大型的程序中。

用来描述结构的一个传统例子是工资记录：每个员工是由一组属性来描述的，如姓名、地址、社会保险号、工资等。其中有些属性可以定义为结构类型，如：姓名可以分成几个部分，地址甚至工资也可能出现相同的情况。在 C语言中，关于结构更典型的例子来自于图形：点由一对坐标定义，矩形由两个点定义，等等。

ANSI C标准中的主要变化是定义了对结构的赋值操作——结构可以拷贝、赋值、传送给函数并被函数返回。很久以来大多数编译程序就支持这一点，只是现在对其性质进行了更精确的定义。自动结构和数组现在也允许初始化。

6.1 结构的基本知识

下面建立一些适用于绘图的结构。最基本的对象是平面上的点，假设用 x与y坐标表示它，且x、y坐标都取整数（见图6-1）。这两个坐标成分可放在结构中，其说明如下：

```
struct  point  {
    int  x;
    int  y;
};
```

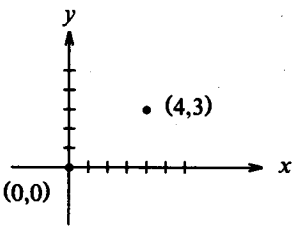

图 6-1

关键字struct用于引入结构说明，它由包含在花括号之内的一系列说明组成。struct后跟的名字是可选的，它是结构的标记（这里为 point）。结构标记用于命名结构，它可以作为花括号内说明表的简写。

在结构中命名的变量称为成员。结构成员或标记和普通变量（即非成员）可以采用相同的名字，不会产生冲突，因为通过上下文分析总可以对其进行区分。另外，不同结构中的成员可以采用相同的名字，尽管从程序风格上说，通常只有密切相关的对象才会使用相同的名字。

一个结构说明定义一种类型。在标志结构成员表结束的右花括号之后，可以像其他基本类型一样跟随一个变量表。例如：

```
struct  {...}  x, y, z;
```

在语法上与

```
int  x, y, z;
```

相似，这两个说明都用于把x、y与z说明为指名类型的变量并且为它们分配存储单元。

若在结构说明之后没有定义变量表，则不需要为之分配存储单元，它仅仅描述了一个结构的模板（或形状）。但是如果结构说明带有标记，那么在以后定义结构实例时可以使用它。例如，对于上面给出的关于point的结构说明，语句

```
struct  point  pt;
```

定义了一个point结构类型的变量pt。结构可以在说明时进行初始化，方法是在其定义后加上初始化符表，初始化符表中与各个成员对应的必须是常量表达式：

```
struct  point  maxpt = { 320, 200 };
```

自动结构也可以通过赋值或调用返回结构类型的函数进行初始化。

在表达式中可以通过如下形式来引用一特定结构中的成员：

结构名.成员

结构成员运算符 . 把结构名与成员名连接起来。例如，可用如下语句打印点 pt的坐标：

```
printf("%d, %d", pt.x, pt.y);
```

也可用如下代码计算原点(0, 0)到pt的距离：

```
double  dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

结构可以嵌套。我们可用对角线上的两个点来定义矩形（见图6-2），相应结构定义如下：

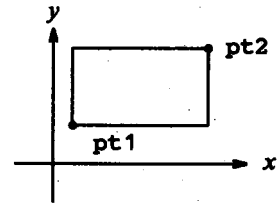

图 6-2

```
struct  rect {
    struct  point  pt1;
    struct  point  pt2;
};
```

所定义的结构rect包含了两个point结构类型的成员。如果如下说明screen变量：

```
struct  rect  screen;
```

那么可以用

```
screen.pt1.x
```

访问screen的成员pt1的x坐标。

6.2 结构与函数

对结构的合法操作只有拷贝、作为一个单元对其赋值、通过 &取其地址以及访问结构成员这几种。拷贝和赋值包括对函数传递变元及从函数返回值。结构之间不可以比较。但是可以用一组成员常量初始化结构，另外自动结构也可以通过赋值来初始化。

为了加深对结构的理解，下面编写几个用于对点和矩形进行操作的函数。至少有三种可能的方法：一是分别传送各个结构成员，二是传送整个结构，三是传送指向结构的指针。这三种

方法各有利弊。

首先给出的函数是函数makepoint，它需要两个整数，并返回一个point结构：

```
/* makepoint: 通过x、y值确定一个点 */
struct  point  makepoint(int x, int y)
{
struct  point  temp;

   temp.x = x;
   temp.y = y;
   return  temp;
}
```

注意，变元和结构成员同名不会引起冲突，事实上，重名强调了两者之间的关系。

现在可以用makepoint动态初始化任意结构，也可以向函数提供结构类型变元：

```
struct  rect  screen;
struct  point  middle;
struct  point  makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x) / 2,
                   (screen.pt1.y + screen.pt2.y) / 2);
```

下面通过一系列的函数对点进行算术运算。例如：

```
/* addpoint: 将两个点相加 */
struct  point  addpoint(struct point p1, struct point p2)
{
   p1.x += p2.x;
   p1.y += p2.y;
   return  p1;
}
```

函数的变元和返回值都是结构类型的。之所以直接将相加所得的结果赋给 p1，而没有使用显式临时变量，是为了说明结构类型参数和其他类型参数一样按值传递。

下面再看另一个例子，函数 ptinrect用于判断一个点是否在给定矩形内。我们采用这样的约定，矩形包括其左边和底边，但不包括它的顶边和右边。

```
/* ptinrect: 如果点p在矩形r 内，函数返回1，否则返回0 */
int  ptinrect(struct point p, struct rect r)
{
   return  p.x >= r.pt1.x && p.x < r.pt2.x
      && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

这里假设矩形是用标准形式表示的，其中 pt1的坐标小于pt2的坐标。下面这个函数返回一个规范形式的矩形。

```
#define  min(a, b)  ((a) < (b) ? (a) : (b))
#define  max(a, b)  ((a) > (b) ? (a) : (b))
```

```
/* canonrect :   将矩形坐标规范化 */
struct  rect  canonrect(struct rect r)
{
      struct  rect  temp;
      temp.pt1.x = min(r.pt1.x, r.pt2.x);
      temp.pt1.y = min(r.pt1.y, r.pt2.y);
      temp.pt2.x = max(r.pt1.x, r.pt2.x);
      temp.pt2.y = max(r.pt1.y, r.pt2.y);
      return  temp ;
}
```

如果传送给函数的结构很大，使用指针时的效率通常要比拷贝结构时高。结构指针类似普通变量的指针。说明语句：

```
struct  point  *pp;
```

将pp定义成一个指向struct point类型的指针。如果pp指向一个point结构，那么* pp就是该结构，而(*pp).x和(*pp).y就是结构成员。若要使用pp，例如可以写出如下的语句：

```
struct  point  origin , *pp;
pp = &origin;
printf ("origin is (%d, %d) \n ", (*pp).x, (*pp).y);
```

(*pp).x中的圆括号是必须的，因为结构成员运算符 . 的优先级比*高。表达式*pp.x的含义是*(pp.x)，因为x不是指针，所以该表达式非法。

因为结构指针的使用频率很高，于是出现了一种简写方式作为可选的标记法。如果 p是结构指针，则

```
p->结构成员
```

表示特定的结构成员。(运算符->是由减号后紧跟>组成的。) 因此也可以这样改写：

```
  printf("origin is (%d, %d) \n ", pp->x, pp->y);
```

. 和->都是从左到右结合的运算符，所以如果有说明

```
struct  rect  r,  *rp = &r;
```

那么以下四个表达式等价：

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

结构成员运算符（. 和->)、函数调用的（）以及下标的[ ]的优先级最高，从而结合最紧密。例如，若给定结构说明

```
struct  {
   int  len;
   char  *str;
}  *p;
```

那么表达式

```
++p->len
```

增加的是len的值而不是p的值。因为->的优先级比++高，所以这个表达式等价于

```
++(p->len)
```

可以使用括号改变结合次序。例如：

```
(++p)->len
```

先进行p加1操作，再对len操作。而在

```
(p++) ->len
```

中，先对len操作，再将p加1（该表达式中括号可以省略）。

同理，*p->str取的是str所指的值；*p->str++先取str所指的值，再将 str加1（与*s++一样）；(*p->str)++将str所指的值加1；*p++->str先取str所指的值，再将p加1。

6.3 结构数组

考虑编写一个统计输入中各个C关键字出现次数的程序。我们需要用一个字符串数组存放关键字名，一个整型数组存放相应关键字的出现次数。一种方法是使用两个独立的数组 keyword和keycount来完成这件事，如下所示：

```
char  *keyword[NKEYS];
int  keycount[NKEYS];
```

注意到这两个数组是并行的，从而可以采用另一种不同的组织方式，即采用结构数组。每个关键字项是一对变量：

```
char  *word;
int  count;
```

于是有一个变量对数组。结构说明

```
struct  key  {
        char  *word;
        int  count;
} keytab[NKEYS];
```

说明了结构类型key，定义了这个类型的结构数组 keytab，并为其分配存储空间。数组 keytab的每个元素都是一个结构。也可以采用如下写法：

```
struct  key  {
        char  *word;
        int  count;
};

struct  key  keytab[NKEYS];
```

因为结构keytab包含一个固定的名字集合，所以在定义时很容易将其说明为外部变量并对其进行初始化。结构初始化与前面所述类似 —— 定义后面跟着括在圆括号中的初始化符表：

```
struct  key  {
   char  *word;
   int  count;
} keytab[ ] = {
   "auto", 0,
```

```
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* … */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
} ;
```

初始化符成对的列出，与结构成员相对应。更精确的做法是，将每一行（即每一结构）的初始化符括在花括号内，如下所示：

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
…
```

若初始化符是简单变量或字符串，并都存在，则内部的花括号可以省略。通常，如果初始化符存在并且方括号[]中为空，那么要通过计算数组keytab中的元素数目来决定数组元素的个数。

在统计关键字出现次数的程序中，首先定义 keytab。在主程序中通过反复调用函数 getword 处理输入，并且每次取一个关键字。通过二分查找函数（见第 3章）查找每一个关键字。注意，关键字列表必须按升序存储在表中。

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

#define  MAXWORD  100

int  getword ( char *, int );
int  binsearch ( char *, struct key *, int );

/* 统计C的关键字 */
main ( )
{
   int  n;
   char  word[MAXWORD];

   while  ( getword (word, MAXWORD ) != EOF )
      if  ( isalpha ( word[0] ))
            if  ( ( n = binsearch ( word, keytab, NKEYS ) ) >= 0 )
                  keytab[n].count++;
   for  ( n = 0;  n < NKEYS;  n++ )
         if  ( keytab[n].count > 0 )
               printf ( "%4d  %s\n ",
                        keytab[n].count, keytab[n].word );
   return  0;
```



```
}

/* binsearch: 在tab[0]到tab[n-1]中寻找匹配的关键字*/
int  binsearch ( char *word, struct key tab[ ], int n )
{
   int  cond;
   int  low,  high,  mid;

   low = 0;
   high = n-1;
   while  ( low <= high )  {
          mid = ( low + high ) / 2;
          if  ( ( cond = strcmp ( word, tab[mid].word ) ) < 0 )
             high = mid - 1;
          else  if  ( cond > 0 )
             low = mid + 1;
          else
             return  mid;
   }
   return  -1;
}
```

稍后将给出函数getword，现在只需知道它每被调用一次，就读入一个字并赋给命名为数组的第一个变元。

NKEYS代表keytab中关键字的个数。尽管可以通过手工计算，但交给机器做会更简单更安全，尤其是在列表会变更的情况下。一种解决办法是在初始化符表的结尾处加上一个空指针，然后循环遍历keytab直到读到尾部空指针。

但实际上并不需要这样做，因为数组的大小在编译时完全确定，等于一个元素的大小乘以元素的个数。所以在这个程序中，元素的个数等于：

```
keytab的大小 / struct key的大小
```

C提供了一个编译时的一元运算符，称为sizeof，可用来计算任一对象的大小。表达式

```
sizeof  对象
```

和

```
sizeof ( 类型名 )
```

返回一个整型值，它等于指定对象或类型所占存储空间的字节数。（严格说来，sizeof的返回值是无符号整数值size_t，它的类型在头文件<stddef.h>中定义。）上述对象可以是变量、数组或是结构。而类型可以是基本类型如整型、双精度浮点等类型，也可以是派生类型如结构、指针等类型。

在这个程序中，关键字的个数等于数组的大小除以单个元素的大小。下面的 #define语句使用了这种方法来设置NKEYS的值：

```
#define  NKEYS ( sizeof keytab / sizeof ( struct key ) )
```

另一种方法是用一个特定元素的大小去除数组的大小：

```
#define  NKEYS ( sizeof keytab / sizeof keytab[0] )
```

这样定义有一个好处，那就是即使类型改变，也不需要做改动。

在条件编译语句#if中不可以使用sizeof，因为预处理程序不对类型名字进行分析。而预处理程序并不计算#define语句中的表达式，所以在#define中使用sizeof是合法的。

下面讨论函数getword。我们给出了更一般性的getword，虽然功能已超出此处程序的要求，但它一点也不复杂。getword从输入中读入下一个字，可以是字符串或是空格符。函数返回值是字的第一个字符，或是文件结束标记EOF，或是字符本身（如果该字符不是字母字符）。

```
/* getword:  从输入中读入下一个字或字符 */
int  getword ( char *word, int lim )
{
   int  c,  getch(void);
   void  ungetch(int);
   char  *w = word;

   while  (isspace(c = getch( )))
      ;
   if  (c != EOF)
      *w++ = c;
   f  (!isalpha(c))  {
      *w = '\0';
      return  c;
   }
   for  (  ;  --lim > 0;  w++)
      if  (!isalnum (*w = getch( )))  {
      ungetch(*w);
      break;
   }
   *w = '\0';
   return  word[0];
}
```

getword使用了第4章中的函数getch和ungetch。当字母数字标记的集合停止时，getword已读入远远不止一个字符。随后ungetch将字符放回到输入中等待下一次调用。getword使用isspace函数来跳过空格符，用isalpha来判断输入是否为字母，用isalnum来识别字母和数字。所有这些函数都在标准头文件<ctype.h>中有定义。

练习6-1　在getword中并没有考虑下划线、字符串常量、注解及预处理程序控制行。请编写一个考虑更完善的getword版本。

6.4　结构指针

为了说明有关指向结构数组的指针的一些问题，本节将重写关键字计数程序，这次是用指针来代替数组下标。

keytab的外部说明不需要改变，但main函数和binsearch则必须做一些修改。修改后的程序如下所示：

```
#include  <stdio.h>
#include  <ctype.h>
#include  <string.h>
#define  MAXWORD  100

int  getword(char *, int)
struct  key  *binsearch(char *, struct key *, int)

/* 计算C中关键字的出现次数；指针版本 */
main( )
{
    char  word[MAXWORD];
    struct  key  *p;

    while  (getword(word, MAXWORD) != EOF)
       if  (isalpha(word[0]))
          if  ((p = binsearch(word, keytab, NKEYS)) != NULL)
             p->count++;
    for  (p = keytab; p < keytab + NKEYS; p++)
       if  (p->count > 0)
          printf("%4d  %s\n", p->count, p->word);
    return  0;
}

/* binsearch: 在tab[0]到tab[n-1]中寻找与读入字相匹配的元素*/
struct  key  *binsearch(char  *word, struct  key  *tab,  int  n)
{
    int  cond;
    struct  key  *low = &tab[0];
    struct  key  *high = &tab[n];
    struct  key  *mid;

    while  (low < high)  {
       mid = low + (high - low) / 2;
       if  ((cond = strcmp(word, mid->word)) < 0)
          high = mid;
       else  if  (cond > 0)
          low = mid + 1;
       else
          return  mid;
    }
    return  NULL;
}
```

有几点需要注意。首先， binsearch**返回一个指向** struct key**的指针而不是一个整数，这在函数原型及** binsearch**中都有说明。如果** binsearch**找到与输入单词相匹配的数组元素，那么返回一个指向该数组元素的指针，否则返回** NULL**。**

第二点，由于要通过指针来访问 keytab**的结构元素。所以在** binsearch**中需要做较大的修改。**

low和high的初始化符现在指向表头和表尾后面一个元素的指针。

不能再简单地通过表达式

```
mid = (low + high) / 2  /* 错误 */
```

来计算中间元素的位置，因为对两个指针进行加法运算是非法的。然而指针的减法却是合法的，high - low的值就是数组元素的个数，故可以用

```
mid = low + (high - low) / 2
```

来将mid置为指向位于high和low正中间的元素。

最重要的变化是要对算法进行修改，以保证不会产生非法的指针，以及保证不访问数组范围之外的元素。问题是 &tab[-1]和&tab[n]均超出了数组tab的范围。前者是绝对非法的，而对后者进行对数组推延也是非法的。然而对数组结束之后第一个元素（即 &tab [n]）的指针运算可以正确运行，这可由语言的定义提供保证。

主程序中有这样的语句：

```
for  (p = keytab; p < keytab + NKEYS; p++)
```

如果p是指针结构，那么对 p的算术运算需考虑结构的大小，所以 p++在p的基础上加上正确的值，从而定位到结构数组的下一个元素，这样使得测试可以保证循环正确结束。

然而，千万不要以为一个结构的大小等于它各成员大小的和。因为对不同对象有不同的对齐需求，所以结构中也有可能出现无名字的“ 洞 ”。例如，假设字符占用一个字节，整数占用四个字节，那么结构

```
struct  {
   char  c;
   int  i;
};
```

则需要8个字节，而不是5个字节。sizeof运算符返回结构大小的正确值。

最后一点与程序的格式有关。当一个函数的返回值是复杂的类型时（如结构指针），例如：

```
struct  key  *binsearch(char *word,  struct key *tab,  int n)
```

函数名字不容易一眼就看清，也不容易使用文本编辑器进行查找。上述语句还可以这样改写：

```
struct  key  *
binsearch(char *word,  struct key *tab,  int n)
```

这纯属个人习惯问题，可以选择自己喜欢的方式并且保持自己的风格。

6.5 自引用结构

假若要处理一个更一般性的问题，即统计输入的各个单词的出现次数。因为预先不知道单词表，所以无法按通常的方法加以排序再使用二分查找。也不能每来一个单词就做一次线性查找，看它是否已存在；若这样的话，程序的执行将花费太长的时间（更确切地说，它的运行时间随输入单词数目而成二次方增长）。那么该如何组织这些数据来有效地处理这一串任意的单词呢？

一种解决方法是：通过将输入单词有序地放入正确位置，从而在任意时刻都保持已输入单

词的有序性。然而这不能简单地通过在线性数组中移动单词来达到目的，尽管是可行的，因为它显然会导致程序执行时间过长。我们将使用一种称做二叉树的数据结构。

每个不同的单词在该树中都是一个节点，每个节点包含：

一个指向该单词的指针
一个出现次数的计数值
一个指向左子树的指针
一个指向右子树的指针

任何节点最多拥有两个子树，当然可能只有一个或一个都没有。

节点的组织方法如下：每个节点的左子树只包含按字典排序小于该节点的那些节点，右子树只包含按字典排序大于该节点的节点。图 6-3是对句子：

now is the time for all good men to come to the aid of their party

中各单词按序插入后形成的树。

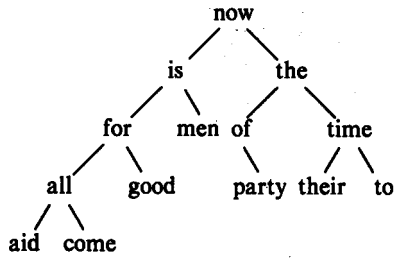

图 6-3

为查找一个新单词是否已在树中，可从根开始，比较新单词与该节点所存单词。若匹配，则得到肯定的答案。若新单词小于该节点中单词，则只在左子树中继续查找，否则在右子树中查找。如在搜寻方向上无子树，则新单词不在树中，事实上该空槽正是存放新加单词的正确位置。因为从任意节点出发的查找使用了它一个子树的查找过程，所以这个过程是递归的。相应地，对于插入和打印操作使用递归过程将是最自然的。

回到一个节点的描述，一个更方便的表示是使用具有以下四个成份的结构：

```
struct tnode {                  /* 树的节点 */
   char *word ;                 /* 指向单词的指针 */
   int count ;                  /* 单词出现的次数 */
   struct tnode *left ;         /* 左子树 */
   struct tnode *right ;        /* 右子树 */
} ;
```

节点的这个递归说明看上去可能有冒险性，但它的确是正确的。一个结构包含它本身的实例是非法的，但：

```
struct tnode *left ;
```

将左子树说明为指向tnode的指针，而不是tnode本身。

我们偶尔也需要自引用结构的一个变形：两个结构相互引用。解决的方法是：

```
struct t {
```

```
    …
    struct s *p;              /* p指向一个s */
} ;
struct s {
    …
    struct t *q               /* q指向一个t */
} ;
```

如果给出一些已编写过的函数（如 getword**）来支持它，那么整个程序的代码就显得惊人的短小。主例程通过**getword**读入单词，并用**addtree**将它们插入到树中。**

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* 单词出现频率计数 */
main()
{
     struct tnode *root;
     char word[MAXWORD];

     root = NULL;
     while (getword(word, MAXWORD) != EOF )
          if (isalpha(word[0]))
              root = addreee( root, word);
     treeprint(root);
     return 0;
}
```

函数addtree**是递归的。由主程序给出一个单词作为树的最顶层（即树的根）。在每一步中，新单词与已存在节点中的单词比较，随后通过对** addtree**的调用而经过左子树或右子树。该单词将最终与树中某节点匹配（在这种情况下计数值加** 1**），或遇上一个空指针（表明必须创建一个节点再加到树中去）。若生成了新节点，则**addtree**返回一个指向新节点的指针，并将该指针存入父节点。**

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree: 在p或p的下方加一个包含w的节点 */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if ( p == NULL) {        /* 新来一个单词 */
```



```
        p = talloc();       /* 创建一个新节点 */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;         /* 重复的单词 */
    else if (cond < 0)      /* 小于该节点则进入左子树 */
        p->left = addtree(p->left, w);
    else                    /* 大于该节点则进入右子树 */
        p->right = addtree(p->right, w);
   return p;
}
```

新节点的存储区通过调用函数talloc而获得，talloc函数返回一个指针，指向能容纳树节点的自由空间。函数strdup将该单词拷贝到一个隐藏位置（稍后将讨论这些函数）。计数值被初始化，两个子树也置为空。当增加新节点时，这部分代码只在树叶端执行。该程序中忽略了 strdup和talloc返回值的错误检查（这显然有点草率）。

treeprint按序打印树，在每个节点，它先打印左子树（小于该单词的所有单词），然后是该单词本身，最后是右子树（大于该单词的所有单词）。如果你对递归操作感到怀疑的话，不妨对上述树进行treeprint操作。

```
/* treeprint: 按序打印树p */
    void treeprint(struct tnode *p)
{
    if (p !=NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

有一点值得注意：若由于单词输入的次序不是随机的而使树变得不平衡，那么程序的运行时间将大大加长。最坏的情况是，若单词已经排好序，则程序进行代模拟线性查找。有一些广义二叉树不受这种最坏情况的影响，但在此对它们不予讨论。

在结束这个例子之前，作为题外话简短讨论一下有关存储分配程序的问题。显然，程序中只有一个存储分配程序是可取的，尽管它需要分配不同的对象。但是如果一个分配程序要处理一些请求，比如指向字符的指针和指向 struct tnode的指针，那么会引起两个问题。第一，它如何满足大多数机器的需求，使得特定类型的对象必须满足对齐限制（例如，整数常常必须分配在偶地址上）？第二点，用什么样的说明能解决这样的问题：一个分配程序必须能返回不同的指针？

对齐需求一般都容易满足，只要确保分配程序总返回满足所有对齐限制的指针，不过代价是浪费一点空间。第5章介绍的alloc函数不保证任何特殊的对齐，所以我们使用能保证对齐的标准库函数malloc。第8章将介绍一种实现malloc函数的方法。

对于任何注重类型检查的语言来说，像 malloc这样的函数的类型说明问题总是令人讨厌。在

C语言中，一个恰当的方法是说明 malloc返回一个指向void类型的指针，然后再显式地将指针强制转换成所需类型。 malloc和有关的其他函数都在标准头文件 <stdlib.h>中做了说明。因此可以把talloc函数写成：

```
#include <stdlib.h>

/* talloc: 创建一个tnode  */
struct tnode *talloc(void)
{
   return  (struct tnode *) malloc(sizeof(struct tnode));
}
```

strdup函数只是把由其变元给出的字符串拷贝到某个安全的位置，这可以通过调用 malloc函数来实现：

```
char *strdup(char s)   /* 拷贝s */
{
      char *p;

      p = (char *) malloc(strlen(s)+1);  /* +1是为了在结尾存入'\0' */
          if (p != NULL)
          strcpy(p, s);
      return p;
}
```

在没有可用空间时，malloc函数返回NULL值；strdup函数的返回值即malloc的返回值，且strdup将错误处理留给它的调用者。

由调用malloc函数而得到的存储区可通过调用 free来释放以重用，参见第7章和第8章。

练习6-2　编写一个程序，读入一段 C程序，并按字母表顺序分组打印变量名字，要求每一组内各变量名字的前 6个字符相同，其余字符可以不同。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。

练习6-3　编写一个交叉引用程序，打印文件中所有单词的列表，并且每个单词也都有一个列表，记录该单词所出现行的行号。对 the、and等嘈杂单词不予考虑。

练习6-4　编写一个程序，根据单词的出现频率降序打印出所输入的各个单词。每个单词前标有它的计数值。

6.6　查找表

为了对结构的方方面面做更深入的讨论，本节将编写一个查找表程序包的内部结构。这个代码很典型，可以在宏处理程序或编译程序的符号表管理例程中找到。例如，考虑 #define语句。当遇上如

```
#define  IN  1
```

之类的程序行时，就要把名字 IN和替换文本 1存入到某个表中。此后，当名字 IN出现在诸如

```
    state=IN;
```

等语句中时，则用1来替换IN。

以下两个函数用来管理名字和替换文本。install(s,t)函数用于将名字s和替换文本t记录到某个表中，其中s和t是字符串。lookup(s)函数在表中查找s，若在某处找到了s，则返回指向该处的指针；若没找到，则返回NULL。

该算法采用的是散列查找方法——将所输入的名字转换成一个小的非负整数，这个整数随后被用来当作指针数组的下标。数组的每个元素指向某个链表的表头，这些链表中各个块用于描述具有某散列值的名字。如果没有与该散列值对应的名字，那么值为NULL（见图6-4）。

链表中的每一个块都是一个结构，包含指向名字的指针、指向替换文本的指针及指向该链表中后继块的指针。空后继指针标记表尾。

```
struct nlist {              /* 链表项 */
    struct nlist *next;     /* 链表中下一表项 */
    char *name;             /* 定义的名字 */
    char *defn;             /* 替换文本 */
};
```

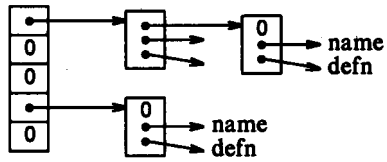

图 6-4

相应的指针数组定义如下：

```
#define  HASHSIZE  101
static struct nlist *hashtab[HASHSIZE];    /* 指针表 */
```

lookup和install函数中的散列函数用于将字符串中每个字符值与前面已求得的部分值杂混相加，并返回对数组大小求模后的余数。这不是最合适的散列函数，但简短有效。

```
/* hash：为字符串s生成散列值 */
unsigned hash(char *s)
{
    unsigned hashval;

    for  ( hashval = 0; *s != '\0';  s++)
         hashval = *s + 31*hashval;
    return hashval % HASHSIZE;
}
```

由于在散列时采用的是无符号算术运算，故保证了散列值非负。

散列过程生成了数组hashtab中的一个起始下标。若一个串与某个起始下标匹配，那么它就存在于该起始下标所指向的链表的某个块中。具体查找过程由lookup函数实现。如果lookup函数发现表项已存在，那么返回指向该表项的指针，否则返回NULL。

```
/* lookup: 在hashtab中查找s */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np !=NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
           return np;        /* 找到s */
    return NULL;                 /* 未找到s */
}
```

lookup函数中的for循环是遍历一个链表的标准方法：

```
for ( ptr = head; ptr != NULL; ptr = ptr->next )
   …
```

install函数借助lookup函数判断待散列的名字是否已存在：如果已存在，那么用新的定义取而代之，否则，将创建一个新的表项。如无足够空间创建新表项，则 install函数返回NULL。

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: 将（name, defn）放入hashtab中 */
struct nlist *install(char *name, char *defn)
{
   struct nlist *np;
   unsigned hashval;

   if ((np = lookup(name)) == NULL) { /* 未找到 */
        np = (struct nlist *) malloc (sizeof(*np));
        if (np == NULL | | (np->name = strdup(name)) == NULL)
            return NULL;
         hashval = hash(name);
         np->next = hashtab[hashval];
         hashtab[hashval] = np;
   } else    /* 已存在 */
         free((void *) np->defn);    / * 释放前一个defn */
    if ((np->defn = strdup(defn)) == NULL)
         return NULL;
     return np;
}
```

练习6-5　编写函数undef，从由lookup和install维护的表中删除一个名字及其定义。

练习6-6　以本节所介绍的函数为基础，编写一个适合 C程序使用的简单版本来实现 #define处理程序（即无变元）。你将发现getch和ungetch函数非常有用。

6.7　类型定义

C提供了一个类型定义（typedef）功能，用来建立新的数据类型名字，例如，说明语句

```
typedef  int  Length;
```

使得名字Length成为int的同义词。类型Length可用于说明、类型转换等，和类型 int完全一样：

```
Length  len, maxlen;
Length  *lengths[];
```

同样，说明语句

```
typedef  char * String;
```

使得String成为char *或字符指针的同义词，随后 String即可被用于说明和类型转换：

```
String  p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc (100);
```

注意typedef中说明的类型在变量名的位置出现，而不是紧接着单词 typedef之后出现。typedef在语法上类似存储类extern，static等。由typedef说明的类型以大写字母开头，以示区别。

作为一个更复杂的例子，可用typedef来重新定义本章前面所介绍的树节点：

```
typedef  struct tnode *Treeptr

typedef  struct  tnode {             /* 树节点 */
      char *word;                    /* 指向文本 */
      int count;                     /* 出现次数 */
      Treeptr left;                  /* 左子树 */
      Treeptr right;                 /* 右子树 */
}  Treenode;
```

程序创建了两个新类型关键字：Treenode（一个结构）和Treeptr（一个指向该结构的指针）。从而函数talloc可相应地修改为：

```
Treeptr talloc (void)
{
   return (Treeptr) malloc(sizeof(Treenode));
}
```

必须强调的是：在任何意义上 typedef都没有创建一个新类型，它只是为某个已存在的类型增加了一个新的名称。在语义上， typedef同样也没有增加新内容，其说明的变量与显式定义说明的变量具有完全相同的属性。实际上， typedef类似于#define语句，只有一点不同：由于可被编译程序解释，故typedef能处理文本替换，而这是预处理程序所力不能及的。例如：

```
typedef  int  (*PFI) (char *, char *);
```

定义了类型PFI，表示“指向（具有两个char *变元的）返回整型值的函数的指针”，这可被用于某些场合，例如，可以用在第5章的排序程序中：

```
PFI strcmp, numcmp;
```

除了看上去整洁美观外，使用 typedef还有两个主要原因。第一，使程序参数化，以提高程序的可移植性。如果typedef说明的数据类型依赖于机器，那么当程序移植到其他机器上时，只需改变typedef类型定义。一个常见的情况是使用 typedef给不同的整型量说明类型，随后可为每个宿主机器选择适当的 short，int和long型整数集。例如标准库中的一些类型，如 size_t和

ptrdiff_t。

typedef的第二个作用是为程序提供更好的文档 —— 一个Treeptr类型显然比一个说明为指向复杂结构的指针更容易理解。

6.8 联合

联合是可以在不同时刻保存不同类型和大小的对象的变量，由编译程序负责跟踪分配大小和对齐需求。联合提供了一种方法，可在单块存储区中管理不同类型的数据，而不需要在程序中嵌入任何依赖于机器的信息。它类似于 Pascal中的变体记录。

作为一个例子（可以在编译程序的符号表管理程序中找到该例子），假设一个常量可能是整型、浮点型或字符指针。一个特定的常量必须保存在一个类型恰当的变量中，然而表管理最方便的做法是让该值占有相同大小存储区，且忽视其类型而存入同一地方。这就是联合的目的 —— 一个单一变量能合法地保存几种类型中的任一种。其语法基于如下结构：

```
union u_tag {
      int ival;
      float fval;
      char *sval;
} u;
```

变量u要足够大以便能保存这三种类型中最大的一个，具体大小依赖于实现。这些类型中的任一个都可赋给 u，随后可在表达式中使用，只要该使用满足一致性：读取的类型必须是最近一次存入的类型。跟踪当前存入联合中的类型是程序员的责任。如果存入类型和提取类型不同，则结果取决于实现。

联合成员访问的语法如下：

```
联合名 . 成员
```

或：

```
联合指针->成员
```

这与访问结构的方式相同。如果用变量 utype来跟踪存入 u中的当前类型，那么可用如下代码：

```
if (utype == INT)
   printf("%d\n", u.ival);
else if (utype == FLOAT)
   printf("%f\n", u.fval);
else if (utype == STRING)
   printf("%s\n", u.sval);
else
   printf("bad type %d in utype\n", utype);
```

联合可在结构和数组中出现，反之亦可。访问结构中联合（或反之）的某一成员的表示法与嵌套结构相同。例如，定义结构数组如下：

```
struct  {
   char *name;
   int flags;
```

```
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

其成员ival的引用为：

```
symtab[i].u.ival
```

而字符串sval的首字符表示为如下两者之一：

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

实际上，在某种意义上，联合也是一个结构，只不过是其所有成员相对于基地址的偏移量为0，结构空间要大到能容纳最“宽”的成员，并且对齐方式适合于联合中的所有成员。对联合的运算与对结构的运算相同，即可以对联合做如下运算：作为一个整体单元赋值或拷贝、取地址及访问一个成员。

一个联合只能用它第一个成员类型的值来初始化，从而上述的联合 u只能用整型值来初始化。

第8章的存储分配程序将说明在特定类型的存储边界上如何使用联合来强制一个变量的对齐。

6.9 位字段

当存储空间很宝贵时，有必要将几个对象打包到一个单一机器字中去。一个常用的方法是使用类似编译程序中符号表的单个位标志集合。外部使用的数据格式（如硬件接口设备）也常需要能从字的部分位中读取数据。

想象一下编译程序中操作符号表的那一部分。程序中的每个标识符都有与其关联的特定信息，例如，它是否为关键字，它是否是外部的且（或）静态的，等等。编码这些信息最简洁的方法是使用在单个字符或整数空间内的单个位标志集合。

通常采用的方法是定义一个与相应位位置有关的“掩码”集，如：

```
#define KEYWORD    01
#define EXTERNAL   02
#define STATIC     04
```

或:

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

这些数字必须是2的某个乘幂。从而访问这些位就变成了使用移位、掩码及第 2章所述按位求反运算符的“简单位”操作。

诸如下面的语句会经常出现：

```
flags |= EXTERNAL | STATIC;
```

它用于在flags中打开EXTERNAL和STATIC，而

```
flags &= ~(EXTERNAL | STATIC);
```

将它们关闭，并且当这两位都为0时，有：

```
if  ((flags & (EXTERNAL | STATIC)) == 0) …
```

为真。

尽管这些术语很容易掌握，但C采用了另一种方法取而代之，即提供在字内直接定义和访问字段的能力，而不是使用按位方式的逻辑运算符。位字段，或简称字段，是“字”中相邻位的集合，所谓“字”是由实现定义的单一存储单元。字段定义和访问的语法基于结构。例如，上述符号表中#define语句可用三个字段的定义来代替：

```
struct {
      unsigned int is_keyword    : 1;
      unsigned int is_extern     : 1;
      unsigned int is_static     : 1;
}  flags;
```

语句定义了一个变量flags，它包含三个1位字段。冒号后的数字标识字段的域宽。字段被说明为无符号整型以保证它们是无符号量。

单一字段的引用方式与其他结构成员相同，如 flags.is_keyword、flags.is_extern等等。字段的特征与小整数相似，与其他整数一样，字段可出现在算术表达式中。故上例可更自然地写成：

```
flags.is_extern = flags.is_static = 1;
```

该语句将is_extern与is_static位打开。语句

```
flags.is_extern = flags.is_static = 0;
```

将is_extern与is_static位关闭。语句

```
if (flags.is_extern == 0 && flags.is_static == 0)
   …
```

用于对is_extern与is_static位进行测试。

有关字段的几乎所有属性都取决于实现。一个字段是否能覆盖字边界由实现定义。字段可无名字，无名字段（只有一个冒号和宽度）被用于填充。特殊宽度 0用于强制在下一个字边界上对齐。

在一些机器上字段的分配从字左端至右端，而在另一些机器上则相反。这意味着尽管字段对维护内部定义的数据结构很有用，但当另外选择外部定义的数据时，必须仔细考虑哪端优先的问题，所以依赖于这些情况的程序是不可移植的。字段也可能只说明成整型，为了方便移植，需显式说明该整型是signed还是unsigned。它们不是数组，并且也没有地址，因此对它们不能应用&运算符。

第7章 输入与输出

输入输出功能并不是C语言本身的组成部分，所以到目前为止，我们并没有过多地强调它们。然而，程序与其环境之间的交互比此前所描述的要复杂得多。本章将描述标准库，介绍一些用于输入输出、字符串处理、存储管理与数学函数以及其他一些用于C程序的功能。讨论的重点将放在输入输出上。

ANSI标准精确地定义了这些库函数，所以这些函数在任何可以使用 C语言的系统中都能以相容的形式存在。如果程序的系统交互部分仅仅使用了由标准库提供的功能，那么这些程序不经修改就可以从一个系统移植到另一个系统。

这些库函数的性质分别在十多个头文件中声明，前面已经遇到过一部分，如<stdio.h>、<string.h>和<ctype.h>。我们不打算把整个标准库都罗列于此，因为我们更关心如何使用标准库编写C程序。附录B对该标准库进行了详细的描述。

7.1 标准输入输出

正如第1章所述，标准库可以实现简单的文本输入输出模式。文本流由一些行组成，每一行的结尾是一个换行符。如果系统不是用这种方式操作的，标准库将尽可能使该系统适应这种模式。例如，标准库可以在输入端将回车符和换行符都转换为换行符，而在输出端进行反向转换。

最简单的输入机制是用getchar函数从标准输入中（一般为键盘）一次读取一个字符：

```
int  getchar(void)
```

getchar函数在每一次被调用时返回下一个输入字符。若遇到文件结束，则返回 EOF。符号常量EOF在头文件<stdio.h>中定义，其值一般为-1，但程序中应该使用EOF而不是-1来测试文件是否结束，以使得程序独立于 EOF的特定值。

在许多情况下，可以使用符号＜让一个文件代替键盘来实现输入重定向：如果程序prog中要使用函数getchar，那么命令行

```
prog < infile
```

就使得程序prog从文件infile（而不是从键盘）中读取字符。实际上，输入开关的实现对程序prog本身是透明的，而且，字符串“ <infile ”也并没有包含在argv的命令行变元中。如果输入通过管道机制来自于另一个程序，那么这种输入开关也是不可见的：在某些系统中，如下命令行：

```
otherprog | prog
```

将运行两个程序otherprog和prog，而且将程序otherprog的标准输出流连接到程序prog的标准输入流上。

函数

```
int  putchar(int)
```

用于输出数据。putchar(c)将字符c送至标准输出流，而标准输出流在缺省情况下为屏幕显示。如果没有错误，那么函数putchar返回所输出的字符；如果有错误出现，那么将返回EOF。同样，输出通常也可以使用

```
>文件名
```

重定向输出到某个文件中。例如，如果程序prog中调用了函数putchar，那么命令行

```
prog > outfile
```

将把程序prog中本应向标准输出设备输出的数据输出到文件outfile中。如果系统支持管道设施，那么命令行

```
prog | anotherprog
```

将把程序prog的标准输出流联接到程序anotherprog的标准输入流上。

函数printf也可用于向标准输出流输出数据。我们在程序中有可能要交叉调用函数putchar和printf，输出将按照函数调用的先后顺序依次产生。

每一个使用输入输出库函数的源程序文件必须在引用这些函数之前包含如下语句：

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

当文件名用一对尖括号<和>括起来时，预处理程序在实现定义的有关位置中查找所指明的文件（例如，在UNIX系统中，文件一般放在目录/usr/include中）。

许多程序只从一个输入流中读取数据并且只向一个输出流输出数据，对于这样的程序，只需用函数getchar、putchar和printf来实现输入输出即可，并且能保证程序的正常启动。尤其是在用重定向将一个程序的输出联结到另一个程序的输入的情况时，程序的输入输出只用这些函数就能实现。例如，考察下面这个程序lower，它用于把输入转换为小写字母的形式：

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

main( )    /* lower: 将输入转换为小写形式*/
{
    int  c;

   while ((c = getchar( )) !=  EOF)
         putchar(tolower(c));
   return  0;
}
```

函数tolower在头文件<ctype.h>中定义，它把大写字母转换为小写形式，并把其他字符原样返回。如前所述，诸如头文件<stdio.h>中的getchar和putchar“函数”以及<ctype.h>中的tolower“函数”一般都为宏，从而避免了对每个字符都进行函数调用的开销。8.5节将介绍它们的实现方法。无论<ctype.h>中的函数在给定的机器上是如何实现的，使用这些函数的程序都可以不必了

解字符集的知识。

练习7-1　编写一个程序，按照放在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。

7.2　格式化输出 —— printf函数

输出函数printf用于将内部数值转换为字符的形式。在前面的有关章节中已经非正式地使用过这个函数。下面的描述只包括常见的一些典型的使用方法，附录 B中给出了其完整的描述。

```
int  printf(char *format, 变元1 , 变元2 , ...)
```

函数printf在输出格式format的控制下，将其变元进行转换与格式化，并在标准输出设备上打印出来。它的返回值为所打印的字符数。

格式化字符串包含两种类型的对象：普通字符和转换规格说明。在输出时普通字符原样不动复制到输出流中，而转换规格说明并不直接输出到输出流中，而是用于控制 printf中变元的转换和打印。每一个转换规格说明都由一个百分号字符 (即%)引入，并以一个转换字符结束。在字符%和转换字符中间可能依次夹有如下成分：

- 负号，用于指定被转换的变元按照左对齐的形式输出。
- 数，用于指定最小字段宽。被转换后的变元将在不小于最小字段宽的宽度中打印出来。如果必要，字段左面（或右面，如果使用左对齐）多余的字符位置用空格填充以保证最小字段宽。
- 小数点，用于将字段宽和精度分开。
- 数，用于表示精度，即指定一个字符串中所要打印的最大字符数，或一个浮点数小数点后的位数，或一个整数最少输出的数字数目。
- 字母h或l，字母h表示将整数作为short类型打印，字母l表示将整数作为long类型打印。

表7-1中列出了所有转换字符。如果 %后面的字符不是一个转换规格说明，则该行为是未定义的。

表7-1　printf函数基本的转换字符

字　符	变元类型；输出形式
d, i	int类型；十进制数
o	int类型；无符号八进制数(不以0开头)
x, X	int类型；无符号十六进制数(不以0x或0X开头)，10~15这六个数分别用a~f或A~F表示
u	int类型；无符号十进制数
c	int类型；单个字符
s	char * 类型；顺序打印字符串中的字符直到遇到空字符 ('\0') 或已打印了由精度指定的字符数为止
f	double类型；十进制小数表示法：[-] *m.dddddd*，其中d的个数由精度指定(缺省值为6)
e, E	double类型；指数形式：[-] *m.dddddd* e±*xx* 或 [-]*m.dddddd* E±*xx*，其中d的数目由精度指定(缺省值为6)
g, G	double类型；如果指数小于-4或大于等于精度，则用%e或%E格式输出，否则用%f格式输出。尾部的0和小数点不打印
p	void * 类型；指针(取决于实现)
%	不转换变元；打印一个百分号

在转换规格说明中，宽度或精度可以用星号*表示，这时，宽度或精度的值通过转换下一个变元（必须为int类型）来计算。例如，为了从字符串s中最多打印max个字符，使用如下语句：

```
printf( "%.*s", max, s);
```

在前面的章节中已经介绍过大部分的格式转换成分，但还未介绍与字符串相关的精度。下面演示了在打印字符串“ hello, world ”(12个字符)时根据不同的转换规格说明而产生的不同结果。我们在每个字段的左面和右面加上冒号，这样可以清晰地表示出字段的宽度。

```
:%s:              :hello, world:
:%10s:            :hello, world:
:%.10s:           :hello, wor:
:%-10s:           :hello, world:
:%.15s:           :hello, world:
:%-15s:           :hello, world:
:%15.10s:         :hello, wor:
:%-15.10s:        :hello, wor:
```

注意：函数printf用它的第一个变元来判断它还有多少个变元及这些变元的类型。如果 printf从第二变元起的变元个数少于第一个变元中指定的个数或它们的类型与第一个变元中指定的类型不符时，那么会得到错误的结果。请注意如下两个函数调用之间的不同之处：

```
printf(s);          /*如果字符串s含有字符%，输出将出错*/
printf("%s", s);    /*正确*/
```

函数sprintf在执行时所进行的转换和函数printf是一样的，但它把输出存放在一个字符串中：

```
int  sprintf(char *string, char *format, 变元1 , 变元2 , ...)
```

sprintf函数和printf函数一样，按照format格式化变元序列，只不过它把结果放到 string中而不是放到标准输出流中。当然， string必须足够大以能放下整个结果。

练习7-2　编写一个程序，实现以合理的方式打印任意输入的功能。至少，它必须根据用户习惯以八进制或十六进制打印非图形字符，并将过长的文本行截断为多个短的文本行。

7.3　变长变元表

本节以函数printf的一个最简化的版本为例，介绍如何编写能够处理带有可变长度的变元表的函数。因为重点在于变元的处理，因此函数 minprintf只处理格式字符串和变元表，格式转换则通过调用函数printf实现。

下面是具有变长变元表的函数printf的正确的说明形式：

```
int  printf(char *fmt, ...)
```

其中，省略号... 表示变元表中变元的数量和类型可能会变化。 ... 只能在变元表的最后位置处出现。由于minprintf不需要像printf函数那样返回实际输出的字符数，因而其说明为如下形式：

```
void  minprintf(char *fmt, ...)
```

编写函数 minprintf的关键在于如何处理一个甚至连名字也没有的变元表。标准头文件<stdarg.h>中包含了一组用于如何处理变元表的宏定义。这个头文件的实现部分随着不同的机器而变化，但它所提供的接口是一致的。

类型va_list用于说明一个依次引用每个变元的变量，该变量在函数 minprintf中称为ap，意思是“ 变元指针 ”。宏va_start将ap初始化为指向第一个无名的变元。在 ap被使用之前，这个宏必须被调用一次。变元表中必须至少有一个有名字的变元，且 va_start根据最后一个有名的变元的值对ap进行初始化。

每调用一次va_arg就返回一个变元并且使 ap指向下一个变元。 va_arg使用一个类型名以决定该返回哪一种类型的对象及如何使 ap指向下一个变元。最后， va_end必须在函数返回之前被调用，完成一些必要的清除工作。

上面这些讨论是实现简化的 printf函数的基础：

```
# include <stdarg.h>

/* minprintf：带有可变变元表的简化的函数printf */
void  minprintf(char *fmt, ...)
{
  va_list  ap;   /*依次指向每一个无名的变元*/
  char  *p, *sval;
  int  ival;
  double  dval;

  va_start(ap, fmt); /*使ap指向第一个无名的变元*/
  for (p = fmt; *p; p++) {
   if(*p != '%') {
      putchar(*p);
      continue;
   }
   switch (*++p) {
   case 'd':
      ival = va_arg(ap, int);
      printf("%d", ival);
      break;
   case 'f':
      dval = va_arg(ap, double);
      printf("%f", dval);
      break;
   case 's':
      for(sval = va_arg(ap, char *); *sval; sval++)
         putchar(*sval);
         break;
      default:
         putchar(*p);
         break;
      }
   }
   va_end(ap);    /*结束时的清除工作*/
}
```

练习7-3　改写minprintf函数，使其能处理printf函数的更多的功能。

7.4 格式化输入——scanf函数

输入函数scanf对应于输出函数printf，它在与后者相反的方向上提供同样多的转换功能。具有变长变元表的函数scanf的说明形式为：

```
int  scanf(char *format, ...)
```

scanf函数从标准输入流读取字符序列，按照format中的转换规格说明对字符序列进行解释，并把结果存储在其余的变元中。下面将讨论格式变元format，其他变元必须都是指针，用于指明相对应的转换后的输入应该存放的位置。和上节一样，本节介绍scanf函数的一些最常用的特征，而不去完整地描述它。

当scanf函数使用完了格式输入串或当一些输入无法与控制说明相匹配时，它就停止运行，并返回成功匹配和赋值的输入项的个数。所以返回值可以作为判断已获得输入值的输入项数的依据。在文件的结尾，EOF被返回。注意这与返回值0不同，0表示下一个输入字符和格式串中的第一个说明不匹配。下一次对这个函数的调用将从上一次转换的最后一个字符的下一个字符开始继续搜索。

另外一个输入函数sscanf用于从一个字符串（而不是标准输入流）读取字符序列：

```
int  sscanf(char *string, char *format, 变元1, 变元2, ...)
```

它按照格式变元format中说明的格式从字符串string中读取字符序列，并把结果存储在变元序列指向的位置中。这些变元必须是指针。

输入格式串一般都包含转换规格说明，用于控制输入的转换。格式化串可能包含如下成分：

- 空格或制表符，在处理过程中将被忽略。
- 普通字符序列(不包括%)，用于匹配输入流中下面尚待读入的非空白符序列。
- 转换规格说明，依次由一个%、一个可选的*（用于赋值屏蔽字符）、一个可选的整数（用于指定最大字段宽）、一个可选的h、l或L字符（用于指明转换对象的宽度）以及一个转换字符组成。

转换规格说明用于控制下一个输入字段的转换过程。一般而言，转换结果放在相应的变元所指向的变量中。但是如果转换规格说明中有用作赋值屏蔽字符的*，则就跳过输入字段，不进行赋值。输入字段是一个由非空白符组成的字符串，它的宽度要么直至下一个空白符，要么直至指定的最大字段宽。这表明scanf函数将越过行边界读取它的输入，因为换行符也是空白符(空白符包括空格符、横向制表符、换行符、回车符、纵向制表符及换页符)。

转换字符用于指明如何解释输入字段。按照C语言按值调用的语义，对应的变元必须是一个指针。转换字符列于表7-2中。

表7-2　scanf函数基本的转换字符

字　符	输入数据；变元类型
d	十进制整数；int *类型
i	整数；int *类型。可以是八进制(以0开头)或十六进制整数(以0x或0X开头)

（续）

字　符	输入数据；变元类型
o	八进制整数(以0开头或不以0开头)；int *类型
u	无符号十进制整数；unsigned int *类型
x	无符号十六进制整数(既可以0x或0X开头，也可不以0x或0X开头); int *类型
c	字符；char * 类型。将下一个输入字符串 (缺省值为 1)存放到指定的位置。该转换规格说明通常不跳过其中的空白符，为了读下一个非空白符，使用 %1s
s	字符串(不加引号)；char *类型，指向一个足以存放该字符串的字符数组并在字符串末尾加一空字符（'\0'）
e, f, g	可带前缀正负号、小数点及指数部分的十进制浮点数； float * 类型
%	两个相连的%用于表示单个%；不进行任何赋值操作

在转换字符d、i、o、u及x的前面可以加上字符h或l。h用于表明变元序列中有一个指向 short 类型而不是int类型的指针，l用于表明变元序列中有一个指向 long类型的指针。类似地，转换字符e、f及g的前面也可以加上l，用于表明变元序列中有一个指向 double类型而不是float类型的指针。

作为第一个例子，下面用函数 scanf进行输入转换来改写第 4章的具有基本运算功能的计算器程序：

```
#include <stdio.h>

main( )          /* 具有基本运算功能的计算器程序 */
{
    double  sum, v;

    sum = 0;
    while (scanf("%f", &v) = = 1)
           printf("\t%.2f\n", sum += v);
    return  0;
}
```

假设我们想读取包含如下数据形式的输入行：

```
25 Dec 1998
```

则相应的scanf语句为：

```
int  day, year;
char  monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

在monthname前面没有取地址运算符&，因为数组名本身就是指针。

普通字符串也可以出现在 scanf的格式串中，但是它们必须与输入中相同的字符串匹配。故可以使用下列的scanf语句读入形如mm/dd/yy的日期数据：

```
int  day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

scanf函数忽略格式字符串中的空格和制表符。除此之外，在读取输入值时，它将忽略空白符（空格、制表符、换行符等等）。为了读取没有固定格式的输入，最好每一次读入一行，然后用sscanf逐个读入。例如，假设我们想读取一些可能包含用上述任一种形式表示的日期数据的行时，可采用下面这个程序：

```
while (getline(line, sizeof(line)) > 0)  {
   if (sscanf(line, "%d %s %d", &day, monthname, &year) = = 3)
       printf("valid: %s\n", line);   /*日期数据形如25 Dec 1988 */
   else if (sscanf(line, "%d/%d/%d", &month, &day, &year) = = 3)
       printf("valid: %s\n", line)    /*日期数据形如mm/dd/yy */
   else
       printf("invalid: %s\n", line);  /*日期形式无效 */
}
```

scanf函数可以和其他输入函数混合起来使用。紧跟在 scanf函数调用之后的下一个输入函数的调用将从scanf没有读取的第一个字符处开始读取数据。

注意：scanf和sscanf函数的所有变元都必须是指针。最常见的错误是将输入语句写成如下形式：

```
scanf("%d", n);
```

而不是写成

```
scanf("%d", &n )
```

编译程序在编译时一般检测不到这类错误。

练习7-4　编写scanf函数的一个简化版本，类似于上一节描述的函数 minprintf。

练习7-5　改写第4章介绍的后缀计算器，用 scanf函数和（或） sscanf函数实现输入和数的转换。

7.5　文件访问

到目前为止，我们所讨论的例子都是从标准输入读取数据并向标准输出输出数据，其中，标准输入和标准输出是由所用操作系统自动提供给程序访问的。

下面编写能访问但还没有和它发生关联的文件的程序。程序 cat可说明这个问题，它把一批有名文件串联后输出到标准输出流上。 cat可用来在屏幕上打印文件，也可用作那些无法通过名字访问文件的程序的通用输入收集器。例如，命令行

```
cat  x.c y.c
```

将在标准输出上打印文件 x.c和y.c的内容（仅此而已）。

程序能够访问文件的关键在于怎样安排有名文件的被读取过程，即怎样将用户所要使用的文件的外部名和访问这些文件的语句联系起来。

方法其实很简单。在读写一个文件之前，必须用库函数 fopen打开它。fopen用诸如x.c或y.c这样的外部名与操作系统进行某些必要的连接和通信（我们不必关心其细节），并返回一个此后用于文件读写操作的指针。

该指针称为文件指针，它指向一个包含文件信息的结构，这些信息包括：缓冲区的位置、缓冲区中当前所指向的字符、是正在读或还是正在写文件、文件是否出错或是否已经到达文件结尾等等。用户不必知道这些细节，因为 <stdio.h>中定义了一个包含这些信息的结构 FILE。在程序中只需做如下说明：

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

本例中，fp是一个指向结构FILE的指针，fopen函数返回一个指向结构FILE的指针。注意FILE像int一样是类型名，而不是结构标记，它由 typedef定义（fopen在UNIX系统中的实现细节将在8.5节讨论）。

在程序可以这样调用fopen函数：

```
fp = fopen(name, mode);
```

fopen的第一个变元是一个包含文件名的字符串。第二个变元是访问方式，它也是一个字符串，用于指定文件打开后的使用方法。允许的方式包括读方式（ "r"）、写（"w"）及添加（"a"）。一些系统还区分文本和二进制文件，后者的访问需要在方式串中添加字符 "b"。

如果以写或添加方式打开一个不存在的文件，那么系统建立所指名的文件。当以写方式打开一个已存在的文件时，该文件原来的内容将被取代，但如果以添加方式打开一个文件，则该文件原来的内容保留不变。读一个不存在的文件会产生一个错误，其他一些操作也有可能导致错误，比如读取一个无读取权限的文件。如果有错误产生， fopen将返回NULL（我们可以更精确地识别错误的类型，参见附录 B.1节中关于错误处理函数的讨论）。

一旦文件被打开，下面所要做的就是读写文件了。有几种方法可以进行这些操作，其中 getc和putc函数是最简单的。 getc返回一个文件中的下一个字符，它需要文件指针来告诉它针对哪一个文件进行操作：

```
int  getc( FILE *fp)
```

getc函数返回fp指向的输入流中的下一个字符。如果到达文件尾或出现错误，那么它返回 EOF。

putc是一个输出函数：

```
int  putc( int c, FILE *fp)
```

putc函数将字符c写入fp指向的文件并返回所写入的字符。如果有错误出现，则返回 EOF。正如getchar和putchar函数一样，getc和putc函数也可以作为宏。

当一个c程序开始运行时，操作系统环境负责打开三个文件并为它们提供相应的指针。这些文件是标准输入、标准输出和标准错误，其相应的指针分别为 stdin、stdout和stderr，它们在<stdio.h>中说明。在大多数环境中， stdin指向键盘，而stdout和stderr则指向显示器；但正如7.1节中所述，stdin和stdout可以被重定向指向文件或管道。

getchar和putchar函数可以用getc、putc、stdin及stdout函数定义如下：

```
#define  getchar( )  getc(stdin)
#define  putchar(c)  putc((c), stdout)
```

为了格式化文件的输入输出，可以使用函数 fscanf和fprintf。它们与scanf和printf函数的区别仅仅在于它们的第一个变元是一个指向所要读写文件的指针，第二个变元则是格式串：

```
int  fscanf( FILE *fp, char *format, ...)
int  fprintf( FILE *fp, char *format, ...)
```

在掌握这些初步知识之后，我们现在就能够编写将一组文件串联起来的 cat程序了。该程序的设计过程和许多其他程序的设计过程类似。如果命令行中带有变元序列，则将它们作为文件名按顺序处理，如果没有变元，则使用标准输入。

```
#include <stdio.h>

/* cat: 将一组文件串联起来，版本1 */
main (int argc, char *argv[ ])
{
    FILE  *fp;
    void  filecopy(FILE *, FILE *);

    if (argc = = 1)      /* 如果命令行没有带变元，则复制标准输入*/
        filecopy(stdin, stdout);
    else
        while (- - argc > 0)
            if ((fp = fopen(*++argv, "r")) = = NULL)  {
                 printf("cat: can't open %s\n", *argv);
                 return  1;
            } else {
                 filecopy(fp, stdout);
                 fclose(fp);
            }
        return  0;
}

/* filecopy: 将文件ifp复制给文件ofp */
void  filecopy(FILE *ifp, FILE *ofp)
{
    int  c;

    while ((c = getc(ifp)) != EOF)
         putc(c, ofp);
}
```

文件指针stdin与stdout是FILE *类型的对象。但它们是常量而不是变量，因此不可能赋值给它们。

函数

```
int  fclose(FILE *fp)
```

执行和fopen相反的操作，它断开由 fopen函数建立的文件指针和外部名之间的连接，释放文件指针以供其他文件使用。因为大多数操作系统都限制了一个程序可以同时打开的文件数，所以当文件指针不再需要时就释放它是一个好的编程思想，就像我们在 cat程序中所做的那样。在输出文件上使用fclose函数还有另外一个原因：它将缓冲区中由 putc函数正在收集的输出写到文件中。当一个程序正常终止时，程序自动为每个打开的文件调用 fclose函数。(如果不再需要使用stdin与

stdout，可以把它们关闭。它们也能通过 freopen函数重新赋值。)

7.6 错误处理 —— stderr和exit函数

cat程序的错误处理部分并不完善。问题在于，如果由于某种原因而造成其中的一个文件不能访问时，相应的诊断信息将在被连接的输出的末尾打印。当该输出指向屏幕时这种处理方法尚可以接受，但如果它指向一个文件或借助于管道而指向另一个程序的输入时则会出现问题。

为完美地解决这个问题，操作系统像提供 stdin和stdout一样自动提供了另一个称为 stderr的输出流。即使重定向了标准输出， stderr上的输出通常也会显示在屏幕上。

下面改写cat程序，使得它在标准错误上显示有关的出错信息。

```
#include <stdio.n>

/* cat: 将一组文件串联起来，版本2 */
main(int argc, char *argv [ ])
{
    FILE  *fp;
    void  filecopy(FILE *, FILE *);
    char  *prog = argv[0];      /* 记下程序名供错误处理使用*/

    if (argc = = 1)         /* 如果命令行没有带变元，则复制标准输入*/
       filecopy(stdin, stdout);
    else
       while ( --argc > 0)
            if ((fp = fopen(*++argv, "r")) = = NULL)  {
            fprintf(stderr, "%s: cant't open %s\n", prog, *argv);
            exit(1);
       } else  {
            filecopy(fp, stdout);
            fclose(fp);
       }
    if (ferror(stdout))  {
       fprintf(stderr, "%s: error writing stdout\n", prog);
       exit(2);
    }
    exit(0);
}
```

该程序通过两种方式报错。首先，将 fprintf函数产生的诊断信息输出到 stderr上，因此诊断信息将会显示在屏幕上而不是消失在管道或一个输出文件中。诊断信息中包含 argv[0]中的程序名，这样当这个程序和其他程序一起使用时，可以识别错误的来源。

其次，程序使用了标准库函数 exit，当它被调用时将终止调用程序的执行。任何调用 exit的进程都能得到它的变元，因此可通过另一个将该程序作为子进程的程序来测试该程序是否成功地执行了。按照惯例，0返回值表示一切正常，而非 0返回值通常表示出现了异常情况。 exit为每个已打开的输出文件调用相应的 fclose函数，以将任何已缓冲的输出写到相应文件中。

在主程序main中，语句

```
return expr
```

等价于

```
exit(expr)
```

但使用函数exit有如下优点：它可以被其他函数调用，并且可以用诸如第 5章中所描述的模式查找函数查找这些调用。

如果流fp上出现一个错误，则函数 ferror返回非0值。

```
int  ferror(FILE *fp)
```

尽管输出错误很少遇到，但它们确实会出现（例如，在一个磁盘装满数据时），因此一个商业程序也应该能检查这种错误。

函数feof(FILE *)与ferror类似，如果遇所指明文件的结尾，那么它返回非 0值。

```
int  feof(FILE *fp)
```

在上面所讨论的这个旨在说明问题的小程序中，不用担心出口返回的状态，但重要的程序都应该小心地返回合理且有意义的数值。

7.7 行输入输出

标准库提供了一个输入函数 fgets，它和前面几章中用过的函数 getline相似：

```
char  *fgets(char *line, int maxline, FILE *fp)
```

fgets函数从 fp所指向的文件中读取下一个输入行（包括换行符）并将它存放于字符数组 line中，它最多可读取 maxline-1个字符。所读取的行以空字符（ '\0'）结尾。在通常情况下， fgets返回 line，但如果遇到了文件结尾或有错误出现，那么返回 NULL（我们编写的getline函数返回所读入行的长度，这个值更有用，当它为 0时意味着已经到达了文件的结尾）。

对于输出，函数fputs将一个字符串（不必包含换行符）写入一个文件中：

```
int  fputs(char *line, FILE * fp)
```

如果有错误出现，则它返回 EOF，否则返回0。

库函数gets和puts的功能相似于 fgets和fputs函数，但它们在 stdin和stdout上进行操作。不同的是，gets函数在读取字符串时将丢弃结尾的换行符 ('\n')，而puts函数在所写出的字符串后还要添加一个换行符。

为了表明诸如 fgets和fputs这样的函数没有什么特别的地方，我们将标准库中它们的代码复制如下：

```
/* fgets: 从iop所指向的文件中最多读取n个字符*/
char  *fgets(char *s, int n, FILE *iop)
{
    register  int c;
    register char  *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
```

```
  if ((*cs++ = c) = = '\n')
       break;
* cs = '\0'
return (c = = EOF && cs = = s) ? NULL : s;
}

/* fputs: 将字符串s输出到iop所指向的文件 */
int  fputs(char *s, FILE *iop)
{
   int  c;

   while (c = *s++)
      putc(c, iop);
   return  ferror(iop) ? EOF : 0;

}
```

ANSI C标准规定：ferror函数在遇到错误时返回非 0值；而fputs函数在遇到错误时则返回 EOF，否则返回一个非负整数值。

用fgets函数很容易实现getline函数:

```
/* getline: 读一个输入行并返回其长度*/
int  getline(char *line, int max)
{
    if (fgets(line, max, stdin) = = NULL)
        return  0;
    else
        return  strlen(line);
}
```

练习7-6　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。

练习7-7　修改第5章的模式查找程序，使得它从一批命名文件中读取输入，或者在没有文件作为该程序的变元时从标准输入中读取输入。当发现一个匹配的行时，相应的文件名应该打印吗？

练习7-8　编写一个程序，打印一批文件，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。

7.8　其他函数

标准库提供了许多功能各异的函数。本节将对其中大多数常用的函数做一个简要的概述。更多的细节和许多没有介绍的其他函数请参见附录 B。

7.8.1　字符串处理函数

前面已经提到过字符串函数 strlen、strcpy、strcat和strcmp，它们都定义在头文件 <string.h> 中。在下面各个函数中，s与t是指向字符类型的指针（即类型为 char *），而c与n则为int类型。

```
strcat(s, t)              将t所指向的字符串连接到s所指向的字符串的末尾
```

strncat(s, t, n)	将t指向的字符串中前n个字符连接到s指向的字符串的末尾
strcmp(s, t)	视s所指向的字符串小于、等于或大于 t所指向的字符串而返回负整数、 0或正整数
strncmp(s, t, n)	除了前n个字符和strcmp一样
strcpy(s, t)	将t所指向的字符串复制到s所指向的位置
strncpy(s, t, n)	将t所指向的字符串中最多前n个字符复制到s所指向的位置
strlen(s)	返回s所指向的字符串的长度
strchr(s, c)	在s所指向的字符串中查找c，若找到则返回指向它第一次出现的位置的指针，否则返回NULL
strrchr(s, c)	在s所指向的字符串中查找c，若找到则返回指向它最后一次出现的位置的指针，否则返回NULL

7.8.2 字符类测试和转换函数

头文件<ctype.h>中的一些函数用于字符的测试和转换。在下面各个函数中， c是一个可用于表示无符号字符或EOF的整数。该函数的返回值类型为 int。

isalpha(c)	若c是字母，则返回一个非0值，否则返回0
isupper(c)	若c是大写字母，则返回一个非0值，否则返回0
islower(c)	若c是小写字母，则返回一个非0值，否则返回0
isdigit(c)	若c是数字，则返回一个非0值，否则返回0
isspace(c)	若c是空格、横向制表符、换行符、回车符和纵向制表符，则返回一个非0值，否则返回0
toupper(c)	返回c的大写形式
tolower(c)	返回c的小写形式

7.8.3 ungetc函数

标准库提供了一个称为 ungetc的函数，它是第 4章所介绍的函数 ungetch的一个在功能上有较多限制的变种。

```
int  ungetc(int c, FILE *fp)
```

该函数将字符c写回文件 fp。如果执行成功，则返回 c，否则返回 EOF。每个文件只能接收一个写回字符。 ungetc函数可以和诸如scanf、 getc或getchar这样的输入函数中的任何一个一起使用。

7.8.4 命令执行函数

函数system(char *s)执行包含在字符串 s中的命令，然后执行当前程序。 s的内容很大程度上取决于所用的操作系统。以 UNIX操作系统上的一个小例子来看，语句

```
system("date");
```

将执行程序 date，它在标准输出上打印当天的日期和时间。 system函数返回一个来自于所执行的命令且依赖于系统的整数状态。在 UNIX系统中，该返回状态是 exit所返回的数值。

7.8.5 存储管理函数

函数malloc和calloc用于动态地分配存储空间。语句

```
void  *malloc(size_t  n)
```

在分配成功时，返回一个指向 n字节大小的未初始化的存储空间的指针，否则返回 NULL。而语句

```
void  *calloc(size_t  n,  size_t  size)
```

在分配成功时，返回一个指向足以容纳由 n个指定大小的对象组成的数组的存储空间的指针，否则返回NULL。该存储空间的每一字节均初始化为 0。

由malloc或calloc函数返回的指针能自动地根据所分配对象的类型进行适当的对齐调整，但它必须强制转换为恰当的类型，正如下例所示：

```
int  *ip;

ip = (int *) calloc(n, sizeof(int));
```

free(p)函数用于释放 p所指向的存储空间，其中 p是此前通过调用 malloc或calloc函数而得到的指针。存储空间的释放顺序没有什么限制，但释放一个不是通过调用 malloc或calloc函数而得到的指针所指向的存储空间是一个可怕的错误。

使用已经释放的存储空间同样是错误的。下面所示的是一个典型的但不正确的代码段，它由一个用于释放列表中数据项所占的存储空间的循环语句组成：

```
for (p = head; p != NULL; p = p->next)     /* 错误的代码段*/
     free(p);
```

正确的处理方法是在释放数据项之前将一切必要的信息先保存起来：

```
for (p = head; p !=  NULL; p = q) {
     q = p->next;
     free(p);
}
```

8.7节给出了一个类似于 malloc函数的存储分配程序的实现，该存储分配程序分配的存储块可以以任意顺序释放。

7.8.6 数学函数

头文件 <math.h>中声明了二十多个数学函数，以下介绍一些经常使用的函数。每个函数带有一个或两个双精度浮点类型的变元，并返回一个双精度浮点数。

sin(*x*)	*x*的正弦函数，其中*x*用弧度表示
cos(*x*)	*x*的余弦函数，其中*x*用弧度表示
atan2(y, *x*)	*y*/*x*的反正切函数，该函数值用弧度表示
exp(*x*)	指数函数e^x
log(*x*)	*x*的自然对数(以e为底)函数(*x* > 0)
log10(*x*)	*x*的常用对数(以10为底)函数(*x* > 0)
pow(*x*, *y*)	x^y
sqrt(*x*)	*x*的平方根函数(*x*≥0)
fabs(*x*)	*x*的绝对值函数

7.8.7 随机数发生器函数

函数rand( )用于生成介于 0和RAND_MAX之间的伪随机整数序列，其中 RAND_MAX 是

在头文件<stdlib.h>中定义的符号常数。下面是一个生成大于等于 0但小于1的随机浮点数的方法：

```
#define  frand( )  ((double) rand( ) / (RAND_MAX + 1.0))
```

(如果所用函数库中已提供了一个用于生成浮点随机数的函数，那么它可能比上面这个函数具有更好的统计学特性。)

函数srand(unsigned)设置rand所用的随机数生成算法的种子值。 2.7节中给出了遵循标准的rand和srand函数的可移植的实现。

练习7-9　诸如isupper这样的函数能够用某种方式实现以节省空间或时间。试研究这两种可能性。

第8章 UNIX 系统界面

UNIX操作系统的服务通过一系列系统调用来实现，这些系统调用实际上是操作系统提供的函数，它们可以被用户程序调用。本章将介绍如何在 C程序中使用一些重要的系统调用。如果读者正在使用UNIX，本章将会对你有直接的帮助。因为在某些情况下，为了获得最大的效率或者为了访问函数库所没有的功能，有必要调用系统调用。然而，即使读者在别的操作系统上使用 C语言，通过学习本章的例子也将会对 C程序设计有更深的了解。不同系统上的 C程序大同小异，只是在细节有一些出入。由于 ANSI C标准库建立在UNIX系统上，通过学习本章中的程序也会有助于了解这个标准库。

本章分三部分讨论：输入 /输出、文件系统和存储分配。其中前两部分要求读者对UNIX系统的外部特性有大概的了解。

第7章中介绍的输入 /输出界面在任何操作系统上都是一样的。在一个具体的系统上，标准库函数的实现必须通过这个系统提供的功能来完成。后面几节将介绍 UNIX系统中实现输入输出的系统调用，并介绍如何用它们实现标准库。

8.1 文件描述符

在UNIX操作系统中，所有的外部设备，包括键盘和显示器，都被当做文件来处理。因此，所有的输入输出都是通过读写文件来完成的，即通过一个单一的界面来处理程序和外部设备之间的通信。

在通常情况下，在读或写文件之前，必须先通知系统，这个过程称为打开文件。如果是写一个文件，那么可能首先需要建立它，否则会丢弃原先已存在的内容。系统将对权限进行检查（比如检查这个文件存在吗？有访问它的权利吗？），如果权限合法，系统将返回给程序一个小的非负整数，此整数称为文件描述符。任何时候文件的输入输出都是通过文件描述符来标识文件，而不是通过文件名。（文件描述符类似于标准库中的文件指针和 MS-DOS中的文件句柄。）系统维护一个已打开文件的所有信息，而用户程序仅仅通过文件描述符就可引用文件。

绝大多数情况下输入输出是通过键盘和显示器屏幕来实现的。因此 UNIX对此做了特别的安排。当命令解释程序（外壳）运行一个程序的时候，它打开三个文件，对应的文件描述符为 0、1、2，分别表示标准输入、标准输出和标准错误输出。如果程序对0进行读，对1和2进行写，那么它可以进行任何输入输出而不必先打开文件。

用户可通过 < 和 > 来重定向程序的I/O：


```
prog < infile > outfile
```

在这种情况下，外壳把文件描述符0和1的缺省赋值改为所命名的文件。通常文件描述符2仍与显示器相关联，以使出错信息输出到显示器。与管道关联的输入输出也有类似的特性。在所有情况下，文件赋值的改变不是由用户程序，而是由外壳来完成的。只要程序使用文件0作为输入，文件1和2作为输出，则它不会知道它的输入从哪里来，输出到哪里去。

8.2 低级I/O——read和write系统调用

输入与输出分别使用read和write系统调用，在C程序中可通过调用函数read和write实现。这两个函数的第一个变元是文件描述符，第二个变元是程序中存放数据的字符数组，第三个变元是所要传输的字节数目。

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

每个调用返回实际传输的字节数目。在读文件时，函数返回的值可能会小于所请求的字节数。若返回值为0，则表示已到了文件的最后；若返回值为-1，则表示发生了某种错误。在写文件时，返回值是实际写入的字节数目。如果返回值与请求写入的字节数目不相等，那么说明发生了错误。

在一次调用中可以读出或写入任意数目的字节。通常这个数目为1，即每次操作1个字符（无缓冲），或是类似于1024或4096这样的与外部设备的块大小相等的值。用更大的值来调用可以获得更高的效率，因为调用的次数减少了。

通过以上的讨论，我们可以编写一个小程序，将输入复制到输出，与第1章中的复制程序在功能上类似。程序可以将任意输入复制到任意输出，因为输入输出可以重定向到任何文件或设备。

```
#include "syscalls.h"

main()   /*从输入复制到输出*/
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
            write(1, buf, n);
    return 0;
}
```

我们已将系统调用的函数原型都放在一个头文件syscalls.h中，本章中的程序都将使用这个头文件。但这个文件的名字不是标准的。

参数BUFSIZ也已在syscalls.h头文件中定义，相对于局部系统来说这个值比较合适。如果文件大小不是BUFSIZ的倍数，那么对read的某一次调用将返回一个较小的字节数（此字节数传给write调用），且下一次调用read将返回0。

为了帮助读者掌握有关概念，下面来说明如何用read和write来构造诸如getchar、putchar等高级函数。例如，以下是getchar函数的一个版本，它通过每次从标准输入读入一个字符来实现无缓冲输入。

```
#include "syscalls.h"

/*getchar: 无缓冲的单字符输入*/
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char c) c :EOF;
}
```

c必须是一个char类型的变量，因为返回语句中的 read函数需要用一个字符指针作为变元。通过将c强制转换为unsigned char类型可以消除符号扩展问题。

getchar的第2个版本一次读入一组字符，但每次只输出一个。

```
#include "syscalls.h"

/*getchar: 简单的带缓冲的版本*/
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) {    /*缓冲区为空*/
        n = read(0, buf, sizeof buf);
        bufp = buf;
     }
     return (--n >= 0) ? (unsigned char) *bufp++ :EOF;
}
```

如果将这两个版本的getchar函数与<stdio.h>文件一起编译，并且getchar是以宏来实现的话，那么有必要用#undef预处理指令来取消对getchar的宏定义。

8.3 open、creat、close和unlink系统调用

除了缺省的标准输入、标准输出和标准错误输出外，必须在读或写文件之前将它显式地打开。系统调用open和creat可用于实现这个功能。

open与第7章讨论的fopen很相似，不同的是前者返回一个 int类型的文件描述符，而后者则返回一个文件指针。如果发生了错误，那么 open将返回-1。

```
#include <fcntl.h>

int fd
int open(char *name, int flags, int perms);

fd = open(name, flags, perms)
```

与fopen一样，变元name是一个包含文件名的字符串。第二个变元 flags是一个int类型的值，说明以何种方式打开文件，它的几个主要值为：

```
O_RDONLY    以只读方式打开文件
O_WRONLY    以只写方式打开文件
O_RDWR      以读写方式打开文件
```

在UNIX系统V版本中，这些常量定义放在头文件 <fcntl.h>中，在伯克利（BSD）版本中这些变量在<sys/file.h>中定义。

在为进行读操作打开一个已存在文件时可以使用如下语句：

```
fd = open ( name, O_RDONLY, 0);
```

在本章的讨论中，open的变元perms的值永远为0。

如果用open打开一个不存在的文件，那么将导致错误。如果要创建一个新的文件或覆盖一个已存在的文件，那么可以调用 creat系统调用：

```
int creat(char *name, int perms);

fd = creat(name, perms);
```

如果creat成功地创建了文件，那么它返回一个文件描述符，否则返回- 1。如果此文件已存在，那么creat将把它的长度截断为 0，从而舍弃了先前存在的内容。用 creat创建一个已存在的文件不会导致错误。

如果所要创建的文件不存在，那么 creat用变元perms设定的权限来创建文件。在 UNIX文件系统中，文件的权限信息用长为 9位的字段来表示，它们分别控制文件的所有者、所有者组和其他成员对文件的读、写和执行访问。这样一个 3位8进制数就可方便地用来说明不同的权限，例如，0755说明文件的所有者可以对它进行读、写和执行操作，而所有者组和其他成员只能进行读和执行操作。

下面通过一个简化的UNIX程序cp来说明creat的用法。该程序将一个文件复制到另一个文件。它仅复制一个文件，不允许用目录作为其第二个变元，并且目标文件的权限不是通过复制得到，而是自定义的。

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666    /*对于所有者、所有者组和其他成员均可读写*/

void error(char *, ...);

/* cp: 从f1复制到f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if  (argc != 3)
         error("Usage: cp from to");
    if  ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
         error("cp: can't open %s", argv[1]);
```



```
    if  ((f2 = creat(argv[2], PERMS)) == -1)
         error("cp: can't create %s, mode %03o", argv[2], PERMS);
        while ((n = read(f1, buf, BUFSIZ)) > 0)
             if (write(f2, buf, n) != n)
                  error("cp: write error on file %s", argv[2]);
        return 0;
}
```

这个程序以固定的权限0666来创建输出文件。利用8.6节将要讨论的stat系统调用，可以获得一个已存在文件的模式，并将此模式赋给它的拷贝。

注意，error类似于printf，可以在调用时带有变长变元表。下面通过 error函数的实现来说明使用printf函数家族的另一个成员标准库函数 vprintf的方法。vprintf函数与printf函数相似，不同的是用一个变元取代了变长变元表，且此变元通过调用 va_start宏进行初始化。同样， vfprintf和vsprintf函数分别与fprintf和sprintf函数匹配。

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

/*error: 打印完错误信息后退出*/
void error(char *fmt, ...);
{
    va_list args;

    va_stat(args, fmt);
    fprintf(stderr, "error: " );
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

一个程序同时打开的文件数目不能超过一个极限（通常为 20）。相应地，如果一个程序需要同时处理多个文件，那么它必须重用文件描述符。函数 close(int fd)用来断开文件描述符和已打开文件间的联结，并释放此文件描述符，以供其他文件使用。 close函数与标准库中的fclose函数相对应，但它不需要刷新缓冲区。如果程序通过 exit函数退出或从主程序中返回，那么所有打开的文件将被关闭。

函数unlink(char *name)用于将文件name从文件系统中删除掉，它对应于标准库函数 remove。

练习8-1　用read，write，open和close系统调用来替换标准库中功能等价的函数，重写第 7章的cat程序，并通过试验比较两个版本的相对执行速度。

8.4　随机访问 —— lseek系统调用

输入输出通常按顺序进行：每次调用 read和write进行读写的位置紧跟在前一次操作的位置之后。但是，有时候需要以任意顺序来访问文件，系统调用 lseek可以在文件中任意移动位置而不读写任何数据。例如，系统调用

```
long lseek(int fd, long offset, int origin);
```

将文件描述符为fd的文件的当前位置设置为offset，移动位置相对于origin而定。随后进行的读写操作从此位置开始。origin的值可以为0、1或2，分别用于指定offset从文件开始、从当前位置开始或从文件结束处开始算起。例如，为了向一个文件后添加内容（在UNIX外壳程序中用重定向符>>或在系统调用fopen中使用变元"a"），那么在写操作之前首先必须先用如下系统调用找到文件的末尾：

```
lseek(fd, 0L, 2);
```

若要返回文件的开始处（反卷），则可以用调用：

```
lseek(fd, 0L, 0);
```

请注意，变元0L也可写为(long)0，或仅仅写为0，只要在系统调用lseek中的说明得当即可。

在使用lseek系统调用时，可以将文件视为一个大数组，其代价是访问速度慢了许多。例如，下面的函数将从文件的任意位置读入任意数目的字节，它返回所读入的数目，若发生错误则返回-1。

```
#include "syscalls.h"

/*get: 从pos位置处读入n个字节*/
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0)    /*移动到位置pos处*/
        return read(fd, buf, n);
    else
        return -1;
}
```

lseek系统调用返回一个long类型的值，此值说明了文件的新位置，若发生错误则返回-1。标准库函数fseek与系统调用lseek类似，区别在于前者的第一个变元是FILE*类型，且当发生错误时返回一个非0值。

8.5 实例——fopen和getc函数的一种实现方法

下面通过一个例子来说明如何把这些系统调用放在一起使用。这个例子是标准库函数fopen和getc的一种实现方法。

回顾在标准库中，文件不是通过文件描述符，而是通过文件指针来描述的。文件指针是一个指向包含文件各种信息的结构的指针，该结构包含有：一个指向缓冲区的指针（通过它可以读入文件的一块内容）、一个计数器（记录缓冲区中剩余的字符数）、一个指向缓冲区中下一个字符的指针、文件描述符以及用于描述读写模式、错误状态的标志等等。

描述文件的数据结构包含在头文件<stdio.h>中，任何需要使用标准输入输出库中函数的程序都必须在源文件中包含它（通过#include指令说明）。此文件也被库中其他函数包含。以下代码摘自一个典型的<stdio.h>文件。其中那些可能会被库中其他函数使用的名字将以下划线开始，因此一般不会与用户程序中的名字冲突，这种命名习惯也适用于其他标准库函数。

```
#define NULL         0
#define EOF          (-1)
```

```
#define BUFSIZ       1024
#define OPEN_MAX     20     /*一次最多可打开的文件数*/

typedef struct _iobuf {
    int cnt;         /*左边的字符数*/
    char *ptr;       /*下一个字符的位置*/
    char *base;      /*缓冲区的位置*/
    int flag;        /*文件访问模式*/
    int fd;          /*文件描述符*/
} FILE;
extern IFLE _iob[OPEN_MAX];

#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])

enum _flags {
    _READ  = 01,     /*以读方式打开文件*/
    _WRITE = 02,     /*以写方式打开文件*/
    _UNBUF = 04,     /*没有缓冲*/
    _EOF   = 010,    /*已到文件的末尾*/
    _ERR   = 020,    /*发生了错误*/
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)   (((p)->flag & _EOF ) != 0)
#define ferror(p) (((p)->flag & _ERR ) != 0)
#define fileno(p) ((p)->fd)

#define getc(p)   (--(p)->cnt >= 0 \
                   ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 \
                   ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()  getc(stdin)
#define putchar(x) putc((x), stdout)
```

宏getc一般先将计数器减1，将指针移到下一个位置，然后返回字符（一个长的 #define 定义可用反斜杠分成几行）。如果计数值变为负值，getc调用函数_fillbuf来填充缓冲区，重新初始化结构的内容，并返回一个字符。返回的字符为unsigned类型，以确保所有的字符为正值。

程序中给出了putc函数的定义，以表明它的操作与getc函数是一样的，当缓冲区满时调用函数_flushbuf。但本章不讨论putc函数的细节。此外，还给出了访问错误输出、文件结束状态和文件描述符的宏。

下面可以开始编写函数fopen。fopen函数的主要操作是打开文件、文件定位以及设置有关标

记以指示相应的状态。它不分配缓冲区空间；缓存的分配是在第一次读文件时由函数 _fillbuf完成的。

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666    /*对所有者、所有者组和其他成员都可读写*/

/*fopen: 打开文件，并返回文件指针*/
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 & (_READ | _WRITE)) == 0)
            break;    /*寻找一个空闲槽*/
    if (fp >= _iob + OPEN_MAX)    /*没有空闲槽*/
        return NULL;

    if (*mode == 'w')
        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)    /*不能访问名字*/
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

这个版本的fopen函数没有涉及标准C的各种访问方式，但将这些方式加进去并不需要增加多少代码。尤其是它不能识别表示二进制访问的 b标志，因为在UNIX系统中这个方式没有定义。同时它也不能识别允许同时进行读和写的 +标志。

对于某一具体的文件，第一次调用 getc函数时会发现计数值为 0，从而需要调用一次函数 _fillbuf。如果_fillbuf发现文件不是以读方式打开的，那么它立即返回 EOF；否则，它将试图分配一个缓冲区（如果读操作是以缓冲方式进行的话）。

一旦建立了缓冲区，_fillbuf就调用read来填充此缓冲区，设置计数值和指针，并返回缓冲区中的第一个字符。随后进行的 _fillbuf调用会发现缓冲区已分配。

```
#include "syscalls.h"

/*_fillbuf: 分配并填充缓冲区*/
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag&(_READ|_EOF|_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL)    /*还未分配缓冲区*/
         if ((fp->base = (char *) malloc(bufsize)) == NULL)
             return EOF;    /*不能分配缓冲区*/
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}
```

剩下的唯一要做的事是让这些函数开始运行。对于 stdin、stdout**和**stderr**，必须定义和初始化数组**_iob**：**

```
FILE _iob[OPEN_MAX] = {    /*stdin, stdout, stderr*/
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 }
};
```

对文件结构的flag**部分的初始化表明要对** stdin**进行读操作、对**stdont**进行写操作、对**stderr**进行缓冲方式的写操作。**

练习8-2　**用不同的字段来替换显式的位操作，重写** fopen**和**_fillbuf**函数。比较相应代码的大小和执行速度。**

练习8-3　**设计和编写函数**_flushbuf、fflush**和**fclose。

练习8-4　**标准库函数:**

```
int fseek(FLIE *fp, long offset, int origin)
```

类似于函数lseek**，所不同的是**fp**是一个文件指针而不是文件描述符，且返回值是一个** int**类型的状态而不是位置值。编写**fseek**，并确保该函数与库中其他函数所使用的缓冲协调一致。**

8.6 实例 —— 目录显示

我们有时还需要对文件系统进行另一种操作，获得关于一个文件的有关信息，而不涉及文件的具体内容。其中一个例子就是目录显示程序（如 UNIX命令ls），它用于打印一个目录中有关文件的名字以及其他一些信息，如文件大小、访问权限等等。 MS-DOS 操作系统的dir命令也有相似功能。

由于在UNIX中目录就是一种文件，因此 ls只需要读此文件就可获得所有的文件名。但是如果需要获取文件的其他信息，如大小，那么需要使用系统调用。在其他一些系统上，甚至获取文件名也需要系统调用，例如在 MS-DOS系统中即如此。我们所需要的是提供一种独立于系统的访问文件信息的途径，即使其实现依赖于系统。

以下将通过程序 fsize来说明这一点。 fsize程序是ls命令的一个特别形式，它输出命令行变元表中所命名的所有文件的大小。如果其中一个文件是目录，那么 fsize程序将对此目录递归调用自己。如果命令行中没有任何变元，那么 fsize程序处理当前目录。

首先复习一下 UNIX文件系统结构。在 UNIX中目录即文件，它包含了一个文件名列表和一些标记文件位置的信息。“ 位置 ” 是一称为 “ 索引节点表 ” 的索引。文件的索引节点是文件存放除其名字以外所有信息的地方。目录项通常包含两个条目：文件名和索引节点号。

遗憾的是，在不同版本的系统中，目录的格式和确切内容是不一样的。因此为了分离出不可移植的部分，我们把任务分成两部分。外面的一层定义了一个称为 Dirent的结构和三个函数 opendir、readdir和closedir，它们提供独立于系统的对目录项中的名字和索引节点号的访问。我们将利用此界面编写 fsize程序，然后展示如何在与 UNIX V 7和UNIX系统V的目录结构相同的系统上实现这些函数。有变化的部分作为习题。

结构 Dirent包含了索引节点号和目录名。文件名的最大长度由 NAME_MAX设定，NAME_MAX的值由系统而定。 opendir返回一个指向结构的指针，此结构称为 DIR，它与结构 FILE类似。结构指针将被 readdir和closedir用到，所有这些信息放在头文件 dirent.h中。

```
#define NAME_MAX 14    /*最长文件名；依赖于系统*/

typedef struct {    /*可移植的目录项*/
    long ino;       /*索引节点号*/
    char name[NAME_MAX+1];    /*名字加结束符'\0'*/
} Dirent;

typedef struct {    /*最小的DIR：无缓冲等*/
    int fd;       /*目录的文件描述符*/
    Dirent d;    /*目录项*/
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

系统调用 stat以文件名作为变元，返回文件的索引节点中的所有信息；若出错则返回 - 1。

```
char *name;
struct stat stbuf;
int stat (char *, struct stat *);

stat(name, &stbuf);
```

它用文件name的索引节点信息填充结构stbuf。描述stat的返回值的结构包含在头文件<sys/stat.h>中，它的一个典型形式如下所示：

```
struct stat    /*由stat返回的索引节点信息*/
{
    dev_t  st_dev;      /*设备*/
    ino_t  st_ino;      /*索引节点号*/
    short  st_mode;     /*模式位*/
    short  st_nlink;    /*与文件的总的链接数*/
    short  st_uid;      /*所有者的id*/
    short  st_gid;      /*所有者组的id*/
    dev_t  st_rdev;     /*对特殊的文件有用*/
    off_t  st_size;     /*文件的字符数大小*/
    time_t st_atime;    /*上一次访问的时间*/
    time_t st_mtime;    /*上一次修改的时间*/
    time_t st_ctime;    /*上一次索引节点改变的时间*/
}
```

该结构中大部分的值已在注释中做了解释。诸如dev_t和ino_t等类型定义在头文件<sys/types.h>中，源文件必须包含此文件。

st_mode项包含了描述文件的标志，这些标志定义在<sys/stat.h>中。我们只需要处理文件类型的有关部分：

```
#define S_IFMT  0160000    /*文件的类型*/
#define S_IFDIR 0040000    /*目录*/
#define S_IFCHR 0020000    /*特别字符*/
#define S_IFBLK 0060000    /*特别块*/
#define S_IFREG 0010000    /*普通*/

/*...*/
```

下面可以开始编写程序fsize。如果由stat调用获得的模式说明某文件不是一个目录，那么文件的大小信息实际上已经得到，可以立即输出。如果此文件是一个目录，那么必须一个文件一个文件地处理。由于该目录可能包含子目录，因此处理过程是递旧的。

主程序（函数）处理命令行变元，它将所有变元传递给函数fsize。

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>        /*读写标志*/
#include <sys/types.h>    /*类型定义*/
#include <sys/stat.h>     /*由stat返回的结构*/
#include "dirent.h"
```

```
void fsize(char *);

/*打印文件大小*/
main(int argc, char **argv)
{
    if (argc == 1 )    /*缺省为当前目录*/
       fsize(".");
    else
       while (--argc > 0)
           fsize(*++argv);
    return 0;
}
```

函数fsize用于打印文件大小。如果此文件是一个目录， fsize首先调用dirwalk函数来处理它所包含的所有文件。注意文件 <sys/stat.h>中的标记名字S_IFMT和S_IFDIR的使用，它们被用来判定文件是不是一个目录。括号是必须的，因为 &的优先级低于==。

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/*fsize: 打印文件name的大小*/
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("%8ld %s\n", stbuf.st_size, name);
}
```

函数dirwalk是通用的，对目录中的每个文件应用一次。它首先打开目录，依次处理其中的每个文件，然后关闭目录返回。因为 fsize函数对每个目录都要调用 dirwalk函数，所以这两个函数是递归调用的。

```
#define MAX_PATH 1024

/*dirwalk: 对dir中的所有文件应用函数fcn*/
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: can't open %s\n", dir);
```

```
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
          || strcmp(dp->name, "..") == 0)
            continue;    /*跳过自己和父目录*/
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
            fprintf(stderr, "dirwalk: name %s / %s too long\n",
                dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

对readdir的调用将返回一个指向下一个文件的指针。如果目录中已没有待处理的文件，那么该函数返回 NULL。每个目录都包含自己和父目录的项 . 和 .. ，在处理时必须略过它们，否则将会导致无限循环。

该层次之下的代码与目录的格式不相关。下一步要做的就是在某个具体的系统上提供一个opendir、readdir和closedir的最小版本。以下的函数针对于 UNIX V7和系统V，它们使用了头文件<sys/dir.h>中的目录信息。

```
#ifndef DIRSIZ
#define DIRSIZ 14
#endif
struct direct    /*目录项*/
{
    ino_t d_ino;    /*索引节点号*/
    char d_name[DIRSIZ];    /*此名字不包含'\0'*/
}
```

某些版本的系统允许有更长的文件名和更复杂的目录结构。

类型ino_t是用typedef定义的类型，它描述了在索引节点表中的索引。在我们通常使用的系统上此类型为 unsigned short，但是这种信息不应放在程序中。因为在不同的系统上可能不同，所以用typedef定义要好一些。所有的“ 系统 ”类型可在文件 <sys/types.h>中找到。

opendir函数首先打开目录，验证此文件是否是一个目录（调用系统调用 fstat，它与stat类似，不同的是它以文件描述符作为变元），然后分配一个目录结构，并保存信息：

```
int fstat(int fd, struct stat *);

/*opendir: 打开目录供函数readdir使用*/
DIR *opendir(char *dirname)
{
    int fd;
```

```
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
      || fstat(fd, &stbuf) == -1
      || (stbuf.st_mode & S_IFMT) != S_IFDIR
      || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
      return NULL;
    dp->fd = fd;
    return dp;
}
```

closedir函数用于关闭目录文件并释放内存空间：

```
/*closedir: 关闭由opendir打开的目录*/
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

最后，函数readdir用read系统调用读出每个目录项。如果一个目录槽当前没有使用（因为删除了一个文件），且它的索引节点号为 0，那么将略过此位置。否则，将索引节点号和目录名字放在static结构中，并返回给用户一个指向此结构的指针。每次调用 readdir函数将覆盖前一次调用所获得的信息。

```
#include <sys/dir.h>    /*局部目录结构*/

/*readdir: 按顺序读取目录项*/
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf;    /*局部目录结构*/
    static Dirent d;    /*返回可移植的结构*/

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
                == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0)    /*目录槽未使用*/
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0';    /*确保程序终止*/
        return &d;
    }
    return NULL;
}
```

虽然fsize程序不通用，但是它例示了一些重要的概念。首先许多程序不是“ 系统程序 ”，它

们仅仅使用由操作系统维护的信息。对于这些程序，重要的是信息表示仅出现在标准头文件中，而且它们仅需包含这些头文件而不必包含相应的说明。其次，可以谨慎地创建一个依赖于系统的对象的界面，但此界面是独立于系统的。标准库中的函数即很好的例子。

习题8-5　修改fsize程序，输出包含在索引节点中的其他信息。

8.7　实例 —— 存储分配程序

第5章已给出了一个功能有限的面向栈的存储分配程序，本节将要编写的版本将不受限制，可以以任意次序调用 malloc和free。malloc在必要时调用操作系统以获取更多的存储空间。这些程序演示了在用一种相对独立于系统的方法编写依赖于系统的代码时应考虑的问题，同时展示了结构、联合和typedef的应用。

malloc不是分配一个编译时已知的固定大小的数组，而是当需要时向操作系统申请空间。由于程序中的其他地方需要申请空间，但并不一定通过调用 malloc实现，因此由malloc管理的空间不一定是连续的。因此空闲的存储空间以空闲块列表的方式保存，每个块包含有一个长度、指向下一块的指针和自己的存储空间。这些块以地址的上升序排列在一起，最后一块（最高地址）指向第一块（见图8-1）。

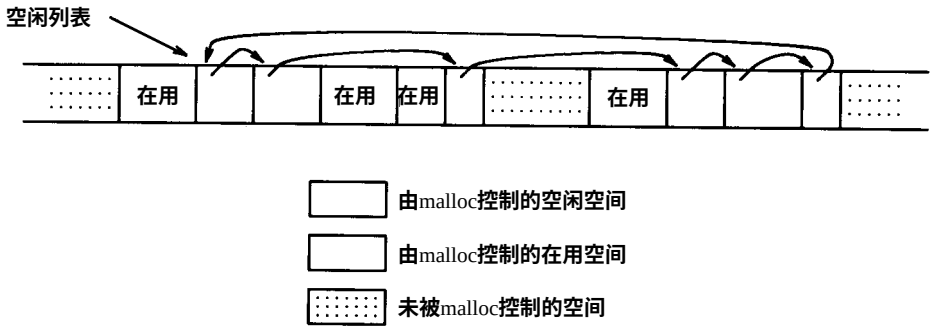

图　8-1

当有申请请求时，扫描空闲块列表以找到一个足够大的块。这个算法称为“最先匹配”；与之相对的算法是“最优匹配”，它寻找满足条件且最小的块。如果这个块刚好与请求的大小相符合，将它从链表中移走并返回给用户。如果这个块太大，那么将它分成两部分：大小合适的块返回给用户，剩下的部分留在空闲块列表中。如果找不到一个足够大的块，那么就从操作系统中申请一个大块加入空闲块列表中。

释放过程也是首先搜索空闲块列表，以找到可以插入被释放块的合适位置。如果被释放块的任一端是一个空闲块，那么将这两个块合成一个更大的块，以确保不会有太多的碎片。因为空闲块列表是以地址的递增顺序链接在一起的，所以很容易判断相邻的块是否空闲。

第5章曾指出了这样一个问题，即确保由 malloc函数返回的存储空间与将要保存的对象能合适地对齐。虽然机器类型各有不同，但是在每一个特定的机器上都有一个最受限的类型，如果此类型可以存储在一个特定的地址中，那么所有其他的类型也可以存放在此地址中。在某些机

器上，最受限的类型是double类型；而在另一些机器上，最受限的类型是 int或long类型。

一个空闲块包含一个指向链表中下一个块的指针、一个关于该块大小的记录和空闲空间本身。位于块的开始处的控制信息称为“ 头部 ”。为了简化块的对齐，所有块的大小都是头部大小的整数倍，且头部已合适地对齐。这是通过一个联合实现的，该联合包含了所期望的头部结构和一个最受限制的对齐类型，在下面这段程序中，我们假定它为 long类型：

```
typedef long Align;             /*以long类型的边界对齐*/
union header {                  /*块的头部：*/
    struct {
        union header *ptr;      /*空闲块列表中的下一块*/
        unsigned size;          /*本块的大小*/
    } ;
    Align x;                    /*强制块的对齐*/
};

typedef union header Header;
```

在这个联合中，Align字段永远不会被使用，它仅仅用于强制每个头部在最坏情况下能够对齐。

在malloc函数中，所请求的以字符为单位的大小将被舍入，以使它是以头部大小为单元的整数倍。实际分配的块将多包含一个单元，用于头部本身；这个值被记录在头部的 size字段中。由malloc函数返回的指针指向空闲空间，而不是块的头部。用户可对获得的存储空间进行任何操作，但是如果在所分配的存储空间之外写入数据，那么可能会破坏块的列表。图 8-2表示由malloc返回的块。

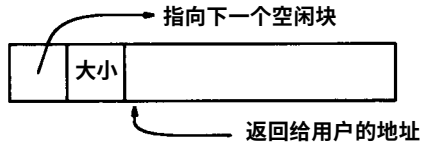

图 8-2

size字段是必需的，因为由 malloc函数控制的块不一定是连续的 —— 我们不可能通过指针算术运算来求这个大小。

变量base表示空闲块的开始部分。如果 freep为NULL，就像第一次调用 malloc函数一样，那么构造一个退化的空闲块列表，它只包含一个大小为 0的块，且这个块指向它自己。无论是哪种情况，都将搜索空闲块列表。搜索从上一次找到空闲块的地方（ freep）开始。这个策略使得列表是同构的。如果找到的块太大，那么把其尾部返回给用户，从而初始块的头部只需要改变 size字段即可。无论在哪种情况下，返回给用户的指针均指向空闲存储空间，即比指向头部的指针大一个单元。

```
static Header base;     /*从空列表开始*/
static Header *freep = NULL;     /*空闲列表的开始指针*/
/*malloc：通用存储分配函数*/
```

```
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {    /*还没有空闲列表*/
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {    /*足够大*/
            if (p->s.size == nunits)  /*正好*/
                prevp->s.ptr = p->s.ptr;
            else {    /*分配末尾部分*/
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)    /*回到空闲列表处*/
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /*没有剩余的存储*/
    }
}
```

函数morecore用于向操作系统请求存储空间，其实现细节因系统的不同而不同。因为向系统请求存储空间是一个代价相对大的操作，所以我们不希望对每个malloc函数调用都进行这个操作，从而morecore函数请求至少NALLOC个单元。这个较大的块在需要时将被分成较小的块。在设置完size字段之后，morecore函数调用free函数把新申请的内存空间插入到这一区域。

UNIX系统调用sbrk (n)返回的指针指向的空间包含 n个额外的字节。如果没有空间，那么sbrk调用返回-1，尽管返回NULL可能是一个更好的设计。-1必须强制转换成char类型，以便与返回值进行比较。而且，强制转换使得该函数不会受不同机器上的指针表示的细节的影响。但仍存在一个假设，即由sbrk调用返回的指向不同块的指针可以进行有意义的比较。标准并没有保证这一点，它只允许指向同一个数组的指针进行比较。这样，这个版本的 malloc函数只能在一般指针比较是有意义的机器之间进行移植。

```
#define NALLOC 1024    /*最小申请单元*/

/*morecore: 向系统申请更多的存储空间*/
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)    /*没有空间*/
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up + 1));
    return freep;
}
```

最后只剩下free。它从freep所指的地址开始，逐个扫描空闲块列表，寻找可以插入空闲块的地方。这个地方可能在两个自由块之间，也可能在列表的末尾。在任一种情况下，如果被释放的块与另一空闲块相邻，那么将这两个块合并起来。唯一的麻烦是让指针指向合适的地方和设置正确的大小。

```
/*free: 将块ap放入空闲块列表中*/
void free(void *ap)
{
    Head *bp, *p;

    bp = (Header *)ap - 1;    /*指向块头*/
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
           break;    /*块freed在列表的开始或末尾*/

    if (bp + bp->s.size == p->s.ptr) {    /*与上一相邻块合并*/
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {    /*与下一相邻块合并*/
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

虽然存储分配处理固有地依赖于具体的机器，但是以上的代码显示了如何控制依赖于机器的部分并将它们组合成一个部分。 typedef和union的使用解决了地址的对齐（假设 sbrk提供了合适的指针）。类型的强制转换使得指针的转换是显式地进行的，而且甚至可以解决设计不好的系统界面问题。虽然在这里细节只涉及存储分配，但是这种方法是通用的，可以应用在别的情

况下。

练习8-6　标准库函数calloc(n,size)返回一个指向n个大小为size的对象的指针，且所有分配的存储空间的值被初始化为0。编写calloc函数，通过调用或修改malloc来实现。

练习8-7　malloc接收任意大小的请求，而不检查它的合理性，而free则认为被释放的块具有合法的大小字段。改进这些函数，使它们有错误检查的能力，从而增加它们的健壮性。

练习8-8　编写函数bfree(p,n)，释放一个包含有n个字符的任意块p，并将它增加到由malloc和free维护的空闲块列表中。通过使用bfree，用户可以在任意时刻增加一个静态或外部数组到空闲块列表中。

附录A 参考手册

A.1 引言

本手册描述的C语言是1988年10月31日提交给ANSI的草案，批准号为“美国国家信息系统标准——C程序设计语言，X3.159-1989”。尽管我们已非常小心，以使这个手册的介绍可以信赖，但它毕竟不是标准本身，而是对标准的一个解释。这个手册的安排基本与标准相似，也与本书的第1版相似，但是对细节的组织是不同的。本手册给出的语法与标准是一样的，只是有少量产生式有所修改，词法元素和预处理器的定义也非形式化。注释部分说明了ANSI标准C与本书第1版介绍的或其他编译器所支持的语言的细微差别。

A.2 词法规则

一个程序由存储在文件中的一个或多个翻译单元组成，程序的翻译分几个阶段完成，这将在A.12节中介绍。翻译的第一阶段完成低级的词法转换，执行由字符#开始的行所引入的指令，并进行宏定义和宏扩展。当预处理（将在A.12节中介绍）完成后，程序就被归约成一个单词序列。

A.2.1 单词

共有6类单词：标识符、关键字、常量、字符串字面值、运算符和其他分隔符。空格、横向和纵向制表符、换行符、换页符和注解（合称空白符）在程序中仅用来分隔单词，因此将被略过。空白符用来分开相邻的标识符、关键字和常量。

如果到某一字符为止的输入流被分成若干单词，那么下一个单词就是可能组成单词的最长的字符串。

A.2.2 注解

注解以字符/*开始，以*/结束。注解不可以嵌套，也不可以出现在字符串中或字符字面值中。

A.2.3 标识符

标识符是一个字母和数字的序列，其第一个字符必须是一个字母，下划线_也被

当做字母。大写和小写字母组成的标识符是不同的。标识符可以任意长。对于内部标识符，至少前31个字母是有意义的，在某些实现中这个值可以更大。内部标识符包括预处理的宏名和其他没有外部连接（见A.11.2节）的名字。有外部连接的标识符的限制要多一些，其实现可能只认为前6个字符是有意义的，而且有可能忽略大小写的不同。

A.2.4 关键字

以下标识符被保留为关键字，它们不能用做别的用途：

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

有些实现还把单词fortran和asm保留为关键字。

注释 关键字const、signed和volatile是ANSI标准中新增加的，enum和void是第1版后新增加的，entry曾经被保留为关键字，但现在已不是了。

A.2.5 常量

共有几种类型的常量，它们每一种都有一个数据类型，基本类型将在A.4.2节讨论。

常量：

整数常量
字符常量
浮点常量
枚举常量

1. 整数常量

整数常量由一串数字序列组成。如果它以0（数字0）开始，那么是八进制数，否则就是十进制数。八进制常量不包括数字8和9。以0x和0X（数字0）开始的数字序列是十六进制数，十六进制数包含到从a~f或从A~F的字母，它们分别表示10~15。

一个整数常量可以以字母u或U为后缀，表示它是一个无符号数；也可以以字母l或L为后缀，表示它是一个长整数。

一个整数常量的类型依赖于它的形式、值和后缀（类型的讨论见A.4节）。如果它没有后缀且是十进制的，那么它的类型很可能是int、long int或unsigned long int。如果它没有后缀且是八进制的或十六进制的，那么它的类型很可能是int、unsigned int、long int或unsigned long int。如果它的后缀为u或U，那么它的类型很可能是unsigned int或unsigned long int。如果它的后缀为l或L，那么它的类型很可能是long int或unsigned long int。

注释 整数常量类型的确定比第1版要详细得多；在第1版中，大的整数常量仅被看做是

long类型的。U后缀是新增加的。

2. 字符常量

字符常量是由单引号括住的一个或多个字符的序列，如 'x' 。单字符常量的值是执行时机器的字符集中的此字符的数值，多字符常量的值由实现定义。

字符常量不包括字符 ' 和换行符，为了表示它们和某些其他的字符，可以使用以下的转义序列（换码序列）：

换行符	NL(LF)	\n	反斜杠	\	\\
横向制表符	HT	\t	问号	?	\?
纵向制表符	VT	\v	单引号	'	\'
回退符	BS	\b	双引号	"	\"
回车符	CR	\r	八进制数	*ooo*	*ooo*
换页符	FF	\f	十六进制数	*hh*	\x*hh*
响铃符	BEL	\a			

转义序列"*ooo*"由反斜杠后跟1、2或3个用来确定对应字符的值的八进制数字组成。一个普通的例子是"\0"（其后没有数字），它表示字符NUL。转义序列"\x*hh*"由反斜杠开始，后跟x，其后是十六进制数字，用来确定对应字符的值。数字的个数没有限制，但如果对应的字符的值超过最大的字符的值，那么该行为是未定义的。对于八进制或十六进制转义字符，如果实现中将类型char看做是有符号的，那么将对字符值进行符号扩展，就好像它被强制转换为 char类型一样。如果"\"后面的字符不是以上所说明的，那么其行为是未定义的。

在C语言的某些实现中，有一个扩展的字符集，扩展的部分不能用 char类型表示。在该扩展集中，常量是由一个前导 L开始（如：L'x'），叫做宽字符常量。这种常量的类型为 wchar_t。这是一个整数类型，定义在标准头文件< stddef.h>中。与通常的字符常量一样，可以使用八进制和十六进制的转义序列；但是，如果值超过 wchar_t可以表示的范围，那么结果是未定义的。

注释 某些转义序列是新增加的，特别是十六进制字符的表示。扩展的字符也是新增加的。通常美国和西欧所用的字符集可以用 char编码；增加wchar_t的主要意图是为了表示亚洲的语言。

3. 浮点常量

一个浮点常量包含有一个整数部分、一个小数点、一个小数部分、一个 e或E，一个可选的有符号整数类型的指数和一个可选的表示类型的后缀（即 f、F、l或L）。整数和小数部分均由数字序列组成。可以没有整数部分或小数部分（但不能二者都没有）。小数点部分或者 e和指数部分可以没有（但不能二者都没有）。浮点常量的类型由后缀确定， F或f后缀表示它是float类型；l或L后缀表明它是long double类型；若没有后缀则是double类型。

注释 浮点常量的后缀是新增加的。

4. 枚举常量

定义为枚举符的标识符是int类型的常量（见A.8.4节）。

A.2.6 字符串字面值

字符串字面值也叫字符串常量，是由双引号括起来的一个字符序列，如 "..."。字符串的类型

为“字符数组”，存储类为static（见A.4节），由给定的字符来初始化。相同的字符串字面值是否看做是不同的取决于具体的实现。如果程序试图改变字符串字面值，那么该行为是未定义的。

我们可以把相邻的字符串字面值连接为一个单一的字符串。任何连接之后，一个空字节 "\0" 被加到字符串的后面以使程序在扫描字符串时知道已到达字符串的末尾。字符串字面值不包含换行符和双引号符；但可以用与字符常量相同的转义序列来表示它们。

与字符常量一样，扩展字符集中字符串字面值以前导字符 L表示，如L"..."。宽字符的字符串字面值的类型为wchar_t的数组，将普通字符串和宽字符的字符串字面值进行连接是未定义的。

注释 如下说明都是ANSI C标准中新增加的：字符串字面值不必相区别、禁止修改字符串字面值以及相邻字符串字面值的连接。宽字符的字符串字面值也是新增加的。

A.3 语法符号

在本手册用到的语法符号中，语法类别由斜体字表示。字面值单词和字符以打字机字体表示。可选类别通常列在不同的行中，但在少数情况下，一长串短的可选项可以表示在一行中，以短语“之一”(one of)标识。任选的终结符或非终结符带有下标 "opt"。例如：

{ 表达式$_{opt}$ }

表示一个括在花括号中的任选的表达式。语法概要在 A.13节中给出。

注释 与本书第1版给出的语法不同，本书给出的语法使表达式运算符的优先级和结合性是显式的。

A.4 标识符的含义

标识符也叫名字，可以指代很多实体：函数，结构标记、联合和枚举，结构或联合的成员，枚举常量，类型定义名字以及对象。一个对象，有时也称为变量，是一个存储区域。它的解释依赖于两个主要属性：它的存储类和它的类型。存储类决定了与该标识的对象相关联的存储区域的生命周期，类型决定了该对象的值的含义。一个名字还有一个作用域和一个连接，作用域即程序中可见此名字的区域，连接决定另一区域中的同一个名字是否指代的是同一个对象或函数。作用域和连接将在 A.11节中讨论。

A.4.1 存储类

存储类共有两类：自动存储类和静态存储类。一个对象说明的几个关键字与上下文一起确定了对象的存储类。自动对象对于一个分程序（见 A.9.3节）来说是局部的，在退出分程序时该对象将消失。如果没有提到存储类说明，或者如果使用了 auto区分符，那么分程序中的说明生成的都是自动对象。说明为register的对象也是自动的，并且（在可能时）存储在机器的快速寄存器中。

静态对象可以局部于某个分程序，或者对所有分程序来说是外部的。不论是哪一种情况，在退出和再进入函数和分程序时其值不变。在一个分程序包括提供函数代码的分程序内，静态对象用关键字 static说明。在所有分程序外部说明的且与函数定义在同一级的对象总是静态的，

可以通过使用static关键字使它们局部于一个特定的翻译单元，这使得它们有内部连接。通过略去显式的存储类或通过使用关键字extern，可以使这些对象全局于整个程序，并有外部连接。

A.4.2 基本类型

有几种基本类型。在附录B中描述的标准头文件<limits.h>定义了在局部实现中每种类型的最大值和最小值。附录B给出的数表示最小的可接受限度。

说明为字符（char）的对象要大到足以存储执行字符集（execution character set）中的任何字符。如果字符集中的某个字符存储在一个char对象中，那么该对象的值等于字符的整数码，并且是非负的。其他量也可存储在char变量中；但其取值范围，特别是其值是否有符号，依赖于具体的实现。

以unsigned char说明的无符号字符与普通字符占用同样的空间，但其值总是非负的。以signed char显式说明的有符号字符也与普通字符占用同样大的空间。

注释 在本书的第1版中没有unsigned char类型，但它的用途很广泛。signed char是新增加的。

除了char类型外，还有3种不同大小的整数类型：short int、int和long int。普通int对象的大小与主机的自然结构一样大，其他大小的整数类型都有特殊的用途。较长的整数至少要占有与较短整数一样的存储空间；但是具体的实现可以使一般整数（int）有与短整数（short int）或长整数（long int）有同样的大小。除非特别说明，整数类型都表示有符号数。

以关键字unsigned说明的无符号整数遵守算术模2n的规则，其中*n*是相应整数表示的位数。这样对无符号数的算术运算永远不会溢出。可以存储在带符号对象中的非负值的集合是可以存储在相应的无符号对象中的值的子集，并且这两个集合的重叠部分的表示是一样的。

单精度浮点数（float）、双精度浮点数（double）和多精度浮点数（long double）中任何类型可能是同义的，但精度由前到后是上升的。

注释 long double是新增加的类型，在第1版中long float与double类型等价，但现在已不再相同。

枚举是具有整型值的一个独特的类型。与每个枚举相关联的是一个有名常量的集合（见A.8.4节）。枚举类型类似于整数类型。但是，如果某个特定枚举类型的对象被赋于的值不是其常量中的一个，或者被赋于的不是一个同类型的表达式，那么枚举类型通常用于编译器以产生警告信息。

因为以上这些类型的对象可以被解释为数字，所以统称它们为算术类型。char类型、int族类型，不论大小如何，是否有符号，都统称为整数类型。类型float、double和long double统称为浮点类型。

void类型说明值的一个空集合，它被用来说明那些不产生任何值的函数的类型。

A.4.3 派生类型

除了基本类型外，我们还可以通过以下几种方法构造派生类型，这些派生类型从概念上说

有无限多个：

给定类型的对象的数组；
返回给定类型的对象的函数；
指向给定类型的对象的指针；
包含一系列不同类型的对象的结构；
包含不同类型的几个对象中任意一个的联合。

一般地，在构造对象时可以递归地使用这些方法。

A.4.4 类型限定符

对象的类型可以有附加的限定符。说明为 const的对象表明此对象的值不可以改变。说明为volatile的对象表明它有与优化相关的特殊属性。限定符既不影响对象的值的范围也不影响其算术属性。限定符将在A.8.2节讨论。

A.5 对象和左值

对象是一个指名的存储区域，左值是指向某个对象的表达式。左值表达式的一个明显的例子是一个有合适类型与存储类的标识符。某些运算符可以产生左值。例如，如果 E是一个指针类型的表达式，那么*E是一个左值表达式，它指代由E指向的对象。名字“左值”来源于赋值表达式E1 = E2，其中左运算分量E1必须是一个左值表达式。对每个运算符的讨论说明了此运算符是否需要一个左值运算分量以及它是否产生一个左值。

A.6 转换

依据运算分量的不同，某些运算符会引起运算分量的值由某个类型转换为另一个类型。本节解释这种转换所产生的结果。A.6.5节将讨论大多数普通运算符所需的转换；对每个运算符的讨论会在需要时做补充。

A.6.1 整提升

在一个表达式中，凡是可以使用整数的地方都可以使用有符号或无符号的字符、短整数和整数的位字段及枚举类型的对象。如果原来类型的所有值都可用 int类型表示，那么原来类型的值就被转换为int类型；否则就被转换为unsigned int类型。这一过程称为整提升。

A.6.2 整数转换

任何整数转换为某个给定的无符号类型的方法是：找出与此整数同余的最小的非负值，其模数为该无符号类型能够表示的最大值加 1。在二进制补码表示中，如果该无符号类型的位模式较窄，那么这就相当于左截取；如果该无符号类型的位模式较宽，那么这就相当于对有符号值进行符号扩展和对无符号值填0。

当任何整数被转换成有符号类型时，如果它可以在新类型中表示出来则其值不变，否则它的值由具体实现定义。

A.6.3 整数和浮点数

当把浮点类型的值转换为整数类型时，其小数点部分将被丢弃掉。如果结果值不能用此整类型来表示，那么其行为是未定义的。特别地，将负的浮点数转换为无符号整类型的结果没有指定。

当把整类型的值转换为浮点类型时，如果该值在该浮点类型可表示的范围内但不能精确表示，那么结果可以是下一个较高的或下一个较低的可表示值。如果该值超过可表示的范围，那么其行为是未定义的。

A.6.4 浮点类型

当一个精度较低的浮点值被转换为有相同或更高精度的浮点类型时，它的值不变。当把一个有较高精度的浮点类型的值转换为精度较低的浮点类型时，如果它的值在可表示范围内，那么结果可以是下一个较高的或下一个较低的表示值。如果结果在范围之外，那么其行为是未定义的。

A.6.5 算术转换

许多运算符都会以相似的方式在运算过程中引起转换并产生结果类型。其效果是将所有运算分量转换为同一类型，并以此作为结果的类型。这种方式的转换称为普通算术转换。

首先，如果一个运算分量为 long double类型，那么另一个也被转换为 long double类型。

否则，如果一个运算分量为 double类型，那么另一个也被转换为 double类型。

否则，如果一个运算分量为 float类型，那么另一个也被转换为 float类型。

否则，同时对两个运算分量进行整提升，然后，如果一个运算分量为 unsigned long int 类型，那么另一个也被转换为 unsigned long int类型。

否则，如果一个运算分量为 long int类型且另一个运算分量为 unsigned int类型，那么结果依赖于long int类型是否可以表示所有的 unsigned int类型的值。如果可以，那么 unsigned int类型的运算分量转换为 long int；如果不可以，那么两个运算分量均转换为 unsigned long int类型。

否则，如果一个运算分量为 long int类型，那么另一个也被转换为 long int类型。否则，如果一个运算分量为 unsigned int类型，那么另一个也被转换为 unsigned int类型。

否则，两个运算分量均为 int类型。

注释 这里有两个变化。第一，对 float运算分量的算术运算可以只用单精度而不是双精度；而在第 1版中指定所有的浮点运算都是双精度。第二，当较短的无符号类型与较长的有符号类型一起运算时，不将无符号类型的属性传递给结果类型；而在第 1版中无符号类型总是处于支配地位。新规则稍微有点复杂，但减少了当无符号数与有符号数混合使用时的麻烦。但当一个无符号表达式与一个同样大小的有符号表达式相比较时仍会得到不期望的结果。

A.6.6 指针和整数

指针值可以加上或减去一个整数类型的表达式，在这种情况下，整数表达式的转换按照对

加法类运算符的讨论进行（见A.7.7节）。

两个指向同一数组中同一类型的对象的指针可以进行减法运算，其结果被转换为整数；转换方式按对减法类运算符的讨论进行（见A.7.7节）。

值为0的整常量表达式或强制转换为类型void *的表达式可通过强制转换、赋值或比较转换为另一种类型的指针。其结果将产生一个空指针，此空指针等于同一类型的另一空指针，但不等于任何指向函数或对象的指针。

某些其他涉及指针的转换也可进行，但其结果依赖于具体的实现。这些转换必须由一个显式的类型转换运算符或强制类型转换（见A.7.5节和A.8.8节）来指定。

指针可以转换为整数类型，只要此类型足够大；所要求的大小依赖于具体的实现。映射函数也依赖于实现。

一个整类型对象可以显式地转换为指针类型。映射总是使一个足够宽的从指针转换来的整数转换回到同一个指针，否则其结果依赖于实现。

指向某一类型的指针可以被转换为指向另一类型的指针，但是如果该指针不指向在存储区域中适当对齐的对象，那么结果指针可能会导致地址异常。指向某对象的指针在转换成一个指向其类型有更少或相同的存储对齐方式的限制的对象时，可以保证原封不动地再转换回来时。“对齐”的概念依赖于实现，但char类型的对象有最少的对齐限制。如将在A.6.8节中讨论的，指针也可以转换为void *类型，并可原封不动地转换回来。

一个指针可以转换为同样类型的另一个指针，除了增加或删除该指针所指的对象类型的限定符（见A.4.4节和A.8.2节）。如果增加了限定符，那么新指针与原指针等价，不同的是多了由限定符带来的限制。如果删除了限定符，那么对基本对象的运算仍受它实际说明中的限定符的限制。

最后，指向一个函数的指针可以转换为指向另一个函数类型的指针，调用转换后指针所指的函数的效果依赖于实现。但是，如果转换后的指针被重新转换为原来的类型，则结果与原来的指针一致。

A.6.7 空类型void

一个void对象的（不存在的）值不可以以任何方式使用，也不能被显式或隐式地转换为一非空类型。因为一个空表达式表示一个不存在的值，这样的表达式只可使用在不需要值的地方，例如作为一个表达式语句（见A.9.2节）或作为逗号运算符的左运算分量（见A.7.18节）。

可以通过强制类型转换将表达式转换为void类型。例如，在表达式语句中一个空的强制类型转换将丢掉函数调用的返回值。

注释　void没有在本书的第1版中出现，但是从本书第1版出版后，就一直被广泛使用着。

A.6.8 指向空类型void的指针

指向任何对象的指针可以被转换为void *类型而不会丢失信息。如果将结果再转换为初始指针类型，那么初始指针被恢复。与在A.6.6节中讨论的、一般需要显式的强制转换的指针到指针

的转换不同，指针可以被赋值为 void *类型指针，也可以赋值给 void *类型指针，并和 void *类型指针进行比较。

注释 对void *指针的解释是新增加的，以前 char *指针扮演通用指针的角色。ANSI标准特别允许void *类型指针和其他对象指针在赋值和关系表达中混用，而对其他的指针的混合使用则要求有显式的类型转换。

A.7 表达式

表达式运算符的优先级与本节中各小节的先后次序相同，即最高优先级的运算符最先介绍。例如，作为加法运算符 +（见A.7.7节）的运算分量的表达式是在 A.7.1节至A.7.6节定义的那些表达式。在每一小节中，各个运算符具有相同的优先级。在每个小节中也讨论了该节所讨论的运算符的左、右结合律。A.13节给出的语法结合了运算符的优先级和结合律。

运算符的优先级和结合律是明确规定的，但是表达式的求值次序除少数例外情况外是没有定义的，尽管子表达式会有副作用。也就是说，除非一个运算符的定义保证了其运算分量以一特定顺序求值，否则具体的实现可以自由地选择任一求值次序，甚至可以交替求值。但是，每个运算符以与它所出现的表达式的句法分析兼容的方式将其运算分量产生的值结合起来。

注释 这个规则取消了以前具有在数学上满足交换律和结合律的运算符的表达式可以任意排列的自由，但可能会在计算时不满足结合律。这个改变仅影响浮点数在接近其精确度限度的计算以及可能发生溢出的情况。

C语言没有定义在表达式求值过程中的溢出、除法检查和其他异常的处理。大多数现有 C语言的实现在进行有符号整数表达式的求值时以及在赋值时忽略溢出异常，但并不是所有实现都这样做。对除数为 0和所有浮点异常的处理，不同的实现有不同的方式，有时候可以用非标准库函数进行调整。

A.7.1 指针生成

对于某类型 *T*，如果某表达式或子表达式的类型为“ *T*的数组 ”，那么此表达式的值是指向数组中第一个对象的指针，并且此表达式的类型被转换为“ 指向 *T*的指针 ”。如果此表达式是一元运算符&、++、--或sizeof的运算分量，或是赋值类运算符或圆点 . 运算符的左运算分量，那么转换不会发生。类似地，类型为“ 返回 *T*的函数 ”的表达式被转换为类型“ 指向返回 *T*的函数的指针 ”，除非此表达式被用作&运算符的运算分量。

A.7.2 初等表达式

初等表达式是标识符、常量、字符串或带括号的表达式。

初等表达式：
- 标识符
- 常量
- 字符串

(表达式)

一个标识符只要是按下面所讨论的方式适当说明的就是初等表达式。其类型由说明指定。如果一个标识符指定一个对象（见 A.5节）且其类型是算术、结构、联合或指针类型，那么它是一个左值。

一个常量是一个初等表达式，其类型依赖于它的形式，见 A.2.5节的讨论。

一个字符串字面值是一个初等表达式。它的初始类型是 char数组类型（对于宽字符字符串，则为wchar_t数组类型），但遵循A.7.1节给出的规则。它通常被修改为指向 char类型（wchar_t类型）的指针，从而结果是指向字符串中第一个字符的指针。在一些初始化程序中不能进行这样的转换，详见A.8.7节。

用括号括起来的表达式是一个初等表达式，它的类型和值与无括号的表达式一致。此表达式是否是左值不受括号的影响。

A.7.3 后缀表达式

后缀表达式中的运算符遵循从左到右的结合规则。

后缀表达式：
  初等表达式
  后缀表达式 [表达式]
  后缀表达式 (变元表达式表$_{opt}$)
  后缀表达式 . 标识符
  后缀表达式 -> 标识符
  后缀表达式 ++
  后缀表达式 - -

变元表达式表：
  赋值表达式
  变元表达式表, 赋值表达式

1. 数组引用

带下标的数组由一个后缀表达式后跟一个括在方括号中的表达式来表示。这两个表达式中要有一个的类型必须为“指向 *T*的指针”，其中*T*是某种类型；另一个表达式的类型必须为整数。下标表达式的类型为 *T*。表达式E1[E2]，在定义上等同于 *((E1)+(E2))。有关数组引用的更多讨论见A.8.6节。

2. 函数调用

函数调用由一个后缀表达式（称为函数命名符）后跟由圆括号括起来的包含一个可能为空的、由逗号分隔的赋值表达式表组成，这些表达式就是函数的变元。如果后缀表达式包含一个在当前作用域中不存在的标识符，那么此标识符就被隐式地说明，就好像说明

```
extern int 标识符 ();
```

在包含此函数调用的最内层分程序中被给出一样。这个后缀表达式（在可能的隐式说明和指针生成之后，见A.7.1节）必须有类型“指向返回 *T*的函数的指针”，其中*T*为某个类型，且函数调

用的值的类型为*T*。

注释 在第1版中，这个类型被限制为函数类型，并且在通过指向函数的指针来调用此函数时必须有一个显式的*运算符，ANSI C标准允许现存的一些编译程序用同样的语法来进行函数调用和通过指向函数的指针来进行函数调用。旧的语法仍然可用。

术语变元用来表示传递给函数调用的表达式，而术语参数则用来表示由函数定义或函数说明所接收的输入对象（或其标识符），通常也可用术语“实际变元（参数）”和“形式变元（参数）”来区分它们。

在准备调用函数时，要对它的每个变元进行复制，所有的变元传递严格地按值进行。函数可能会改变其参数对象的值（即变元表达式值的拷贝），但这个改变不会影响变元的值。然而，可以将指针作为变元传递，以使函数可以改变指针所指向的对象的值。

函数可以用两种方式说明。在新的方式中，参数的类型是作为函数类型的一部分显式指定的，这种说明称为函数原型。在旧的方式中，参数类型没有说明。函数说明在A.8.6节和A.10.1节讨论。

如果在一个函数说明的作用域中函数是以旧方式说明的，那么按以下方式对每个变元进行缺省变元提升：对每个整类型变元进行整提升（见 A.6.1节），将每个float类型的变元转换为double类型。如果调用时变元的数目与函数定义中参数的数目不等，或者某个变元的类型提升后与相应的参数类型不一致，那么函数调用的结果是未定义的。类型一致性依赖于函数定义是以新方式进行的还是以旧方式进行的。如果是旧方式的，那么类型一致性检查将在提升过的调用的变元类型和提升过的参数类型之间进行；如果定义是新方式的，那么提升过的变元类型必须与没有提升过的参数本身的类型一致。

如果在函数调用的作用域中函数说明是以新方式进行的，那么变元将被转换为函数原型中的相应参数类型，就像是赋值一样。变元数目必须与显式说明的参数数目相同，除非函数说明的参数表以省略号（，...）结束。在这种情况下，变元的数目必须等于或超过参数的数目；其后无显式指定类型的参数与之对应的变元要进行缺省的变元提升，如前面段落中所述。如果函数定义是以旧方式进行的，那么在调用中可见的原型中的每个变元类型必须与相应函数定义中的参数类型一致（函数定义中的参数类型已进行过变元提升）。

注释 这些规则特别复杂，因为必须要考虑到新旧方式函数的混合使用。应尽可能避免新旧方式混合使用。

变元的求值次序没有指定。不同的编译器的实现方式各不相同。然而，在进入函数前变元和函数命名符是完全求值的，包括所有的副作用。对任何函数都可以进行递归调用。

3. 结构引用

一个后缀表达式后跟一个圆点和一个标识符仍是一个后缀表达式。第一个运算分量表达式的类型必须是一个结构或联合，标识符必须是结构或联合的成员名字。结果值是结构或联合的指名的成员，其类型是对应成员的类型。如果第一个表达式是一个左值且第二个表达式的类型不是数组类型，那么整个表达式是一个左值。

一个后缀表达式后跟一个箭头（由-和>组成）和一个标识符仍是一个后缀表达式。第一个运算分量表达式必须是一个指向结构或联合的指针，标识符必须指名结构或联合的一个成员，结

果指向指针表达式所指向的结构或联合的指名成员，结果类型是对应成员的类型。如果成员类型不是数组类型那么整个表达式是一个左值。

这样，表达式E1->MOS与(*E1).MOS等价。结构和联合将在A.8.3节讨论。

注释　在本书的第1版中，已经规定了在这样的表达式中，成员的名字必须属于后缀表达式所指定的结构或联合，但是这个规则并没有强制实行。最新的编译程序和 ANSI强制规定这一点。

4. 后缀加一与减一运算符

一个后缀表达式后跟一个++或--运算符仍是一个后缀表达式。表达式的值是运算分量的值。当执行完此表达式后，运算分量的值加 1（++）或减1（--）。这个运算分量必须是一个左值。对运算分量的限制和运算细节的详细讨论见加法类运算符（ A.7.7节）和赋值类运算符（ A.7.17节）。其结果不是左值。

A.7.4　一元运算符

表达式中的一元运算符遵循从右到左的结合原则。

```
一元表达式：
  后缀表达式
  ++ 一元表达式
  -- 一元表达式
  一元运算符 强制转换表达式
  sizeof 一元表达式
  sizeof (类型名)

一元运算符：任意一个
  & * + - ~ !
```

1. 前缀加一与减一运算符

以运算符++或--为前缀的一元表达式仍是一个一元表达式。运算分量将被加 1（++）或减1（--），整个表达式的值是经过加减以后的值。该运算分量必须是一个左值。对运算分量的限制和运算细节的讨论详见加法类运算符（见A.7.7节）和赋值类运算符（见A.7.17节）。结果不是左值。

2. 地址运算符

一元运算符&用于计算运算分量的地址。该运算分量必须是一个既不能指向位字段，也不能指向说明为register的对象的左值或函数类型。结果值是一个指针，指向由左值所指的对象或函数。如果运算分量的类型为 *T*，那么结果的类型为指向 *T*的指针。

3. 间接寻址运算符

一元*运算符表示间接寻址，它返回其运算分量所指向的对象或函数。如果它的运算分量是一个指针且所指向的对象是算术、结构、联合或指针类型，那么它是一个左值。如果表达式的类型为“ 指向*T*的指针 ”，那么结果类型为*T*。

4. 一元加运算符

一元+运算符的运算分量必须是算术类型，其结果是运算分量的值。如果运算分量是整类型，

那么就要进行整提升，结果类型是经过提升后的运算分量的类型。

注释　一元+运算符是ANSI C标准新增加的，增加它是为了与一元-运算符对称。

5. 一元减运算符

一元-运算符的运算分量必须是算术类型，结果为运算分量的负值。如果运算分量是整类型，那么就要进行整提升。有符号数的负值的计算方式为：将提升所得到的类型中的最大值减去提升过的运算分量的值，然后加1；但0的负值仍为0。结果类型为提升过的运算分量的类型。

6. 二进制求反运算符

一元~运算符的运算分量必须是整类型，结果为运算分量的二进制反码。在运算过程中要对运算分量进行整提升。如果运算分量是无符号类型的，那么结果是通过由提升后的类型的最大值减去运算分量的值得到的值。如果运算分量是有符号的，那么结果的计算方式为：将提升后的运算分量转换为相应的无符号类型，进行二进制求反运算，再将结果转换为有符号类型。结果的类型为提升后的运算分量的类型。

7. 逻辑非运算符

运算符!的运算分量必须是算术类型或是一个指针。如果运算分量等于 0，那么结果为1，否则结果为0。结果类型为int。

8. sizeof运算符

sizeof运算符用于求存储其运算分量类型的对象所需要的字节数。运算分量或者为一个未求值的表达式，或者为一个由括号括起的类型名字。当 sizeof被用于char类型时，其值为1；当用于数组时，其值为数组中字节的总数。当用于结构或联合时，结果是对象中的字节数，包括任何使对象平铺为数组所需要的填充空间：有n个元素的数组的大小是一个元素大小的n倍。此运算符不能用于函数类型和不完全类型的运算分量，也不能用于位字段。结果是一个无符号整形常量，具体的类型由实现定义。在标准头文件<stddef.h>（见附录B）中，这一类型被定义为size_t类型。

A.7.5　强制转换

以括号括起来的类型名开头的一元表达式将导致表达式的值被转换为指名的类型。

强制转换表达式：
　一元表达式
　(类型名字) 强制转换表达式

这个结构称为强制转换。类型名字将在A.8.8节描述。转换的结果已在A.6节讨论过。包含强制转换的表达式不是左值。

A.7.6　乘法类运算符

乘法类运算符*、/和%遵循从左到右的结合规则。

乘法类表达式：
　强制转换表达式
　乘法类表达式 * 强制转换表达式
　乘法类表达式 / 强制转换表达式
　乘法类表达式 % 强制转换表达式

*和/的运算分量必须为算术类型，%的运算分量必须为整类型。对这些运算分量要进行常规算术转换，并预测结果类型。

二元运算符*表示乘法。

二元运算符/求得第一个运算分量被第二个运算分量除所得的商，而运算符 %求得相应的余数。如果第二个运算分量为 0，那么结果没有定义。其他情况下 (a/b)*b + a%b等于a永远成立。如果两个运算分量均为非负，那么余数是非负的且小于除数，否则，仅可保证余数的绝对值小于除数的绝对值。

A.7.7 加法类运算符

加法类运算符+和-遵循从左到右的结合规则。如果运算分量有算术类型，那么要进行常规的算术转换。对于每个运算符有更多的可能类型。

加法类表达式：
  乘法类表达式
  加法类表达式 + 乘法类表达式
  加法类表达式 - 乘法类表达式

运算符+作用的结果为两个运算分量的和。数组中指向一个对象的指针可以和一个任何整类型的值相加，后者将通过乘以所指对象的大小被转换为地址偏移量。相加的和是一个指针，它与初始指针有相同的类型，并指向同一数组中的另一个对象，此对象与初始对象之间有合适的偏移量。因此，如果P是一个指向数组中某个对象的指针，那么表达式 P+1是指向数组中下一个对象的指针。如果相加的和所指定的指针不在数组的范围，且不是数组末尾的第一个位置，那么结果没有定义的。

注释　允许指针指向数组的末尾是ANSI C新增加的，它使得我们可以像通常一样对数组元素建立循环。

运算符-作用的结果是两个运算分量的差值。可以从一个指针减去一个任意整类型的值，该运算的转换规则和条件与加法相同。

如果指向同一类型的两个指针相减，那么结果是一个有符号整类型数，表示所指向的对象之间的偏移量。相邻的对象之间的偏移量为 1。结果的类型依赖于具体的实现，但在标准头文件<stddef.h>中定义为ptrdiff_t。只有当指针指向的对象属于同一数组时，差值才有意义。然而，如果P指向数组的最后一个成员，那么(P+1) -P的值为1。

A.7.8 移位运算符

移位运算符<<和>>遵循从左到右的结合规则。每个运算符的运算分量必须为整类型，并且遵循整提升原则。结果的类型是提升过的左运算分量的类型。如果右运算分量为负值，或者大于等于左运算分量的位数，那么结果没有定义。

移位表达式：
  加法类表达式
  移位表达式　<< 加法类表达式

移位表达式>> 加法类表达式

E1<<E2的值为E1（按位模式解释）左移 E2个位。如果不发生溢出，此值等同于 E1乘以2^{E2}。E1>>E2的值为E1右移E2个位。如果E1为无符号数或为非负值，那么右移等同于 E1除以2^{E2}。其他情况结果由具体实现定义。

A.7.9 关系运算符

关系运算符遵循从左到右的结合规则，但这个规则没有多大作用。 a<b<c在语法分析时被解释为(a<b)<c，并且a<b的值为0或1。

关系表达式：
移位表达式
关系表达式 < 移位表达式
关系表达式 > 移位表达式
关系表达式 <= 移位表达式
关系表达式 >= 移位表达式

当所指定的关系为假时，运算符 <（小于）、>（大于）、<=（小于等于）和 >=（大于等于）均返回0；当关系为真时，它们均返回 1。结果的类型为 int类型。如果运算分量为算术类型，那么要进行通常的类型转换。可以将指向同一类型的对象的指针进行比较（忽略任何限定符），其结果依赖于所指对象在地址空间中的相对位置。指针比较只对相同对象的部分有定义：如果两个指针指向同一个简单对象，那么它们的值相等；如果指针指向同一个结构的不同成员，那么指向后说明的成员的指针有较大的值；如果指针指向同一个联合的不同成员，那么它们有相同的值；如果指针指向一个数组的不同成员，那么它们的比较值等于对应的下标的比较值。如果指针P指向数组的最后一个成员，那么 P+1比P大，尽管P+1已指向数组的界外。其他情况指针的比较没有定义。

注释 这些规则允许指向同一个结构或联合的不同成员的指针进行比较，从而放宽了第 1 版所述的限制。这些规则还使得与指向正好超出数组末尾的指针的比较合法化。

A.7.10 相等类运算符

相等类表达式：
关系表达式
相等类表达式 == 关系表达式
相等类表达式 != 关系表达式

运算符==（等于）和!=（不等于）与关系运算符相似，但优先级不同。（任何时候只要a<b与c<d有相同的真值，那么a<b == c<d的值就为1。）

相等类运算符与关系运算符有相同的规则，但这类运算符还允许做如下比较：指针可以与值为0的常量表达式或指向void的指针进行比较。参见A.6.6节。

A.7.11 按位与运算符

按位与表达式：

相等类表达式
按位与表达式 & 相等类表达式

在进行按位与运算时要进行通常的算术转换，结果为运算分量的按位与。该运算符仅适用于整类型运算分量。

A.7.12 按位异或运算符

按位异或表达式：
按位与表达式
按位异或表达式 ^ 按位与表达式

在进行按位异或运算时要进行通常的算术转换，结果为运算分量的按位异或。该运算符仅适用于整类型运算分量。

A.7.13 按位或运算符

按位或表达式：
按位异或表达式
按位或表达式 | 按位异或表达式

在进行按位或运算时要进行通常的算术转换，结果为运算分量的按位或。该运算符仅适用于整类型运算分量。

A.7.14 逻辑与运算符

逻辑与表达式：
按位或表达式
逻辑与表达式 && 按位或表达式

运算符&&遵循从左到右的结合规则。如果两个运算分量都不为 0，那么它返回1，否则返回0。与&不同，&&确保从左到右的求值次序：首先计算第一个运算分量，包括所有的副作用，如果为0，那么整个表达式的值为0。否则计算右运算分量，如果为0，那么整个表达式的值为0；否则为1。

两个运算分量不需是同一类型的，但是每一个运算分量必须为算术类型或者是指针。结果为int类型。

A.7.15 逻辑或运算符

逻辑或表达式：
逻辑与表达式
逻辑或表达式 || 逻辑与表达式

运算符||遵循从左到右的结合规则。如果有一个运算分量不为0，那么它返回1；否则返回0。与|不同，||确保从左到右的求值次序：首先计算第一个运算分量，包括所有的副作用，如果不为 0，那么整个表达式的值为1。否则计算右运算分量，如果不为0，那么整个表达式的值为1；否则为0。

两个运算分量不需是同一类型的，但是每一个运算分量必须为算术类型或者是指针。结果

为int类型。

A.7.16 条件运算符

条件表达式：
逻辑或表达式
逻辑或表达式 ？ 表达式 ： 条件表达式

首先计算第一个表达式(包括所有的副作用)，如果该表达式的值不为0，那么结果为第二个表达式的值，否则结果为第三个表达式的值。第二个和第三个运算分量中只有一个会被计算到。如果第二个和第三个运算分量为算术类型，那么要进行通常的算术转换以使它们有一个共同的类型，这个类型就是结果的类型。如果它们都是void类型，或是同一类型的结构或联合，或是指向同一类型的对象的指针，那么结果的类型为共同的类型。如果其中一个运算分量是指针，而另一个是常量0，那么0被转换为指针类型，并且结果为指针类型。如果一个运算分量为指向 void的指针，而另一个为普通指针，那么另一个指针被转换为指向void的指针，并且这是结果的类型。

在比较指针的类型时，指针所指对象的类型的任何类型限定符（见 A.8.2节）将被忽略，但这些限定符都可被结果的类型继承。

A.7.17 赋值表达式

有几个赋值运算符，它们均从左到右结合。

赋值表达式：
条件表达式
一元表达式 赋值运算符 赋值表达式

赋值运算符：任意一个
= *= /= % = += -= <<= >>= &= ^= |=

所有这些运算符要求左运算分量为左值，并且此左值可以修改，它不可以是数组、不完全类型或函数。同时其类型不能有const限定符，如果它是结构或联合，那么它的任意一个成员或递归子成员不能有const限定符。赋值表达式的类型是其左运算分量的类型，值是赋值发生后存储在左运算分量中的值。

在用=的简单赋值中，用表达式的值替换左值所指向的对象的值。下面几个条件中必须有一个条件成立：或者两个运算分量均为算术类型，在此情况下右运算分量的类型通过赋值转换为左运算分量的类型；或者两个运算分量为同一类型的结构或联合；或者一个运算分量是指针，另一个运算分量是指向 void的指针；或者左运算分量是指针，右运算分量是值为 0的常量表达式；或者两个运算分量均为指向同一类型的函数或对象的指针，除了右运算分量可能没有 const或volatile说明。

形式为E1 *op*= E2的表达式等价于E1=E1 *op* (E2)，唯一的区别是前者E1只求值一次。

A.7.18 逗号运算符

表达式：

赋值类表达式
表达式，赋值表达式

由逗号分开的一对表达式的求值次序为从左到右，并且左表达式的值被丢弃掉。结果的类型和值就是右运算分量的类型和值。在开始计算右运算分量以前，计算左运算分量所带来的副作用将被完成。在逗号有特别含义的上下文中，如在函数变元表（见 A.7.3节）和初始化符表（A.8.7节）中，所要求的语法单元是一个赋值表达式。这样逗号运算符仅出现在一个圆括号组中，例如，函数调用

```
f(a, (t=3, t+2), c)
```

包含有三个变元，其中第二个变元的值为 5。

A.7.19 常量表达式

从语法上看，常量表达式是局限于运算符的某一个子集的表达式：

常量表达式：
条件表达式

一些上下文要求表达式的值为常量，例如：在 switch语句中的case后面、作为数组的边界和位字段的长度、作为枚举常量的值、用在初始化符中以及在某些预处理表达式中。

除非作为sizeof的运算分量，否则常量表达式中可以不包含赋值、增一或减一运算符、函数调用或逗号运算符。如果要求常量表达式为整类型，那么它的运算分量必须由整数、枚举、字符和浮点常量组成。强制类型转换必须指定为整类型，任何浮点常量都被强制转换为整数。此规则必须将数组、间接指向、取地址和结构成员运算排除在外（但是， sizeof可以带任何运算分量）。

初始化符中的常量表达式可以有更大的范围。运算分量可以是任意类型的常量，一元运算符&可以用于外部和静态对象以及以常量表达式为下标的外部和静态数组。一元运算符 &也可以在出现无下标的数组或函数时被隐式地应用。初始化符计算的值必须或者为一个常量，或者为已说明的外部或静态对象的地址与一个常量的和或差。

允许在#if后面的整类型常量的范围较小，不可以是 sizeof表达式、枚举常量和强制转换。参见A.12.5节。

A.8 说明

说明用于指定每个标识符的含义，它们并不需要保留与每个标识符相关的存储空间。保留存储空间的说明称为定义。说明的形式为：

说明：
说明区分符 初始化说明符表$_{opt}$

初始化说明符表中的说明符包含了被说明的标识符；说明区分符由一系列类型和存储类区分符组成。

说明区分符：
存储类区分符 说明区分符$_{opt}$

类型区分符　说明区分符$_{opt}$
类型限定符　说明区分符$_{opt}$

初始化说明符表：
初始化说明符
初始化说明符表，初始化说明符

初始化说明符：
说明符
说明符 = 初始化符

说明符将在稍后讨论（见A.8.5节），它们包含了被说明的名字。一个说明必须包含至少一个说明符，或者其类型区分符必须说明一个结构标记、一个联合标记或枚举的成员。不允许空的说明。

A.8.1 存储类区分符

存储类区分符为：

存储类区分符：
```
auto
register
static
extern
typedef
```

关于存储类的含义已在A.4节讨论过。

区分符auto和register使得被说明的对象有自动存储类，它们仅可用在函数中。这种说明也起着定义的作用，并预留存储空间。带有 register区分符的说明等价于带有 auto区分符的说明，不同的是前者暗示了被说明的对象将被频繁地访问。只有很少的对象被真正放在寄存器中，并且只有特定类型才可以。所受的限制依赖于具体的实现。然而，如果一个对象被说明为 register，那么就不能对它应用一元运算符 &，不论是显式地还是隐式地应用。

注释　计算一个被说明为register而实际为auto的对象的地址是非法的，这是一个新的规则。

区分符static使得被说明的对象具有静态存储类，可以用在函数内或函数外。在函数内，该区分符使得存储空间被分配，起着定义的作用。对于在函数外的效果，参见 A.11.2节。

用在函数内的extern说明用于指明被说明对象的存储空间在别处定义。对于在函数外的效果，见A.11.2节。

typedef区分符没有预留存储空间，之所以称之为存储类区分符，只是为了语法描述上的方便。我们将在A.8.9节讨论它。

在一个说明中最多只能有一个存储类区分符，如果没有存储类区分符被指定，那么就使用如下规则：在函数内说明的对象被认为具有 auto存储类；在函数内说明的函数被认为具有 extern存储类；在函数外说明的对象与函数被认为具有带外部连接的静态存储类。参见 A.10节至A.11节。

A.8.2 类型区分符

类型区分符定义如下：

类型区分符：
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  结构或联合区分符
  枚举区分符
  类型定义名字

在long**和**short**这两个类型区分符中最多有一个可同时与** int**一起说明；在**int**缺省时含义也是一样的。**long**可与**double**一起说明。** signed**和**unsigned**这两个类型区分符中最多有一个可同时与** int**、**int**的**short**和**long**的变种或**char**一起指定。** signed**和**unsigned**可以单独出现，这种情况下默认为**int**。**signed**区分符对强制**char**对象带符号位是非常有用的；对其他整类型也允许带** signed**，但这是多余的。**

除了上面这些情况，在一个说明中至多只能给出一个类型区分符。如果说明中没有类型区分符，则默认为int**。**

类型也可以用限定符限定，以指定被说明对象的特殊性质。

类型限定符：
  const
  volatile

类型限定符可与任何类型区分符一起出现。 const**对象可被初始化，但随后不能再被赋值。**volatile**对象没有独立于实现的语义。**

注释 const**和**volatile**性质是**ANSI**标准新增加的。**const**的作用是声明可以放在只读存储器中的对象，并可能为优化提供机会。** volatile**的作用是使实现屏蔽可能的优化。例如，对于具有内存映像输入/输出的机器，指向设备寄存器的指针可被说明为指向** volatile**的指针，目的是防止编译程序通过指针明显删除多余的引用。除了需要诊断改变** const**对象的明显企图，一个编译程序可能会忽略这些限定符。**

A.8.3 结构和联合说明

结构是由不同类型的有名成员序列组成的对象。联合也是对象，在不同时刻，它含有许多不同类型成员中的任意一个。结构和联合区分符具有相同形式。

结构或联合区分符：
  结构或联合标识符$_{opt}$ {结构说明表}

结构或联合标识符
结构或联合：
struct
union

结构说明表是对结构或联合成员的说明序列：

结构说明表：
结构说明
结构说明表 结构说明
结构说明：
区分符限定符表 结构说明符表；
区分符限定符表：
类型区分符 区分符限定符表$_{opt}$
类型限定符 区分符限定符表$_{opt}$
结构说明符表：
结构说明符
结构说明符表，结构说明符

通常，一个结构说明符就是对结构或联合成员的说明符。结构成员也可能包含指定的位数，这样的成员也叫做位字段，或称为字段，其长度通过跟在说明符后的冒号之后的常量表达式来指定。

结构说明符：
说明符
说明符$_{opt}$：常量表达式

一个形如

结构或联合标识符 { 结构说明表 }

的类型区分符说明了其中的标识符是由结构说明表指定的结构或联合的标记。我们可以在同一作用域或内层作用域内的后续说明中通过在不包含结构说明表的区分符中使用标记来表示同一类型：

结构或联合标识符

如果一个区分符中只有标记而无结构说明表并且没有说明标记，那么该区分符说明了一个不完整类型。具有不完整结构或联合类型的对象可被上下文引用，只要该处不需要知道它们的大小。例如，在指定一个指针或用类型定义新建一个类型名字的说明（而不是定义）中，都可引用不完整类型，其余情况则不允许。在其后如果具有该标记的区分符再次出现并包含了结构说明表，那么该类型就成为完整类型。即使是在包含结构说明表的区分符中，在该结构说明表内所说明的结构或联合类型也是不完整的，一直到花括号 }终止该区分符时，所说明的类型才成为完整类型。

结构中不能包含不完整类型的成员。因此，不能说明包含自身实例的结构或联合。然而，除了可以命名结构或联合类型外，标记还允许定义自引用结构；由于可以说明指向不完整类型的指针，结构和联合可包含指向自身实例的指针。

一个非常特殊的规则适用于如下形式的说明：

结构或联合标识符；

它用于说明结构或联合，但没有说明表和说明符。即使所说明的标识符是在外层作用域已说明过的结构或联合的标记（参见 A.11.1节），该说明仍使得该标识符成为在当前作用域内的一个新的不完整类型的结构或联合的标记。

注释　这是ANSI中的一个新的比较难理解的规则。它旨在处理在内层作用域说明的相互递归调用结构，但这些结构的标记可能已在外层作用域中说明。

具有结构说明表而无标记的结构或联合区分符用于建立一个唯一的类型，它只能被它所在的说明直接引用。

成员和标记的名字不会相互冲突，也不会与普通变量冲突。一个成员名字不能在同一结构或联合中出现两次，但相同的成员名字可在不同的结构或联合中使用。

注释　在本书的第1版中，结构或联合的成员名与其父辈无关联。然而在 ANSI标准制定前，这种关联在编译程序中早已普遍。

一个结构或联合的非位字段成员可以具有任意对象类型。一个位字段成员（它无需说明符，因而可无名）具有类型int，unsigned int或signed int，并且被解释成用位表示其长度的整类型对象。int类型位字段是否要当做有符号数则依赖于实现。结构的相邻位字段成员以依赖于实现的方式被一起放到依赖于实现的存储单元中去。如果在另一位字段之后的某一位字段无法全部存入已被前面位字段部分填充的存储单元中，那么可将它分成两部分存入相邻的存储单元，或者可以填充该单元。我们可以用宽度为 0的无名位字段来强制做这种填充，从而下一位字段将从下一分配单元的边界开始存储。

注释　在处理位字段方面，ANSI标准比第1版更加依赖于实现。为了将位字段存储为无条件地依赖于实现，应当参照这一语言规则。具有位字段的结构可被方便地用来节省存储空间（代价是增加了指令空间和访问字段的时间）。同时，带位字段的结构也可被用来描述在位层次上的存储布局，不过这是一个不方便的方法。在第二种情况下，必须了解局部实现的规则。

结构成员的地址值按它们说明的顺序递增。一个非位字段结构成员根据其类型在地址边界上对齐，因而，在结构中可能会存在无名空穴。若指向一结构的指针被强制转换成指向该结构第一个成员的指针类型，那么结果将指示第一个成员。

联合可以被看成是结构，其所有成员起始偏移量都为 0，并且它的大小足以容纳它的任一成员。任一时刻它至多只能存储其中一个成员。如果指向某一联合的指针被强制转换成指向一个成员的指针类型，那么结果将指向那个成员。

结构说明的一个简单例子如下：

```
struct tnode {
    char tword[20];
    int count;
    struct tnode * left;
    struct tnode * right;
};
```

该结构包含一个具有 20个字符的数组、一个整数以及两个指向类似结构的指针。一旦给出这样

的说明，说明

```
struct tnode s, *sp;
```

就将声明s为所给定类型的结构，sp为所给定类型结构的指针。有了这些说明，表达式

```
sp->count
```

就表示由sp所指向结构的count字段，而

```
s.left
```

就表示结构s的左子树指针；

```
s.right->tword[0]
```

就表示s右子树中tword成员的第一个字符。

我们通常无法检查联合的某一成员，除非已用该成员给联合赋值。然而，有一个特殊的情况可以简化联合的使用：如果一个联合包含有共享一个公共初始序列的若干结构，并且该联合当前包含有这些结构中的某一个，那么引用这些结构中任一结构的公共初始部分是允许的。例如，下述这段程序是合法的：

```
union {
      struct{
           int type;
      } n;
      struct {
           int type;
           int intnode;
      } ni;
      struct {
           int type;
           float floatnode;
      } nf;
   } u;
   ...
   u. nf.     type = FLOAT;
   u.nf.floatnode = 3.14;
   ...
   if (u.n.type == FLOAT)
      ... sin(u.nf.floatnode) ...
```

A.8.4 枚举

枚举类型是这样一种特殊的类型，它的值包含在一个有名常量集合中，这些常量就叫做枚举符（枚举常量）。枚举区分符的形式借鉴了结构和联合区分符的形式。

枚举区分符：
  枚举标识符$_{opt}$ {枚举符表}
  枚举标识符
枚举符表：
  枚举符

枚举符表，枚举符
枚举符：
标识符
标识符 = 常量表达式

枚举符表中的标识符被说明为int类型的常量，它们可以出现在需要常量的任何地方。如果没有带有=的枚举符出现，那么相应常量值从0开始，且当从左至右读取说明时枚举常量值加1。带有=的枚举符对联系的标识符给出所指定的值，其后的标识符从所指定值开始继续递增。

在同一作用域内各个枚举符的名字必须互不相同，也不能和普通变量名字相同，但其值不必是不同的。

枚举区分符中标识符的作用与结构区分符中结构标记的作用类似。它命名了一个特定的枚举类型。除了不存在不完整枚举类型外，枚举区分符在具有或缺少标记和枚举符表时的规则与结构或联合的那些规则相同。无枚举符表的枚举区分符标记必须指向作用域内的一个具有枚举符表的区分符。

注释　相对于本书第1版，枚举类型是新概念，但作为C语言的一部分已有好些年了。

A.8.5 说明符

说明符的语法如下：

说明符：
指针$_{opt}$　直接说明符
直接说明符：
标识符
(说明符)
直接说明符　[常量表达式$_{opt}$]
直接说明符　(参数类型表)
直接说明符　(标识符表$_{opt}$)
指针：
*类型限定符表$_{opt}$
*类型限定符表$_{opt}$ 指针
类型限定符表：
类型限定符
类型限定符表　类型限定符

说明符的结构与间接、函数及数组表达式类似，组合方式也相同。

A.8.6 说明符的含义

说明符表出现在类型和存储类区分符序列之后。每个说明符说明一个唯一的主标识符，该标识符是直接说明符产生式的第一个备选。存储类区分符可直接作用于该标识符，但其类型依赖于其说明符的形式。当说明符的标识符出现在具有与该说明符相同形式的表达式中时，该说明符就被看做是断言。

若只考虑说明区分符（参见A.8.2节）的类型部分及特定的说明符，那么一个说明具有形式“T　D”，其中T是类型，D是说明符。在不同形式的说明中，标识符的类型可用这个概念来归纳

描述。

在说明T D中（其中D是不加任何修饰的标识符），D的类型是T。

在说明T D中，若D具有

(D1)

的形式，则D1中标识符的类型与D中标识符的类型相同。圆括号不改变类型，但可改变复杂说明符的绑定。

1. 指针说明符

在说明T D中，如果D具有如下形式：

* 类型限定符表$_{opt}$ D1

且在说明T D1中的标识符的类型是“类型修饰符 T”，那么D的标识符的类型是“类型修饰符类型限定符表指向T的指针”。星号*后的限定符用于指针本身，而不是指针所指向的对象。

例如，考虑如下说明

```
int *ap[];
```

这里ap[]起到D1的作用，说明“int ap[]”使得ap的类型为“整数数组类型”，类型限定符表为空，且类型修饰符为“的数组”。因此，实际说明使ap具有“指向int的指针数组”类型。

作为另一个例子，说明

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

说明了一个整数i和一个指向整数的指针pi。常量指针cpi的值不能修改，该指针总是指向同一位置，尽管它所指之处的值可以改变。整数ci是常量，也不能被修改（尽管可以初始化，如本例中所示）。pci的类型是“指向const int的指针”，pci本身可以被改变而指向另一处，但它所指之处的值不能通过pci赋值来改变。

2. 数组说明符

在说明T D中，如果D具有形式

D1 [常量表达式$_{opt}$]

且在说明T D1中标识符的类型是“类型修饰符 T”，那么D的标识符类型是“类型修饰符 T的数组”。如果存在常量表达式，则该常量表达式必须为整类型且值大于 0。若数组缺少用于指定上界的常量表达式，那么该数组类型是不完整类型。

数组的成份类型可以是算术类型、指针类型、结构类型或联合类型，也可以是另一个数组（以生成多维数组）。数组成份的类型必须是完整类型，绝不能是不完整类型的数组或结构。这就意味着，对于多维数组，只有第一维可以缺省。一个不完整数组类型的对象的类型，可以通过对该类型的另一个完整说明（见A.10.2节）或通过对其初始化（见A.8.7节）来使它完整。例如，

```
float fa[17],  *afp[17];
```

说明了一个浮点数组和一个指向浮点数的指针数组，而

```
static int  x3d[3][5][7];
```

则说明了一个静态的三维整型数组，其大小为 3×5×7。具体而言，x3d是一个由三个项组

成的数组，每一个项都是由5个数组组成的数组，5个数组中的每一个又都是由7个整数组成的数组。x3d、x3d[i]、x3d[i] [j]与x3d[i] [j] [k]都可以合理地出现在一个表达式中，前三者是数组类型，最后一个是整数类型。更明确地说，x3d [i] [j]是一个有7个整数元素的数组，x3d [i]是含有5个由7个整数构成的数组的数组。

数组下标运算规定E1[E2]与*(E1+E2)等同。因此，尽管看上去不对称，但下标运算是可交换的运算。根据作用于+和数组的转换规则(参见A.6.6节、A.7.1节与A.7.7节)，若E1是数组且E2是整数，那么E1 (E2)代表E1的第E2个成员。

在本例中，x3d [i] [j] [k]等价于* （x3d [i] [j]+k)。第一个子表达式x3d [i] [j]按A.7.1节所述转换成类型“指向整数数组的指针”，而根据A.7.7节中所述，加法运算涉及乘以整数大小的操作。它所遵循的规则是：数组按行存储（最后一维下标变动最快），且说明中的第一维下标用于决定数组所需的存储区大小，但第一维下标在下标计算的其他方面不起作用。

3. 函数说明符

在一个新方式的函数说明T D中，如果D具有形式

```
D1（参数类型表）
```

并且在说明T D1中标识符的类型是“类型修饰符 T”，那么D的标识符的类型是“具有返回T的变元参数类型表的类型修饰符函数”。

参数的语法定义为：

```
参数类型表：
    参数表
    参数表，…
参数表：
    参数说明
    参数表，参数说明
参数说明
    说明区分符 说明符
    说明区分符 抽象说明符opt
```

在这个新的说明中，参数表说明了参数的类型。作为一个特殊情况，无参数的新方式函数的说明符具有一个参数类型表，该表仅包含关键字void。若参数类型表以省略号“，...”结束，那么该函数接受的参数个数可比显式描述的参数个数多，参见A.7.3节。

若参数类型是数组或函数，则按照参数转换规则（见A.10.1节）将它们转换为指针。在参数的说明区分符中唯一允许的存储类区分符是register，除非函数定义以函数说明符为首，否则该存储类说明符将被忽略。类似地，如果参数说明的说明符中包含标识符，且函数定义不以该函数说明符为首，那么该标识符超出了作用域。不涉及标识符的抽象说明符将在A.8.8节讨论。

在旧方式的函数说明T D中，如果D具有形式

```
D1（标识符表opt）
```

并且在说明T D1中的标识符的类型是“类型修饰符 T”，那么D的标识符类型为“未指定返回T的变元的类型修饰符函数”。参数（若存在的话）具有形式

```
标识符表：
    标识符
```


标识符表，标识符

在旧方式说明符中，除非在函数定义的首部使用了说明符，否则标识符表必须缺省（参见 A.10.1 节）。说明不提供有关参数类型的信息。

例如，说明

```
int f(), *fpi(), (*pfi)();
```

说明了一个返回整数类型的函数 f、一个返回指向整数的指针的函数 fpi以及一个指向返回整数的函数的指针 pfi。它们都未说明参数类型，因此都属旧方式的说明。

在新方式的说明

```
int strcpy(char *dest, const char *source), rand(void);
```

中，strcpy是返回整数类型的函数，具有两个参数，第一个是字符指针，第二个是指向常量字符的指针。参数名字即是有效注解。第二个函数 rand不带参数且返回类型 int。

注释 到目前为止，带参数原型的函数说明符是由 ANSI标准引入的最主要的语言变化。它们优于第1版中的“旧方式”说明符，因为它们提供了函数调用时的错误检测和参数强制转换，不过代价是在引入时带来了混乱和迷惑，而且还必须兼容这两种形式。为了兼容，不得不在语法上做一些手脚，即采用 void作为无参数新方式函数的显式标记。

带有变长变元表的函数采用的省略号“，…”也是新的标准，它和标准头文件 <stdarg.h>中的宏共同形式化了一个机制，该机制在第 1版中虽被禁止但可非正式地接受。

这些表示法来自C++。

A.8.7 初始化

在说明一个对象时，对象的初始化说明符可为其指定一个初始值。初始化符紧随 =之后，它或是一个表达式，或是一列嵌套在花括号中的初始化符。一列初始化符可以以逗号结束，这使得格式简洁优美。

初始化符：
  赋值表达式
  {初始化符表}
  {初始化符表，}

初始化符表：
  初始化符
  初始化符表，初始化符

静态对象或数组的初始化符中的所有表达式必须是如 A.7.19节中所述的常量表达式。如果初始化符是用花括号括起来的初始化符表，那么 auto或register对象或数组的初始化符中的表达式也同样必须是常量表达式。然而，若自动对象的初始化符是一个单一表达式，那么它不必是常量表达式，但必须符合对象赋值的类型要求。

注释 第1版不支持自动结构、联合或数组的初始化。而 ANSI标准是允许的，但只能通

过常量结构，除非可用简单表达式表示初始化符。

一个未显式初始化的对象将被隐式初始化，它（或它的成员）被赋以常量 0。未显式初始化的自动对象的初始值是没有定义的。

指针或算术类型对象的初始化符是一个单一表达式，但可能括在花括号中。该表达式将赋值给相关对象。

结构的初始化符可以是具有相同类型的表达式，也可以是按其成员次序括在花括号中的初始化符表。无名的位字段成员被忽略，故不被初始化。若表中初始化符的数目比结构的成员数少，那么尾随的剩余结构成员将被初始化为 0。初始化符的数目不能比成员数多。

数组的初始化符是括在花括号中的数组成员的初始化符。若数组大小未知，那么初始化符的数目将决定数组的大小，从而使得数组类型变得完整。若数组大小固定，则初始化符的数目不能超过数组成员的数目。若初始化符的数目比数组成员数目少，则尾随的剩余数组成员将被初始化为0。

作为一个特殊情况，字符数组可用字符串字面值初始化。字符串的各个字符依次初始化数组中的相应成员。类似地，宽字符字面值（参见 A.2.6节）可初始化wchar_t类型的数组。若数组大小未知，则数组大小将由字符串中字符数（包括结尾空字符）决定。若数组大小固定，则不计结尾空字符，字符串中字符数不能超过数组大小。

联合的初始化符可以是具有相同类型的表达式，也可以是括在花括号中的联合的第一成员的初始化符。

注释　第1版不允许对联合初始化。“第一成员”规则显得有点笨拙，但没有新语法很难进行概括。除了允许联合至少以一个原始方式被显式初始化，这一 ANSI规则还给出了非显式初始化的静态联合的精确语义。

聚集是一个结构或数组。若一个聚集包含聚集类型的成员，则初始化时将递归使用初始化规则。如下情况将在初始化中省略括号：若聚集的成员也是一个聚集，且该成员的初始化符以左花括号开头，那么后继部分用逗号隔开的初始化符表将初始化子聚集的成员。初始化符的数目不允许超过成员的数目。然而，如果子聚集的初始化符不以左花括号开头，那么表中只有足够元素被认为是子聚集的成员，任何剩余成员将用来初始化下一个子聚集所在的聚集的成员。

例如，

```
int x[ ] = {1, 3, 5};
```

说明并初始化x为一个具有三个成员的一维数组，因为数组未指定大小且有三个初始化符。

```
float y[4][3] = {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 },
} ;
```

是一个完全用花括号隔开的初始化： 1、3和5这三个数初始化数组 y[0]的第一行，即 y[0][0]、y[0][1]和y[0][2]。同样，另两行初始化 y[1]和y[2]。因初始化符的数目不够，因此元素 y[3]被初始化为0。确切地说，如下说明能得到相同的结果：

```
float y[4] [3] = {
   1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

y的初始化符以左花括号开始，但 y[0]的初始化符与其不同，因此 y[0]的初始化使用了表中三个元素。同理，y[1]使用了随后的三个，接着 y[2]使用了最后三个。另外，

```
float y[4] [3] = {
   { 1 }, { 2 }, { 3 }, { 4 }
} ;
```

用于初始化 y 的第一列（将 y 当做一个二维数组），其余的默认初始化为 0。

最后，

```
char msg[] = "Syntax error on line %s\n" ;
```

说明了一个字符数组，对其元素用了一个字符串字面值进行初始化，该数组大小包括终结空字符。

A.8.8 类型名字

在有些上下文中（例如，在需要显式指定强制类型转换时、在函数说明符中说明参数类型时以及在作为 sizeof 的变元时），必须提供数据类型的名字。这一目的可以使用类型名字来达到。在语法上，类型名字就是该类型的某一对象的说明，只是省略了该对象的名字。

类型名：
  区分符限定符表 抽象说明符$_{opt}$

抽象说明符：
  指针
  指针$_{opt}$ 直接抽象说明符

直接抽象说明符：
  （抽象说明符）
  直接抽象说明符$_{opt}$ [常量表达式$_{opt}$]
  直接抽象说明符$_{opt}$ [参数类型表$_{opt}$]

如果该结构是说明中的一个说明符，那么就有可能唯一确定标识符在抽象说明符中出现的位置。所指名的类型从而就与假想标识符的类型相同。例如，

```
int
int *
int *[3]
int (*)[ ]
int *( )
int (*[ ])(void)
```

等6个说明分别命名了类型“整数”、“指向整数的指针”、“包含3个指向整数的指针的数组”、“指向未指定整数个数的数组的指针”、“未指定参数的函数，函数返回指向整数的指针”以及“大小未指定的数组，该数组的指针指向无参函数，每个参数返回一个整数”。

A.8.9 类型定义

如果一个说明的存储类区分符是 typedef，那么该说明并不说明对象，而是定义用于命名类型的标识符。这些标识符就称为类型定义名字。

类型定义名字：
  标识符

类型定义说明以通常的方式把一个类型指派给其说明符中的每个名字（参见A.8.6节）。此后，每个这样的类型定义名字在语法上就等同于相关类型的类型区分符关键字。

例如，在定义

```
typedef   long Blockno, *Blockptr;
typedef   struct { double r, theta; } Complex;
```

之后，结构

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

都是合法的说明。b的类型是long类型，bp的类型是“指向long类型的指针”，而z的类型是所指定的结构，zp的类型是指向该结构的指针。

typedef类型定义并不引入新类型，它只是可以用另一种方式说明的类型的同义词。在本例中，b具有与其他任何long类型对象相同的类型。

typedef类型定义名字可在内层作用域内重新说明，但必须给出一个类型区分符的非空集合。例如，

```
extern Blockno;
```

没有重新说明Blockno，但

```
extern int Blockno;
```

重新说明了Blockno。

A.8.10 类型等价

如果两个类型区分符表包含相同的类型区分符集合（把某些类型区分符的潜在等价关系考虑在内，例如，单个long就是long int），那么这两个类型区分符表就是等价的。具有不同标记的结构、联合和枚举是不相同的，不带标记的联合、结构或枚举指定各不相同的类型。

如果两个类型在展开了其中任何 typedef类型并删除了所有函数参数标识符后，它们的抽象说明符（见A.8.8节）是相同的，直到类型区分符表的等价，那么就称这两个类型是相同的。数组大小和函数参数类型在这里是有效因素。

A.9 语句

如果不特别指明，语句都是顺序执行的。语句执行都有一定的结果，但没有值。语句可分为几种。

语句：
  带标号语句
  表达式语句
  复合语句
  选择语句
  循环语句
  跳转语句

A.9.1 带标号语句

语句可带有标号前缀。

带标号语句：
  标识符：语句
  case　常量表达式：语句
  default：语句

由标识符构成的标号用于说明该标识符，标识符标号的唯一用途是作为 goto**语句的转向的目标。标识符的作用域是当前函数。因为标号有自己的名字空间，它们不会与其他的标识符混淆并且不能被重新说明。参见** A.11.1**节。**

case**标号和**default**标号用在**switch**语句中（参见** A.9.4**节）。**case**标号的常量表达式必须为整型。**

标号本身不会改变程序控制流。

A.9.2 表达式语句

大部分语句为表达式语句，其形式如下：

表达式语句：
  表达式$_{opt}$；

大多数表达式语句为赋值语句或函数调用语句。表达式的所有副作用在下一个语句执行前完成。如果没有表达式，则称做空语句。空语句通常用来提供一个空体给循环语句或用来放置标号。

A.9.3 复合语句

当需要把若干个语句作为一个语句来使用时，可以使用复合语句（也称“分程序”）。函数定义的体就是复合语句。

复合语句：
  {说明表$_{opt}$}　语句表$_{opt}$
说明表：
  说明
  说明表　说明
语句表：
  语句
  语句表　语句

如果在一分程序外说明的标识符又出现在分程序内的说明表中，那么外部说明在分程序内

就被屏蔽了（参见A.11.1节）。在同一分程序内一个标识符只能说明一次。此规则适用于同一名字空间的标识符（参见A.11节），不同名字空间的标识符被认为是不同的。

自动对象在每次进入分程序时按说明的顺序初始化。如果执行了一跳转语句进入分程序，则不进行初始化。静态对象仅在程序开始执行前初始化一次。

A.9.4 选择语句

选择语句有如下几种控制流形式：

选择语句：
```
if （表达式） 语句
if （表达式） 语句 else 语句
switch （表达式） 语句
```

在两种形式的if语句中，表达式都必须为算术或指针类型。首先计算表达式的值包括所有的副作用，如果不等于0则执行第一个子语句。在第二种形式中，如果表达式为0，则执行第二个子语句。通过将else与同一嵌套层中遇到的最近的未匹配else的if相连接可以解决else的歧义性。

switch语句根据表达式的不同取值将控制转向相应的分支。关键字switch之后用圆括号括起来的表达式必须为整类型。由switch语句控制的子语句一般是复合语句。该子语句中的任一语句可带一个或多个case标号（参见A.9.1节）。控制表达式要进行整提升(参见A.6.1节)，case常量被转换为整提升后的类型。与同一switch语句相关的两个case常量在转换后不可能有相同的值。也可能至多有一个default标号与一个switch语句相关。switch语句可以嵌套，一个case或default标号与包含它的最小的switch相关。

在switch语句执行时，首先计算表达式的值及其副作用，并将其值与每个case常量比较，如果某case常量与表达式的值相同，控制转向与case标号匹配的语句。如果没有case常量与表达式匹配，并且如果有default标号，控制转向有标号的语句。如果没有case常量匹配，并且也没有default标号，则switch语句的所有子语句都不执行。

注释　在本书第1版中，switch语句的控制表达式与case常量都必须为int类型。

A.9.5 循环语句

循环语句用于指定程序段的循环执行。

循环语句：
```
while （表达式） 语句
do 语句 while （表达式）;
for （表达式opt ;表达式opt ;表达式opt )语句
```

在while语句和do语句中，只要表达式的值不为0，则其中的子语句一直重复执行。表达式必须为算术或指针类型。while语句在语句执行前测试表达式并计算其副作用，而do语句在每次循环后测试。

在for语句中，第一个表达式被计算一次，以此对循环初始化。对该表达式的类型没有限制。第二个表达式必须为算术或指针类型，在每次开始循环前计算其值。如果该表达式的值等于0，

那么for语句终止执行。第三个表达式在每次循环后计算，以重新对循环进行初始化，它的类型也没有限制。所有表达式的副作用在计算其值后立即完成。如果在子语句中不包含 continue语句，那么语句

```
for (表达式1；表达式2；表达式3)  语句
```

等价于

```
表达式1；
while (表达式2) {
  语句
  表达式3；
}
```

for语句的三个表达式中任一表达式都可缺省。在第二个表达式缺省时等价于测试一个非 0常量。

A.9.6 跳转语句

跳转语句用于无条件地转移控制。

```
跳转语句：
  goto  标识符；
  continue；
  break;
  return  表达式opt；
```

在goto语句中，标识符必须为位于当前函数中的标号（参见 A.9.1节）。控制转移到标号所指定的语句。

continue语句只能出现在循环语句内，它将控制转向包含此语句的最内层循环部分。更精确地说，在下面任一语句内，如果 continue语句不包含在更小的循环语句中，则 continue语句与 goto contin语句等价。

```
while (…) {          do {              for (…) {
    …                …                 …
contin: ;            contin: ;         contin: ;
}                    }while (…);       }
```

break语句只能用在循环语句或 switch语句中，用于终止包含这样语句的最内层循环语句的执行，并将控制转向到被终止语句的下一个语句。

return语句用于将函数返回到调用者，当 return语句后跟一表达式时，其值返回给函数调用者。像赋值一样，该表达式被转换为其所在函数返回的类型。

函数的自然结束等价于一个不带表达式的 return语句。在两种情况下返回值都是没有定义的。

A.10 外部说明

提供给C编译程序处理的输入单元称做翻译单元，它由一外部说明序列组成，这些外部说明或者是说明，或者是函数定义。

翻译单元：
　外部说明
　翻译单元　外部说明
外部说明：
　函数定义
　说明

就像在分程序中说明的作用域为整个分程序一样，外部说明的作用域是其所在的翻译单元。外部说明除了只能出现在函数代码可能出现的位置外，其语法规则与其他说明一样。

A.10.1　函数定义

函数定义有如下形式：

函数定义：
　说明区分符$_{opt}$　说明符　说明表$_{opt}$　复合语句

在说明区分符中可以使用的存储类区分符只能是 extern或static，关于这两个存储类区分符之间的区别参见A.11.2节。

函数可返回一算术类型、结构、联合、指针或 void，但不能返回函数或数组。函数说明中的说明符必须显式指定所说明的标识符具有函数类型，也就是说，必须包含如下两种形式之一（参见A.8.6节）：

直接说明符（参数类型表）
直接说明符（标识符表$_{opt}$）

其中，直接说明符为一标识符或用圆括号括起来的标识符。特别地，不能通过 typedef获得一个函数类型。

使用第一种形式的说明符的函数定义为新方式的函数定义，其参数及其类型都在参数类型表中说明，函数说明符后的说明表不能缺少。除非参数类型表中只有 void类型（表示该函数没有参数），参数类型表中的每个说明都必须包含一个标识符。如果参数类型表以“，…”结束，那么在调用该函数时所用的变元数就可多于参数数。在标准头文件 <stdarg.h>中定义的、在附录B中介绍的va_arg宏机制被用来表示额外的变元。变参函数必须至少有一个指名参数。

使用第二种形式的说明符的函数定义为旧方式的函数定义，标识符表给出了参数的名字，这些参数的类型由说明表指定。未做说明的参数默认为 int。说明表必须只说明在标识符表中指名的参数，不能进行初始化，register是唯一可以使用的存储类区分符。

在这两种函数定义中，参数可理解为是在组成函数体的复合语句刚开始执行时说明的，因此在该复合语句中不能重新说明与参数同名的标识符（但可以像其他的标识符一样在该复合语句内的分程序中重新说明参数标识符）。如果某一参数说明为“某一类型 *type*的数组”，那么该说明会被自动调整，使该参数为“指向类型 *type*的指针”。类似地，如果一参数说明为“返回某一类型*type*的函数”，那么该说明会被调整使该参数为“指向返回类型 *type*的函数的指针”。在调用函数时，必要时要对变元进行类型转换并赋值给参数，参见A.7.3节。

注释　新方式函数定义是在ANSI标准中新引入的。关于提升的细节也有些细微的变化。第1版指定的float类型的参数说明被调整为 double类型。当在函数内部生成一指向参数

的指针时，区别就显而易见了。

下面是一个新方式函数定义的完整的例子：

```
int max ( int a, int b, int c )
{
   int m;
   m = ( a > b ) ? a : b ;
   return ( m > c ) ? m : c ;
}
```

这里int是说明区分符，max (int a, int b, int c)是函数说明符，{ … }是给出函数代码的分程序。相应旧方式的定义为：

```
int max ( a, b, c )
int a, b, c ;
{
   /* … */
}
```

这里max (a, b, c)是说明符，int a, b, c ；是参数的说明表。

A.10.2 外部说明

外部说明用于指定对象、函数及其他标识符的特性。术语“外部”指它们位于函数外部，并且不直接与关键字 extern连接。对外部说明的对象可以不指定存储类，也可指定为 extern或static。

对同一标识符的几个外部说明可存在于同一翻译单元中，只要它们类型和连接一致，并且对该标识符至多只有一个定义。

对一个对象或函数的两个说明只要遵循 A.8.10节讨论的规则就认为是类型一致的。此外，如果两个说明的区别在于：一个类型为不完整结构、联合或枚举类型（参见 A.8.3节），而另一个是带同一标记的对应的完整类型，那么也认为这两个类型是一致的。此外，如果一个类型是不完整数组类型（参见 A.8.6节）而另一个类型为完整数组类型，除此之外其他都相同，那么也认为这两个类型是一致的。最后，如果一个类型指定了一个老式函数，而另一个类型指定了带参数说明的在其他方面相同的新式函数，也认为它们是一致的。

如果对一个对象或函数的第一个外部说明包含 static区分符，那么该标识符有内部连接，否则它有外部连接。关于连接将在 A.11.2节讨论。

一个对象的外部说明若带初始化符则该说明就是定义。如果一个外部说明不带初始化符并且不含extern区分符，那么它就是一个暂时定义。如果一个对象的定义出现在翻译单元中，那么所有暂时定义都被认为仅仅是多余的说明；如果在该翻译单元中不存在对该对象的定义，那么其暂时定义变为一个初始化符为 0的单一的定义。

每个对象都必须正好有一个定义。对具有内部连接的对象，这个规则分别作用于每个翻译单元，这是因为内部连接的对象对每个翻译单元是独一无二的。对于具有外部连接的对象，这个规则作用于整个程序。

注释　虽然单一定义规则在阐述上与本书第 1版有所不同，但它在效果上与这里所述是一样的。有些实现通过将暂时定义的概念一般化而放宽了这个限制。在另一种阐述中，对一个程序的所有翻译单元中的外部连接对象的所有暂时定义被一起考虑，而不是在各个翻译单元中分别考虑，在 UNIX系统中通常就采用的这种方法，并且被认为是该标准的一般扩展。如果一个定义出现在程序中的某个地方，那么暂时定义仅被认为是说明，但如果没有定义出现，则所有暂时定义变为具有初始化符的定义。

A.11　作用域与连接

一个程序中所有单元不必同时编译。源文本可保存在包含翻译单元的若干个文件中，预编译过的程序段可以从库中装入。程序中函数间的通信可以通过调用和操纵外部数据来实现。

因此，我们就要考虑两种类型的作用域：第一种是标识符的词法作用域，它是体现标识符特性的程序文本区域；第二种是与外部连接的对象和函数相关的作用域，它决定在各个已编译的翻译单元内标识符之间的连接。

A.11.1　词法作用域

标识符可以在若干个名字空间中使用而互不影响。如果在不同的名字空间中使用同一标识符，那么即使是在同一作用域内，这个标识符也可用于不同的目的。名字空间包含如下几种：对象、函数、类型定义名字和枚举常量；标号；结构标记、联合标记和枚举标记；每个结构或联合的各自成员。

注释　这些规则与本手册第 1版所述有几点不同。标号以前没有自己的名字空间；结构和联合分别有各自的名字空间，在某些实现中枚举标记亦是如此；把不同种类的标记放在同一名字空间中是一种新的限制。与第 1版的最大不同是每个结构和联合都为其成员创立不同的名字空间，因此同一名字可出现在若干个不同结构中。这一规则在近几年十分常用。

在一个外部说明中的对象或函数标识符的词法作用域从其说明结束开始直到所在翻译单元的结束。函数定义中参数的作用域在定义函数的分程序开始处开始贯穿整个函数，函数说明中参数的作用域在说明符的末尾处结束。分程序头部中说明的标识符的作用域是其所在的整个分程序。标号的作用域是其所在函数。结构、联合、枚举标记或枚举常量的作用域从其出现在类型区分符中开始，到翻译单元结束（对外部说明）或分程序结束（对函数内的说明）。

如果一标识符显式地在分程序（包括组成函数的分程序）头部中说明，任何分程序外部说明的标识符将被覆盖直到分程序结束。

A.11.2　连接

在翻译单元中，具有内部连接的同一对象或函数标识符的所有说明均指同一实体，并且该对象或函数对这个翻译单元是唯一的。具有外部连接的同一对象或函数标识符的所有说明也指同一实体，并且该对象或函数在整个程序中共享。

如A.10.2节所述，如果使用了static区分符，那么对一标识符的最初的外部说明给出了标识符内部连接，否则，给出该标识符外部连接。如果在一分程序内对一个标识符的说明不包含extern区分符，那么该标识符无连接，并且对于函数是唯一的。如果这样的说明中包含extern区分符，并且在包含该分程序的作用域中对该标识符的外部说明是活动的，那么该标识符与外部说明具有相同的连接，并表示同一对象或函数。但是，如果没有外部说明是可见的，那么其连接是外部的。

A.12 预处理

预处理程序用于执行宏替换、条件编译和引入指名的文件。以#开始的命令行（“#”前可以有空格）就是预处理程序处理的对象。这些命令行的语法独立于语言的其他部分，它们可在任何地方出现，其作用可延续到所在翻译单元的末尾（与作用域无关）。行边界是有实际意义的，每一行都单独分析（关于如何将若干行连起来，参见A.12.4节）。对预处理程序而言，单词就是任何语言的单词，或者像在#include指令（参见A.12.4节）中用做文件名字的字符序列。此外，所有未做其他定义的字符都被认为是单词。但是，除空格和横向制表符之外的空白符的效果在预处理程序指令行中是没有定义的。

预处理本身是在逻辑上连续的几个阶段完成的，在某些特殊的实现中可以缩减。

1) 首先，把如A.12.1节所述的三字符序列替换为其等价字符，如果操作系统环境需要，还要在源文件的各行之间插入换行符。

2) 把指令行中位于换行符前的反斜杠符“\”删除掉，从而把各指令行连接起来（参见A.12.2节）。

3) 把程序分成用空白符隔开的单词，把注解替换为一个空白符。接着执行预处理指令，并进行宏扩展（参见A.12.3节至A.12.10节）。

4) 把字符常量和字符串字面值中的换码序列（参见A.2.5节与A.2.6节）替换为其等价字符，然后把相邻的字符串字面值连接起来。

5) 通过收集必要的程序和数据，并将外部函数和对象的引用与其定义相连接，翻译经过以上处理得到的结果，然后与其他程序和库连接起来。

A.12.1 三字符序列

C源程序的字符集是7位ASCII码的子集，但它是ISO 646-1983不变代码集（Invariant Code Set）的超集。为了使程序能用这种缩减字符集表示，如下所示的所有三字符序列都要用相应的单个字符替换，这种替换在其他任何处理之前进行。

```
??=     #       ??(     [       ??<     {
??/     \       ??)     ]       ??>     }
??'     ^       ??!     |       ??-     ~
```

除此之外不进行其他替换。

注释 三字符序列是ANSI标准新引入的。

A.12.2 行连接

通过把以反斜杠“\”结束的指令行末尾的反斜杠和其后的换行符删除掉，可以将若干指令行合并成一行。这种处理要在分离单词之前进行。

A.12.3 宏定义和扩展

形如

```
# define 标识符 单词序列
```

的控制行用于使预处理程序在此后将指定标识符的各个实例用给定的单词序列替换，单词序列前后的空白符都被丢掉。第二次用 #define指令定义同一标识符是错误的，除非第二次定义中的单词序列与第一次相同（所有的空白分割符被看做是相同的）。

形如

```
# define  标识符（标识符表） 单词序列
```

的指令行是一个带有参数（由标识符表指定）的宏定义，其中第一个标识符与圆括号“（”之间没有空格。像第一种形式一样，单词序列前后的空白符都被丢弃掉。如果要对宏重定义，那么就要求其参数个数和拼写及其单词序列都必须与前面的定义相同。

形如

```
# undef  标识符
```

的控制行用于使指定标识符的预处理程序定义不被考虑。将 #undef应用于未知标识符（即未用#define指令定义的标识符）不是错误。

当一个宏以第二种形式定义时，由宏标识符（后面可以跟一个空白符）及后随的用一对圆括号括起来的由逗号分隔的单词序列组成了一个宏调用。宏调用的变元是用逗号分隔的单词序列，用引号或嵌套的括号括起来的逗号不用于分隔变元。在处理过程中，变元不进行宏扩展。宏调用时变元的数目必须与其定义时参数的数目匹配。在变元被分离后，先将前导和结尾的空白符删除，然后在替换宏的单词序列时，用由各个变元产生的单词序列替换对应的未用引号括住的参数标识符。除非在替换序列中参数有前导＃，或者前导或尾随有 ##，在插入前，要对宏调用的变元序列进行检查，在必要时进行扩展。

有两个特殊的操作符会影响替换过程。首先，如果在替换单词序列中参数有直接前导＃，那么要用双引号 ″括住对应的参数，然后将＃和参数标识符用被引号括住的变元替换。在字符串字面值或字符常量两边或内部的每个双引号 ″或反斜杠 \ 前要插入一反斜杠\。

其次，如果两种类型的宏中无论哪一种宏的定义单词序列包含有一个 ##操作符，那么，在参数替换后要把 ##及其前后的空白符删除掉，以便将相邻单词连接起来形成一新的单词。如果如此所产生的单词是无效的，或者，如果结果依赖于 ##操作符的处理顺序，那么，其结果没有定义。同样，##也可能不出现在替换单词序列的开头或结尾。

对这两种类型的宏，都要重复扫描替换单词序列以找到更多的定义标识符。但是，一旦一个给定的标识符在给定扩展中被替换，当再次扫描再遇到此标识符时就不再对其进行替换，而是保持不变。

即使宏扩展的最终值以#开始，也不认为其是预处理指令。

注释　有关处理宏扩展的细节在ANSI标准中比第1版描述得更详细，最重要的变化是加入了#和##操作符，这使得引用与连接成为可能。一些新规则，特别是与连接有关的规则是很奇怪的(参见下面的例子)。

例如，这种功能可在下面用做“显式常量”：

```
#define    TABSIZE        100
int        table[TABSIZE];
```

定义

```
#define    ABSDIFF(a, b)  ((a) > (b) ? (a) - (b) : (b) - (a))
```

定义了一个用于返回其两个参数变元差的绝对值的宏。与函数不同的是，变元与返回值可以是任意算术类型甚至可以是指针。同样，变元可能有副作用，要计算两次，一次是进行测试，另一次产生值。

对定义

```
#define  tempfile(dir)  #dir "/%s"
```

宏调用tempfile(/usr/tmp)将产生

```
"/usr/tmp" "/%s"
```

接着被连接为一个单一字符串。在定义

```
#define cat(x, y)  x ## y
```

后，宏调用cat (var, 123) 产生var123。但是，宏调用cat (cat (1, 2), 3))没有定义：##的存在限制了外部调用的变元的扩展。因此将产生单词串

```
cat    (    1    ,    2    ) 3
```

并且)3不是合法的单词，它是由第一个变元的最后一个单词与第二个变元的第一个单词连接而成。如果再进行第二层的宏定义：

```
#define xcat(x, y)  cat(x, y)
```

那么就会得到所要的效果：xcat (xcat (1, 2), 3)将产生123，因为xcat自身的扩展不包含##操作符。

类似地，ABSDIFF(ABSDIFF(a, b), c)将产生所期望的完全扩展的结果。

A.12.4 文件包含

形如

```
# include <文件名>
```

的控制行用于将该行替换成由文件名所命名的文件的内容。文件名中不能出现大于字符 >或换行符。如果文件名中包含字符"、 ' 、\或 /*，那么其行为没有定义。预处理程序将在依赖于实现的有关位置中查找所指名的文件。

类似地，形如

```
# include "文件名"
```

的控制行首先从最初的源文件的目录开始搜索指名的文件 (搜索过程依赖于实现)，如果没找到指

名的文件，那么就像在第一种形式中哪样处理，即在实现定义的有关位置上接着查找所指名的文件。在文件名中使用字符 ' 、\ 或 *仍然是没有定义的，但可以使用字符 > 。

最后，形如

```
# include 单词序列
```

的、与上述两种情况都不匹配的指令通过像对普通文本一样扩展单词序列来解释。必须将其解释成具有<...>或"..."两种形式之一，然后再按上述方式进行相应处理。

#include文件可以嵌套。

A.12.5 条件编译

对一个程序的某些部分可以进行条件编译，这时可按照如下所示的语法进行：

```
预处理程序条件：
if-行  文本  elif-部分  else-部分opt  #endif

if-行：
  # if  常量表达式
  # ifdef  标识符
  # ifndef  标识符

elif-部分：
  elif-行  文本
  elif-部分opt

elif-行：
  # elif  常量表达式

else-部分：
  else-行  文本

  else-行：
  # else
```

其中，每个条件编译指令 (if-行、elif-行、else-行及#endif)在程序中均单独占一行。预处理程序依次对#if及后继的#elif指令中的常量表达式进行计算，直到发现有一个指令的常量表达式有非 0 值，并删去值为 0的指令行后面的文本。常量表达式不为 0的#if和#elif指令之后的文本像其他程序代码一样进行编译。在这里，“ 文本 ”由任何不属于该条件编译指令结构的程序代码组成，它可以包含预处理指令，也可以为空。一旦预处理程序发现某个 #if或#elif条件编译指令中的常量表达式的值不为 0并选择紧随其后的程序代码供以后的编译阶段使用时，就删去后继的 #elif和#else条件编译指令及相应的文本。如果 #if与后继的所有#elif指令中的常量表达式的值都为 0，并且该条件编译指令链中包含一条 #else指令，那么就选择在 #else指令之后的文本。除了对条件编译指令的嵌套进行检查之外，该条件编译指令的不活动分支（即条件为假的分支）所控制的文本都要跳过去。

在#if及#elif指令中的常量表达式中可以进行通常的宏替换。除此之外，任何形如

```
defined  标识符
```

或

```
defined  (标识符)
```

的表达式在进行宏扫描之前要进行替换，如果该标识符在预处理程序中已有定义，那么就用 1L 来替换它，否则，用0L来替换。在预处理程序进行宏扩展之后的任何标识符用 0L来替换。最后，每个整数类型的常量都被预处理程序认为其后面跟有后缀 L，以便把所有的算术运算都当作是在长整数类型或无符号长整数类型的运算分量之间进行的运算。

进行上述处理之后的常量表达式 (见A.7.19节)满足如下限制：它必须是整形的，并且其中不包含sizeof与强制转换运算符或枚举常量。

控制行

```
# ifdef  标识符
# ifndef  标识符
```

分别等价于：

```
# if  defined  标识符
# if  !  defined  标识符
```

注释 #elif是ANSI C中新引入的条件编译指令，尽管它已经在某些预处理程序中被实现。defined预处理运算符也是新引入的。

A.12.6 行控制

为利于其他预处理程序生成C程序，形式为

```
#  line  常量  "文件名"
#  line  常量
```

之一的预处理指令，为了错误诊断的目的，使编译程序相信下一行源代码的行号被置为十进制整数常量，当前的输入文件由标识符命名。如果缺少带双引号的文件名部分，那么就不改变当前正被编译的源文件的名字。对行中的宏进行扩展后再进行解释。

A.12.7 错误信息生成

形如

```
#  error  单词序列opt
```

的预处理程序行用于使预处理程序打印包含该单词序列的诊断信息。

A.12.8 编译指示

形如

```
#  pragma  单词序列opt
```

的控制行（叫做编译指示）使得处理程序完成依赖于实现定义的动作。不能识别的编译指示被忽略掉。

A.12.9 空指令

形如

#

的预处理程序行无任何作用。

A.12.10 预定义名字

有些标识符是预定义的，它们被扩展后产生特定的信息。它们与预处理程序表达式操作符defined都不能用undefined取消其定义或重新进行定义。

_ _LINE_ _	包含当前源文件的行号的十进制常量。
_ _FILE_ _	包含正被编译的源文件名字的字符串字面值。
_ _DATE_ _	包含源文件的编译日期的字符串字面值，其形式为“ Mmm dd yyyy ”。
_ _TIME_ _	包含编译时间的字符串字面值，其形式为“ hh:mm:ss ”。
_ _STDC_ _	整型常量1。它表示该标识符只有在与标准一致的实现中被定义为 1。

注释 #error与#pragma是ANSI 标准中新引入的。这些预定义的预处理程序宏也是新引入的，其中的一些宏已经在某些编译程序中实现。

A.13 语法

以下是对本附录前面部分所讨论的语法的一个简要概括。它们的内容完全相同，但所给的顺序不同。

本语法未定义终结符整形常量、字符常量、浮点常量、标识符、字符串和枚举常量。以打字机字体的形式表示的单词和符号是字面方式的终结符[⊖]。本语法可以机械地转换为自动语法分析程序生成器（parser-generator）可以接受的输入。除了增加语法标记以表明产生式中的可选项外，还有必要扩展“ 之一 ”结构，并（根据语法分析程序生成器的规则）复制每个带有 $_{opt}$ 符号的产生式，一次有该符号而一次没有。进一步更改即删除产生式

类型定义名字：标识符

使类型定义名字成为终结符。这个语法可被 YACC语法分析程序生成器接受。但有一个冲突，是由if-else的结构歧义性产生的。

翻译单元：
  外部说明
  翻译单元 外部说明

外部说明：
  函数定义
  说明

⊖ 打字机打印的各个字符所占的宽度相同。作者在这里做这种要求是为了区分非终结符与终结符。在译成中文时由于非终结符被译成中文，而终结符保持不动，两者已有明确的区别，故在书中不再做此要求。——译者注

函数定义：
  说明区分符$_{opt}$ 说明符 说明表$_{opt}$ 复合语句

说明：
  说明区分符 初始化说明符表$_{opt}$;

说明表：
  说明
  说明表 说明

说明区分符：
  存储类区分符 说明区分符$_{opt}$
  类型区分符 说明区分符$_{opt}$
  类型限定符 说明区分符$_{opt}$

存储类区分符：
  `auto register static extern typedef`
  之一

类型区分符：
  `void char short int long float double signed unsigned`
  结构或联合区分符 枚举区分符 类型定义名字
  之一

类型限定符：
  `const volatile`
  之一

结构或联合区分符：
  结构或联合 标识符$_{opt}$ { 结构说明表 }
  结构或联合 标识符

结构或联合：
  `struct union`
  之一

结构说明表：
  结构说明
  结构说明表 结构说明

初始化说明符表：
  初始化说明符
  初始化说明符表, 初始化说明符

初始化说明符：
  说明符
  说明符 = 初始化符

结构说明：
  区分符限定符表　结构说明符表；
区分符限定符表：
  类型区分符　区分符限定符表$_{opt}$
  类型限定符　区分符限定符表$_{opt}$

结构说明符表：
  结构说明符
  结构说明符表，结构说明符

结构说明符：
  说明符
  说明符$_{opt}$：常量表达式

枚举区分符：
  enum 标识符$_{opt}$ { 枚举符表 }
  enum 标识符

枚举符表：
  枚举符
  枚举符表，枚举符

枚举符：
  标识符
  标识符 = 常量表达式

说明符：
  指针$_{opt}$ 直接说明符

直接说明符：
  标识符
  (说明符)
  直接说明符 [常量表达式$_{opt}$]
  直接说明符 (参数类型表)
  直接说明符 (标识符表$_{opt}$)

指针：
  * 类型限定符表$_{opt}$
  * 类型限定符表$_{opt}$ 指针

类型限定符表：
  类型限定符
  类型限定符表 类型限定符

参数类型表：
  参数表
  参数表，...

参数表：
  参数说明
  参数表，参数说明

参数说明：
  说明区分符 说明符
  说明区分符 抽象说明符$_{opt}$

标识符表：
  标识符
  标识符表，标识符

初始化符：
  赋值表达式
  { 初始化符表 }
  { 初始化符表，}

初始化符表：
  初始化符
  初始化符表，初始化符

类型名字：
  区分符限定符表 抽象说明符$_{opt}$

抽象说明符：
  指针
  指针$_{opt}$ 直接抽象说明符

直接抽象说明符：
  (抽象说明符)
  直接抽象说明符$_{opt}$ [常量表达式$_{opt}$]
  直接抽象说明符$_{opt}$ (参数类型表$_{opt}$)

类型定义名字：
  标识符

语句：
  带标号语句
  表达式语句
  复合语句
  选择语句
  循环语句
  跳转语句

带标号语句：
  标识符：语句
  case 常量表达式：语句
  default：语句

表达式语句：
  表达式$_{opt}$;

复合语句：
  { 说明表$_{opt}$ 语句表$_{opt}$ }

语句表：
  语句
  语句表 语句

选择语句：
  if (表达式)语句
  if (表达式) 语句 else 语句
  switch (表达式) 语句

循环语句：
  while (表达式)语句
  do 语句 while (表达式) ;
  for (表达式$_{opt}$; 表达式$_{opt}$; 表达式$_{opt}$) 语句

跳转语句：
  goto 标识符;
  continue ;
  break ;
  return 表达式$_{opt}$;

表达式：
  赋值表达式
  表达式, 赋值表达式

赋值表达式：
  条件表达式
  一元表达式 赋值运算符 赋值表达式

赋值运算符：
  = *= /= % = += -= <<= >>= &= ^= |=
  之一

条件表达式：
  逻辑或表达式
  逻辑或表达式 ? 表达式 : 条件表达式

常量表达式：
  条件表达式

逻辑或表达式：
  逻辑与表达式
  逻辑或表达式 ¦¦ 逻辑与表达式

逻辑与表达式：
  按位或表达式
  逻辑与表达式 && 按位或表达式

按位或表达式：
  按位异或表达式
  按位或表达式 | 按位异或表达式

按位异或表达式：
  按位与表达式
  按位异或表达式 ^ 按位与表达式

按位与表达式：
  相等类表达式
  按位与表达式 & 相等类表达式

相等类表达式：
  关系表达式
  相等类表达式 == 关系表达式
  相等类表达式 != 关系表达式

关系表达式：
  移位表达式
  关系表达式 < 移位表达式
  关系表达式 > 移位表达式
  关系表达式 <= 移位表达式
  关系表达式 >= 移位表达式

移位表达式：
  加法类表达式
  移位表达式 << 加法类表达式
  移位表达式 >> 加法类表达式

加法类表达式：
  乘法类表达式
  加法类表达式 + 乘法类表达式
  加法类表达式 - 乘法类表达式

乘法类表达式：
  强制转换表达式
  乘法类表达式 * 强制转换表达式
  乘法类表达式 / 强制转换表达式
  乘法类表达式 % 强制转换表达式

强制转换表达式：
  一元表达式
  (类型名字) 强制转换表达式

一元表达式：
  后缀表达式
  ++ 一元表达式
  -- 一元表达式
  一元运算符 强制转换表达式
  sizeof 一元表达式
  sizeof （类型名字）

一元运算符：
  & * + - ~ !
  之一

后缀表达式：
  初等表达式
  后缀表达式 [表达式]
  后缀表达式 (变元表达式表$_{opt}$)
  后缀表达式 . 标识符
  后缀表达式 -> 标识符
  后缀表达式 ++
  后缀表达式 --

初等表达式：
  标识符
  常量
  字符串
  (表达式)

变元表达式表：
  赋值表达式
  变元表达式表 , 赋值表达式

常量：
  整型常量
  字符常量
  浮点常量
  枚举常量

以下是预处理程序的语法，它概述了控制行的结构，但不适合机械性语法分析。它包含符号文本（即通常的程序文本）、非条件预处理程序控制行或完全的预处理程序条件结构。

控制行：
  # define 标识符 单词序列
  # define 标识符 (标识符, ..., 标识符)单词序列
  # undef 标识符
  # include <文件名>
  # include "文件名"
  # include 单词序列
  # line 常量 "文件名"

```
  #  line 常量
  #  error 单词序列opt
  #  pragma 单词序列opt
  #
  预处理器条件指令

预处理器条件指令：
  if-行  文本  elif-部分  else-部分opt  #endif
if行：
  #  if 常量表达式
  #  ifdef 标识符
  #  ifndef 标识符
elif-部分：
  elif-行 文本
  elif-部分opt
elif-行：
  #   elif 常量表达式
else-部分：
  else-行 文本
else-行：
  #  else
```


附录B 标 准 库

本附录为ANSI标准中定义的库的概要。标准库不是C语言本身的部分，但是支持标准C的环境将会提供该库中的函数说明、类型以及宏定义。我们这里没有讨论某些用途有限的函数以及某些可以简单地从其他函数合成的函数，我们也没有讨论多字节字符，同时也没有讨论程序局部环境问题，也就是说忽略了那些依赖于当地语言、国籍或文化的性质。

标准库中的各个函数、类型以及宏分别在以下标准头文件中说明：

```
<assert.h> <float.h> <math.h> <stdarg.h> <stdlib.h>
<ctype.h> <limits.h> <setjmp.h> <stddef.h> <string.h>
<errno.h> <locale.h> <signal.h> <stdio.h> <time.h>
```

我们可以用以下方式访问头文件：

```
#include <头文件>
```

头文件可以被多次以任意顺序包含。头文件必须被包含在外部声明和外部定义之外，在使用头文件的任何声明之前都必须先包含它。一个头文件不一定是一个源文件。

以一个下划线开头的外部标识符保留给标准库使用，其他所有以一个下划线和一个大写字母开头的标识符和以两个下划线开头的标识符也都被保留给标准库使用。

B.1 输入与输出：<stdio.h>

在<stdio.h>头文件中所定义的用于输入和输出的函数、类型以及宏的数目将近占整个标准库的三分之一。

流是磁盘或其他外围设备中存储的数据的源点或终点。尽管在一些系统中（如在著名的UNIX系统中），文本流和二进制流是没有差别的，但是标准库中还是提供了这两种流。文本流是由文本行组成的序列，每一行有 0个或多个字符并以'\n'结尾。在某些环境中也许需要把文本流转换成其他表示形式（例如把 '\n'映射成回车符和换行符）或从其他表示形式转换过来。二进制流是未经处理的字节组成的序列，这些字节记录着内部数据，具有如下性质：如果在同一系统中把写出的二进制流再读入进来，读入的和写出的内容是完全相同的。

通过打开一个流可以将该流与一个文件或设备关联起来，这一关联可以通过关闭流而终止。打开一个文件将返回一个指向 FILE类型对象的指针，该指针中记录有用于控制该流的所有必要的信息。在不会引起歧义性的情况下，我们在下文中将不再区分

“文件指针”和“流”。

在程序开始执行时，stdin、stdout和stderr这三个流已经被打开。

B.1.1 文件操作

下列函数用于处理与文件有关的操作。其中 size_t类型是由运算符 sizeof产生的无符号整类型。

```
FILE *fopen(const char *filename, const char *mode)
```

fopen函数用于打开以 filename所指内容为名字的文件，返回与之相关联的流。如果打开操作失败，那么返回NULL。

访问方式mode的合法值如下：

```
"r"   打开文本文件用于读。
"w"   创建文本文件用于写，并删除已存在的内容（如果有的话）。
"a"   添加；打开或创建文本文件用于在文件末尾写。
"r+"  打开文本文件用于更新（即读和写）。
"w+"  创建文本文件用于更新，并删除已存在的内容（如果有的话）。
"a+"  添加；打开或创建文本文件用于更新和在文件末尾写。
```

后三种方式允许对同一文件进行读和写。在读和写的交替过程中，必须调用 fflush函数或文件定位函数。如果方式值在上述字符之后再加上 "b" ，如"rb"或"w+b"等，那么表示文件是二进制文件。文件名 filename长度的最大值为 FILENAME_MAX个字符。一次最多可打开 FOPEN_MAX个文件。

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

freopen函数用于以参数 mode指定的方式打开参数 filename指定的文件，并使该文件与参数 stream指定的流相关联。它返回指向 stream的指针；若出错则返回 NULL。freopen函数一般用于 stdin、stdout和stderr等标准流的重定向。

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

对输出流，fflush函数用于将已写到缓冲区但尚未写出的全部数据都写到文件中。对输入流，其结果是未定义的。如果写的过程中发生错误则返回 EOF，否则返回0。fflush(NULL)函数用于刷新所有的输出流。

```
int fclose(FILE *stream)
```

fclose函数用于刷新stream的全部未写出数据，丢弃任何未读的缓冲区内的输入数据并释放自动分配的缓冲区，最后关闭流。若出错则返回 EOF，否则返回0。

```
int remove(const char *filename)
```

remove函数用于删除参数filename指定的文件，这样，随后进行的打开该文件的操作将失败。如果失败，则返回一个非0值。

```
int rename(const char *oldname, const char *newname)
```

rename函数用于改变文件的名字。如果操作失败，则返回一个非 0值。

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

tmpfile函数用于以方式wb+创建一个临时文件，该文件在被关闭或程序正常结束时被自动删除。该函数的返回值为一个流；如果创建文件失败，则返回 NULL。

```
char *tmpnam(char s[L_tmpnam])
```

tmpnam(NULL)函数用于创建一个不同于现存文件名字的字符串，并返回一个指向一内部静态数组的指针。 tmpnam(s)函数将所创建的字符串存储在数组 s中，并将它返回作为函数值。 s中至少要有L_ tmpnam个字符的空间。 tmpnam函数在每次被调用时均生成不同的名字。在程序的执行过程中，最多只能确保生成 TMP_ MAX个不同的名字。注意tmpnam函数只是用于创建一个名字，而不是创建一个文件。

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

setvbuf函数用于控制stream所指定的流的缓冲区。在读、写以及其他任何操作之前必须调用此函数。当mode的值为 _IOFBF时，将进行完全缓冲。当 mode的值为 _IOLBF时，对文本文件表示行缓冲；当 mode的值为 _IONBF时，表示不设置缓冲。如果变元 buf的值不是NULL，则setvbuf函数将buf指向的区域作为指定流的缓冲区；否则就分配一个缓冲区。 size决定缓冲区的大小。如果出错则setvbuf返回一个非0值。

```
void setbuf(FILE *stream, char *buf)
```

如果buf的值为NULL，那么stream所指定流的缓冲区将被关闭；否则 setbuf函数等价于(void) setvbuf(stream，buf，_IOFBF，BUFSIZ)。

B.1.2 格式化输出

printf类函数用于提供格式化的输出转换。

```
int fprintf(FILE *stream, const char *format, ...)
```

fprintf函数用于按照 format说明的格式把变元表中变元的内容进行转换，并写入 stream指向的流。返回值是实际写入的字符数。若出错则返回一个负值。

格式化字符串由两种类型的对象组成：普通字符（它们被拷贝到输出流）与转换规格说明（它们决定变元的转换和输出格式）。每个转换规格说明均以字符%开头，以转换字符结束。在“%”与转换字符这二者之间依次可以有以下转换字符：

- 标记（可以以任意顺序出现）　　用于修改变换说明：

`-`　　指定被转换的变元在其字段内左对齐。

`+`　　指定在输出的数前面加上正负号。

空格　如果第一个字符不是正负号，那么在其前面附加一个空格。

`0`　　对于数值转换，说明在输出长度小于字段宽时，加前导0。

`#`　　指定其他输出形式。对于o格式，第一个数字必须为零。对于x或X格式，指定在输出的非0值前加0x或0X。对于e、E、f、g和G格式，指定输出总有一个小数点。对于g和G格式，指定输出值后面无意义的0将被保留。

- 一个指定最小字段宽的数。转换后的变元输出宽度至少要达到这个数值。如果变元的字符数小于此数值，那么在变元左边（或右边，按要求而定）添加前导字符。前导字符通常为空格，但是如果设置了0前导指示符，那么前导字符为0。
- 点号用于把字段宽和精度分开。
- 精度，对于字符串，说明输出字符的最大数目；对于 e、E和f格式，说明输出的小数位数；对于 g和G格式，说明

输出的有效位数；对于整数，说明输出的最小位数（必要时可加前导0）。

- 长度修饰符h、l、L "h"表示对应的变元按short或unsigned short类型输出。"l"表示对应的变元按long或unsigned long类型输出。"L"表示对应的变元按long double类型输出。

在格式串中字段宽和精度二者或其中任何一个均可用 *来指定，此时该值可通过转换下面的变元来求得，这些变元必须为 int类型。

转换字符和它们的命令如表 B-1所示。如果%后面的字符不是转换字符，那么该行为是未定义的。

表B-1 printf 函数的转换字符

格 式 码	变元类型；转换效果
d，i	int；有符号十进制表示法
o	int；无符号八进制表示法（无前导 0）
x，X	int；无符号十六进制表示法（无前导 0X和0x），对0x用abcdef，对0X用ABCDEF
u	int；无符号十进制表示法
c	int；单个字符，转换为 unsigned char类型后
s	char *；输出字符串直到遇到'\0'或者已达到由精度指定的字符数
f	double；形如[-] *mmm.ddd*的十进制浮点数表示法，*d*的数目由精度确定。缺省精度为6位，精度为0时不输出小数点
e，E	double；形如[-]*m.ddddde*±*xx*或[-] *m.ddddd*E±*xx*的十进制表示法。*d*的数目由精度确定，缺省精度为6位。精度为0时不输出小数点
g，G	double；当指数值小于- 4或大于等于精度时，采用 %e 或%E的格式；否则采用 %f的格式。尾部的0与小数点不打印
p	void *；输出指针值（具体表示与实现有关）
n	int *；到目前为止以此格式调用 printf输出的字符的数目将被写入到相应变元中。不进行变元转换
%	不进行变元转换；输出符号 %

```
int printf(const char *format, ...)
```

printf(...)函数等价于 fprintf(stdout,...)。

```
int sprintf(char *s, const char *format, ...)
```

sprintf函数与 printf函数的功能基本相同，但输出到数组 s中，并以'\0'结束。s必须足够大，以便能装下输出结果。该函数返回实际输出的字符数，不包括 '\0'。

```
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
```

vprintf、vfprintf与vsprintf这几个函数与对应的 printf函数等价，但变元表由 arg代替。arg由宏va_start初始化，由va _arg调用。参见B.7节对<stgarg.h>头文件的讨论。

B.1.3 格式化输入

scanf类函数用于提供格式化输入转换。

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf函数用于在格式串format的控制下从流stream中读入字符，把转换后的值赋给后续各个变元，在此每一个变元都必须是一个指针。当格式串format结束时函数返回。如果到达文件的末尾或在变元被转换前出错，那么该函数返回EOF；否则返回实际被转换并赋值的输入项的数目。

格式字符串format通常包含有用于指导输入转换的转换规格说明。格式字符串中可以包含：

- 空格或制表符，它们将被忽略。
- 普通字符（“%”除外），与输入流中下一个非空白的字符相匹配。
- 转换规格说明，由一个%、一个赋值屏蔽字符*（任选）、一个用于指定最大字段宽的数（任选）、一个用于指定目标字段的字符(h、l或L)（任选）以及一个转换字符组成。

转换规格说明决定了输入字段的转换方式。通常把结果保存在由对应变元指向的变量中。然而，如果转换规格说明中包含有赋值屏蔽字符“*”，例如%*s，那么就跳过对应的输入字段，不进行赋值。输入字段是一个由非空白符组成的字符串，当遇到空白符或到达最大字段宽（如果有的话）时，对输入字段的读入结束。这意味着scanf函数可以跨越行的界限来读入其输入，因为换行符也是空白符（空白符包括空格、横向制表符、纵向制表符、换行符、回车符和换页符）。

转换字符说明了输入字段的解释含义，对应的变元必须是指针。合法的转换符如表B-2所示。

表B-2 SCANF函数的转换字符

字 符	输入数据；变元类型
d	十进制整数；int *
i	整数；int *。该整数可以是以0开头的八进制数，也可以是以0x或0X开头的十六进制数
o	八进制整数（可以带或不带前导0）；int *
u	无符号十进制整数；unsigned int *
x	十六进制整数（可以带或不带前导0x或0X）；int *
c	字符；char *。按照字段宽的大小把读入的字符保存在指定的数组中，不加入字符'\0'。字段宽的缺省值为1。在这种情况下，不跳过空白符;如果要读入下一个非空白符，使用%1s
s	由非空白符组成的字符串（不包含引号）；char*。该变元指针指向一个字符数组，该字符数组有足够空间来保存该字符串以及在末尾添加的'\0'
e、f、g	浮点数；float *。float浮点数的输入格式为：一个任选的正负号，一串可能包含小数点的数字和一个任选的指数字段。指数字段由字母e或E以及后跟的一个可能带正负号的整数组成
p	用printf("%p")函数调用输出的指针值；void *
n	将到目前为止此调用所读的字符数写入变元；int *。不读入输入字符。不增加转换项目计数
[...]	用方括号括起的字符集中的字符来匹配输入，以找到最长的非空字符串；char *。在末尾添加字符'\0'。格式[]...]表示字符集中包含字符]
[^...]	用不在方括号括起的字符集中的字符来匹配输入，以找到最长的非空字符串；char *。在末尾添加字符'\0'。格式[^]...]表示字符集中包含字符]
%	字面值%，不进行赋值

如果变元是指向short类型而不是int类型的指针，那么在d、i、n、o、u和x这几个转换字符前可以加上字母h。如果变元是指向long类型的指针，那么在这几个转换字符前可以加上字母l。

如果变元是指向double类型而不是float类型的指针，那么在e、f和g这几个转换字符前可以加上字母l。如果变元是指向long double类型的指针，那么在e、f和g这几个转换字符前可以加上字母L。

```
int scanf(const char* format,...)
```

scanf(...)函数与fscanf(stdin, ...)等价。

```
int sscanf(char *s,const *format,... )
```

sscanf(s, ...)函数与scanf(...)等价，不同的是前者的输入字符来源于s。

B.1.4 字符输入输出函数

```
int fgetc(FILE *stream)
```

fgetc函数用于以unsigned char类型返回stream流中的下一个字符（转换为int类型）。如果到达文件的末尾或发生错误，则返回EOF。

```
char *fgets(char *s, int n, FILE *stream)
```

fgets函数用于读入最多n-1个字符到数组s中。当遇到换行符时，把换行符保留在数组s中，读入不再进行。数组s以'\0'结尾。fgets函数返回数组s，如果到达文件的末尾或发生错误，则返回NULL。

```
int fputc(int c, FILE *stream)
```

fputc函数用于把字符c（转换为unsigned char类型）输出到流stream中。它返回所写的字符，若出错则返回EOF。

```
int fputs(const char *s, FILE *stream)
```

fputs函数用于把字符串s（不包含字符'\n'）输出到流stream中；它返回一个非负值，若出错则返回EOF。

```
int getc(FILE *stream)
```

getc函数等价于fgetc，不同之处为：当getc函数被定义为宏时，它可能多次计算stream的值。

```
int getchar(void)
```

getchar函数等价于getc(stdin)。

```
char *gets(char *s)
```

gets函数用于把下一个字符串读入到数组s中，并把读入的换行符替换为字符'\0'。它返回数组s，如果到达文件的末尾或发生错误则返回NULL。

```
int putc(int c, FILE *stream)
```

putc函数等价于fputc，不同之处为：当putc函数被定义为宏时，它可能多次计算stream的值。

```
int putchar(int c)
```

putchar(c)函数等价于putc(c,stdout)。

```
int puts(const char *s)
```

puts函数用于把字符串 s和一个换行符输出到 stdout。如果发生错误，那么它返回 EOF；否则返回一个非负值。

```
int ungetc(int c, FILE *stream)
```

ungetc用于把字符 c（转换为 unsigned char类型）写回到流 stream中，下次对该流进行读操作时，将返回该字符。对每个流只能写回一个字符，且此字符不能是 EOF。ungetc函数返回被写回的字符；如果发生错误，那么它返回 EOF。

B.1.5 直接输入输出函数

```
size_t fread(void  *ptr, size_t size, size_t nobj, FILE *stream)
```

fread函数用于从输入流 stream中读入最多 nobj个长度为 size的对象到 ptr指向的数组中，并返回所读入的对象数，此返回值可能小于 nobj。实际执行状态可用函数 feof或ferror得到。

```
size_t fwrite(const void *ptr, size_t size, size_t nobj,
                        FILE *stream)
```

fwrite函数用于把 ptr所指向的数组中 nobj个长度为 size的对象输出到流 stream中，并返回被输出的对象数。如果发生错误，则返回一个小于 nobj的值。

B.1.6 文件定位函数

```
int fseek(FILE *stream, long offset, int origin)
```

fseek函数用于给与 stream流相关的文件定位，随后的读写操作将从新位置开始。对于二进制文件，此位置被定位在由 origin开始的 offset个字符处。origin的值可能为 SEEK_SET（文件开始处）、SEEK_CUR（当前位置）或 SEEK_END（文件结束处）。对于文本流，offset必须为 0，或者是由函数 ftell返回的值（此时 origin的值必须是 SEEK_SET）。fseek函数在出错时返回一个非 0 值。

```
long ftell(FILE *stream)
```

ftell函数用于返回与 stream流相关的文件的当前位置。在出错时返回 -1L。

```
void rewind(FILE *stream)
```

rewind(fp)函数等价于 fseek(fp，0L，SEEK_SET)与clearerr(fp)这两个函数顺序执行的效果。

```
int fgetpos(FILE *stream, fpos_t *ptr)
```

fgetpos函数用于把 stream流的当前位置记录在 *ptr中，供随后的 fsetpos函数调用时使用。若出错则返回一个非 0值。

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

fsetpos函数用于将流 stream的位置定位在 ptr所指向的位置。 *ptr的值是在函数 fgetpos调用时记录下来的。若出错则返回一个非 0值。

B.1.7 错误处理函数

当发生错误或到达文件末尾时，标准库中的许多函数将设置状态指示符。这些状态指示符可被显式地设置和测试。另外，（定义在<errno.h>中的）整数表达式errno可包含一个出错序号，该数将进一步给出最近一次出错的信息。

```
void clearerr(FILE *stream)
```

clearerr函数用于清除与流stream相关的文件结束指示符和错误指示符。

```
int feof(FILE *stream)
```

feof函数用于在与stream流相关的文件结束指示符被设置时返回一个非0值。

```
int ferror(FILE *stream)
```

ferror函数用于在与stream相关的文件出错指示符被设置时返回一个非0值。

```
void perror(const char *s)
```

perror(s) 函数用于输出字符串s以及与全局变量errno中的整数值相对应的出错信息，具体出错信息的内容依赖于实现。该函数的功能类似于

```
fprintf(stderr, "%s: %s\n", s, "出错信息")
```

参见B.3节所介绍的函数strerror。

B.2 字符类测试：<ctype.h>

头文件<ctype.h>中说明了一些用于测试字符的函数。每个函数的变元均为 int类型，变元的值必须是EOF或可用unsigned char类型表示的字符，函数的返回值为 int类型。如果变元c满足所指定的条件，那么函数返回非0值（表示真）；否则返回值为0（表示假）。这些函数如下：

isalnum(c)：函数isalpha(c)或isdigit(c)为真。
isalpha(c)：函数isupper(c)或islower(c)为真。
iscntrl(c)：c为控制字符。
isdigit(c)：c为十进制数字。
isgraph(c)：c是除空格外的可打印字符。
islower(c)：c是小写字母。
isprint(c)：c是包括空格的可打印字符。
ispunct(c)：c是除空格、字母和数字外的可打印字符。
isspace(c)：c是空格、换页符、换行符、回车符、横向制表符和纵向制表符。
isupper(c)：c是大写字母。
isxdigit(c)：c是十六进制数字。

在七位ASCII字符集中，可打印字符是从 0x20(' ')到0x7E('~')之间的字符；控制字符是从0(NUL)到0x1F(US)之间的字符和字符0x7F(DEL)。

另外，还有两个函数用于字母的大小写间的转换：

int tolower(int c)　　把c转换为小写字母
int toupper(int c)　　把c转换为大写字母

如果c是一个大写字母，那么tolower(c)返回对应的小写字母；否则返回c。如果c是一个小写字母，那么toupper(c)返回对应的大写字母；否则返回c。

B.3 字符串函数：<string.h>

在头文件<string.h>中定义了两组字符串函数。第一组函数的名字以str开头；第二组函数的名字以mem开头。只有函数memmove对重叠对象间的拷贝进行了定义，而其他函数则未对重叠对象间的拷贝进行定义。比较类函数将其变元视为unsigned char类型的数组。

在下面的各个函数中，变量s和t的类型为char *，cs和ct的类型为const char *，n的类型为size_t，c的类型为int（已被转换成char类型）。

```
char *strcpy(s,ct)
```

strcpy函数用于把字符串ct（包括'\0'）拷贝到字符串s当中，并返回s。

```
char *strncpy(s,ct,n)
```

strncpy函数用于把字符串ct中最多n个字符拷贝到字符串s中，并返回s。如果ct[⊖]中少于n个字符，那么就用一个'\0'补充。

```
char *strcat(s,ct)
```

strcat函数用于把字符串ct连结到s的尾部，并返回s。

```
char *strncat(s,ct,n)
```

strncat函数用于把字符串ct中最多n个字符连结到字符串s的尾部，并以'\0'结束；返回s。

```
int strcmp(cs,ct)
```

strcmp函数用于比较字符串cs和ct；当cs<ct时它返回一个负数；当cs>ct时它返回一个正数；当cs==ct时它返回0。

```
int strncmp(cs,ct,n)
```

strncmp函数用于将字符串cs中至多n个字符与字符串ct相比较。当cs<ct时它返回一个负数；当cs>ct时它返回一个正数；当cs==ct时它返回0。

```
char *strchr(cs,c)
```

strchr函数用于返回一个指向字符串cs中字符c第一次出现的位置的指针；如果cs中不包含c，那么该函数返回NULL。

```
char *strrchr(cs,c)
```

strrchr函数用于返回一个指向字符串cs中字符c最后一次出现的位置的指针；如果cs中不包含c，那么函数返回NULL。

```
size_t strspn(cs,ct)
```

strspn函数用于返回字符串cs中由字符串ct中的字符构成的第一个子串的长度。

```
size_t strcspn(cs,ct)
```

strcspn函数用于返回字符串cs中由不在字符串ct中的字符组成的第一个字串的长度。

⊖ 原书这个标识符为t，疑有误。——译者注

```
char *strpbrk(cs,ct)
```

strpbrk函数用于返回指向字符串 ct中的任意字符第一次出现在字符串 cs中的位置的指针；如果cs中没有与ct相同的字符，那么返回NULL。

```
char *strstr(cs,ct)
```

strstr函数用于返回指向字符串 ct第一次出现在字符串 cs中的位置的指针；如果 cs中不包含字符串ct，那么返回NULL。

```
size_t strlen(cs)
```

strlen函数用于返回字符串cs的长度。

```
char *strerror(n)
```

strerror函数用于返回指向与错误序号 n对应的错误信息字符串的指针（错误信息的具体内容依赖于实现）。

```
char *strtok(s,ct)
```

strtok函数在s中搜索由ct中的分界符界定的单词。见下面的讨论。

对strtok(s，ct)的一系列调用将把字符串 s分成许多单词，这些单词以 ct中的字符为分界符。第一次调用时s非空，它搜索s，找出由非ct中的字符组成的第一个单词，将 s中的下一个字符替换为'\0'，并返回指向单词的指针。随后，每次调用 strtok函数时（由s的值是否为NULL指示），均返回下一个由非ct中的字符组成的单词。当s中没有这样的单词时返回NULL。每次调用时字符串ct可以不同。

以mem开头的函数以字符数组的方式操作对象，其主要目的是提供一个高效的函数接口。在下面的函数中，s和t的类型均为void *，cs 和ct的类型均为const void *，n的类型为size_t，c的类型是int（被转换为unsigned char类型）。

```
void *memcpy(s,ct,n)
```

memcpy函数用于把字符串ct中的n个字符拷贝到s当中，并返回s。

```
void *memmove(s,ct,n)
```

memmove函数的功能与 memcpy相似，不同之处是当发生对象重叠时，该函数仍能正确执行。

```
int memcmp(cs,ct,n)
```

memcmp函数用于把cs的前n个字符与ct相比较，其返回值与strcmp的返回值相同。

```
void *memchr(cs,c,n)
```

memchr函数用于返回指向 c在cs中第一次出现的位置指针，如果在 cs的前n个字符当中找不到匹配，那么返回NULL。

```
void *memset(s,c,n)
```

memset函数用于把s中的前n个字符替换为c，并返回s。

B.4 数学函数：<math.h>

头文件<math.h>中说明了数学函数和宏。

宏EDOM和ERANGE（定义在头文件<error.h>中）是两个非0整常量，用于引发各个数学函数的定义域错误和值域错误；HUGE_VAL是一个double类型的正数。当变元取值在函数的定义域之外时，就会出现定义域错误。在发生定义域错误时，全局变量 errno的值被置为EDOM，函数的返回值视具体实现而定。如果函数的结果不能用 double类型表示，那么就会发生值域错误。当结果上溢时，函数返回HUGE_VAL并带有正确的符号（正负号），errno的值被置为ERANGE。当结果下溢时，函数返回0，而errno是否被设置为ERANGE视具体实现而定。

在下列函数中，x和y的类型为double，n的类型为int，所有函数的返回值的类型均为 double。三角函数的角度用弧度表示。

sin(x)	x的正弦。
cos(x)	x的余弦。
tan(x)	x的正切。
asin(x)	$\sin^{-1}(x)$，值域为$[-\pi/2, \pi/2]$，其中x $[-1, 1]$。
acos(x)	$\cos^{-1}(x)$，值域为$[0, \pi]$，其中x $[-1, 1]$。
atan(x)	$\tan^{-1}(x)$，值域为$[-\pi/2, \pi/2]$。
atan2(y,x)	$\tan^{-1}(y/x)$，值域为$[-\pi, \pi]$。
sinh(x)	x 的双曲正弦。
cosh(x)	x 的双曲余弦。
tanh(x)	x 的双曲正切。
exp(x)	幂函数e^x。
log(x)	自然对数$\ln(x)$，其中$x>0$。
log10(x)	以10为底的对数$\log10(x)$，其中$x>0$。
pow(x,y)	x^y。如果x=0且y≤0或者如果x<0且y不是整数，那么产生定义域错误。
sqrt(x)	x的平方根，其中$x \geq 0$。
ceil(x)	不小于x的最小整数，x的类型为double。
floor(x)	不大于x的最大整数，x的类型为double。
fabs(x)	绝对值$\|x\|$。
ldexp(x,n)	$x \cdot 2^n$。
frexp(x,int *exp)	把x分成一个在[1/2,1）区间的真分数和一个 2的幂数。将真分数返回，幂数保存在*exp中。如果x为0，那么这两部分均为0。
modf(x,double *ip)	把x分成整数和小数两部分，两部分均与 x有相同的正负号。函数返回小数部分，整数部分保存在 *ip中。
fmod(x,y)	求x/y的浮点余数，符号与 x相同。如果y为0，那么结果由具体实现而定。

B.5 实用函数：<stdlib.h>

在头文件<stdlib.h>中说明了用于数值转换、内存分配以及具有其他相似任务的若干函数。

```
double atof(const char *s)
```
atof函数用于把字符串s转换成double类型；该函数等价于。
```
    strtod(s, (char**)NULL)。
```

```
int  atoi(const char *s)
```
atoi函数用于把字符串s转换成int类型；该函数等价于
```
    (int)strtol(s, (char**)NULL, 10)。
```

```
long atol(const char* s)
```
atol函数用于把字符串s转换成long类型；该函数等价于
```
    strtol(s, (char**)NULL, 10)
```

```
double strtod(const char *s, char **endp)
```

strtod函数用于把字符串 s的前缀转换成 double类型，在转换时跳过 s的前导空白符。除非endp为NULL，否则该函数把指向 s中未转换部分（ s的后缀部分）的指针保存在 *endp中。如果结果上溢，那么函数就返回带有适当符号的 HUGE_VAL；如果结果下溢，那么函数返回 0。在这两种情况下，errno均被置为ERANGE。

```
long strtol(const char *s, char **endp, int base)
```

strtol函数用于把字符串s的前缀转换成long类型，在转换时跳过s的前导空白符。除非endp为NULL，否则该函数把指向s中未转换部分（s的后缀部分）的指针保存在 *endp中。如果base的取值在2到36之间，那么转换的输入是以该基数进行的；如果 base的取值为 0，那么基数为八进制、十进制或十六进制。以0为前导的是八进制，以0x或0X为前导的是十六进制。无论在哪种情况下，字母均表示10到base -1之间的数字。如果 base值是16，可以加上前导 0x或0X。如果结果上溢，那么函数依据结果的符号返回 LONG_MAX或LONG_MIN，同时将errno的值置为ERANGE。

```
unsigned long strtoul(const char* s, char **endp, int base)
```

strtoul函数的功能与strtol函数相同，只是结果为unsigned long类型，错误值为ULONG_MAX。

```
int rand(void)
```

rand函数用于产生一个 0到RAND_MAX之间的伪随机整数。 RAND_MAX的取值至少为32767。

```
void srand(unsigned int seed)
```

srand函数用参数seed作为生成新的伪随机数序列的种子。种子 seed的初值为1。

```
void *calloc(size_t nobj, size_t size)
```

calloc函数为由nobj个大小为size的对象组成的数组分配足够的内存，并返回指向所分配区域的第一个字节的指针；若内存不足以满足要求，则返回NULL。该空间的初始大小为0字节。

```
void *malloc(size_t size)
```

malloc函数用于为大小为 size的对象分配足够的内存，并返回指向所分配区域的第一个字节的指针；若内存不足以满足要求，则返回 NULL。不对分配的内存区域进行初始化。

```
void *realloc(void *p, size_t size)
```

realloc函数用于将p所指向的对象的大小改为 size个字节。如果新分配的内存比原内存大，那么原内存的内容保持不变，增加的空间不进行初始化。如果新分配的内存比原内存小，那么新内存保持原内存区单元的内容。 realloc函数返回指向新分配空间的指针；若不能满足要求，则返回NULL，在这种情况下，原p指向的内存区保持不变。

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

free函数用于释放p所指向的内存空间；当 p的值为NULL时，该函数不起作用。 p必须指向先前用动态分配函数malloc、realloc或calloc分配的空间。

```
void abort(void)
```

abort函数用于使程序非正常终止。其功能与 raise(SIGABRT)类似。

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

exit函数用于使程序正常终止。 atexit函数以与注册的相反顺序被调用，此时，所有已打开的文件被刷新，所有已打开的流被关闭，控制返回给环境。 status的值如何被返回依具体的实现而定，但用0值表示终止成功。也可使用值EXIT_SUCCESS和EXIT_FAILURE。

```
int atexit(void (*fcn)(void))
```

atexit函数用于注册在程序正常终止时所要调用的函数 fcn。如果注册失败，那么返回非0值。

```
int system(const char *s)
```

system函数用于把字符串 s传送给执行环境。如果 s的值为NULL，那么在存在命令处理程序时，返回非0值。如果s的值不是NULL，那么返回值与具体的实现有关。

```
char * getenv(const cahr *name)
```

getenv函数用于返回与name有关的环境字符串。如果该字符串不存在，那么返回 NULL。其细节与具体的实现有关。

```
void * bsearch(const void *key, const void *base,
               size_t n, size_t size,
               int (*cmp)(const void *keyval, const void *datum)
```

bsearch函数用于在base[0]...base[n-1]之间查找与*key匹配的项。 cmp函数在其第一个变元（查找关键字，keyval）小于第二个变元（表项，datum）时必须返回一个负值；在其第一个变元等于第二个变元时必须返回零；在其第一个变元大于第二个变元时必须返回一个正值。数组 base中的项必须按升序排列。 bsearch函数返回一个指向匹配项的指针；若不存在匹配的项则返回NULL。

```
void qsort(void *base, size_t n, size_t size,
                int (*cmp)(const void *, const void *))
```

qsort函数用于对由 n个大小为 size的对象构成的数组 base进行升序排序。比较函数 cmp与bsearch函数中的描述相同。

```
int abs(int n)
```

abs函数用于返回其int类型变元n的绝对值。

```
long labs(long n)
```

labs函数用于返回其long类型变元n的绝对值。

```
div_t div(int num, int denom)
```

div函数用于求num/denom的商和余数，并把结果分别保存在结构类型 div_t的两个int类型的数quot和rem中。

```
ldiv_t ldiv(long num, long denom)
```

ldiv函数用于求num/denom的商和余数，并把结果分别保存在结构类型 ldiv_t的两个long类型的数quot和rem中。

B.6 诊断：<assert.h>

assert宏用于为程序增加诊断功能：

```
void assert(int表达式)
```

当

```
assert(表达式)
```

执行时，如果表达式的值为0，那么assert宏将在标准出错输出流stderr输出一条如下所示的信息：

```
Assertion failed: 表达式, file 文件名, line nnn
```

然后调用abort终止执行。其中的源文件名和行号来自于预处理程序宏 _ _FILE_ _和_ _LINE_ _。

如果在头文件<assert.h>被包含时定义了宏NDEBUG，那么宏assert将被忽略。

B.7 变长变元表：<stdarg.h>

头文件<stdarg.h>提供了依次处理含有未知数目和类型的函数变元表的功能。

假设函数f含有可变数目的变元，*lastarg*是它的最后一个有名参数，然后在 f内说明一个类型为va_list的变量ap，它将依次指向每个变元：

```
va_list ap;
```

在访问任何未命名的变元前必须用 va_start宏对ap进行初始化：

```
va_start( va_list ap, lastarg);
```

此后，宏va_arg的每次执行将产生一个与下一个未命名的变元有相同类型和值的值，它同时还修改ap，以使下一次使用va_arg时返回下一个变元：

```
type va_arg(va_list ap, 类型);
```

当所有的变元被处理完毕之后 f要退出执行以前，必须调用一次宏 va_end：

```
void va_end(va_list ap);
```

B.8 非局部跳转：<setjmp.h>

头文件<setjmp.h>中的说明提供了一种避免通常的函数调用和返回顺序的途径，特别地，它允许立即从一个多层嵌套的函数调用中返回。

```
int setjmp(jmp_buf env)
```

setjmp宏用于把当前状态信息保存到env中，供以后longjmp函数恢复状态信息时使用。如果是直接调用setjmp，那么返回值为0；如果是由于调用longjmp而调用setjmp，那么返回值非0。setjmp只能在某些特定的情况下调用，如在if语句、switch语句及循环语句的条件测试部分以及一些简单的关系表达式中：

```
if (setjmp(env) == 0)
   /*直接调用setjmp时进行的处理*/
else
   /*由于调用longjmp函数而进行的处理*/
```

```
void longjmp(jmp_buf env, int val)
```

longjmp用于恢复由最近一次调用setjmp时所保存到env中的状态信息。当它执行完时，程序就像setjmp宏调用刚刚执行完并返回非0值val那样继续执行。包含setjmp宏调用的函数的执行不必已经终止。所有可访问的对象的值都与调用longjmp时相同，唯一的例外是，那些调用setjmp宏的函数中的非volatile自动变量如果在调用setjmp宏后有了改变，那么就变成未定义的。

B.9 信号处理：<signal.h>

头文件<signal.h>中提供了一些用于处理程序运行期间所引发的异常条件的功能，如处理来源于外部的中断信号或程序执行期间出现的错误等异常事件。

```
void (*signal(int sig, void (*handler)(int))) (int)
```

signal用于确定处理以后的信号的方法。如果handler的值是SIG_DFL，那么就采用由实现定义的缺省行为；如果handler的值是SIG_IGN，那么就忽略该信号；否则，调用handler所指向的函数（变元的类型为信号的类型）。有效的信号包括：

SIGABRT	异常终止，如由abort发出的信号。
SIGFPE	算术运算出错，如除数为0或溢出。
SIGILL	非法函数映像，如非法指令。
SIGINT	交互式意图，如中断。
SIGSEGV	非法访问存储器，如访问不存在的内存单元。
SIGTERM	发送给本程序的终止请求信号

signal对指定信号返回handler的前一个值；如果出现错误，那么返回值为SIG_ERR。

当随后出现信号sig时，该信号被恢复为其缺省行为，然后调用信号处理函数，如(*handler)(sig)。如果从信号处理程序中返回，那么程序就从发信号时的位置重新开始执行。

信号的初始状态由实现定义。

```
int raise(int sig)
```

raise用于向程序发送信号sig。如果发送不成功，就返回一个非0值。

B.10 日期与时间函数：<time.h>

头文件<time.h>中说明了一些用于处理日期与时间的类型和函数。其中的一部分函数用于处理当地时间，因为时区等原因，当地时间与日历时间可能不相同。clock_t和time_t是两个用于表

示时间的算术类型，而struct tm则用于存放日历时间的各个成分。tm的各个成员的用途及取值范围如下：

int tm_sec；	从当前分钟开始经过的秒数（0，61）。
int tm_min；	从当前这小时开始经过的分钟数（0，59）。
int tm_hour；	自午夜（0，23）。
int tm_mday；	本月中的天数（1，31）。
int tm_mon；	从1月起经过的月数（0，11）。
int tm_year；	从1900年起经过的年数。
int tm_wday；	从星期天起经过的天数（0，6）。
int tm_yday；	从1月1日起经过的天数（0，365）。
int tm_isdst；	夏令时标记。

其中，tm_isdst在使用夏令时时其值为正，在不使用夏令时时其值为 0。如果该信息不能使用，那么tm_isdst的值为负。

```
clock_t clock(void)
```

clock函数用于返回程序自开始执行到目前为止所占用的处理机时间；如果处理机时间不可使用，那么返回值为-1。clock()/CLOCKS_PER_SEC是以秒为单位表示的时间。

```
time_t time(time_t *tp)
```

time函数用于返回当前日历时间；如果日历时间不能使用，那么返回值为 -1。如果tp的值不为NULL，那么同时把返回值赋给*tp。

```
double difftime(time_t time2, time_t time1)
```

difftime函数用于返回time2-time1的值（以秒为单位）。

```
time_t mktime(struct tm *tp)
```

mktime函数用于将结构*tp中的当地时间转换为time_t类型的日历时间（可被time函数使用）。结构中各成分具有上面所示范围之内的取值。 mktime函数返回转换成得到的日历时间；如果该时间不能表示，那么就返回-1。

下面四个函数用于返回指向可被其他调用重写的静态对象的指针。

```
char *asctime(const struct tm *tp)
```

asctime函数用于将结构*tp中的时间转换成如下所示的字符串形式：

```
    Sun Jan  3  15:14:13  1998\n\0
char *ctime(const time_t *tp)
```

ctime函数用于将结构*tp中的日历时间转换成当地时间。它等价于如下函数调用

```
        asctime(localtime(tp))
struct tm *gmtime(const time_t *tp)
```

gmtime函数用于将*tp中的日历时间转换成世界协调时（ UTC）。如果UTC不能使用，那么

该函数返回NULL。函数名字“gmtime”有其历史意义⊖。

```
struct tm *localtime(const time_t *tp)
```

localtime函数用于将结构*tp中的日历时间转换成当地时间。

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *tp)
```

strftime函数用于根据fmt的值把结构*tp中的日期与时间信息转换成指定的格式并存储到s所指向的数组中，其中fmt类似于printf函数中的格式说明，它由0个或多个转换规格说明与普通字符组成。普通字符（包括终结行的空字符'\0'）都原封不动地拷贝到s中。每个%*c*按照下面所描述的格式用与当地环境相适应的值来替换。写到s中的字符数不得大于smax。Strftime函数返回实际写到s中的字符数（不包括空字符'\0'）；如果所产生的字符数多于smax，该函数返回值为0。

在strftime函数中可以使用的转换规格说明及其所表示的含义如下：

%a	一星期中各天的缩写名。
%A	一星期中各天的全名。
%b	缩写月份名。
%B	月份全名。
%c	当地时间和日期表示。
%d	用整数表示的一个月中的某一天（01~31）。
%H	用整数表示的时（24小时制，00~23）。
%I	用整数表示的时（12小时制，01~12）。
%j	用整数表示的一年中各天（001~366）。
%m	用整数表示的月份（01~12）。
%M	用整数表示的分（00~59）。
%p	与AM与PM对应的当地表示方法。
%S	用整数表示的秒（00~61）。
%U	用整数表示一年中的星期数（00~53，将星期日看作是每周的第一天）。
%w	用整数表示一周中的各天（0~6，星期日为0）。
%W	用整数表示一年中的星期数（00~53，将星期一看作是每周的第一天）。
%x	当地日期表示。
%X	当地时间表示。
%y	不带世纪号的年份（00~99）。
%Y	完整年份表示（带世纪号）。
%Z	时区名字（在可以使用时）。
%%	%本身。

B.11 由实现定义的限制：<limits.h>和<float.h>

头文件<limits.h>中定义了用于表示整类型大小的常量。以下所列的值是可接受的最小值，

⊖ gmtime是英文“Greenwich Mean Time（格林尼治标准时间）”的缩写。——译者注

在实际系统中可以使用更大的值。

CHAR_BIT	8	char类型的位数
CHAR_MAX	UCHAR_MAX或SCHAR_MAX	char类型的最大值
CHAR_MIN	0或SCHAR_MIN	char类型的最小值
INT_MAX	+32767	int类型的最大值
INT_MIN	-32767	int类型的最小值
LONG_MAX	+2147483647	long类型的最大值
LONG_MIN	-2147483647	long类型的最小值
SCHAR_MAX	+127	signed char类型的最大值
SCHAR_MIN	-127	signed char类型的最小值
SHRT_MAX	+32767	short类型的最大值
SHRT_MIN	-32767	short类型的最小值
UCHAR_MAX	255	unsigned char类型的最大值
UINT_MAX	65535	unsigend int类型的最大值
ULONG_MAX	4294967295	unsigned long类型的最大值
USHRT_MAX	65535	unsigned short类型的最大值

下面列出的名字是<float.h>的一个子集，是与浮点算术运算相关的一些常量。给出的每个值代表相应量的一个最小取值。各个实现可以定义适当的值。

FLT_RADIX	2	指数表示的基数，如2、16
FLT_ROUNDS		加法的浮点舍入规则
FLT_DIG	6	精度的十进制数字数
FLT_EPSILON	1E-5	使1.0+x (1.0成立的最小的x
FLT_MANT_DIG		基数为FLT_RADIX的尾数中的数字数
FLT_MAX	1E+37	最大浮点数
FLT_MAX_EXP		使FLT_RADIX^n-1可表示的最大的n
FLT_MIN	1E-37	最小规范化的浮点数
FLT_MIN_EXP		使10^{n}为规范化数的最小的n
DBL_DIG	10	精度的十进制字数
DBL_EPSILON	1E-9	使1.0+x 1.0成立的最小的x
DBL_MANT_DIG		基数为FLT_RADIX的尾数中的数字数
DBL_MAX	1E+37	最大双精度浮点数
DBL_MAX_EXP		使FLT_RADIX^n-1可以正常表示的最大的n
DBL_MIN	1E-37	最小双精度浮点数
DBL_MIN_EXP		使10^{n}为规范化数的最小的n

附录C 变更小结

自本书第1版出版以来，C语言的定义已经经历了一定的变化。几乎每一次变化都是对原语言的一次扩充，同时每一次扩充都是经过仔细设计的，保持了与现有版本的兼容，其中有一些修正了原版本中的歧义性描述；另一些是对已有版本的变更。许多新增功能都是随AT&T所提供的编译程序的文档一同发布的，并被此后的其他C编译程序供应商采用。前不久，ANSI标准化委员会在对C语言标准化时采纳了其中绝大部分的变动，并作了其他一些重要修正。ANSI的报告甚至在正式的C标准发布之前就被一些编译程序提供商部分地先期使用了。

本附录总结了本书第1版所定义的C语言与ANSI新标准之间的差别。这里仅讨论语言本身，不涉及其环境和库。尽管环境和库也是标准的重要组成部分，但它们与第1版几乎无可比之处，因为第1版并没有试图规定一个环境或库。

- 与第1版相比，标准C中关于预处理的定义更加细致，并进行了扩充：它明显是以单词为基础的；增加了连结单词的运算符（ ##）和产生字符串的运算符（#）；增加了新的控制行（如#elif和#pragma）；明确允许重新说明由同样单词序列所组成的宏；字符串中的参数不再被替换。允许在任何地方使用反斜杠字符“\”进行行的连接，而不仅仅是在字符串和宏定义中。参见A.12节。
- 所有内部标识符的最小有效长度增加为31个字符；具有外部连接的标识符的最小有效长度仍然为6个单字符（在很多实现中都允许更长的标识符）。
- 由双问号“??”引入的三字符序列允许表示某些字符集中缺少的字符。定义了#\^[]{}|~等转义字符，见A.12.1节。注意到三字符序列的引入可能会改变包含“??”的字符串的意义。
- 引入了新的关键字（void、const、volatile、signed和enum）。关键字entry将不再使用。
- 定义了在字符常量和字符串字面量中使用的新的转义序列。如果由“\”和其后的字符所构成的不是转义序列，那么其结果是未定义的。参见A.2.5节。
- 所有人都喜欢的小小变化：8和9不作为八进制数字。
- 新标准引入了更大的后缀集合，使得常量的类型更加明确：U或L用于整数，F或L用于浮点数。它同时也细化了有关无后缀常量类型的规则（参见A.2.5节）。
- 相连的字符串被连结在一起。

- 提供了表示宽字符字符串字面值和字符常量的方法，参见 A.2.6节。
- 同其他类型一样，对字符类型也可以使用关键字 signed或unsigned明确地说明是有符号的还是无符号的。 long float作为double的同义词这种独特的用法被放弃了，但可以用 long double来说明有更高精度的浮点数。
- 曾经有段时间， unsigned char类型是有效的。新标准引入了关键字 signed，用来明确表示字符和其他整类型对象的符号。
- 很多编译程序在几年前就实现了 void类型。新标准引入了 void *类型作为通用指针类型；在此之前用char *来担负这一角色。同时，明确地规定了在不进行强制转换时有关在指针与整数之间以及不同类型的指针之间运算的规则。
- 新标准明确指定了算术类型的取值范围的最小值，并在两个头文件（ <limits.h>和<float.h>）中给出了每一种特定实现的特征。
- 新增加的枚举类型是第 1版中所没有的。
- 标准采用了C++中的类型限定符的概念，如const（参见A.8.2节）。
- 字符串不再是可以修改的，因此可以放在只读内存区中。
- “ 通常的算术转换 ”已被改变，特别地，“ 整数总是被转换为无符号数，浮点数总是被转换为双精度数 ”已更改为 “ 提升到最小的足够大的类型 ”。参见A.6.5节。
- 旧的赋值类运算符如 “ =+ ”等已不再存在。同时，赋值类运算符现在作为单个单词；而在第1版中，它们是两个单词，中间可以用空白符分开。
- 在编译程序中，不再可以将数学上可结合的运算符当做计算上也是可结合的。
- 为了和一元减一运算符 “ － ”对称，引入了一元增一运算符 “ ＋ ”。
- 指向函数的指针可以用作函数的标记符，而不需要显式的 *运算符。参见A.7.3节。
- 结构可以被赋值、作为变元传给函数以及被函数返回。
- 允许对数组应用求地址运算符，其结果为指向数组的指针。
- 在第1版中， sizeof运算符运算的结果为 int类型，但随后很多编译程序都将此结果实现为unsigned类型。标准明确了该运算符的结果类型依赖于具体的实现，但要求其类型定义应包含在标准头文件 <stddef.h>中。关于求两个指针的差的类型 (ptrdiff_t)也有类似的改变。参见A.7.4节与A.7.7节。
- 地址运算符&不可应用于被说明为register的对象，即使具体的实现未将这种对象存放在寄存器中。
- 移位表达式的类型是其左运算分量的类型，右运算分量不能提升结果。参见 A.7.8节。
- 标准允许创建一个指向数组最后一个元素下一个位置的指针，并允许对其进行算术和关系运算。参见A.7.7节。
- 标准（从C++吸收）引入了函数原型说明的表示法。在函数原型中可以说明参数的类型，可以说明带有可变参数表的函数，并允许以一种一致的方法来处理。参见 A.7.3节、A.8.6和B.7节。旧风格的函数仍然被接受，但有一定限制。
- 标准禁止空的说明，即没有说明符以及不包含至少一个结构、联合或枚举的说明。另一方面，只说明一个结构或联合标记的说明是对该标记的重新说明，即使它已在外作用域中说

明过。

- 禁止没有任何限定符或说明符的外部数据说明（只是一个说明符）。
- 在某些实现中，如果在内部分程序中包含了一个 extern说明，那么该说明将对所在文件的其他部分可见。标准明确规定这种说明的作用域仅为该分程序。
- 参数的作用域被扩展到函数的复合语句中，因此，在函数顶层的变量说明不能隐藏参数。
- 标识符的名字空间有一些不同。标准将所有的标记放在一个单独的名字空间中，同时也为标号引入了一个单独的名字空间，参见 A.11.1节。结构或联合的成员名将与其所属的结构或联合相关联。（这已经是许多实现的共同做法了。）
- 联合可以被初始化，初始化符指向其第一个成员。
- 有自动存储类的结构、联合和数组可以以一种受限的方式初始化。
- 明确指定大小的字符数组可以用与此长度相同的字符串字面值初始化（“ \0 ”被不知不觉地挤掉了）。
- switch语句的控制表达式和 case标号可以是任何整类型。

附录D 索 引

（以下按字母排序）

G

H

I

L

T

U

（以下按拼音排序）

计算机科学丛书

中文版

《C程序设计语言》	Kernighan等著/徐宝文 等译/28.00元
《数据库系统实现》	Garcia-Molina等著/杨冬青 等译/45.00元
《分布式系统设计》	Wu著/高传善 等译/30.00元
《数据库系统导论》	Date著/孟小峰 等译/66.00元
《计算机网络实用教程》	Dean 著/陶华敏 等译/65.00元
《计算机网络实用教程实验手册》	Dean 著/陶华敏 等译/15.00元
《设计模式：可复用面向对象软件的基础》	Gamma 著/吕建 等译/35.00元
《计算机网络与因特网》	Comer 著/徐贤良 等译/40.00元
《程序设计实践》	Kernighan 著/裘宗燕 译/20.00元
《软件需求》	Wiegers著/陆丽娜 等译/19.00元
《C程序设计教程》	Deitel 著/薛万鹏 等译/33.00元
《C++程序设计教程》	Deitel 著/薛万鹏 等译/22.00元
《专家系统原理与编程》	Giarratano 著/印鉴 等译/49.00元
《数据仓库》	Inmon 著/王志海 等译/25.00元
《UNIX操作系统设计》	Bach 著/陈葆珏 等译/33.00元
《计算理论导引》	Sipser 著/张立昂 等译/30.00元
《编译原理及实践》	Louden 著/冯博琴 等译/39.00元
《数字逻辑：应用与设计》	Yarbrough 著/朱海滨 等译/49.00元
《数据通信与网络教程》	Shay 著/高传善 等译/40.00元
《TCP/IP详解 卷1：协议》	Stevens著/范建华 等译，谢希仁校/ 45.00元
《TCP/IP详解 卷2：实现》	Stevens著/陆雪莹 等译，谢希仁校/ 78.00元
《TCP/IP详解 卷3：TCP事务协议、HTTP、NNTP和UNIX域协议》	Stevens著/胡谷雨 等译，谢希仁校/ 35.00元
《UNIX环境高级编程》	Stevens著/尤晋元 等译/55.00元
《计算机文化》	Parsons 著/朱海滨 等译/50.00元
《C++编程思想》	Eckel著/刘宗田 等译，袁兆山 等校/39.00元
《信息系统原理》	Stair著/张靖 等译/42.00元
《计算机信息处理》	Mandell著/尤晓东 等译/38.00元
《现代操作系统》	Tanenbaum 著/陈向群 等译/40.00元
《UNIX编程环境》	Kernighan 著/陈向群 等译/24.00元
《最新网络技术基础》	Palmer著/严伟 译/20.00元
《可扩展并行计算：技术、结构与编程》	Hwang 著/陆鑫达 等译/49.00元

《数据库管理系统基础》	Pratt 著/陆洪毅 等译/20.00元
《数据通信与网络》	Forouzan 著/潘伦 等译，吴时霖 校/48.00元
《Java编程思想》	Eckel著/京京工作室译/60.00元
《数据结构、算法与应用——C++语言描述》	Sahni 著/汪诗林 译，王广芳校/49.00元
《软件工程：JAVA语言实现(第4版)》	Schach 著/袁兆山 等译/38.00元
《软件工程：实践者的研究方法(第4版)》	Pressman 著/黄柏素、梅宏 译/48.00元
《数据库系统概念(第3版)》	Silberschatz 著/杨冬青、唐世渭 等译/49.00元
《Internet技术基础》	Comer著/袁兆山 等译/18.00元

影印版

《计算机网络(第2版)》	Peterson 著/65.00元
《人工智能》	Nilsson 著/45.00元
《并行计算机系统结构(第2版)》	Culler 著/88.00元
《计算机体系结构：量化研究方法(第2版)》	Hennessy, Patterson 著/88.00元
《计算机组织与设计——硬件/软件接口(第2版)》	Patterson, Hennessy 著/80.00元
《可扩展并行计算——技术、结构与编程》	Hwang 著/69.00元
《离散数学及其应用(第4版)》	Rosen 著/59.00元
《数据通信与网络》	Forouzan 著/59.00元
《数据结构算法与应用——C++语言描述》	Sahni 著/66.00元
《软件工程：JAVA语言实现(第4版)》	Schach 著/51.00元
《软件工程——实践者的研究方法(第4版)》	Pressman 著/68.00元
《通信网络基础(第2版)》	Walrand 著/32.00元
《数据库系统概念(第3版)》	Silberschatz 著/65.00元
《高性能通信网络（第2版）》	Walrand 著/64.00元
《高级计算机体系结构》	Hwang 著/59.00元
《系统分析与设计（英文版）》	Satzinger 著/60.00元