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UNIX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 B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语句的第一个变元的%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二个版本：

```
#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，在写浮点常数时最好还是把它写成带小数点，即使该浮点常数取的是整数值，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，它通常把字符打印，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2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1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1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);
}
```

%f可用于float与double类型，%.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for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while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A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nc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、空白字符（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一个版本：

```
#include  <stdio.h>
/* 统计各个数字、空白字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", 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、空白字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北幂的函数 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  */
/*  (老风格版本)  */
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中仍可以使用老风格的函数说明与定义，至少它可以作为一个过渡阶段。但我们还是强烈建议读者：在可以使用支持新风格的编译程序时，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被用作临时变量，（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。当把数组名用作变元时，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应作成变量或符号参数吗？

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字符之后。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。）