Microsoft 4.0版

C 语 言

程 序 设 计

（上 卷）

北京中国科学院希望电脑公司

一九八八年四月

Microsoft 4.0版

C 语 言

数

程 序 设 计

（上 卷）

北京中国科学院希望电脑公司

一九八八年四月

目　录

第一章 导 论

开始，我们对C作一速成介绍。我们的目标是指出实际程序中语言的实质性成分，而不去陷入细节、形式规则和例外情况。因此，不追求完整性，或者甚至不追求严密性（只是举的例子要正确）。我们希望尽快地使你达到能编写有用的程序的水平，把我们的注意力集中在基本的方面：变量和常量，算术运算，控制流，函数和输入输出基据原理。我们故意从这一章中舍去C的一些在写较大的程序时非常重要的特性。如指针、结构、大部分C运算符，若干控制流语句和无数细节。

这种方法当然有它的弱点。最显著的弱点是从一个地方看不出任何一个特定语言特性的完整内容，而且由于导论的简短扼要还可能带错路，由于不能利用C的全部功能，例子不能象可能做到的那样精确和合理。我们尽量把这种影响减少到最低限度。

另一个弱点是以后的章节必定要重复本章的一些内容。我们希望这些重复对你有所帮助而不是增加烦脑。

无论如何，有经验的程序人员可从本章的内容中断定他们自己的程序设计需要。初学者通过写他们自己小的和类似的程序作为补充。两种人都可用它作为一个框架子，挂上在第二章开始的更详细的描述。

1.1 开始

学习一个新程序设计语言的唯一途径是用它写程序。要写的第一个程序对所有语言都是相同的：

打印字

```
hello, world
```

这是基本的障碍；为了跨越此障碍，你必须会在某个地方创建程序文本，成功地编译，装入它，运行它，并在你输出的地方找到它。掌握了这些呆板的细节，其它任何事情相对地就容易了。

在C中，打印“hello, world”的程序是：

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

如何运行此程序取决于你正使用的系统。

现在就程序本身作些说明。一个C程序不管它的长度多长，都由一个或几个“函数”所组成，它们指定了实际要做的计算操作。C中的函数类似于Fortran 程序中的函数和子程序，或者PL/I，Pascal等中的过程。在我们的例子中，main 就是这样的函数。通常可以任意给函数取你喜欢的名字，但main是一个特殊的名字——你的程序在main的开始处开始执行，这意味着，每一个程序总必须在某个地方有一个main函数。main常常要引用其它函数来执行它的工作，有些来自同一个程序，另一些则来自以前写好的函数库中。

函数间数据通讯的一种方法是通过参数。函数名后面的括号把参数表括起来，这里main

是无参函数，用（　）表示。花括号｛　｝把构成函数的语句括起来，它类似于PL／I中的DOEMD或Algol，Pascal中的begin-end等等。函数名后面跟以用括号括起来的参数表示调用该函数。没有Fortran或PL／I中的那种CALL语句。即使没有参数也必须带括号。

```
Printf(″hello, world／n″)
```

语句行是一个函数调用，调用名叫Printf的函数，带参数"hello, world／n″。printf是库函数，它在终端上（除非指定了其它目标）打印输出。这里是印出组成参数的字符串。

括号在双引号"…″里的任何数目的字符序列叫做字符串或字符串常数。目前我们使用的字符串将只作为printf和其它函数的参数。

在字符串中的序列／n是C中的换行字符的表示方法，在打印时，它使终端前进到下一行的左首页边。如果你不用／n（这是有价值的试验），就会发现你的输出未用换行结束。把换行字符放入printf参数的唯一方法是用／n，如果你试图采用象

```
printf("hello, world
″);
```

这样的办法，C编译程序毫不客气地印出关于缺引号的诊断信息。

printf决不自动提供换行，所以可以用多次调用一步步地构成输出行。我们已经写的第一个程序与

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

一样产生完全相同的输出。

注意＼n只代表一个字符。象＼n那样的转义序列为表示难于得到的或看不到的字符提供了通用的和可扩充的手段。此外，C中还用＼t表示制表字符，＼b表示退一格，＼″表示双引号，＼＼表示反斜线＼本身。

1.2　变量和算术运算

下面的程序用公式C＝（5＼9）（F-32）印出下列华氏温度和摄字温度的对照表：

F	C
0	—17.8
20	—6.7
40	4.4
60	15.6
…	…
260	126.7
280	137.8
300	148.9

这里是程序本身：

```
\* print Fahrenheit-Celsius table
   for f=0,20, ···, 300 * /
main( )
{
   int lower, upper, step;
   float fahr, celsius;
   lower=0;      /* lower limit of temperature table */
   upper=300;    /* upper limit */
   step=20;      /* step size */
   fahr=lower;
   while( fahr<=upper ) {
         celsius=(5.0/9.0) * (fahr-32.0);
         printf("%4.0f%6.1f/n", fahr, celsius);
         fahr=fahp+step;
   }
}
```

头两行

```
  /* print Fahrenheit-Gelsius table
     for f=0, 20, ···, 300 * /
```

是一个注解，在这里扼要地说明程序要干什么。在/*和*/之时的任何字符，编译程序是不管的，为使程序易于理解，可以随便使用它。注解可出现在空白或换行能出现的任何地方。

在C中，所有变量都必须在使用之前加以说明，通常说明出现在函数开始处，在所有可执行语句之前。如果你忘了说明，将得到来自编译程序的诊断信息。说明由类型和具有该类型的变量表组成，如：

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

类型int表示所列的变量均是整数。float表示浮点数，即数中可带有小数部分。int和float的精度均依赖于你使用的具体机器。例如，在PDP-11中，int是16位带符号的数，也就是在-32768到+32767之间的数。float数长32位，达到大约七位有效数字，数量范围约在10^{-38}和10^{+38}。

除了int和float之外C还提供了若干其它的基本数据类型：

char	字符——一个单字节
short	短整数
long	长整数
double	双精度浮点数

这些对象的大小均是依赖于机器的，将在第二章中详述。还有这些基本类型的数组，结构和联合，对它们的指针，以这些基本类型的值为结果的函数，所有这些将在一定的时候遇到。

温度转换程序中的实际计算从下述赋值语句开始：

```
lower = 0;
upper = 800;
step = 20;
fahr = lower;
```

它们为变量赋初值。每个语句均用分号结束。

表中每一行用同一方法计算，所以我们使用循环，每一行重复一次，这就是while语句的目的：

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

检查括号中的条件。如果为真（fahr小于或等于upper），执行循环体（花括号{和}括起来的所有语句）。然后再对条件进行检查，如仍为真，再执行循环体。当检查结果为假时（fahr比upper大了）循环结束，从该循环之后的语句继续执行。在本章序中，它后面没有语句了，所以程序结束。

while中的体可以象温度转换程序那样，是括在花括号的一个或几个语句，如

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

不管那一种情况，由while所控制的语句缩进去一个制表停的位置，使得循环中那些语句一目了然。这种缩进去的办法强调了程序的逻辑结构。尽管C在语句的位置方面是十分自由的，适当的缩进去和空白的使用使程序易读是重要的。我们建议每行只写一个语句，一般在运算符两端都留有空白。花括号的位置不太重要，我们选择了普遍用的方法之一。选择适合你用的一种方法，然后始终用它。

多数工作是在循环体中干的。计算摄氏温度，用语句

```
clesius = ( 5.0/9.0 ) * ( fahr-32.0 );
```

赋给clesius。用5.0／9.0而不用看起来比较简单的5／9的原因是，在C中与许多语言一样，整数除去要截断，任何小数部分都被舍去。因此，5／9是0，所有温度都变成0了。在常数中的小数点指出它是浮点数，所以5.0／9.0是0.555…，这正是我们所希望的。

即使fahr是浮点数，32会在作减法之前，自动地转换成浮点数（32.0），我们还是写32.0而不写32。作为一种风格，即使浮点常数具有整值，最好也写上明显的小数点，它向程序的阅读者强调了他们的浮点性质，并且确保编译程序按照你的方式来工作。

整数转换成浮点数的详细规则在第二章中详述。现在应注意，赋值语句

```
fahr = lower;
```

和测试

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

都按希望的那样工作——在做运算之前整数都转换成浮点数。

这个例子还指出了一些printf是怎样工作的printf实际上是一个通用的格式转换函数，在第七章中我们将完整地描述它。它的第一个参数是要印出的字符串，每用一个%号都指示其它（第二个、第三个，…）某个参数要被替换，并且指出用什么格式印出它。例如，在语句

```
printf ( "%4.0f%6.1f\n", fahr, celsius );
```

中，转换描述%4.0f指出一个浮点

数要用至少四个字符宽度的空间印出它，在小数点后不带数字。%6.1f说明另一个数至少占6个位置，小数点后带一个数字，类似于Fortran中的F6.1或者PL／I中的F（6，1）。描述中的一部分可以省略%6f指出一个数要占至少6个字符宽度；%.2f要求小数点后面有两个位置，但对它的宽度不加限制；%f只说作为浮点数印出一个数。printf还能识别%d为十六进制数，%o为八进制数，%x为十六进制数，%c为字符，%s为字符串，%%为%本身。

在printf第一个参数中的每个%构造与其相应的第二个、第三个参数等配成对。它们必须按数和类型正确地排列，否则将得到无意义的结果。

附带说明，printf不是C语言的一部分；在C本身中并不定义输入或输出。关于printf没有什么不可思议的。它是一个有用的函数，为C程序正常存取的标准子程序库的一部分。为了把精力集中在C本身，在第七章之前我们对I／O不多加涉及。具体地说，我们把格式化的输入推迟到那时候再说。如果你必须输入数，请读第七章7.4节中关于函数scanf的论论。除了scanf是读输入而不是写输出之外，scanf和printf很相似。

1.3 For语句

正如期望的那样，可以有许多不同的方法来写一个程序。奔我们用不同的方法写温度转换程序。

```
main( )     /*Fahrenheit-Celsius table*/
{
    int fahr;
    for(fahr=0; fahr<=300; fahr=fahr+20)
        printf("%4d%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

此程序产生了相同的结果，但看上去确实很不一样。主要的变化在于取消了许多变量；只留下int型的fahr变量（用来指出printf里的%d转换）。在for语句（它本身是一种新的构造）中，上、下限和步长只是作为常数出现，计算摄氏温度的表达式不是单独的赋值语句，而是作为printf的第三个参数出现。

最后的一个改变体现了C语言的一条很普通的规则——凡是允许使用某种类型变量的值的地方，都可以使用该类型的表达式。为了与%6.1f相匹配，printf的第三个参数是一个浮点值，因此在这儿可出现任何浮点表达式。

for本身是个循环，是while的一般化。如果将它与while作一比较，那么可把它的操作弄得更清楚。它分成三部分，彼此用分号分开。第一部分

```
fahr=0
```

只做一次，发生在真正进入循环之前。第二部分是测试或者是控制循环的条件：

```
fahr<=300
```

判断此条件，如为真，执行循环体（这里只有一个printf语句）。然后再进行重新初始化

```
fahr=fahr+20
```

并且再判断条件。当条件为假时，则结束循环。如while那样，循环体可以是单个语句，也可以是括在花括号里的一组语句，初始化和重新初始化部分可以是任何单个表达式。

使用while语句还是for语句可以随意选择，选择的依据也是显而易见的。for通常适用

于初始化和重新初始化都是单个语句，并且在逻辑上相关联的循环，因为它比while紧凑并把循环控制语句集中在一个地方。

1.4 符号常数

在离开温度转换程序之前，我们最后再研究它一下，把300和20那样的“魔术般的数”填在程序里不是一个好办法。它们几乎没有给以后要读此程序的人转达任何信息，也很难用系统的方法改变它们。幸亏C提供了避免写这种数的方法。在程序的开始处，可用 *define 构造把一个符号名或符号常数定义为特定的字符串。以后编译程序对不带引号出现的所有名字用相应的字符串来代替。实际上，替换此名字的可以是任意行文，而不限于数。

```
#define  LOWER   0    /*lowerlimit of table*/
#define  UPPER   300  /*upper limit*/
#define  STEP    20   /*step size*/
{
int fahr;
for (fahr=LOW  R; fahr<=UPPER; fahr=fahr+STEP)
   printf("%4d%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

量LOMER，UPPER和STEP都是常数，所以不需要在说明中出现。符号名通常用大写书写，因此读的时候与小写的变量名能很好地区分开。注意在定义的末尾不带分号。因为在被定义的名字后面整个行都被代替，所以在for语句中会出现太多分号。

1.5 一组有用的程序

现在我们要研究一组对字符数据进行简单操作的程序，可以发现许多程序正是这里讨论的基本模式的扩充。

字符输入和输出

标准库中提供了一次读，写一个字符的函数。每调用一次，getchar( )就取下一个输入字符，返回该字符作为函数的值。也就是说，在

```
c=getshar( )
```

之后，变量c中包含了输入的下一个字符。字符通常来自终端，但等到第七章我们才来讨论这个要求。

函数putchar( )是getchar的补充：

```
putchar(c)
```

把变量c的内容在某输出介质（通常也是终端）上印出。对putchar和printf调用可以交织在一起，输出将按照调用的顺序表现出来。

和printf一样，getchar和putchar没有什么特殊的规定。它们都不是C语言的一部分，但可以普遍适用。

文件复制

有了getchar和putchar后，无须对I/O有更多的了解，就可以写出大量有用的代码。最简单的例子是把输入一次一个字符地复制到输出的程序。概括为：

```
取一个字符
while（字符不是文件结束标志）
    输出刚读入的字符
```

取一个新字符

把它转换成c程序就是：

```
main( )   *copy input to output; 1st version*/
{
    int c;
    c=getchar( );
    while(c! =EOF) {
        putchar(c);
        c=getchar( );
    }
}
```

关系运算符！=表示“不等于”。

主要的问题是检测输入的末端。按规定，getchar在遇到输入末端时，它返回的不是合法字符的值，用这种方法，程序可检测出输入已完了，唯一复杂的地方是，通常关于文件的末端值有两种约定。我们用符号名EOF表示这种值（不管它是什么值）的办法把问题搁置起来。实际上，EOF可能是－1或0，为了能正常的工作，在程序前面必须有

```
*define EOF -1
```

或

```
*defineEOF 0
```

两者中适当的一个。通过使用符号常数EOF表示getchar遇到文件结束时返回的值，保证程序中只有一件事依赖于专门的数字值。

我们把c说成是int，而不是char，所以它可以保存getchar返回的值。如同将要在第二章中看到的，此值实际上是int，因为它除了能表示所有可能的字符外还必须能表示EOF。

有经验的c程序员还可把这个复制程序写得更紧凑些。在c中，诸如

```
c=getchar(    )
```

这样的赋值可以在表达式中使用，它的值仅仅是要赋给等号左边的值。如果把对c赋一个字符的赋值放在while的测试部分里，文件复制程序可写成：

```
main( )   /*copy input to output; 2nd version*/
{
  int c;
  while( (c=geichar( ) )! =EOF)
      putchar(c);
}
```

该程序取一个字符，赋给c，然后测试此字符是否是文件结束标志。如果不是，执行while中的体，打印此字符。再重复while。当最后达到输入结束时，while结束，main也结束了。

这种形式的程序把输入集中在一起——现在只有对getchar的一个调用——因而把程序紧缩了。在测试部分嵌入赋值是c中允许有价值的简洁的情况之一（这可能会失去控制，建立不可理解的代码，虽然这是我们试图制止的倾向）。

必须承认放条件中括住赋值的括号确实十分必要。！ = 的优先数比 = 的优先数高，这意味着，如果没有括号的话，关系测试！将在赋值！ = 之前完成。所以语句

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

等价于

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

因此将产生不希望的结果，即按照getchar调用遇到的是否是文件结束，而把c置成0或1（在这方面更多的情况见第二章）。

字符计数

下一个程序对字符计数，这是一个精制的小品main ( ) /* count characters in input */

```
{
    long nc;
    nc = 0;
    while (getchar ( ) ! = EOF)
        ++nc;
}
```

语句

```
++nc;
```

指出一个新的运算符 + +，含义为增加 1。你可以将它写成nc = nc + 1，但 + +nc更紧凑，通常效率也较高。对应的运算符 - - 表示减少 1。运算符 + +和 - - 可以是前辍运算符（ + +nc）也可以是后辍运算符（nc + +），在第二章中将指出，这两种形式在表达式中有不同的值，但 + +nc和nc + +都要增加，当前我们考虑前辍形式。

字符计数程序把它的计数累积在一个long变量而不是在int中。在PDP-11上，int 的最大值是32767，如果说明成int，比较小的输入就会使计数器溢出，在Honeywell和IBM的C中，long和int是同义的并且比较大。转换说明 %ld 通知 printf 相应的参数是 long 整数。

为了处理十分大的数，可以用double（双倍长float）。为了说明写循环的不同方法，我们用for语句而不用while语句。

```
main ( ) /* count characters in input */
{
    double nc;
    for (nc = 0; getchar ( ) ! = EOF; ++nc)
        ;
    printf ("%.0f\n", nc);
    }
```

printf用%f既代表float，也代表double，%.0f抑制不存在的小数部分的打印。

这里的for循环体是空的，因为所有工作已经在测试部分和重新初始化部分做了。但是C的语法规则要求for语句有一个体。这里孤伶伶的一个分号（技术上叫空语句）就是为了满足此要求。我们把它放在单独一行的目的是为了看得比较清楚。

在离开字符计算程序前，观测一下如果没有字符输入的话，while或for的测试在最开

始调用getchar时就失败，所以程序产生0，得到正确的结果。这是很重要的事情。while和for的特点是在进入循环体之前的最开始处进行测试。如果没有事做，什么也不做，甚至意味着不通过循环体。当处理象“没有字符”那样的输入，程序必须有理解力地办事。while和for语句有助于合理处理带有边界条件的事情。

行计数

下一个程序对它输入中的行进行计数。假定输入行是由换行字符／n终止的，此字符已经谨慎地附在写出的每一行中。

```
main ( )     / * count lines in Input * /
{
  int c, nl;
  nl = 0;
  while ( ( c = getchar ( ) ) ! = EOF )
        if ( c = = '\n' )
          + + nl;
    printf ( "%d\n", nl ) ;
}
```

while的体现在由if组成，它依次控制增量 + + nl。if语句测试括号里的条件，如果是真，执行跟在后面的语句（或用花括号括起的一组语句）。我们已经将它写得缩进去一点以便指出什么受什么控制。

双等号 = = 是c中“等于”的表示法（类似Fortran中的.EQ.）。用此符号将相等测试与赋值语句用的单个等号相区别。因为在典型的C程序中赋值的出现差不多是的字符集中的数字值。它被称之为字符相等测试的两倍，所以赋值运算符是它的一半长是合适的。

任何单个字符可以写在单引号之间，产生的值等于该字符在机器常数。例如，'A'是字符常数，在ASCII字符集中它的位为65，是字符A的内部表示。当然用'A'比用65好，因为它的含义明显并且独立于特定的字符集。

在字符串中使用的转义序列在字符常数里同样也是合法的，所以在测试和算术表达式中，'\n'代表换行字符的值。必须注意'\n'是单个字符，在表达式中等价于单个整数，另一方面"\n"是一个字符串，它碰巧只包含一个字符。字符串与字符的比较将在第二章中进一步讨论。

字计数

我们一组有用程序中的第四个是对行，字和字符进行计数，按照松散的定义，字是不包含空白，制表或换行的字符序列。（这是UNIX实用程序WC的简单结构形式）

```
 *define    YES 1
 *define    NO 0
main ( ) / * count lines, words, chars in input * /
{
    int c, nl, nw, nc, inword;
    inword = NO;
    nl = nw = nc = 0;
    while ( ( c = getchar ( ) ) ! = EOF ) {
```

```
        ++nc;
        if(c=='\n')
            ++nl;
        if(c==''||c=='\n'||c=='\t')
            inword=NO;
        else if(inword==NO){
            inword=YES;
            ++nw;
        }
    }
    printf("%d%d%d\n", nl, nw, nc);
}
```

每当程序遇到字的第一个字符，便对它计数。变量inword记录程序究竟当前在一个字中还是不在一个字中，一开始“不在一个字中”赋予值NO。为了使程序易读，我们喜欢用符号常数YES和NO，而不用文字值1和0。当然在这样的小程序中差别不大，但在较大的程序中最初用这种方法来写虽然稍麻烦，但却增加了清晰性，因而是值得的。还可发现，凡数字只作为符号常数出现的程序，容易作广泛的修改。

行

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

把三个变量置成0。这不是特殊的情况，它的后果是一次赋值得一个值，赋值从右向左进行。

实际上，它好象：

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

运算符||意思是或（OR），所以行

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

的含义是“如果c是空白或者c是制表字符…”（转义序列\t是制表符的可见表示。）对“与”运算（AND）相应有&&运算符。由&&或||联结在一起的表值或求值是从左到右进行的，并且只要知道了它的真假值，就停止求值过程。因此，若C包括了空白，就没有必要去检查其是否包含换行或制表字符，所以这些检查就不做了。在这里，关于这一点并不特别重要，但在比较复杂的情况，这是很有意义的，我们很快就会看到这一点。

例子中还用了C中的else语句，它指出如果if语中条件部分为假时做的另一动作。一般形式是

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

在if-else中的两个语句做一个并且只做一个。若表达式是真，执行语句1；否则，执行语句2。事实上，每个语句还可以是很复杂的。在字计数程序中else后面的那个语句就是if语句，它控制花括号中两个语句的执行。

1.6 数组

让我们写一个程序计算每个数字出现的次数，空的字符（空白，制表和换行）出现次数

和其它所有字符出现的次数。当然这是人为的程序，允许我们能在一个程序中阐述C的若干内容。

有十二类输入，所以使用数组来保存每个数字出现的次数比用十个单独的变量要方便得多。这里是程序的一种形式：

```
main ( )   / * count digits, white space, others * /
{
    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 ( > = ,0' & &c< = '9' )
            + +ndigit[ c-'0' ];
        else if ( c = = ' ' | |c = = '\n' | |c = = '\t' )
            + +nwhiet;
        else
            + +nother;
    printf ( "digits = " );
    for ( i = 0; i<10;  + +i )
        printf ( "%d", ndigit[ i ] );
    printf ( "\nwhite space = %d, other = %d\n",
          nwhite, nother );
```

说明：

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

说明ndigit是10个整数的数组。在C中，数组的下标总是从0开始（而不象Fortran或PL／I中从1开始），所以它的元素是ndigit[0]，ndigit[1]，…，ndight[9]。这反映在对数组初始化和打印数组的for循环中。

下标可以是任何整型表达式，其中当然包括象i那样的整型变量和整常数。

此特定程序紧紧地依赖于数字的字符表示的特性。例如：测试

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

确定c中的字符是否是数字。如果是数字，此数字的数值是

```
    c—'o'
```

这只有在下述情况下才是正确的：'0'，'1'，… 等都是正确的且按升序排列，在'0'和'9'之间除了数字外不出现其它字符。幸亏对所有通常的字符集，这种情况总是成立的。

按照定义，包含char和int的算术运算均在进行运算之前将它们转换成int，所以在算术运算的过程中char变量和常数实质上等价于int。这是十分自然和方便的。例如，c—'0'是个整表达式，按c中存贮的字符是'0'到'9'，而对应地取0到9之间的值，因此是数组ndigit的合法下标。