第3章

最简单的C程序设计——顺序程序设计

有了前两章的基础，现在可以编写简单的C程序了。要编写出一个正确的C语言程序，需要两方面的知识：一是根据问题的要求，设计出解题的具体步骤，这一步骤称为设计算法。二是用C语言写出程序，以便计算机能正确地执行。这两方面的知识是缺一不可的。也就是说，既要懂得算法，能设计算法，同时需要掌握C语言的知识，能够灵活地运用它们写出可供计算机执行的C程序。

本章介绍有关算法的知识，同时也介绍最简单、最基本的C语句，并把这二者紧密结合起来，引导读者编写最简单的C语言程序，为以后逐步深入的学习、编写较复杂的C程序打下初步的基础。

3.1 算法是程序的灵魂

3.1.1 什么是算法

在前面两章中，读者已经清楚地看到：计算机所进行的一切操作都是由程序决定的，程序是由人们事先编写好并输入给计算机的。从前面的程序中可知，一个程序包括以下两个方面的内容：

（1）**对数据的描述**。在程序中要指定数据的类型和数据的组织形式，即**数据结构**(data structure)。

（2）**对操作的描述**。即操作步骤，也就是**算法**(algorithm)。

数据是操作的对象，操作的目的是对数据进行加工处理，以得到期望的结果。打个比方，厨师制作菜肴，需要有菜谱，菜谱上一般应包括：①配料，指出应使用哪些原料；②操作步骤，指出如何使用这些原料，按规定的步骤加工成所需的菜肴，没有原料是无法加工成所需菜肴的。面对同一些原料可以加工出不同风味的菜肴。作为程序设计人员，必须认真考虑和设计数据结构和操作步骤(即算法)。著名计算机科学家沃思(Nikiklaus Wirth)提出一个公式：

数据结构 + 算法 = 程序

这是对面向过程程序的概括。实际上,一个程序除了以上两个主要要素之外,还应当采用适当的程序设计方法(例如对结构化程序设计方法)进行程序设计,并且用某一种计算机语言表示。因此,算法、数据结构、程序设计方法和语言工具 4 个方面是一个程序设计人员所应具备的知识。在设计一个程序时要综合运用这几方面的知识。在这 4 个方面中,**算法是灵魂,数据结构是加工对象,语言是工具,编程需要采用合适的方法**。

算法是解决**"做什么"**和**"怎么做"**的问题。程序中的操作语句,实际上就是算法的体现。显然,不了解算法就谈不上程序设计。本书不是一本专门介绍算法的教材,也不是一本只介绍 C 语言语法规则的使用说明。本书的目的是使读者通过学习,能够知道怎样编写一个 C 程序,并且能够编写出不太复杂的 C 程序。通过一些实例把以上 4 个方面的知识结合起来,介绍如何编写一个 C 程序。

首先通俗地说明什么是算法。做任何事情都有一定的内容和步骤。例如,你想从北京去天津开会,首先要去买火车票,然后按时乘坐地铁到北京站,登上火车,到天津站后坐汽车到会场,参加会议;你要买电视机,先要选好货物,然后开票、付款、拿发票、取货,打车回家;要考大学,首先要填报名单,交报名费,拿到准考证,按时参加考试,得到录取通知书,到指定学校报到注册等。这些步骤都是按一定的顺序进行的,缺一不可,次序错了也不行。我们从事各种工作和活动,都必须事先想好进行的步骤,然后按部就班地进行,才能避免产生错乱。实际上,在日常生活中,由于已养成习惯,所以人们并不意识到每件事都需要事先设计出"行动步骤"。例如吃饭、上学、打球、做作业等,事实上都是按照一定的规律进行的,只是人们不必每次都重复考虑它而已。

不要认为只有"计算"的问题才有算法。广义地说,为解决一个问题而采取的方法和步骤,都称为"算法"。例如,描述太极拳动作的图解,就是"太极拳的算法"。一首歌曲的乐谱,也可以称为该歌曲的算法,因为它指定了演奏该歌曲的每一个步骤,按照它的规定就能演奏出预定的曲子。

对同一个问题,可以有不同的解题方法和步骤。例如,求 1+2+3+…+100,即 $\sum_{n=1}^{100} n$。有人可能先进行 1+2,再加 3,再加 4,一直加到 100,而有的人采取这样的方法:100+(1+99)+(2+98)+…+(49+51)+50=100+49×100+50=5050。还可以有其他的方法。当然,方法有优劣之分。有的方法只需进行很少的步骤,而有些方法则需要较多的步骤。一般来说,希望采用方法简单、运算步骤少的方法。因此,为了有效地进行解题,不仅需要保证算法正确,还要考虑算法的质量,选择合适的算法。

本书所关心的当然只限于计算机算法,即计算机能执行的算法。例如,让计算机算 1×2×3×4×5,或将 100 个学生的成绩按高低分数的次序排列,是可以做到的,而让计算机去执行"替我理发"或"做一碗红烧肉",是做不到的(至少目前如此)。

计算机算法可分为两大类别:**数值运算算法**和**非数值运算算法**。数值运算的目的是求数值解,例如求圆面积、求方程的根、求一个函数的定积分、判断某年是否闰年等,都属于数值运算范围。非数值运算包括的面十分广泛,最常见的是用于事务管理领域,例如图书检索、学生成绩管理、商品销售管理、对一个单位的成员按工资排序等。目前,计算机在非数值运算方面的应用远远超过了在数值运算方面的应用。由于数值运算有现成的模

型，可以运用数值分析方法，因此对数值运算的算法的研究比较深入，算法比较成熟。对各种数值运算都有比较成熟的算法可供选用。人们常常把这些算法汇编成册（写成程序形式），或者将这些程序存放在磁盘上，供用户调用。例如有的计算机系统提供“数学程序库”，使用起来十分方便。而非数值运算的种类繁多，要求各异，难以规范化，因此只对一些典型的非数值运算算法（例如排序算法）作比较深入的研究。其他的非数值运算问题，往往需要使用者参考已有的类似算法，重新设计解决特定问题的专门算法。本书不可能罗列所有算法，只是想通过一些典型算法的介绍，帮助读者了解如何设计一个算法，并引导读者举一反三。

在写程序之前，必须想清楚“做什么”和“怎么做”。“做什么”往往是从题目或任务中可以看出来或整理出来的（例如：求三角形的面积、求一元二次方程等），而“怎么做”则要由程序设计者去思考和设计的。“怎么做”包括两方面的内容：一是要做哪些事情才能达到解决问题的目的；二是决定做这些事情的先后次序。这就是“算法”所要解决的问题。

3.1.2　怎样表示算法

想好一个算法后，应该采用适当的方式表示出来，以便根据它编写程序。

1. 用自然语言表示算法

自然语言就是人们日常使用的语言，可以是汉语、英语或其他语言。用自然语言表示通俗易懂，但文字冗长，容易出现歧义性。自然语言表示的含义往往不大严格，要根据上下文才能判断其正确含义。假如有这样一句话：“张先生对李先生说他的孩子考上了大学”。请问是张先生的孩子考上大学还是李先生的孩子考上大学呢？光从这句话本身难以判断。此外，用自然语言来描述包含条件判断和循环的算法，不很方便。因此，除了那些很简单的问题以外，一般不用自然语言描述算法。

2. 用流程图表示算法

流程图是用一些图框来表示各种操作。美国国家标准化协会 ANSI（American National Standard Institute）规定了一些常用的流程图符号（见图 3.1），已为世界各国程序工作者普遍采用。

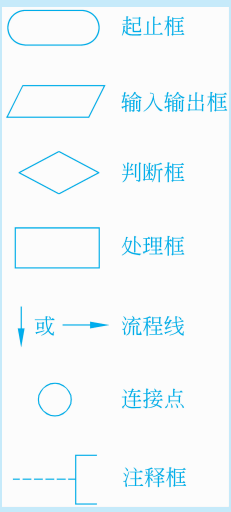

图　3.1

如判断一个数是否偶数的算法，用流程图表示如图 3.2。图中的菱形框用来判断“m 能否被 2 整除”，如果能，就输出该数“是偶数”，否则，就输出该数“不是偶数”。

输出 1 到 10 的算法，用流程图表示如图 3.3。

先把 1 赋给变量 n，即置 n 的初值为 1，然后判别 n 的值是否小于或等于 10，今 n 的值小于 10，故应执行“输出 n 的值”，输出数值 1。然后执行 n＝n＋1，即将 n 的值加 1 后再赋给 n，故 n 变成 2 了，再返回去判断 n 是否小于或等于 10，今 n 的值小于 10，再输出 n 的值（今为 2），再使 n 变成 3，再判断 n 是否小于或等于 10……如此反复循环，直到第 10 次，输出完 n 的当前值 10 后，n 变成 11

了,再判断时,n 已大于 10 了,所以不再执此“输出 n 的值”和“使 n 增值 1”的操作,算法结束。

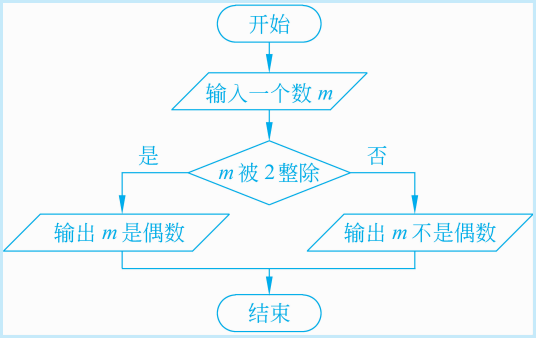

图 3.2

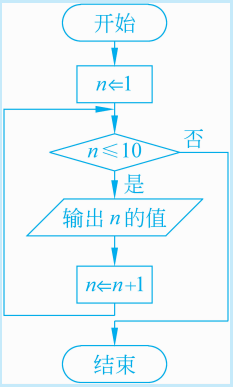

图 3.3

用这种流程图用图形表示算法,直观形象,易于理解。但画图比较麻烦,需要占用较大的纸面面积,而且修改比较困难,现在使用不多了。

3. 用 N-S 流程图表示算法

1973 年,美国学者 I. Nassi 和 B. Shneiderman 提出了一种新的流程图形式。在这种流程图中,完全去掉了带箭头的流程线。全部算法写在一个矩形框内,在该框内还可以包含其他的从属于它的框,或者说,由一些基本的框组成一个大的框。这种流程图又称 **N-S 结构化流程图**(N 和 S 是两位美国学者的英文姓氏的首字母)。这种流程图适于结构化程序设计,而且作图简单,占面积小,一目了然,因而很受欢迎。

N-S 流程图用以下的流程图符号。

(1) 顺序结构。顺序结构用图 3.4 形式表示。表示执行完 A 操作后,接着执行 B 操作。

(2) 选择结构。选择结构用图 3.5 表示。当 p 条件成立时执行 A 操作,p 不成立则执行 B 操作。

图 3.2 可以改用 N-S 流程图表示,如图 3.6 所示。

图 3.4

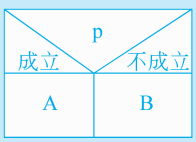

图 3.5

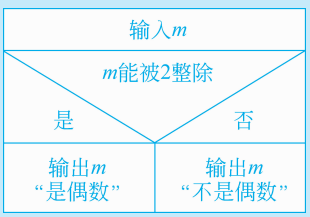

图 3.6

(3) 循环结构。循环结构可用图 3.7 形式表示。图 3.7 表示当 p_1 条件成立时反复执行 A 操作,直到 p_1 条件不成立为止。

输出 1 到 100 的算法，用 N-S 流程图表示如图 3.8 所示。它的流程与图 3.3 相同。

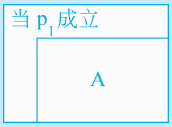

图　3.7

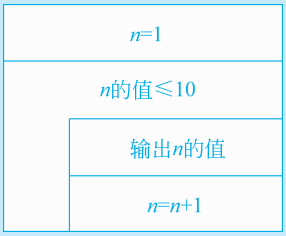

图　3.8

在本章和后续各章中，读者将会看到在程序设计中怎样使用流程图。

4. 用伪代码表示算法

用传统的流程图和 N-S 图表示算法直观易懂，但画起来比较费事，在设计一个算法时，可能要反复修改，而修改流程图是比较麻烦的。因此，流程图适宜于表示一个算法，但在设计算法过程中使用不是很理想（尤其是当算法比较复杂、需要反复修改时）。为了设计算法时方便，常用一种称为**伪代码**（pseudo code）的工具。

伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。它如同一篇文章一样，自上而下地写下来。每一行（或几行）表示一个基本操作。它不用图形符号，因此书写方便，格式紧凑，也比较好懂，也便于向计算机语言算法（即程序）过渡。

例如，“输出 x 的绝对值”的算法可以用伪代码表示如下：

```
if x is positive then
    print x
else
    print-x
```

它好像一个英语句子一样好懂，在西方国家用得比较普遍。

也可以用汉字伪代码。例如：

```
若 x 为正
    输出 x
否则
    输出-x
```

也可以中英文混用，例如：

```
if x 为正
    print x
else
    print-x
```

将计算机语言中的关键字用英文表示,其他的可用汉字。总之,以便于书写和阅读为原则。用伪代码写算法并无固定的、严格的语法规则,只要把意思表达清楚,并且书写的格式要写成清晰易读的形式。

在以上几种表示算法的方法中,具有熟练编程经验的专业人士喜欢用伪代码,初学者喜欢用流程图或 N-S 图,比较形象,易于理解。本书主要使用 N-S 图表示算法。

3.2 程序的三种基本结构

在上一节介绍 N-S 流程图时已提到了程序中用到的三种结构:顺序结构、判断结构和循环结构。在本节中再作进一步介绍。

一个程序包含一系列的执行语句,每一个语句使计算机完成一种操作。在写程序时,要仔细考虑各语句的排列顺序,程序中语句的顺序不是任意书写而无规律的。假如一个程序的流程如同图 3.9 那样无规律地跳转,虽然它也能执行并得到正确的结果,但是在阅读这样的程序时,很难清晰地理解其算法。这样的程序是难阅读、难修改、难维护的。写程序应当遵循一定的规律,尽量避免不必要的跳转,最好是使各语句按照从上到下的顺序排列,在执行时也是按从上到下的顺序执行。1966 年,Bohra 和 Jacopini 提出了以下 3 种基本结构,如果用这 3 种基本结构作为算法的基本单元来编写程序,就能实现上面的目的。

(1) **顺序结构**。各操作步骤是顺序执行的,如图 3.10 所示,虚线框内是一个顺序结构。其中 A 和 B 两个框是顺序执行的。即在执行完 A 框所指定的操作后,必然接着执行 B 框所指定的操作。顺序结构是最简单的一种基本结构。

(2) **选择结构**。选择结构又称**判断结构**或**分支结构**,根据是否满足给定的条件而从两组操作中选择一种操作。如图 3.11 所示。虚线框内是一个选择结构。此结构中必包含一个判断条件 p(以菱形框表示),根据给定的条件 p 是否成立而选择执行 A 组操作框或 B 组操作。p 所代表的条件可以是“x<0”或“x>y”,“a+b <c+d”等,详见第 4 章。

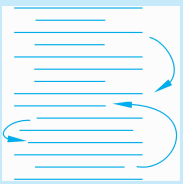

图 3.9

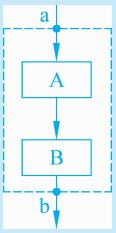

图 3.10

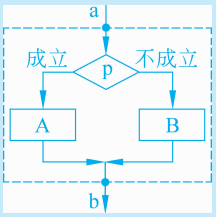

图 3.11

第 1 章例 1.3 中的 if 语句

```
if  (x>y) z=x;                    //如果满足 x>y 条件,执行 z=x
else z=y;                         //如果不满足 x>y 条件,执行 z=y
```

就是一个选择结构。

注意：无论 p 条件是否成立，只能执行 A 操作或 B 操作之一，不可能既执行 A 操作又执行 B 操作。无论走哪一条路径，在执行完 A 或 B 之后，就结束了。A 或 B 两个操作中可以有一个是空操作，即不执行任何操作，如图 3.12 所示。

(3) **循环结构**。它又称重复结构，即在一定条件下反复执行某一部分的操作。图 3.13 所示的就是一种循环结构。执行过程是：当给定的条件 p 成立时，执行 A 操作，执行完 A 后，再判断条件 p 是否成立，如果仍然成立，再执行 A，如此反复执行 A，直到某一次 p 条件不成立为止，此时不执行 A，而脱离循环结构。

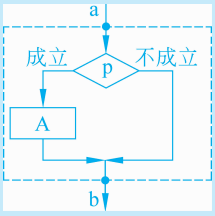

图　3.12

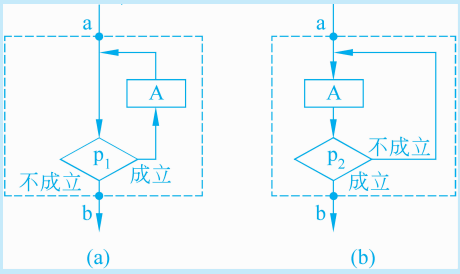

图　3.13

一个良好的程序，无论多么复杂，都可以由这三种基本结构组成。用这三种基本结构构造算法和编写程序，就如同用一些预构件盖房子一样方便，程序结构清晰。有人形容这三种基本结构像项链中的珍珠一样排列整齐、清晰可见。用这三种基本结构构成的程序称为**“结构化程序”**。

C 语言提供了实现三种基本结构的语句，如用 if 语句可以实现选择结构，用循环语句(for 语句，while 语句) 可以实现循环结构。凡是能提供实现三种基本结构的语句的语言，称为结构化语言。显然，C 语言属于结构化语言。

本章只介绍能实现顺序结构的语句，它们只执行最简单的操作。

3.3　C 语句综述

和其他高级语言一样，C 语言的语句用来向计算机系统发出操作指令。一个语句经编译后产生若干条机器指令。一个实际的程序应当包含若干语句。从第 1 章已知，一个程序是一个或多个函数组成的，在一个函数的函数体中一般包括两个部分：**声明部分**和**执行部分**(有的简单的程序可以不包含声明部分，而只有执行语句，如第 1 章中的例 1.1)。执行部分是由若干个语句组成的。C 语句都是用来完成一定操作任务的。声明部分的内容不称为语句。如“int a;”不是一条 C 语句，它不产生机器操作，而只是对变量的定义。

C 程序结构可以用图 3.14 表示。即一个 C 程序可以由若干个源程序文件(分别进行编译的文件模块)组成，一个源文件可以由若干个函数和预处理指令以及全局变量声明部分组成(关于“全局变量”见第 7 章)。一个函数一般都是由数据声明部分和执行语句组成。

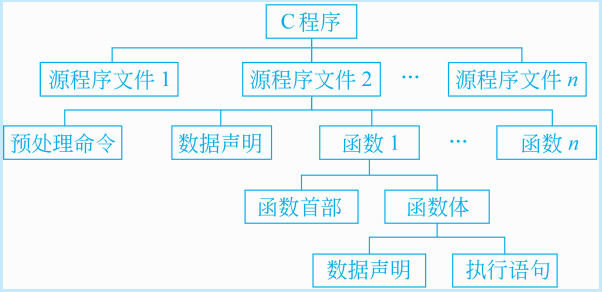

图 3.14

C 语句分为以下 5 类。

(1) **控制语句**。控制语句用于完成一定的控制功能。C 只有 9 种控制语句,它们是:

① if()…else… (条件语句,用来实现选择结构)
② switch (多分支选择语句)
③ for()… (循环语句,用来实现循环结构)
④ while()… (循环语句,用来实现循环结构)
⑤ do…while() (循环语句,用来实现循环结构)
⑥ continue (结束本次循环语句)
⑦ break (中止执行 switch 或循环语句)
⑧ return (从函数返回语句)
⑨ goto (转向语句,现已基本不用了)

上面 9 种语句表示形式中的括号“()”表示括号中是一个“判别条件”,“…”表示内嵌的语句。例如:“if()…else…”的具体语句可以写成:

```
if(x>y) z=x;else z=y;
```

其中 x>y 是一个“判别条件”,z=x;和 z=y;是语句,这两个语句是内嵌在 if…else 语句中的。这个 if…else 语句的作用是:先判别条件“x>y”是否成立,如果 x>y 成立,就执行内嵌语句“z=x;”;否则就执行内嵌语句“z=y;”。

(2) **函数调用语句**。函数调用语句由一个函数调用加一个分号构成,例如:

```
printf("This is a C statement.");
```

printf 是一个函数,上面的语句是调用 printf 函数,后面加一个分号。此外没有其他的内容。

(3) **表达式语句**。表达式语句由一个表达式加一个分号构成,最典型的是,由赋值表达式构成一个赋值语句。例如:

```
a=3
```

是一个赋值表达式,而

```
a=3;
```

是一个赋值语句。可以看到一个表达式的最后加一个分号就成了一个语句。一个语句必须在最后出现分号，分号是语句中不可缺少的组成部分，而不是两个语句间的分隔符号。例如：

```
i=i+1                                    (是表达式,不是语句)
i=i+1;                                   (是语句)
```

任何表达式都可以加上分号而成为语句，例如：

```
i++;
```

是一个语句，作用是使 i 的值加 1。又例如：

```
x+y;
```

也是一个语句，作用是完成 x＋y 的操作，它是合法的，但是并不把 x＋y 的和赋给另一变量，所以它并无实际意义。

表达式能构成语句是 C 语言的一个重要特色。其实“函数调用语句”也是属于表达式语句，因为函数调用(如 sin(x))也属于表达式的一种。只是为了便于理解和使用，才把“函数调用语句”和“表达式语句”分开来说明。由于 C 程序中大多数语句是表达式语句(包括函数调用语句)，所以有人把 C 语言称作“表达式语言”。

(4) **空语句**。下面是一个空语句：

```
;
```

即只有一个分号的语句，它什么也不做。有时用来作流程的转向点(流程从程序其他地方转到此语句处)，也可用来作为循环语句中的循环体(循环体是空语句，表示循环体什么也不做)。

(5) **复合语句**。可以用 {} 把一些语句括起来成为复合语句。例如下面是一个复合语句：

```
{z=x+y;
 t=z/100;
 printf("%f",t);
}
```

注意：复合语句中最后一个语句中最后的分号不能忽略不写。

C 语言允许一行写几个语句，也允许一个语句拆开写在几行上，书写格式无固定

要求。

本章介绍几种顺序执行的语句,在执行这些语句的过程中不会发生流程的控制转移(即不按顺序执行的跳转)。

3.4 赋值表达式和赋值语句

3.4.1 赋值表达式

1. 赋值运算符

赋值符号“=”就是赋值运算符,它的作用是将一个数据赋给一个变量。如“a=3”的作用是执行一次赋值操作(或称赋值运算)。把常量 3 赋给变量 a。也可以将一个表达式的值赋给一个变量。

* 2. 复合的赋值运算符

在赋值符“=”之前加上其他运算符,可以构成复合的运算符。如果在“=”前加一个“+”运算符就成了复合运算符“+=”。例如,可以有:

a+=3　　等价于　a=a+3

x*=y+8　　等价于　x=x*(y+8)

x%=3　　等价于　x=x%3

以“a+=3”为例来说明,它相当于使 a 进行一次自加 3 的操作。即先使 a 加 3,再赋给 a。同样,“x*=y+8”的作用是使 x 乘以(y+8),再赋给 x。

为便于记忆,可以这样理解:

① a+=b　　(其中 a 为变量,b 为表达式)

② a+=b　　(将有下划线的“a+”移到“=”右侧)

③ a=a+b　　(在“=”左侧补上变量名 a)

注意,如果 b 是包含若干项的表达式,则相当于它有括号。例如,以下 3 种写法是等价的:

① x%=y+3

② x%=(y+3)

③ x=x%(y+3)　　(不要错写成 x=x % y+3)

凡是二元(二目)运算符,都可以与赋值符一起组合成复合赋值符。有关算术运算的复合赋值运算符有:

+=,-=,*=,/=,%=

C 语言采用这种复合运算符,一是为了简化程序,使程序精练,二是为了提高编译效率,能产生质量较高的目标代码。专业人员喜欢使用复合运算符,程序显得专业一点。对

初学者来说，不必多用，首要的是保持程序清晰易懂。我们在此作简单的介绍，是为了便于阅读别人编写的程序。

复合的赋值运算符可以不必详细讲授，由学生自己阅读，有一定了解即可。

3. 赋值表达式

由赋值运算符将一个变量和一个表达式连接起来的式子称为“赋值表达式”。它的一般形式为：

```
变量 赋值运算符 表达式
```

如“a＝5”是一个赋值表达式。对赋值表达式求解的过程是：先求赋值运算符右侧的“表达式”的值，然后赋给赋值运算符左侧的变量。一个表达式应该有一个值，例如，赋值表达式“a＝3＊5”的值为 15，执行表达式后，变量 a 的值也是 15。赋值运算符左侧的标识符称为“**左值**”(left value，简写为 lvalue)。意思是位置在赋值运算符的左侧。并不是任何对象都可以作为左值的，变量可以作为左值，而表达式 a＋b 就不能作为左值，常变量也不能作为左值，因为常变量不能被赋值。出现在赋值运算符右侧的表达式称为“**右值**”(right value，简写为 rvalue)。显然左值也可以出现在赋值运算符右侧，因而凡是左值都可以作为右值。例如：

```
b=a;                    //b 是左值
c=b;                    //b 也是右值
```

赋值表达式中的“表达式”，又可以是一个赋值表达式。例如：

```
a=(b=5)
```

括号内的“b＝5”是一个赋值表达式，它的值等于 5。执行表达式“a＝(b＝5)”相当于执行“b＝5”和“a＝b”两个赋值表达式。因此 a 的值等于 5，整个赋值表达式的值也等于 5。从附录 C 可以知道赋值运算符按照“自右而左”的结合顺序，因此，“(b＝5)”外面的括号可以不要，即“a＝(b＝5)”和“a＝b＝5”等价，都是先求“b＝5”的值(得 5)，然后再赋给 a，下面是赋值表达式的例子：

```
a=b=c=5                 (赋值表达式值为 5,赋值后 a、b、c 的值均为 5)
a=5+(c=6)               (表达式值为 11,a 值为 11,c 值为 6)
a=(b=4)+(c=6)           (表达式值为 10,a 值为 10,b 等于 4,c 等于 6)
a=(b=10)/(c=2)          (表达式值为 5,a 等于 5,b 等于 10,c 等于 2)
```

请分析下面的赋值表达式：

```
(a=3*5)=4*3
```

先执行括号内的运算，将 15 赋给 a，然后执行 4＊3 的运算，得 12，再把 12 赋给 a。最

后 a 的值为 12,整个表达式的值为 12。读者可以看到:(a=3*5)出现在赋值运算符的左侧,因此赋值表达式(a=3*5)是左值。请注意,在对赋值表达式(a=3*5)求解后,变量 a 得到值 15,此时赋值表达式(a=3*5)=4*3 相当于(a)=4*3,在执行(a=3*5)=4*3 时,实际上是将 4*3 的积 12 赋给变量 a,而不是赋给 3*5。正因为这样,赋值表达式才能够作为左值。

赋值表达式作为左值时应加括号,如果写成下面这样就出现语法错误:

```
a=3*5=4*3
```

因为 3*5 不是左值,不能出现在赋值运算符的左侧。

将赋值表达式作为表达式的一种,使赋值操作不仅可以出现在赋值语句中,而且可以以表达式形式出现在其他语句(如输出语句、循环语句等)中,例如:

```
printf("%d",a=b);
```

如果 b 的值为 3,则输出 a 的值(也是表达式 a=b 的值)为 3。在一个语句中完成了赋值和输出双重功能。这是 C 语言灵活性的一种表现。

3.4.2 赋值过程中的类型转换

如果赋值运算符两侧的类型一致,则直接进行赋值。如:

```
i=6                                    (假设 i 已定义为 int 型)
```

如果赋值运算符两侧的类型不一致,但都是数值型或字符型时,在赋值时要进行类型转换。类型转换是由系统自动进行的。转换的规则是:

(1) 将实型数据(包括单、双精度)赋给整型变量时,先对实数取整(即舍去小数部分),然后赋予整型变量。如 i 为整型变量,执行“i=3.56”的结果是使 i 的值为 3,以整数形式存储在整型变量中。

(2) 将整型数据赋给单精度或双精度型变量时,数值不变,但以实数形式存储到变量中,如将 23 赋给 float 变量 f,先将 23 转换成 23.00000,再按指数形式存储在 f 中。如果将 23 赋给 double 型变量 d,即执行 d=23,则将 23 补足有效位数字为 23.00000000000000,然后以双精度数形式存储到变量 d 中。

(3) 将一个 double 型数据赋给 float 变量时,截取其前面 7 位有效数字,存放到 float 变量的存储单元(4 个字节)中。但应注意数值范围不能溢出。例如:

```
double d=123.456789e100;
f=d;
```

f 无法容纳如此大的数,出现溢出的错误。

将一个 float 型数据赋给 double 变量时,数值不变,有效位数扩展到 16 位,在内存中以 8 个字节存储。

(4) 字符型数据赋给整型变量时，将字符的 ASCII 码赋给整型变量。如：

```
i='a';                                  //已定义 i 为整型变量
```

由于字符'a'的 ASCII 码为 97，因此，赋值后 i 的值为 97。

(5) 将一个占字节多的整型数据赋给一个占字节少的整型变量或字符变量(例如把一个 4 字节的 long 型数据赋给一个 2 字节的 short 型变量，或将一个 2 字节的 int 型数据赋给 1 字节的 char 型变量)，只将其低字节原封不动地送到该变量(即发生截断)。例如：

```
i=289;                                  //假设已定义 i 为整型变量
c='a';                                  //假设已定义 c 为字符变量
c=i;                                    //假设将一个占 2 字节的 int 型数据赋给 char 型变量
```

赋值情况见图 3.15。c 的值为 33，如果用“%c”输出 c，将得到字符“!”(其 ASCII 码为 33)。

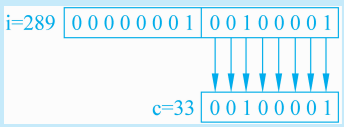

图　3.15

要避免进行这种赋值，因为赋值后数值可能发生失真。如果一定要进行这种赋值，应当保证赋值后数值不会发生变化，即所赋的值在变量的数值范围内。如将上面的 i 值改为 123，就不会发生失真。

如果将一个**有符号整数**赋给**无符号整型变量**，或将一个**无符号整数**赋给**有符号整型变量**，是按照存储单元字节中原样传送的。请参阅 3.9 节提高部分。

3.4.3　赋值语句

赋值语句是由赋值表达式加上一个分号构成。从前面的例子中已知道，赋值表达式的作用是将一个表达式的值赋给一个变量，因此赋值表达式具有计算和赋值双重功能。程序中的计算功能主要是由赋值语句来完成的。在 C 程序中，赋值语句是用得最多的语句。如：

```
s=2*3.14159*r;                          //计算圆周长，赋值给变量 s
```

在前面的叙述中已知：在一个表达式中可以包含另一个表达式。既然赋值表达式是一种表达式，因此它就可以出现在其他表达式之中，例如：

```
if((a=b)>0) t=a;
```

按语法规定 if 后面的括号内是一个“条件”，例如可以是：“if(x>0)…”。现在在 x 的位置上换上一个赋值表达式“a=b”，其作用是：先进行赋值运算(将 b 的值赋给 a)，然后判断 a 是否大于 0，如大于 0，执行 t=a。注意，在 if 语句中的“a=b”不是赋值语句而是赋值表达式，以上的写法是合法的。如果写成：

```
if((a=b;)>0)t=a;                        //"a=b; "是赋值语句
```

就错了。在 if 的条件中可以包含赋值表达式,但不能包含赋值语句。由此可以看到,C 语言把赋值语句和赋值表达式区别开来,增加了表达式的种类,使表达式的应用几乎“无孔不入”,能实现其他语言中难以实现的功能。

注意:要区分赋值表达式和赋值语句。
赋值表达式的末尾没有分号,赋值语句的末尾必须有分号。
在一个表达式中可以包含一个或多个赋值表达式,但决不能包含赋值语句。

3.4.4 变量赋初值

程序中常需要对一些变量预先设置一个初值。设置初值既可以用赋值语句去实现,也可以在定义变量的同时使变量初始化,后者更为方便。例如:

```
int a=3;                          //指定 a 为整型变量,初值为 3
double f=3.56;                    //指定 f 为双精度型变量,初值为 3.56
char c='a';                       //指定 c 为字符变量,初值为'a'
```

也可以对被定义的变量中的一部分变量赋初值。其余的变量不赋初值。例如:

```
int a,b,c=5;
```

指定 a、b、c 为整型变量,但只对 c 初始化,c 的初值为 5,b 和 c 未被赋初值。

如果对几个变量赋予同一个初值,应写成

```
int a=3,b=3,c=3;
```

表示 a、b、c 的初值都是 3。不能写成

```
int a=b=c=3;
```

一般变量的初始化不是在编译阶段完成的(只有静态存储变量和外部变量的初始化是在编译阶段完成的),而是在程序运行时执行本函数时赋予初值的,相当于有一个赋值语句。例如:

```
int a=3;
```

相当于

```
int a;                            //定义 a 为整型变量
a=3;                              //赋值语句,将 3 赋给 a
```

又如:

```
int a,b,c=5;
```

相当于

```
int a,b,c;                    //指定 a、b、c 为整型变量
c=5;                          //将 5 赋给 c
```

3.5　数据输入输出的概念

输入输出是程序中最基本的一种操作，几乎每一个 C 程序都包含输入输出。因为要进行运算，就必须给出数据，而运算的结果当然需要输出，以便人们应用。没有输出的程序是没有意义的。

在讨论程序的输入输出时要注意以下几点。

(1) 所谓**输入输出是以计算机主机为主体而言的**。从计算机向输出设备(如显示器、打印机等)输出数据称为**输出**，从输入设备(如键盘、鼠标、磁盘、光盘、扫描仪等)向计算机输入数据称为**输入**。见图 3.16。

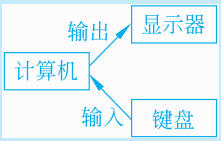

图　3.16

(2) **C 语言本身不提供输入输出语句**，输入和输出操作是由 C 函数库中的函数来实现的。在 C 标准函数库中提供了一些输入输出函数，例如，printf 函数和 scanf 函数。读者在使用它们时，千万不要误认为它们是 C 语言提供的“输入输出语句”。printf 和 scanf 不是 C 语言的关键字，而只是库函数的名字。实际上人们可以不用 printf 和 scanf 这两个名字，而另外编写一个输入函数和一个输出函数，用来实现输入输出的功能，采用其他的名字作为函数名。

C 提供的函数以库的形式存放在 C 的编译系统中，它们不是 C 语言文本中的组成部分。在第 1 章中曾介绍，不把输入输出作为 C 语句的目的是使 C 语言编译系统简单，因为将语句翻译成二进制的指令是在编译阶段完成的，没有输入输出语句就可以避免在编译阶段处理与硬件有关的问题，可以使编译系统简化，而且通用性强，可移植性好，在各种型号的计算机和不同的编译环境下都能适用，便于在各种计算机上实现。各种 C 编译系统提供的系统函数库是各软件公司根据用户的需要编写的，并且已编译成目标文件(.obj 文件)。它们在程序连接阶段与由源程序经编译而得到的目标文件(.obj 文件)相连接，生成一个可执行的目标程序(.exe 文件)。如果在源程序中有 printf 函数，在编译时并不把它翻译成目标指令，而是在连接阶段与系统函数库相连接后，在执行阶段中调用函数库中的 printf 函数。

在不同的编译系统所提供的函数库中，函数的数量、名字和功能是不完全相同的。不过，有些通用的函数(如 printf 和 scanf 等)，各种编译系统都提供，成为各种系统的标准函数。

C 语言函数库中有一批“标准输入输出函数”，它是以标准的输入输出设备(一般为终端设备)为输入输出对象的。其中有 putchar(输出字符)、getchar(输入字符)、printf

(格式输出)、scanf(格式输入)、puts(输出字符串)、gets(输入字符串)。本章主要介绍前面 4 个最基本的输入输出函数。

(3) **在使用系统库函数时,要在程序中使用预编译指令"#include"**。如:#include <stdio.h> 。目的是将有关的"头文件"的内容包括到用户源文件中。在头文件中包含了调用函数时所需的有关信息。例如在使用标准输入输出库函数时,要用到"stdio.h"头文件中提供的信息。stdio 是 standard input & output(输入和输出)的缩写。文件后缀中"h" 是 header 的缩写。#include 指令都是放在程序的开头,因此这类文件被称为**头文件**。"stdio.h"头文件包含了与标准 I/O 库有关的变量定义和宏定义以及对函数的声明。在调用标准输入输出库函数时,文件开头应该有以下预编译指令:

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

或

```
#include "stdio.h"
```

其作用是在程序编译时,系统会将"stdio.h"头文件的内容调出来放在此位置,取代本#include 行。这样在本程序模块中就可以使用这些内容了。

说明:以上两种#include 指令形式的区别是:(1)用尖括号形式(如<stdio.h>)时,编译系统直接找到 C 编译系统所在的子目录(C 库函数头文件一般都是和 C 编译系统存放在同一个子目录中的)中去找所要包含的文件(如 stdio.h),这种方式称为**标准方式**。(2)用双撇号形式(如"stdio.h"),在编译时,系统先在用户当前目录中寻找要包含的文件,若找不到,再按标准方式查找。

一般情况下,用户用#include 指令是为了使用系统库函数,因此要包含系统提供的相应头文件,所以用标准方式为宜,以提高效率。如果用户想包含的头文件不是系统提供的相应头文件,而是用户自己编写的文件(这种文件一般都存放在用户当前目录中),这时应当用双撇号形式,否则会找不到所需的文件。如果文件不在当前目录中,可以在双撇号中写出文件路径(如#include "C:\temp\file1.h"),以便系统能从中找到所需的文件。

应养成这样的习惯:只要在本程序文件中使用标准输入输出库函数时,一律加上#include <stdio.h>指令。

从 3.6 节起,将由浅入深地介绍怎样进行数据的输入和输出。

3.6 字符数据的输入输出

先介绍最简单的输入输出,即只向计算机输入一个字符或从计算机输出一个字符。上节已经介绍,在 C 程序中的输入输出是由函数实现的。C 函数库中提供输入一个字符的函数 putchar 和输出一个字符的函数 getchar。它们是最简单的,也是最容易理解的。在掌握了字符的输入输出后,下节再介绍数值数据的输入输出。

3.6.1　用 putchar 函数输出一个字符

想从计算机向显示器输出一个字符，要调用系统函数库中的 putchar 函数（字符输出函数）。putchar 函数的一般形式为：

```
putchar(c)
```

putchar 是 put character（给字符）的缩写，很容易记忆。C 语言的函数名大多是可以见名知意的，不必死记。putchar(c)的作用是输出字符变量 c 的值，显然它是一个字符。

例 3.1　先后输出几个字符。

编写程序：

可以很容易写出程序，先后调用几次 putchar 函数，就能输出几个字符。

```
#include <stdio.h>
int main ()
  {char a,b,c;                          //定义 3 个字符变量
   a='B';b='O';c='Y';                   //给 3 个字符变量赋值
   putchar(a);                          //向显示器输出字符 B
   putchar(b);                          //向显示器输出字符 O
   putchar(c);                          //向显示器输出字符 Y
   putchar ('\n');                      //向显示器输出一个换行符
   return 0;
  }
```

运行结果：

```
BOY
```

运行时连续输出以下 3 个字符，然后换行。

程序分析：

从此例可以看出，用 putchar 函数可以输出能在显示器屏幕上显示的字符，也可以输出屏幕控制字符，如 putchar('\n')的作用是输出一个换行符，使输出的当前位置移到下一行的开头。如果将最后 4 个语句改为以下一行：

```
putchar(a);putchar('\n');putchar(b); putchar('\n');putchar(c); putchar('\n');
```

则输出结果为：

```
B
O
Y
```

如果把上面的程序改为以下这样，请思考输出结果。

```
#include <stdio.h>
int main ( )
  {int a,b,c;                          //定义 3 个整型变量
   a=66;b=79;c=89;                     //给 3 个整型赋值
   putchar(a);                         //向显示器输出变量 a 的值
   putchar(b);                         //向显示器输出变量 b 的值
   putchar(c);                         //向显示器输出变量 c 的值
   putchar ('\n');                     //向显示器输出一个换行符
   return 0;
  }
```

请读者上机运行此程序,可以看到运行时同样输出:

```
BOY
```

从前面的介绍已知:在程序中整型数据与字符数据是相通的(但应注意,整型数据的值应在字符的 ASCII 代码范围内),而 putchar 函数是输出字符的函数,它只能输出字符而不能输出整数,由于 66 是字符 B 的 ASCII 代码,因此,putchar(66)输出字符 B。其他类似。

结论:putchar(c)中的 c 可以是字符变量或整型变量(其值在字符的 ASCII 代码范围内),当然也可以是字符常量或整型常量,如 putchar('B')或 putchar(66)。

也可以用 putchar 函数输出其他转义字符,例如:

```
putchar('\101')                        (输出字符 A)
putchar('\'')                          (输出单撇号字符')
putchar('\015')                        (八进制数 15 等于十进制数 13,从附录 A 查出 13 是
                                       "回车"的ASCII 代码,因此输回车,不换行,使输出的
                                       当前位置移到本行开头)
```

3.6.2 用 getchar 函数输入一个字符

为了向计算机输入一个字符,要调用系统函数库中的 getchar 函数(字符输入函数)。getchar 函数的一般形式为:

```
getchar(c)
```

getchar 是 get character(取得字符)的缩写,getchar 函数的作用是从计算机终端(一般是显示器的键盘)输入一个字符,即计算机获得一个字符)。getchar 函数没有参数,其一般形式为:

```
getchar()
```

getchar 函数的值就是从输入设备得到的字符。注意：getchar 函数只能接收一个字符。如果想输入多个字符就要用多个 getchar 函数。

例 3.2　用 getchar 函数输入字符。

编写程序：

```
#include <stdio.h>
int main ( )
{ char a,b,c;                  //定义字符变量 a,b,c
  a=getchar();                 //从键盘输入一个字符,送给字符变量 a
  b=getchar();                 //从键盘输入一个字符,送给字符变量 b
  c=getchar();                 //从键盘输入一个字符,送给字符变量 c
  putchar(a);                  //将变量 a 的值输出
  putchar(b);                  //将变量 b 的值输出
  putchar(c);                  //将变量 c 的值输出
  putchar('\n');               //换行
  return 0;
}
```

运行结果：

```
BOY↙                           (连续输入 BOY 后,按 Enter 键,字符才送到内存中的存储单元)
BOY                            (输出变量 a,b,c 的值)
```

说明：在用键盘输入信息时，并不是在键盘上敲一个字符，该字符就立即送到计算机中的。这些字符先暂存在键盘的缓冲器中，只有按了 Enter 键才把这些字符一起输入到计算机中，按先后顺序分别赋给相应的变量。

如果在运行时，在输入一个字符后马上按 Enter 键，会得到什么结果？

运行结果为：

```
B↙                             (输入字符 B 后马上按 Enter 键)
O↙                             (输入字符 O 后马上按 Enter 键)
B                              (输出 B 后换行)
O                              (输出 O 后换行)
```

请思考是什么原因？

注意：第 1 行输入的不是一个字符 B，而是两个字符：B 和换行符，其中字符 B 赋给了变量 a，换行符赋给了变量 b。第 2 行接着输入两个字符：O 和换行符，其中字符 O 赋给了变量 c，换行符没有送入任何变量。在用 putchar 函数输出变量 a，b，c 的值时，就输出了字符 B，然后输出换行，再输出字符 O，然后执行 putchar('\n')，换行。

提示：执行 getchar 函数不仅可以从输入设备获得一个可显示的字符，而且可以获得在屏幕上无法显示的字符，如控制字符。

用 getchar 函数得到的字符可以赋给一个字符变量或整型变量,也可以不赋给任何变量,而作为表达式的一部分,在表达式中利用它的值。例如,例 3.2 可以改写如下:

```
#include <stdio.h>
  int main ( )
{ putchar(getchar());                    //将接收到的字符输出
  putchar(getchar());                    //将接收到的字符输出
  putchar(getchar());                    //将接收到的字符输出
  putchar('\n');
  return 0;
 }
```

因为第 1 个 getchar 函数得到的值为'B',因此 putchar 函数输出'B',第 2 个 getchar 函数得到的值为'O',因此 putchar 函数输出'O',第 3 个情况相同。

也可以在 printf 函数中输出刚接收的字符:

```
printf("%c",getchar());                  //%c 是输出字符的格式声明
```

在执行此语句时,先从键盘输入一个字符,然后用输出格式声明%c 输出该字符。

3.7 简单的格式输入与输出

上一节介绍了最简单的输入输出(只输入输出一个字符),而实际上在程序中往往需要输入输出各种类型的数据(如整型、单精度型、双精度型、字符型等),需要有一种函数,能处理各种类型数据的输入输出。

在 C 程序中,数据的输入输出主要用 printf 和 scanf 函数来实现的。这两个函数是**格式输入输出函数**,在进行输入输出时,程序设计人员必须指定输入输出数据的格式,即根据数据的不同类型指定不同的格式。

C 提供的输入输出格式比较多,也比较烦琐,初学时不易掌握,更不易记住。用的不对就得不到预期的结果,不少编程人员由于掌握不好这方面的知识而浪费了大量调试程序的时间。为了使读者便于掌握,我们分两步来介绍。在本节中,先介绍简单的格式输入输出,这是不难理解的。有了这些基本知识后,就可以顺利地进行后续内容的学习了,也可以进行一般的编程工作了。在本章的提高部分(第 9 节)将进一步介绍格式输入输出的各种规定,以便在进一步编程时有所遵循。

在初学时不必花许多精力去死抠每一个细节,重点掌握最常用的一些规则即可。其他部分可在需要时随时查阅。学习这部分的内容时最好边看书边上机练习,通过编写和调试程序的实践逐步深入而自然地掌握输入输出的应用。

3.7.1 用简单的 printf 函数输出数据

前面各章节中已用到了 printf 函数(格式输出函数),它的作用是向终端(或系统隐含