第3章 最简单的 C 程序设计——顺序程序设计

在第 1 章中介绍了几个简单的 C 程序,在第 2 章中介绍了程序中用到的一些基本要素(常量、变量、运算符、表达式等),它们是构成程序的基本成分。从本章开始将陆续介绍怎样利用 C 语言编写程序。本章介绍最简单的程序结构——顺序程序结构。

3.1 程序的三种基本结构

一个程序包含一系列的执行指令,每一个指令使计算机完成一种操作。程序中的指令不是任意书写而无规律的。1966 年,Bohra 和 Jacopini 提出了以下 3 种基本结构,用这 3 种基本结构作为表示一个良好算法的基本单元。

(1) 顺序结构。各操作步骤是顺序执行的,如图 3-1 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。

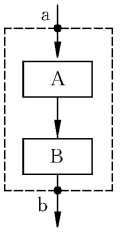

图 3-1

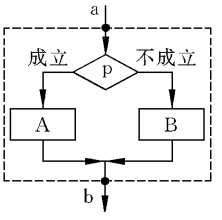

图 3-2

(2) 选择结构。选择结构又称选取结构或分支结构,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 3-2 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。

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 3-3 所示。

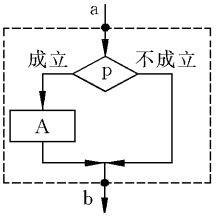

图　3-3

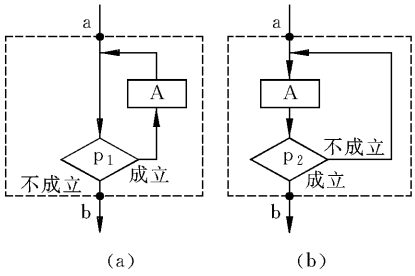

图　3-4

(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 3-4 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。

一个良好的程序，无论多么复杂，都是由这 3 种基本结构组成的。本章只介绍构成顺序结构的语句，它们只执行最简单的操作。

3.2　C 语句概述

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，一个程序是由若干函数组成的，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)。执行部分是由语句组成的。根据 C89 标准，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 3-5 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，一个源程序文件可以由若干个函数和预处理命令以及全局变量声明部分组成(关于“全局变量”见第 7 章)，一个函数体由数据声明部分和执行语句组成。

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 for( )…　　(循环语句，用来实现循环结构)

③ while( )…　　(循环语句，用来实现循环结构)

④ do…while( )　　(循环语句，用来实现循环结构)

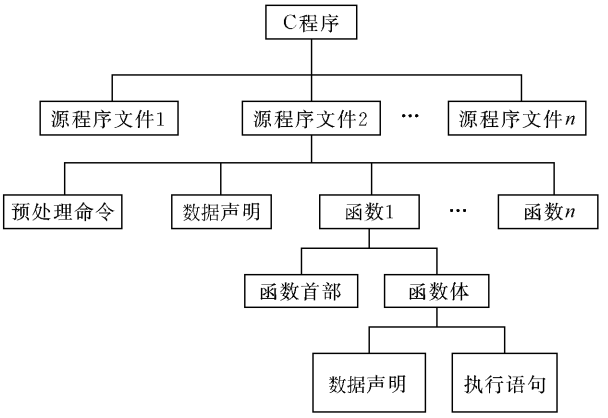

图 3-5

⑤ continue （结束本次循环语句）
⑥ break （中止执行 switch 或循环语句）
⑦ switch （多分支选择语句）
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.");
```

（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 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 语言称做“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。

3.3　赋值表达式和赋值语句

3.3.1　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。

3. 赋值表达式的写法

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,a,b,c 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。

赋值表达式也可以包含复合的赋值运算符。例如:

```
a+=a-=a*a
```

也是一个赋值表达式。如果 a 的初值为 12,此赋值表达式的求解步骤如下:

① 先进行“a-=a*a”的运算,它相当于 a=a-a*a,a 的值为 12-144=-132。

② 再进行“a+=-132”的运算,相当于 a=a+(-132),a 的值为 -132-132=-264。

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。

3.3.2 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) 将浮点型数据(包括单、双精度)赋给整型变量时,先对浮点数取为整(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、双精度变量时,数值不变,但以浮点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,再存储在 f 中。如将 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,然后以双精度浮点数形式存储到变量 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 */
```

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

赋值情况见图 3-6。c 的值为 33, 如果用"%c"输出 c,将得到字符"!"(其 ASCII 码为 33)。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。

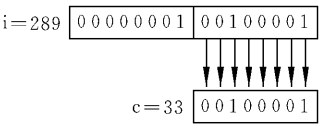

图 3-6

(6) 将有符号整数赋值给长度相同的无符号整型变量时,按字节原样赋值(连原有

的符号位也作为数值一起传送)。

例 3.1　有符号整数赋值给无符号整型变量,数据会失真。

```
#include <stdio.h>
 void main ()
 { unsigned a;                    /* 定义 a 为无符号整型变量 */
   int b;                         /* 定义 b 为有符号整型变量 */
   b = -1;
   a = b;
   printf("%u\n",a);
 }
```

由于 a 是无符号整型变量,因此,不能用%d 输出格式符,而要用输出无符号数的“%u”格式符。在 Turbo C 环境下运行的结果为

```
65535                            (即 2^16 - 1)
```

在 Visual C++ 6.0 环境下运行的结果为

```
4294967295                       (即 2^32 - 1)
```

赋值情况见图 3-7。如果 b 为正值,且在 0 ~ 32767 之间,则赋值后数值不变。

(7) 将无符号整数赋值给长度相同的有符号整型变量时,应注意不要超出有符号整型变量的数值范围,否则会出错。

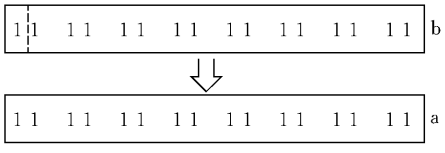

图　3-7

例 3.2　无符号整数赋值给有符号整型变量,注意数值范围。

```
#include <stdio.h>
void main ()
{ unsigned a;
  int b;
  a = 65535;                     /* 两个字节 16 位全为 1 */
  b = a;                         /* b 的数值范围为 -32768 ~ 32767 */
  printf("%d\n",b);
}
```

在执行“b = a”赋值时,将 a 的两个字节(全 1)原样赋给 b,由于 b 的数值范围为 -32768 ~ 32767,显然不能正确反映 65535,对一个有符号的整型数据来说,第 1 个二进位是 1 表示此数是一个负数,16 位全 1 是 -1 的补码。

在 Turbo C 环境下运行,输出结果为 -1。如果在 Visual C++ 环境下运行,将 a = 65535 改为 4294967295,输出结果也是 -1。为什么要做此改动,请读者思考。

以上的赋值规则看起来比较复杂,其实,不同类型的整型数据间的赋值归根结底就是一条:按存储单元中的存储形式直接传送。只要学过补码知识的,对以上规则是不难理解的。如果对补码知识不熟悉,对此可不必深究,只要知道在什么情况下可能出现不想得到的结果,注意避免这种情况出现即可。在学习本书时不必死记,这部分内容可以通过学

生自学和上机实践来掌握。

由于 C 语言使用灵活,在不同类型数据之间赋值时,常常会出现意想不到的结果,而编译系统并不提示出错,全靠程序员的经验来找出问题。这就要求编程人员对出现问题的原因有所了解,以便迅速排除故障。

如果读者想对此进一步了解,可参阅作者所著《C 程序设计(第三版)》第 3 章,在该书中做了较详细的说明。

3.3.3 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 */
```

C 语言的赋值语句具有其他高级语言的赋值语句的一切特点和功能。但也应当注意到它们的不同。

(1) C 语言中的赋值号“=”是一个运算符,在其他大多数语言中赋值号不是运算符。

(2) 要区别赋值表达式和赋值语句。其他多数高级语言没有“赋值表达式”这一概念。赋值表达式可以包括在其他表达式之中,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。在 if 语句中的“a=b”不是赋值语句而是赋值表达式,这样写是合法的。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。

3.3.4 变量赋初值

程序中常需要对一些变量预先设置一个初值。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,后者更方便。例如:

```
int a=3;                    /* 指定 a 为整型变量,初值为 3 */
float f=3.56;               /* 指定 f 为浮点型变量,初值为 3.56 */
char c='a';                 /* 指定 c 为字符变量,初值为'a' */
```

也可以使被定义的变量的一部分赋初值。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。

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;
```

初始化不是在编译阶段完成的(只有在第 7 章中介绍的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;                    /* 指定 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 */
```

3.4　数据输入输出的概念及在 C 语言中的实现

在讨论输入输出时要注意以下几点。

(1) 所谓输入输出是对计算机主机而言的。从计算机向外部输出设备(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、扫描仪等)向计算机输入数据称为输入。

(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,而只是函数的名字。实际上完全可以不用 printf 和 scanf 这两个名字,而另外编两个输入输出函数,用其他函数名。C 提供的函数以库的形式存放在系统中,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