第三章 控制结构--顺序结构

总体要求

- 了解顺序结构概念
- 了解输入输出概念
- 掌握 C 语言标准函数库函数的使用
- 熟练掌握字符输入输出函数的使用规则
- 熟练掌握格式输入输出函数的使用规则

核心技能点

- 具备对格式输入输出函数使用能力

扩展技能点

- 对字符输入输出函数使用技巧
- 对格式输入输出函数的使用技巧

相关知识点

- C 语言顺序结构程序的编写
- VC++编译器用法

学习重点

- 熟练掌握 scanf()函数和 printf()函数的使用

在第一章里，笔者介要介绍了高级程序设计语言的三种控制结构：顺序、选择和循环。这三种基本结构可以反复迭代，构成各种复杂的程序。C 语言提供了多种语句来实现这些程序结构。

在第二章里读者学习了 C 语言最基本的元素：变量和表达式，但这些元素不能单独完成算法所要求的逻辑运算。它们必须被组织在语句当中，才能完成相应的功能。因此，在这一章里，主要介绍 C 语句的概念，以及顺序结构的构成，然后详细讲解最基本的有输入输出语句。

3.1 问题引入

在第二章提出的解决方案中，只是提出了如何表示单个学生完整信息的建议，而没有涉及到信息的采集和输出，还不能满足比赛组织者的最基本要求。如果要完成这两项功能，那么就必须学习 C 语言是如何完成信息的输入和输出的。

从计算机向显示屏、打印机等外部设备传出数据称为**“输出（output）”**，从如键盘、磁盘等等外部设备向计算机传入数据称为**“输入（input）”**。这是两项非常基本的操作，任何一种语言都必须实现它们。一般地，如 BASIC 语言，都有相应的输入输出语句。

而在 C 语言中，情况就有些不一样了。C 语言没有相应的输入输出语句，而完成数据输入输出是由标准库函数来完成的。这些标准输入输出库函数都是声明在一个名为**“stdio.h”**的文件中，称之为**“头文件（header file）”**。在编写的源代码中，可以使用预编译预处理命令**“#include”**将头文件包含进来，其形式为：

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

或

```
#include “stdio.h”
```

头文件 stdio.h 中声明了所有与基本输入输出相关的函数原型，将在后面的内容中详细学习。

3.2 C 语句概述

C 表达式可以完成指定的运算，但它们只是类似于自然语言中的词组或短语，不能表达完整的信息。

而要完整表达信息，必须将它们放在 C 的语句（statement）中。C 语句是程序中能够独立执行的最小单位，算法中的每一个步骤都会被转换成 C 的一条或多条语句来实现。

C 语句的典型标志是分号“;”。任何一条 C 语句都会以;结束。

{漏泄语句后的分号是初学者常犯的错误之一。从 PASCAL、BASIC 转为 C 的程序员尤其要注意这个细节。}

3.2.1 C 语句的分类

根据语句的功能，C 语句可以大致分为以下几类：

1. 空语句

单独的;可以构成一条语句，形如：

```
;
```

这称为**空语句**，因为它什么都不做。空语句往往出现在需要语句占位但又不需要做任何工作的地方。

2. 表达式语句

任何一个合法的 C 表达式跟上一个;就构成一条**表达式语句**。以下语句都是合法的：

```
2;          //常量表达式语句
grade;      //变量表达式语句
score + 3; //算数表达式语句
```

表达式语句完成相应的运算。但需要提到的是，虽然表达式会产生一个结果，但在语句执行完成后，这个结果被忽略了。因此，上面三条语句没有任何意义。与空语句一样，这样的表达式语句主要起占位的作用，而且很少出现在代码中。

然而，不是所有的表达式语句都没有意义。有两类表达式语句在我们的程序中会经常出现，它们就不仅仅是起到占位的作用了，还会产生有用的结果。

(1) 当表达式是一个赋值表达式时，我们称该语句为**赋值语句**。例如：

```
score = 92 + 3;
```

隐含有赋值操作的自增和自减也属于此类语句。例如：

```
++grade;
score--;
```

(2) **函数调用语句**，它的主体是一个函数表达式。例如：

```
printf(“%d”, score);
```

因为函数内部会做一些具体的操作，所以这样的语句也能产生有用的结果。

3. 流程控制语句

上述讲到的语句可以按照一定的逻辑顺序构成顺序执行的语句序列，但当程序的流程遇到选择或需要重复执行的情况，就需要特定的流程控制语句来实现。C 的流程控制语句有三大类：

- 选择语句：有 if 和 switch 两种；
- 循环语句：有 while、do…while、for 三种；
- 无条件转移语句：有 break、continue、return、goto 四种。

这些流程控制语句的功能和用法将在后续的章节中陆续展开。

4. 复合语句（语句块）

在某些特定的场合，一条语句不能完成指定的功能，而需要将多条语句组合在一起。在这种情况下，我们往往将这些语句用一对{}括起来，称为**“复合语句**（语句块，或简称块）”，被视为是一条语句。例如：

```
{
    a = 2;
    b *= a;
    c = b % 3;
}
```

上例的{}中虽然包含了三条语句，但仍属于一条复合语句。

复合语句有一个有趣的也许是非常有用的一个特性。复合语句其实带有分程序的特征，而分程序是允许在其内部声明变量的，因此，可以在复合语句（其实是所有）的左花括号{后面声明变量。当然，这些变量的使用范围仅局限在那个分程序块（复合语句）内部。

5. 标号语句

标号语句是在普通语句的基础上在其前面加上一个语句标号构成的。语句标号有三种：

(1) label1: 语句

(2) case 表达式: 语句

(3) default: 语句

情况（1）中，“label1”就是**语句标号**。语句标号就像一张标签，“贴”在语句前面，将这条语句标示出来，以便成为 goto 语句或其他跳转函数的转移目标。这种语句标号是一个普通标识符，因此遵循所有关于用户自定义标识符的规定。

另外两种语句标号只能出现在 switch 语句中。

3.2.2 非语句的情况

有些 C 语言的语法分量虽然可以带有“;”，但它们不是可以执行的语句。以下提到的这些都不是 C 语言的可执行语句：

- 变量/常量声明
- 函数声明
- 编译预处理指令

变量/常量声明的形式我们已经学过，后二者我们将在后面的章节中学习到。

现在我们通过一个已经学过的程序来看看 C 语句的情况：

```
/* ex1-1.c
   这是一个很简单的 C 程序，
   但是却包含了重要的 C 程序的特征。*/   //以上三行为注释，不是语句

#include <stdio.h>    //包含头文件，属于编译预处理指令，不是语句

int main()    //函数头部声明，不是语句
{
    int a;          //变量声明，不是语句

    a = 3 * 4; //赋值语句
    printf("3 * 4 = %d\n", a);  //函数调用表达式语句

    return 0;       //流程控制语句：转移语句
}
```

这里对变量/常量声明有一些特别说明：

1. 在所有左花括号{后面、switch 语句中的 case 标号后面都可以出现声明，包括变量/常量声明、函数原型声明；
2. 所有声明都必须出现在包含这些声明的语句块中的第一条可执行语句之前。例如：例 ex1-1 中的 int a;就出现在第一条可执行语句 a = 3 * 4;这条赋值语句之前，而下面的例子是错误的：

```
{
    int a;      //ok
    a = 3 * 4;
```

```
    int b; //错误，声明出现在可执行语句之后
    b = ++a;
}
```

{在任何程序需要的地方出现变量/常量声明是 C++的语法，因为 C++视所有的声明为语句。如果在编辑程序的时候误将一个出现上述情况的 C 程序保存为.cpp 格式，那么 C++编译器不会认为这是一个错误。但如果程序员后来发现文件类型错误而将其改成.c 格式，那么 C 编译器将会报出错误。}

习题 3-1	请将上面的代码扩充成一个完整的 C 程序，保存成.c 格式。看看在编译时会出现什么样的错误提示？
习题 3-2	将 3-1 中的.c 文件的后缀改为.cpp，然后编译这个文件，看看又会出现什么情况？

3.3 顺序控制结构

在例 ex1-1 的 main()函数中，三条可执行语句按照特定的逻辑顺序依次排列起来，而它们的执行也将按照这个顺序依次执行。这就是**顺序控制结构**的概念。

顺序结构是三种控制结构中最简单的一种。从全局的角度来看，一个函数中的所有语句的排列都属于顺序结构。但是，顺序结构虽然简单，但其中语句排列的顺序却一定是按照某种特定的逻辑来确定的。而这种逻辑顺序是不能打乱的，否则将会使程序的运行结果不正确。例如：例 ex1-1 中三条可执行语句的顺序就不能改变。究其原因，是因为输出语句依赖于赋值语句的结果，而返回语句铁定是逻辑顺序中最后一条。

但就局部而言，当前后几条语句没有结果依赖关系的时候，这些语句的顺序是可以交换的。例如：以下两条语句可以交换顺序而不影响逻辑的正确性：

```
a = 0;
b = 1;
```

顺序结构因其简单而容易掌握，因此这里就不再赘述了。在下面的内容中，将会着重介绍 C 的输入输出方式。

3.4 字符输入/输出

这一节介绍 C 标准 I/O 函数库中最简单的、也是最容易理解的字符输入输出函数 putchar()和 getchar()。

3.4.1 getchar()函数（字符输入函数）

函数 getchar()是从标准输入设备（键盘）上输入一个字符，该函数没有参数，其原型为：

```
int getchar();
```

函数的返回值就是从输入设备得到一个字符（的 ASCII 码）。当该函数执行时，程序暂停下来等待用户的输入。一旦用户在键盘上按下一个可打印字符建（例如字母、数字、符号等等），那么输入的字符将在光标处回显出来；当用户按下回车键时，输入完成，函数的执行结束，并将输入字符返回到调用处。此时，可以用一个 char（或 int）变量来接受这个返回值。如果输入过程中发生错误，则函数返回 EOF 值。

{ EOF 是一个在头文件 stdio.h 中预定义的符号常量，用于表示文件结束，并被常用于错误指示。}

【例 3-1】 输入单个字符

```
//ex3-1.c
#include  <stdio.h>

int main()
{
    char c;
```

```
    printf("Please input a character\n");
    c = getchar();  //输入一个字符，存放在变量 c 中
    printf("You typed in %c.\n", c);

    return 0;
}
```

程序的结果是：

```
Please input a character:
x↙
You typed in X.
```

使用 getchar()函数还应注意几个问题：

1. getchar()将所有输入都视为字符，并且返回该字符的 ASCII 码。例如键入 0 时，得到的不是数 0，而是字符'0'（ASCII 码为 48）；
2. getchar()函数只能处理可打印字符。如果按下键盘上的 F1-F12、esc、insert/delete、光标控制键、shift/ctrl/alt/windows 键等，函数一般会忽略按键而继续等待；
3. getchar()是一个被称为“带缓冲输入（buffered input）”的函数。这类输入函数要求在输入完成后比须按一次回车键函数才能返回，否则将会一直等待。如果不输入任何字符而直接回车，那么函数将返回回车键的 ASCII 码—10；
4. getchar()只接收单个字符。当输入多于一个字符时，只接收第一个字符。例如：当输入序列是 abcde↙，那么 getchar()只返回字符'a'。不过，多余的输入将会被存放到系统设置的缓冲区（buffer）中。当下一次调用 getchar()时，函数不会等待而是立即返回。此例中将会返回字符'b'；
5. getchar()多数情况下出现在赋值语句中。如果单独出现，那么其功能是使程序暂停，好让用户观察现阶段程序的运行情况，按下回车见后程序继续执行；
6. getchar()不能输入宽字符。

{宽字符是用 16 位二进制表示的字符，常用于表示汉字、日文等非英语字符。getchar()函数要求输入的是一个用 8 位二进制表示的可打印的 ASCII 字符。}

习题 3-3　请编写一个程序来验证上述提到的注意问题。

3.4.2 putchar()函数（字符输出函数）

用于输出的函数 putchar()的功能是向标准输出设备（屏幕）上输出一个字符，其原型为：

int putchar(int ch);

它将变量 ch 以可打印的字符形式显示在输出设备上。ch 可以是常量、变量、转义字符或表达式等，其数据类型可以是字符型或整型。如果是整型数据，那么它代表的是与一个字符相对应的 ASCII 码值。

函数的返回值是要输出的字符。如果发生错误，则函数返回 EOF。

【例 3-2】输出字符

```
//ex3-2.c
#include <stdio.h>

int main()
{
    char ch = 'B';
    putchar(ch);          //输出字符变量 ch 的可打印形式（B）
    putchar('B');         //输出大写字母 B
    putchar('\x42');      //用转义序列来输出字母 B
    putchar(0x42);        //直接用 ASCII 码值输出字母 B
```

```
    putchar('\n');          //换行

    return 0;
}
```

程序的结果是：

```
BBBB

_
```

{符号_表示光标所处的位置。}

3.5 格式化输入/输出

getchar()和 putchar()只能处理单个的字符。如果要输入输出 int、double 等数据，那么就可以使用 C 语言标准函数库中提供的带格式输入输出函数——printf()函数和 scanf()函数。格式化的意思是我们可以定制输入输出的格式，包括数据的类型、显式格式等等。

3.5.1 格式化输出函数 printf()

printf()函数的原型为：

int printf(const char *format, arguments…);

其中，arguments 是一个由逗号隔开的输出参数列表。

printf()函数的功能是将输出列表中的数据按格式字符串 format 指定的格式输出。其中，format 中可以包含**格式占位符**，用以对应输出列表中的数据。如果 format 中不包含格式占位符，那么格式字符串在输出时将按原样打印，起到提示的作用。

printf()的返回值是打印出的字符数目，在出错的情况下会返回一个负值。如果 format 是个空串，那么会发生一个错误。

在一般情况下，我们会忽略 printf()函数的返回值。

现在我们来详细讨论格式占位符。格式占位符由**“%”**符号开头，后跟**格式字符**，用来指定输出项的类型、形式、长度、小数位数等格式。如**“%d”**表示按十进制整型输出，**“%f”**表示按十进制浮点型输出，**“%c”**表示按字符型输出等。

格式占位符的一般形式为：

%[标志][长度修饰符][域宽][.精度]类型转换字符

1. 类型转换字符

类型转换字符用以表示输出数据的类型，其格式符和意义表 3-1 所示：

表 3-1 类型转换字符

类型转换字符	含义
d, i	以十进制形式输出带符号整数(正数不输出符号)
o	以八进制形式输出无符号整数(不输出前缀 0)
x, X	以十六进制形式输出无符号整数(不输出前缀 0x)。x 用 0x，X 用 0X
u	以十进制形式输出无符号整数
f, F	以小数形式输出单、双精度实数
e, E	以指数形式输出单、双精度实数
g, G	以 e、f 中较短的输出宽度输出单、双精度实数
a, A	以十六进制形式输出整数，包含前缀 0x。a 用小写，例如 0xa7，A 用大写，例如 0xA7
c	输出单个字符
s	输出字符串
%	%本身

2. 标志

表 3-2 标志字符

标志	含义
-	结果左对齐，右边填空格
+	输出正号或负号
#	对 c，s，d，u 类无影响；对 o 类，在输出时加前缀 0，对 x 类，在输出时加前缀 0x；对 e,g,f 类当结果有小数时才给出小数点
0	对 d，i，o，u，x，X，a，A，e，E，f，F，g 和 G，填充前导 0 到指定宽度

3. 长度修饰符

表 3-3 长度修饰符

长度修饰符	含义
h	输出短整型
l	对 d，i，o，u，x 或 X，输出长整形；对 c 和 s，输出宽字符（串）。对浮点格式无影响
L	对 a，A，e，E，f，F，g 和 G，按双精度浮点数输出

注：表 3-1、3-2、3-3 中只列出了最常用的符号

4. 域宽

用十进制整数来表示输出的最少位数。若实际位数多于定义的宽度，则按实际位数输出，若实际位数少于定义的宽度则补以空格或 0。

5. 精度

精度格式符以“.”开头，后跟十进制整数。如果输出数，则表示小数的位数；如果输出的是字符，则表示输出字符的个数；若实际位数大于所定义的精度数，则截去超过的部分。

一般地，printf()的格式字符串中的格式占位符与输出参数列表中参数的数目是一样的，并且按出现的顺序一一对应。如有：

```
printf(“%d%c%f”, a, b, c);
```

则格式占位符%d 对应变量 a，%c 对应变量 b，%f 对应变量 c，表示将变量 a、b、c 分别按十进制整数、字符和单精度浮点（float）格式输出。

如果参数数目多于格式占位符数目，那么编译器不认为这是一个严重错误，但多出的那些参数将会被忽略而不会输出。

如果参数数目少于格式占位符数目，那么编译器也不认为这是一个严重错误，但输出结果要是编译器的行为而定。

【例 3-3】printf()函数的使用。

在这个例子中，将展示格式转换字符、浮点精度和域宽的使用。

```
//ex3-3.c
#include <stdio.h>

int  main()
{
    int a = -1;
    float b = 1243.9341;
    double c = 24212345.2478;
    char d = 'A';
    int e = 65;

    printf("a=%d, %5d, %u, %o, %x\n",a, a, a, a, a, a);
```

```
    printf("b=%f, %1.2f, %5.4f, %e\n", b, b, b, b);
    printf("c=%lf, %8.3lf, %g\n",c, c, c, c);
    printf("d=%c, %8c\n", d, d);
    printf("|%10s|\n", "abcdefg");
    printf("|%-10s|\n", "abcdefg");
    printf("%c, %d\n", d, d);
    printf("%d, %c\n", e, e);

    return 0;
}
```

程序的结果如下：

```
Xxxxx
```

请读者注意输出结果中的一些细节：

1. 负数转换成无符号整数后会得到意想不到的结果，如本例中的-1 用 u 格式输出后结果为 4294967295。不过，这个结果不是随即产生的，而是遵循一定的规律的。本例中，机器字长是 32 位，那么可以发现，$4294967295=2^{32}+(-1)$；
2. 如果待输出的参数个数多于格式占位符的个数，那么多出的参数将会被忽略，如本例中第一条 printf()语句，指定了 5 个占位符，但却给出了 6 个参数，因此最后一个多出的参数被忽略，没有被输出。不过，这个参数代表的表达式的值却被计算了，其结果也被忽略；
3. 浮点数的存储值可能与指定的值有细微的差异，如本例中 b=1243.9341，但存储结果是 1243.934082；
4. 如果指定的宽度短于输出数的位数，那么指定宽度无效，数据会原样输出，但精度仍会其作用，如本例中 b 用%1.2f 格式输出，虽然宽度指定为 1，但由于 b 的有效位数是 8，所以指定宽度无效，但精度 2 却使小数部分被截断；
5. double 类型的数据应该采用%lf 格式输出，否则会产生不可预知的错误结果；
6. 如果指定的精度短于存储值，那么输出结果将四舍五入，例如本例中%8.3lf 输出 c 时；
7. 类型转换字符应该与对应参数的类型匹配。如果二者不匹配，那么类型转换字符将起决定性作用。也就是说，不管数据的类型是什么，printf()都将坚持用类型转换字符指定的格式输出。很明显地，这里将会出现隐式的类型转换。如本例中用%d 格式输出字符'A'，其结果是'A'的 ASCII 码 65。相反，用%c 格式输出整数 65，则这个整数会被认为是某个字符的 ASCII 码，因而其输出结果是个字符。

习题 3-4　请编写一个程序输出如下图形：

```
*
**
***
****
```

3.5.2 格式化输入函数 scanf()

scanf()函数的原型为：

int scanf(const char *format, arguments…);

其中，arguments 是由逗号隔开的**地址列表。**

scanf()函数的功能是：

1. 从标准输入设备（一般情况下是键盘）输入的数据；
2. 按照指定的格式将这些数据转换成指定的类型；
3. 将这些数据依次存放到地址列表指定的参数中。

函数的返回值是被正确转换的数据的个数。如果在转换过程中发生错误，那么这个返回值可能少于参数的个数，甚至是 0。

scanf()的格式字符串与 printf()的极为相似，这里就不在赘述了，只强调格式字符串中需要注意的几点：

1．如果格式字符串中包含不属于格式占位符的字符，那么在输入时必须原样输入这些字符；

2．每个格式占位符间可以没有间隔，或者用空格、制表符（按 tab 键输入）隔开；

3．格式占位符的数目应该与地址列表中参数的数目相同。如果前者少于后者，则多余的参数作为表达式会被计算，但其本身和结果都被忽略。

{一些程序员喜欢用逗号,作为输入项的分割符。那么，在 scanf()的格式化字符串中，就应该在每个格式占位符间用逗号隔开。反过来，如果格式占位符用,隔开，那么在输入时必须在每个输入项间键入,。如果格式占位符间是空格、制表符甚至没有间隔，那么在输入时可以用空格、制表符分开各输入项。}

特别强调，scanf()的参数是一个地址列表，这与 printf()的参数类表有很大的区别。每个地址都有如下形式：

&变量名

如果变量名是一个数组名或指针名，那么就不能再用&符号。关于地址、数组和指针的内容将在后续的章节中讲述，这里只需要读者记住这种用法。

【例 3-4】scanf()函数的使用：从键盘输入三个整型变量的值

```
//ex3-4.c
#include <stdio.h>

int main()
{
    int a, b, c;

    scanf(“%d”, &a)
    scanf(“%d,%d”, &b, &c); //请注意,的存在

    printf(“a=%d, b=%d, c=%d\n”, a, b, c);

    return 0;
}
```

程序的输出结果是：

```
11↙
22,33↙
a=11, b=22, c=33
```

{请读者注意在输入 b 和 c 时，由于格式字符串中包含了一个不属于任何类型转换符的符号’,’，因此在输入时必须输入这个逗号，否则输出结果将不正确。}

在多数情况下，scanf()工作的很好。但在一些特殊的场合，却会造成一些困扰。

1．输入 double 类型的数据时

如果有声明 double d，那么使用

```
scanf(“%f”, &d);
```

将得不到想要的结果。正确的输入格式是使用%lf 格式：

```
scanf(“%lf”, &d);
```

2．输入的 char 类型数据不是第一项时

如果 char 数据的输入是在其他数据之后，那么就必须小心使用 scanf()。例如有声明

```
int i;
```

自忠建版权

```
char c;
```

和输入：

```
scanf("%d%c", &i, &c); //注意两个格式占位符间没有分隔符
```

如果有如下形式的键盘输入：

```
123 A↙
```

那么，i 会得到正确的输入 123，但 c 的值却不是'A'，而是空格！为什么会有这样的结果呢？

这里先要简单解释一下 scanf()的工作原理。与 getchar()一样，scanf()也是一种带缓冲的输入。在用户输入一行信息并按回车后，输入的信息会以字符串的形式先存放到系统缓冲区中，然后 scanf()再从这个缓冲区中读取数据。在此例中，scanf()“知道”第一个输入项应该是整数，因此它尝试依次从缓冲区中读取数字字符。一旦遇到第一个不属于数字的字符，那么该次读取结束，以前读入的那些字符将按规则转化成整数并存放到指定的变量 i 中。此后，scanf()试图按第二个格式（字符格式）读取后续的数据。而此时，缓冲区中的当前字符（也就是终止了上一次读取的字符）是空格，刚好符合格式要求，因此 c 得到空格也就不奇怪了。此时，未读取的字符'A'留在了缓冲区中。

为解决上述问题，可以使用如下方式：

```
scanf("%d %c", &i, &c); //两个格式占位符间加入一个空格
```

在这种情况下，scanf()的格式字符串中有不属于格式占位符的字符（此例中是空格），那么在输入的时候必须在输入项间加入一个空格以匹配要求。而例中的输入刚好符合要求。

另一种解决问题的方法是不改变 scanf()的格式字符串，而在输入时将字符数据紧接在整数后面：

```
123A↙
```

第三种解决方案是用 getchar()函数来读取字符数据。

3．在一条 scanf()后用另一条 scanf()或 getchar()读取字符数据时

如果有声明

```
int i;
char c = 'A';
```

和语句序列

```
scanf("%d", &i);
scanf("%c", &c); //或 c = getchar();
printf("%d,%c", i, c);
```

当用户键入如下输入后，

```
123↙
```

第二条 scanf()或 getchar()将不会等待用户输入而立即返回，并得到如下输出结果：

```
123,

_
```

这又是为什么呢？

请读者注意在输出结果中，当前光标位置前有一个空行，但程序代码并没有明确要求输出空行（即打印一个'\n'字符）。显然，唯一合理的解释就是字符 c 的值是'\n'。读者不禁要问：这里才完成第一次输入，而第二次输入还没有开始呢！即使要打印，那么 c 的值也应该是'A'，怎么也不会是'\n'啊！

造成以上错误结果的“罪魁祸首”仍然是 scanf()的缓冲特性。在此例中，所有用户输入，包括最后输入的回车符都被存入了缓冲区。这个标志着输入结束的回车符将使 scanf()的第一次输入结束，但它仍然留在了缓冲区中。而当第二条带缓冲的输入函数执行时，如果要求的输入碰巧是一个字符，那么残留的回车符正好满足这个需求，因此函数立即返回，而指定的字符变量将会得到'\n'。

这是 scanf()函数的一个“臭名昭著”的历史遗留问题。解决问题的方法似乎比较笨拙，但却很有效，就是在每一条 scanf()后面加上一条 getchar()调用以吸收残留的'\n'。例如：

```
scanf("%d", &i); getchar();
scanf("%c", &c);
```

但这种方案还是有一个缺陷。如果输入是 123ABC↙X 并期望 i 到 123 而 c 得到 X，那么结果还是事与愿违：c 的值是'B'，因为 getchar()只能吸收掉'A'。所以，一个更有效的方法是使用 fflush()函数，具体做法如下：

```
scanf("%d", &i); fflush(stdin);
scanf("%c", &c);
```

其中，fflush(stdin)将丢弃标准输入缓冲区中的所有残留，从而使结果正确。

关于 scanf()更多的情况需要读者在实践中加以总结，这里就不再深入讨论了。

{如果读者在 Windows 环境下编写程序，那么本章讲述的所有输入输出函数都只能在控制台应用程序中使用。这些应用程序在运行时都有一个特点，就是会打开一个拥有简单外观的控制台窗口。请注意，这个窗口是 Windows 系统为控制台应用程序提供的运行平台，并不是程序本身产生的。那些拥有真正 Windows 窗口的应用程序都使用其它的方法进行输入输出。}

习题 3-5　请编写一个程序来验证上述提到的关于 scanf()的问题。

3.6 编译预处理

在前面我们提到，几乎每一个程序都会用#include 命令来包含头文件。这里，#include 是一条编译预处理命令。那么什么是编译预处理呢？

简而言之，编译预处理就是编译器在真正地编译源代码之前，预先扫描源代码，如果发现有需要提前处理的情况，那么就对源代码进行加工改造，然后再对改造后的源代码进行编译以生成二进制目标代码。

有三种常有的编译预处理命令，这里简要介绍它们的功能和使用。

3.6.1 文件包含

正如读者熟悉的那样，文件包含使用#include 命令，其语法如下：

#include <源文件名>

或

#include "源文件名"

其功能是将**源文件名**指定的文件"抄写"在#include 出现的地方，这使得源代码变得更长了。

两种不同的括号指明了如何在磁盘上搜索源文件。<>的潜台词是：编译器，请你到系统指定的文件夹下去搜索头文件，而这个指定文件夹一般都跟编译器位于同一个文件夹中，其名为 include；而""则告诉编译器，先在当前文件夹下搜索，如果没找到，则到指定文件夹下去查找。如果包含的头文件没有找到或不存在，则编译器会报告一个错误。

原则上你可以包含任意 C 源代码到你的程序中，但一般情况下我们只是包含头文件（一般以.h 为后缀）。头文件有系统预先准备好的和用户自己编写的两种。但无论是那一种，一般都包含了函数的原型声明、外部变量声明、宏定义等非执行语句。

3.5.2 宏

宏定义的语法如下：

#define 符号名　　字符串

其中，字符串不用""括起来。

在编译预处理阶段，一旦编译器发现了宏，那么在编译之前，编译器将把在源程序中所有出现*符号名*的地方用**字符串**来代替。例如有宏定义：

```
#define INFLATEDRATE  0.05
```

和语句

```
price = price * (1.0 + INFLATEEDRATE);
```

那么在编译预处理时，上面的语句将会被替换成：

```
price = price * (1.0 + 0.05);
```

然后再进行正式的编译。

可以看到，宏一般用于定义符号常量，以避免在程序中出现大量的神仙数字，从而在一定程度上降低代码的维护成本。

使用宏需要注意一些细节。宏只是一个字面串，程序中出现宏的地方仅在编译预处理阶段被直接替换，正如读者在 Word 中使用 Ctrl+H 来替换文档中的匹配字符串那样。正是由于这个原因，设计不好，宏将可能会引入错误。比如，因为宏定义不是一条可执行语句，所以一般不要在其后加上分号。例如将上面的宏定义改成：

```
#define INFLATEDRATE  0.05;
```

那么在替换时语句将变成：

```
price = price * (1.0 + 0.05;);
```

显然这是一条错误的语句。

在这里强烈建议读者，在可能的情况下尽量使用 const 取代#define 来定义符号常量，以避免#define 可能引入的问题。

3.5.3 条件编译

我们常用的条件编译命令如下：

```
#ifdef（或者 #ifndef） 符号名
  代码段 1
[#else
  代码段 2]
#endif
```

其功能是：如果定义了（或者没有定义）符号名，那么编译代码段 1，否则编译代码段 2。其中，#else 分支是可以省略的。

条件编译的一个非常有用的功能就是设置头文件保护（header guard）。例如，有以下自定义头文件 myheader.h，其内容如下：

```
#define INFLATEDRATE  0.05
```

而在源文件 myprog.c 中，由于直接或间接的原因将 myheader.h 包含了两次以上，那么编译器在编译的时候就会报错：符号 INFLATEDRATE 被重复定义了。为了避免这种情况，我们可以将 myheader.h 改成这样：

```
#ifndef __myheader__
#define __myheader__

#define INFLATEDRATE  0.05

#endif
```

那么，当编译器处理第一次头文件包含时，因符号__myheader__并未定义，夹在#ifndef 和#endif 中的内容会被处理；而在第二次或更多的包含中，处理过程正好相反。这样一来，就会有效解决重复包含的问题。

避免头文件被重复包含的另一种方式是在头文件的第一行插入如下预处理指令：

```
#pragma once
```

其含义是保证头文件只被包含一次。但这条预处理指令强烈依赖于编译器的实现，因此可能不能适用于所有的编译器。

3.7 解决方案

学习完本章的内容，相信读者已经能够完成信息输入输出的设计。现在就接着第二章的解决方案来继续深化案例的解决。

在 solution1.c 中，读者可以明显地看到程序分成了两个相对独立的部分：一个是关于类型（方案中主要是枚举类型）的定义部分，另一个是完成功能的实现部分。那么，我们首先要做的一件是就是把这两部分分开，分别放在不同的 C 文件中。一般情况下，把类型定义部分放在一个后缀为.h 的头文件中，而将程序实现部分放在一个后缀为.c 的文件中，并且在这个.c 文件中包含前面提到的.h 文件。

根据上面的思路，我们将 solution1.c 改造成如下两部分：

1. solution.h 的代码

```
//solution.h
#pragma once

//枚举类型及其常量声明，用于变量 gender 的取值范围
enum GENDER { MALE = ‘M’, FEMALE = ‘F’ };

//枚举类型及其常量声明，用于变量 level 的取值范围
enum LEVEL { levelA = ‘A’, levelB, levelC, levelD, levelE };
```

2. solution2.c 的代码

```
//solution2.c
#include <stdio.h>
#include “solution.h”

int main()
{
  //学生信息声明
  int   studentID;          //学生编号
  char name[30];            //姓名，用字符数组存放。姓名的最大长度不超过 30 个字符
  enum GENDER gender;       //性别，取值 MALE 和 FEMALE 之一
  int grade;                //年龄
  int score;                //得分
  enum LEVEL level;         //等级

  printf(“学号：”); scanf(“%d”, &studentID); getchar();
  printf(“姓名：”); scanf(“%s”, name); getchar();
  printf(“性别：”); scanf(“%c”, &gender); getchar();
  printf(“年级：”); scanf(“%d”, &grade); getchar();
  printf(“得分：”); scanf(“%d”, &score); getchar();

  printf(“\n 您输入的学生信息如下：\n”);
  printf(“学号\t 姓名\t 性别\t 年级\t 得分\n”);
  printf("-----------------------------------\n");
  printf(“%d\t%s\t%c\t%d\t%d\n”, studentID, name, gender, grade, score);

  return 0;
```

自主建版权

```
}
```

请读者仔细体会程序代码是如何格式化信息的输出样式的。

{在后续的章节中，随着知识点的增多，解决方案的代码肯定会不可避免地变长。为了节省篇幅，后续的解决方案将不再出现完整的程序，而只是针对某个局部的主体代码。}

习题 3-6　请读者依照笔者的思路将程序编写完整，然后测试其正确性。

本章小结

C 语言的格式输入输出的规定比较繁琐，而输入输出却是最基本的操作，几乎每一个程序都包含输入输出。不少编程人员由于掌握不好这方面知识而浪费了大量调试程序的时间。因此本章比较仔细地介绍了字符输入输出函数——getchar()函数和putchar()函数，格式输入输出函数——scanf 函数和 printf 函数，以便在编程时有所遵循。但在学习时重点掌握常用的一些规则即可，其他部分可在需要时随时查阅。

其后我们简要讲述了编译预处理，它广泛地应用于头文件包含、符号常量定义、程序调试等方面。

自忠建版权