第 3 章　键盘输入与屏幕输出

📖 内容关键词

✍ C 语句分类

✍ 字符输入/输出函数

✍ 格式输入/输出函数

📖 重点与难点

✍ 表达式与表达式语句的区别

✍ scanf()语句的正确用法

✍ 输入/输出数据时的格式控制

📖 典型实例

✍ 以不同格式输入和输出两个整型数

3.1　C 语句分类

（1）控制语句。C 语言只有如下 9 种控制语句（Control Statement）。

```
if~else        for()~         continue
switch         while()        ~goto
break          do~while()     return
```

（2）变量定义语句。由类型关键字后接变量名（如果有多个变量名，则用逗号分隔）和分号构成的语句，如“int a, b, c;”。

值得注意的是，变量定义语句不是可执行语句。它只是将某些信息传递给编译器，通知编译器变量的类型是什么，以便编译器为其预留出相应大小的存储空间，因为不同类型的变量在内存中占据不同大小的存储空间。

（3）表达式语句。由表达式后接一个分号构成的语句。

（4）函数调用语句。表达式必须是有值的，而函数调用不一定是有返回值的。在 C 语言中，没有专门的输入/输出语句，输入/输出操作通常是通过调用输入/输出函数来实现的。

（5）复合语句。两条或两条以上的语句序列，用一对花括号括起来构成的语句。

（6）空语句。只有一个分号构成的语句，表示什么也不做。

3.2　表达式语句

顺序结构是最简单的程序结构。在顺序结构程序中，程序的执行是按照语句书写的顺序

来完成的，赋值操作是顺序结构中最常见的操作。但是在 C 语言中，没有专门的赋值语句，赋值操作通常是用赋值表达式后接一个分号（;）构成赋值表达式语句实现的。例如，c = a + b 只是一个赋值表达式，而

```
c = a + b; //赋值表达式语句
```

则是一个表达式语句。表达式语句与表达式在概念上是完全不同的。

3.3 复合语句和空语句

1. 复合语句

两条或两条以上的语句序列，用一对花括号括起来构成的语句，称为复合语句（Compound Statement），也称为语句块（Block）。其一般形式为：

```
{
   语句1
   语句2
   ……
   语句n
}
```

例如，下面的复合语句：

```
{
   temp = x;
   x = y;
   y = temp;
}
```

在逻辑上形成一个整体，在语法上等同于一条语句，可被当作一条语句来处理，这样就为程序设计带来了方便。在放置单条语句的任何地方都可以放置一条复合语句。

通常，将一组逻辑相关的语句组放在一起构成一条复合语句。例如，第 4 章要介绍的条件和循环语句在语法上只允许带一条语句，而要处理的操作往往需要多条语句才能完成，这时采用复合语句就可以解决这个问题。

【编程提示】在复合语句中定义的变量只能在复合语句中使用。

【例 3.1】 这个程序用于演示复合语句中定义的变量只能在复合语句中使用。

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 0;
5      {
6          int a = 1;
7          printf("In: a = %d\n", a);
8      }
9      printf("Out: a = %d\n", a);
10     return 0;
11 }
```

程序的运行结果如下：

```
In: a = 1
Out: a = 0
```

在这个程序中，虽然复合语句里面的变量 a 和外面的变量 a 是同名的，但它们代表不同的变量。在复合语句里面定义的变量 a，只能在里面引用，而在复合语句外面定义的变量 a，则只能在外面引用，从输出结果即可看出这一点。

2. 空语句

在表达式语句中，如果没有任何表达式而只有一个分号，则称为空语句（Null Statement）。空语句的形式如下：

```
;              //空语句
```

空语句什么也不做，只表示语句的存在。既然如此，那么空语句还有什么作用呢？

采用自顶向下（Top-down）的方法（参见第 4 章 4.7 节）设计程序时，在一些未设计完成的模块中，通常都暂时放一条空语句，留待以后对模块实现时再进行扩充。例如：

```
int main(void)
{
    DataInitialize();    //调用用户自定义函数
    /*......*/
    DataOutput();        //调用用户自定义函数
    return 0;
}
void DataInitialize()    //定义用户自定义函数
{
    ;                    //空语句
}
void DataOutput()        //定义用户自定义函数
{
    ;                    //空语句
}
```

3.4　基本的输入/输出操作

C 语言中没有提供专门的输入/输出语句，输入/输出操作是通过调用 C 的标准库函数来实现的。C 的标准函数库中提供许多用于标准输入/输出操作的库函数，使用这些标准输入/输出函数时，只要在程序的开始位置加上如下一行编译预处理命令即可：

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

它的作用是：将输入/输出函数的头文件 stdio.h 包含到用户源文件中。其中，h 为 head 之意，std 为 standard 之意，i 为 input 之意，o 为 output 之意。

本章只涉及 ANSI C 标准定义的输入/输出函数。由于各种机器间的差异太大，因此 ANSI C 没有定义实现各种屏幕控制（如鼠标定位）或图形显示的函数，多数 C 编译程序都在标准 C 的基础上增加了自己的显示和图形库函数，当然，这些函数只能运行于特定的编译环境。

3.4.1　字符输入/输出

getchar()和 putchar()是专门用于字符输入/输出的函数。其中，getchar()用于从键盘读一个

字符，它等待击键，待用户击键后，将读入值返回，并自动将用户击键结果回显到屏幕上；putchar()则把字符写到屏幕的当前光标位置。这两个函数的使用格式如下：

```
变量 = getchar();
putchar( 变量 );
```

【例 3.2】 下面这个程序用于演示如何使用函数 getchar()和 putchar()。

```
1  #include  <stdio.h>
2  int main(void)
3  {
4       char  ch;
5       printf("Press a key and then press Enter:");
6       ch = getchar();          //从键盘输入一个字符,并将该字符存入变量 ch
7       printf("You pressed ");
8       putchar(ch);             //在屏幕上显示变量 ch 中的字符
9       putchar('\n');           //输出一个回车换行符
10      return 0;
11 }
```

程序首先执行第 5 行可执行语句，这时会在屏幕上显示如下提示信息：

```
Press a key and then press Enter:
```

然后执行第 6 行可执行语句，程序等待用户从键盘输入一个字符，例如用户从键盘输入一个字符 A，并按一下回车键（以下用↙表示用户输入一个回车键），那么程序继续向下执行第 7 行语句，在屏幕上显示如下信息：

```
You pressed
```

接着执行第 8 行语句，在“You pressed”的后面再显示一个字符 A，即

```
You pressed A
```

最后执行第 9 行语句，目的是将光标移到下一行的起始位置。

假如将第 7 ~ 8 行的语句复制并粘贴到第 9 行以后，那么程序的运行结果如下：

```
Press a key and then press Enter: A ↙
You pressed A
Press a key and then press Enter: B ↙
You pressed B
```

而此时，如果删掉第 9 行语句，那么程序的运行结果如下：

```
Press a key and then press Enter: A ↙
You pressed A Press a key and then press Enter: A↙
You pressed A
```

从这个例子可以得出以下三点结论。

① 函数 putchar()的作用是向终端显示器屏幕输出一个字符。这个字符可以是可打印字符，也可以是转义序列，函数 putchar()的参数就是待输出的字符。

② 函数 getchar()的作用是从系统隐含指定的输入设备(即终端键盘)输入一个字符，按回车键表示输入结束。函数 getchar()没有参数，函数的返回值就是从终端键盘读入的字符。

③ 输出'\n'的作用是将光标移到下一行的起始位置处。

3.4.2　格式输入/输出

【例 3.3】　以整型格式输入一个变量的值，然后以整型格式输出这个变量的值。

```
1  #include  <stdio.h>
2  int main(void)
3  {
4       int  var;
5       printf("Please enter a number and then press Enter:");
6       scanf("%d", &var);
7       printf("The number you entered was %d\n", var);
8       return 0;
9  }
```

在这个程序中，用到了一个格式输入函数 scanf()和一个格式输出函数 printf()。函数 scanf()的作用是按指定格式要求和数据类型，读入若干数据给相应的变量，而函数 printf()的作用是输出一行字符串，或者按指定格式和数据类型输出若干变量的值。

程序运行时，首先执行第 5 行语句，向屏幕输出如下一行字符串：

```
Please enter a number and then press Enter:
```

输出这行字符串的目的是给用户显示一个提示信息，提示用户做什么和如何做。输出这行提示信息后，程序开始执行第 6 行语句，函数 scanf()的第一个参数为格式控制字符串"%d"，输入数据的格式和类型一般都在控制字符串中指定。本例中，格式控制字符串中的%d 指定输入数据为整型，当程序执行到该函数调用语句时，程序将一直等待直到用户从键盘输入一个整型数据且按回车键为止。假设用户输入了整型数据 6，那么 6 将被送给变量 var 存储。注意，第 6 行语句中的变量 var 前的运算符&为取地址运算符，&var 表示取变量 var 的地址，在第 6 行语句中，&var 指出输入数据将要存入的内存地址，正如寄信需要写上邮寄地址一样，函数 scanf()要求在其参数中指定输入数据的存储地址。

当程序执行到第 7 行语句时，除显示字符串"The number you entered was"以外，还会在其后显示整型变量 var 的值 6。与函数 scanf()类似，函数 printf()中也有一个格式控制字符串，其中的%d 是指定按十进制整型格式输出变量 var 的值。

根据以上分析可知，例 3.3 程序的运行结果应为：

```
Please enter a number and then press Enter: 6 ↙
The number you entered was 6
```

若要输入/输出实型数据，则将格式转换说明%d 改成%f 即可。请看下例。

【例 3.4】　上机运行本例的程序，根据程序提示输入数据，然后观察并写出程序运行结果。

```
1  #include  <stdio.h>
2  int main(void)
3  {
4       float  var;
5       printf("Please enter a number and then press Enter:");
6       scanf("%f", &var);
7       printf("The number you entered was %f\n", var);
```

```
8       return 0;
9   }
```

仿照例 3.3 分析可知，该程序的运行结果如下：

```
Please enter a number and then press Enter: 6.2 ↙
The number you entered was 6.200000
```

这里，按%f 格式输出实型数据时，除非特别指定，否则隐含输出 6 位小数。

除%d 和%f 这两种常用的格式转换说明外，还有其他格式转换说明。下面对 scanf()和 printf()这两个函数的用法进行归纳总结。

1. 函数 printf()的一般格式

```
printf(格式控制字符串);
printf(格式控制字符串, 输出值参数表);
```

格式控制字符串（Format Control String）是用双引号括起来的字符串，也简称为格式字符串（Format String），输出值参数表中可有多个输出值，也可没有（当只输出一个字符串时）。一般，格式控制字符串包括两部分：如表 3-1 所示的格式转换说明符（Conversion Specifiers）和需原样输出的普通的文本字符（Literal characters）。

表 3-1 函数 printf()的格式转换说明符

格式转换说明符	用 法
%d 或%i	输出带符号的十进制整数，正数的符号省略
%u	以无符号的十进制整数形式输出
%o	以无符号的八进制整数形式输出，不输出前导符 0
%x	以无符号十六进制整数形式（小写）输出，不输出前导符 0x
%X	以无符号十六进制整数形式（大写）输出，不输出前导符 0x
%c	输出一个字符
%s	输出字符串
%f	以十进制小数形式输出实数（包括单、双精度），整数部分全部输出，隐含输出 6 位小数，输出的数字并非全部是有效数字，单精度实数的有效位数一般为 7 位，双精度实数的有效位数一般为 16 位
%e	以指数形式（小写 e 表示指数部分）输出实数，要求小数点前必须有且仅有 1 位非零数字
%E	以指数形式（大写 E 表示指数部分）输出实数
%g	根据数据的绝对值大小，自动选取 f 或 e 格式中输出宽度较小的一种使用，且不输出无意义的 0
%G	根据数据的绝对值大小，自动选取 f 或 E 格式中输出宽度较小的一种使用，且不输出无意义的 0
%p	以主机的格式显示指针，即变量的地址
%n	令 printf()把自己到%n 位置已经输出的字符总数放到后面相应的输出项所指向的整型变量中，printf()函数返回后，%n 对应的输出项指向的变量中存放的整型值为出现%n 时已经由 printf()函数输出的字符总数，%n 对应的输出项是记录该字符总数的整型变量的地址
%%	显示百分号%

格式控制字符串描述了输出格式。在一个格式控制字符串中可以有多个转换说明。输出值参数表逐个对应格式控制字符串中的每个转换说明。每个格式转换说明（Conversion Specification）都以一个%开始、以一个格式字符作为结束，用于指定各输出值参数的输出格式。

输出值参数表是需要输出的数据项的列表，这些输出数据项可以是变量或表达式，输出值参数之间用逗号分隔。输出值的数据类型应与格式转换说明符相匹配。每个格式转换说明符和输出值参数表中的输出值参数是一一对应的，如果没有输出值参数，那么格式控制字符串中就不再需要格式转换说明符了。

【例 3.5】　下面这个程序用于演示格式转换说明符%f、%e 和%g 的用法。

```
#include  <stdio.h>
int main(void)
{
      double  f1 = 1.123456789e+10;
      double  f2 = 3.14;
      printf("%%f: %f\n", f1);
      printf("%%e: %e\n", f1);
      printf("%%g: %g\n", f1);
      printf("%%f: %f\n", f2);
      printf("%%e: %e\n", f2);
      printf("%%g: %g\n", f2);
     return 0;
}
```

在 Visual C++ 6.0 环境下，程序的运行结果如下：

```
%f: 11234567890.000000
%e: 1.123457e+010
%g: 1.12346e+010
%f: 3.140000
%e: 3.140000e+000
%g: 3.14
```

在这个例子中，为了输出%号，用了连续两个%来输出%号，因此%%后面的字符不再是格式字符，而是作为普通字符原样输出到屏幕上。对于实数 1.123456789e+10，使用%e 时输出的宽度较小，而对于实数 3.14，使用%f 时输出的宽度较小。

所有浮点数转换说明符的默认精度都是 6。转换说明符%f、%e 或%E 的输出精度是指小数点后的小数位数，而转换说明符%g 或%G 的输出精度是指包含小数点左边数字在内的有效数字的最大个数。例如 1.123456789e+10 用转换说明符%g 输出的结果是 1.12346e+010，输出结果中有 6 个有效数字。而用转换说明符%e 输出的结果是 1.123457e+010，输出结果中有 7 个有效数字，小数位数是 6。

需要注意的是，转换说明符%E、%e 和%g 会在输出时对数据进行舍入处理，而转换说明符%f 则不会。所以，输出数据时，一定要确保用户知道，格式化处理使得输出的数据有可能是不精确的。此外，采用某些编译器时，浮点数输出结果的指数部分的+号后边只显示两位数字。

【例 3.6】　这个程序分别用%p、%x 和%X 三种格式符输出一个整型变量的地址值。

```
#include  <stdio.h>
int main(void)
{
```

```
4       int  var = 12;
5       printf("%%p: %p\n", &var);
6       printf("%%x: %x\n", &var);
7       printf("%%X: %X\n", &var);
8       return 0;
9   }
```

在 Visual C++ 6.0 下，程序的运行结果如下：

```
%p: 0013FF7C
%x: 13ff7c
%X: 13FF7C
```

注意，这个程序在不同的操作系统和编译环境下的运行结果可能是不同的。不过，可以确定的是，在 32 位系统环境下，地址值是一个 32 位数。

2．函数 scanf()的一般格式

scanf(格式控制字符串，参数地址表)；

其中，格式控制字符串是用双引号括起来的字符串，它包括格式转换说明符和分隔符两部分。函数 scanf()的格式转换说明符通常由%开始并以一个格式字符结束，用于指定各参数的输入格式，具体如表 3-2 所示。

表 3-2　函数 scanf( )的格式转换说明符

格式转换说明符	用　法
%d 或%i	输入十进制整数
%o	输入八进制整数
%x	输入十六进制整数
%c	输入一个字符，空白字符（包括空格、回车、制表符）也作为有效字符输入
%s	输入字符串，遇到第一个空白字符（包括空格、回车、制表符）时结束
%f 或%e	输入实数，以小数或指数形式输入均可
%%	输入一个百分号%

参数地址表是由若干变量的地址组成的列表，这些参数之间用逗号分隔。函数 scanf()要求必须指定用来接收数据的变量的地址，每个转换说明符都对应一个存储数据的目标地址。如果没有指定存储数据的目标地址，虽然编译器不会提示出错信息，但会导致数据无法正确地读入指定的内存单元中。当函数 scanf()成功调用时，返回值为成功赋值的数据项数；出错时，则返回 EOF（EOF 是“End Of File”的缩写词，表示文件结尾，它是一个在头文件<stdio.h>中定义的整数型的符号常量，C 标准只是将 EOF 定义成一个负整数，通常被定义为−1，但并不一定是−1。因此在不同的系统中，EOF 可能取不同的值）。

3．函数 printf()中的格式修饰符

在函数 printf()的格式说明中，在%和格式符之间的位置，还可插入如表 3-3 所示的格式修饰符，用于指定输出数据的最小域宽（Field Width）、精度（Precision）、对齐方式等。

表 3-3　函数 printf( )的格式修饰符

格式修饰符	用　　法
英文字母 l	加在格式符 d、i、o、x、u 之前用于输出 long 型数据
英文字母 L	加在格式符 f、e、g 之前用于输出 long double 型数据
英文字母 h	加在格式符 d、i、o、x 之前用于输出 short 型数据
最小域宽 m （整数）	指定输出项输出时所占的列数 当 m 为正整数时，若输出数据宽度小于 m，则在域内向右靠齐，左边多余位补空格；当输出数据宽度大于 m 时，按实际宽度全部输出；若 m 有前导符 0，则左边多余位补 0 若 m 为负整数，则输出数据在域内向左靠齐
显示精度.n （大于等于 0 的整数）	精度修饰符位于最小域宽修饰符之后，由一个圆点及其后的整数构成 对于浮点数，用于指定输出的浮点数的小数位数 对于字符串，用于指定从字符串左侧开始截取的子串字符个数
*	当最小域宽 m 和显示精度 .n 用*代替时，表示它们的值不是常数，而由 printf()函数的输出项按顺序依次指定
空格	在没有输出加号的正数前面输出一个空格
+（加号）	在正数前面输出一个加号，在负数前面输出一个减号。这样可以对齐输出具有相同数字位数的正数和负数
0（零）	在输出的数据前面加上前导符 0，以填满域宽
#	当使用八进制转换说明符 o 时，在输出数据前面加上前导符 0 当使用十六进制转换说明符 x 或 X 时，在输出数据前面加上前导符 0x 或 0X 当以转换说明符 e、E、f、g 或 G 输出的浮点数没有小数部分时，强制输出一个小数点（通常，只有小数点后有数字时才会输出小数点） 对于 g 或 G 转换说明符，末尾的 0 不会被删除

注：用 Visual C++在汇编级跟踪可知，调用函数 printf()时，float 型的参数都是先转化为 double 型后再传递的，所以%f 可以输出 double 和 float 两种类型的数据，或者说，输出 double 型数据可以使用%lf 或%f。

【例 3.7】 上机运行下面的程序，然后写出运行结果。

```
#include  <stdio.h>
int main(void)
{
      float  f1 = 100.15799, f2 = 12.55, f3 = 1.7;
      int    n1 = 123, n2 = 45, n3 = 6;
      printf("printf WITHOUT width or precision specifications:\n");
      printf("%f\n%f\n%f\n", f1, f2, f3);
      printf("%d\n%d\n%d\n", n1, n2, n3);
      printf("printf WITH width and precision specifications:\n");
      printf("%5.2f\n%6.1f\n%3.0f\n", f1, f2, f3);
      printf("%5d\n%6d\n%3d\n\n", n1, n2, n3);
      return 0;
}
```

程序的运行结果如下：

```
printf WITHOUT width or precision specifications:
100.157990
12.550000
1.700000
```

```
123
45
6
printf WITH width and precision specifications:
100.16
  12.6
  2
  123
    45
  6
```

这说明，当按照指定的精度打印浮点数时，打印的数值将是小数部分**舍入**（Rounded）到指定的小数点后位数的结果，而存储在内存中的浮点数值是不变的。

还可以通过格式控制字符串后面的实参列表中的整型表达式来指定域宽和精度。方法是：在格式控制字符串中域宽或精度的位置上写上一个星号*。这时，程序先计算实参列表中对应的整型表达式的值，然后用其替换星号。例如，下面这条语句：

```
printf("%*.*f", 7, 2, 100.15799 );
```

将以 7 为域宽，2 为精度，输出右对齐的 100.16。

4．函数 scanf()中的格式修饰符

在函数 scanf()的%与格式符之间也可插入如表 3-4 所示的格式修饰符。

表 3-4　函数 scanf()的格式修饰符

格式修饰符	用　　法
英文字母 l	加在格式符 d、i、o、x、u 之前用于输入 long 型数据； 加在格式符 f、e 之前用于输入 double 型数据
英文字母 L	加在格式符 f、e 之前用于输入 long double 型数据
英文字母 h	加在格式符 d、i、o、x 之前用于输入 short 型数据
域宽 m（正整数）	指定输入数据的宽度（列数），系统自动按此宽度截取所需数据
忽略输入修饰符*	表示对应的输入项在读入后不赋给相应的变量，即让 scanf()函数从输入流中读取任意类型的数据并将其丢弃，而不是将其赋值给一个变量，因此也称为赋值抑制字符（Assignment suppression character）。

注：函数 scanf()没有显示精度.n 格式修饰符，即用函数 scanf()输入实型数据时不能指定显示精度。

【常见错误】在输入函数 scanf()的格式控制字符串的转换说明符中指定显示精度是错误的，只有在输出函数 printf()的格式转换说明符中才能够指定显示精度。

【例 3.8】 上机运行下面的程序，然后写出运行结果。

```
1  #include  <stdio.h>
2  int main(void)
3  {
4       int  a, b;
5       printf("Please input a and b:");
6       scanf("%2d%*2d%2d", &a, &b);
7       printf("a=%d, b=%d, a+b = %d\n", a, b, a+b);
8      return 0;
9  }
```

在程序第 6 行语句中，格式说明符%*2d 中的*为忽略输入修饰符，表示对应该格式说明符的输入项在读入后不赋给任何变量，%2d 中的 2 为域宽附加格式说明，表示从输入数据中按指定宽度 2 来截取所需数据。因此，当输入 123456 时，该程序的运行结果如下：

```
Please input a and b: 123456 ↙
a=12, b=56, a+b = 68
```

若将第 6 行语句中的%*2d 修改为%*2s，并且仍输入 123456，则程序的运行结果仍为：

```
Please input a and b: 123456 ↙
a=12, b=56, a+b = 68
```

读者不难分析，当输入 12345678 时，程序的输出结果仍为：

```
a=12, b=56, a+b = 68
```

注意，在用函数 scanf()输入非字符型数据时，以下几种情况都认为数据输入已结束：

- 输入空格符、回车符、制表符（Tab）。
- 达到指定域宽。
- 输入非数字字符。

例如，例 3.8 程序运行时，如果输入 12345a，则程序运行结果如下：

```
Please input a and b: 12345a↙
a=12, b=5, a+b = 17
```

其中，在从输入数据中按指定宽度 2 来读取第 3 个数据时遇到非法输入字符 a，于是第 3 个输入的数据为 5。

*3.4.3　使用函数 scanf()时需要注意的问题

1. 函数 scanf()对输入数据的格式要求

用 scanf()输入数据时，除格式控制字符串中格式说明以外的其他字符，都必须原样输入。

【例 3.9】　阅读下面程序，并回答问题。

```
1  #include  <stdio.h>
2  int main(void)
3  {
4       int  a, b;
5       scanf("%d %d", &a, &b);
6       printf("a = %d, b = %d\n", a, b);
7       return 0;
8  }
```

问题 1　当要求程序输出结果为 a = 12,b = 34，用户应该如何输入数据？

问题 2　当限定用户输入数据以逗号为分隔符，即输入数据格式为：

```
12, 34↙
```

时，应修改程序中的哪条语句？怎样修改？

问题 3　当程序第 5 行语句修改为如下语句时，用户应该如何输入数据？

```
scanf("a = %d, b = %d", &a, &b);
```

问题 4　当限定用户输入数据为以下格式：

```
1234↙
```

同时要求程序输出结果为 a = 12,b = 34 时，应修改程序中的哪条语句？怎样修改？

问题 5 当限定用户输入数据为以下格式：

```
12↙
34↙
```

同时要求程序输出结果为 a = "12", b = "34"时，应修改程序中的哪条语句？怎样修改？

问题 6 设计程序使得用户可以用任意字符作为分隔符进行数据的输入，对于这样的要求，应该如何修改程序？

下面让我们来逐一解答上述问题。

问题 1 解答 因为程序第 5 行语句中的两个格式转换说明符%d 之间是空格符，这个空格符作为普通字符必须在输入时原样输入，即输入数据之间应以空格作为分隔符。所以，当要求程序输出结果为 a = 12, b = 34 时，用户应该按以下格式输入数据：

```
12 34↙
```

问题 2 解答 当限定用户输入数据以逗号为分隔符，即输入数据格式为

```
12, 34↙
```

时，显然，应该将第 5 行语句修改为

```
scanf("%d, %d", &a, &b);
```

该语句要求输入数据之间必须以逗号分隔，否则将无法正确输入数据。

问题 3 解答 当程序第 5 行语句修改为如下语句：

```
scanf("a = %d, b = %d", &a, &b);
```

时，用户应在数据输入时将字符串"a = "和"b = "原样输入，即按以下格式输入数据：

```
a = 12, b = 34↙
```

问题 4 解答 当限定用户输入数据为以下格式：

```
1234↙
```

同时要求程序输出结果为 a = 12, b = 34 时，应将程序第 5 行语句修改为

```
scanf("%2d%2d", &a, &b);
```

这样在输入数据时，可以自动按照指定宽度从输入的数据中截取所需数据。

问题 5 解答 当限定用户输入数据为以下格式：

```
12↙
34↙
```

同时要求程序输出结果为 a = "12", b = "34"时，应修改程序为：

```
1  #include  <stdio.h>
2  int main(void)
3  {
4       int  a, b;
5       scanf("%d%d", &a, &b);
6       printf("a = \"%d\", b = \"%d\"\n", a, b);
7       return 0;
8  }
```

这里，函数 printf()格式控制字符串中的字符'\" '是转义序列，代表双引号字符。这是因为，双引号已经被用作表示字符串的定界符，这样就导致双引号不能被打印出来，所以为了打印双引号，就需要将其放到转义字符的后面构成转义序列，以这种方式来打印双引号。

问题 6 解答 在要求以任意字符作为分隔符进行数据输入时，使用忽略输入修饰符可给用户带来方便。

```
#include  <stdio.h>
int main(void)
{
     int  a, b;
     scanf("%d%*c%d", &a, &b);
     printf("a = %d, b = %d\n", a, b);
     return 0;
}
```

请读者上机分别按如下几种数据输入格式进行数据的输入，并观察屏幕显示结果，检验程序能否都打印出 a = 12, b = 34 的结果。

格式 1　以回车符作为数据分隔符：

```
12↙
34↙
```

格式 2　以空格符作为数据分隔符：

```
12 34↙
```

格式 3　以逗号作为数据分隔符：

```
12,34↙
```

格式 4　以制表符作为数据分隔符：

```
12      34↙
```

格式 5　以字符-作为数据分隔符：

```
12-34↙
```

2. %c 格式符在应用中存在的问题及其解决方法

【例 3.10】　按下述格式，从键盘输入一个整数加法表达式：

```
操作数1  +  操作数2
```

然后计算并输出表达式的计算结果，形式如下：

```
操作数1  +  操作数2  = 计算结果
```

编写程序如下：

```
#include  <stdio.h>
int main(void)
{
     int    data1, data2;
     char  op;
     printf("Please enter the expression data1 + data2\n");
     scanf("%d%c%d",&data1, &op, &data2);
     printf("%d%c%d = %d\n", data1, op, data2, data1+data2);
     return 0;
}
```

从键盘先后输入 12、空格、+、空格和 3 后的程序运行结果如下：

```
Please enter the expression data1 + data2
12 + 3↙
12 3129 = 3141
```

这个程序的运行结果显然不对，这是为什么呢？

发生这种错误的原因显然是因为数据没有被正确读入，先来看我们是如何输入数据的吧。当程序提示输入数据时，首先输入一个整型数 12，紧接着输入一个空格字符，然后输入字符'+'，紧接着再输入一个空格字符，最后输入整型数 3。当输入数据 12 时，12 被函数 scanf()用 d 格式符正确地读给变量 data1。然而，其后输入的空格字符却被函数 scanf()用 c 格式符读给了变量 op，当然，data2 也因此无法得到整型值 3。我们只要做个简单的测试，即重新修改输入格式，就可以验证我们的上述分析结果。下面是程序两次运行的结果：

第 1 次测试（先后输入 12、空格和 3）的结果：

```
Please enter the expression data1 + data2
12 3↙
12 3 = 15
```

第 2 次测试（先后输入 12、+和 3）的结果：

```
Please enter the expression data1 + data2
12+3↙
12+3 = 15
```

在第 1 次测试中，输入的 12、空格符、3 分别被读给整型变量 data1、字符型变量 op、整型变量 data2。而在第 2 次测试中，输入的 12、字符'+'、3 分别被读给整型变量 data1、字符型变量 op、整型变量 data2。这说明在用%c 格式符读入字符时，空格字符和转义序列（包括回车和制表符）都会作为有效字符输入，这是使用函数 scanf()时特别需要注意的一点。

【例 3.11】 编程从键盘先后输入整型、字符型和实型数据，要求每输入一个数据就显示一个数据的类型和数据值。

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int   a;
5      char  b;
6      float  c;
7      printf("Please input an integer:");
8      scanf("%d", &a);
9      printf("integer: %d\n", a);
10     printf("Please input a character:");
11     scanf("%c", &b);
12     printf("character: %c\n", b);
13     printf("Please input a float number:");
14     scanf("%f", &c);
15     printf("float: %f\n", c);
16     return 0;
17 }
```

程序的运行结果如下：

```
Please input an integer:12↙
integer: 12
Please input a character: character:

Please input a float number:3.5↙
```

```
float: 3.500000
```

显然，这个程序和例 3.10 一样，问题也是出在%c 格式符上面，输入 12 以后按的回车键被当作有效字符读给字符型变量 b 了。解决这个问题有如下两种方法。

方法 1　用函数 getchar()将前面数据输入时存于缓冲区中的回车符读入，避免被后面的字符型变量作为有效字符读入。

```
#include  <stdio.h>
int main(void)
{
     char  a, b, c;
     printf("Please input c:");
     scanf("%c", &c);
     getchar();//将存于缓冲区中的回车符读入，避免被后面的变量作为有效字符读入
     printf("Please input b:");
     scanf("%c", &b);
     getchar();//将存于缓冲区中的回车符读入，避免被后面的变量作为有效字符读入
     printf("Please input a:");
     scanf("%c", &a);
     getchar();//将存于缓冲区中的回车符读入，避免被后面的变量作为有效字符读入
     printf("c=%c,b=%c,a=%c\n", c, b, a);
     printf("ASCII: c:%d,b:%d,a:%d\n", c, b, a);
     return 0;
}
```

方法 2　在%c 前面加一个空格，将前面数据输入时存于缓冲区中的回车符读入，避免被后面的字符型变量作为有效字符读入。就程序可读性而言，这个方法更好一些。

```
#include  <stdio.h>
int main(void)
{
     char  a, b, c;
     printf("Please input c:");
     scanf("%c", &c);
     printf("Please input b:");
     scanf(" %c", &b);       //在%c 前面加一个空格，将存于缓冲区中的回车符读入
     printf("Please input a:");
     scanf(" %c", &a);       //在%c 前面加一个空格，将存于缓冲区中的回车符读入
     printf("c=%c,b=%c,a=%c\n", c, b, a);
     printf("ASCII: c:%d,b:%d,a:%d\n", c, b, a);
     return 0;
}
```

程序的运行结果如下：

```
Please input c: a↙
Please input b: b↙
Please input a: c↙
```

```
c=a, b=b, a=c
ASCII: c:97, b:98, a:99
```

按上述方法修改例 3.10 的程序，即在%c 前面加一个空格，那么无论以如下哪种方式输入加法算式，都能得到正确的结果：

```
12  +  3↙
```

或者

```
12↙
+↙
3↙
```

3. 对输入非数字字符的检查与错误处理

由于 scanf()不进行参数类型匹配检查，因此，当输入数据类型与格式字符不匹配时，编译器不提示出错信息，但会导致程序不能正确读入数据。即使输入数据类型与格式字符相符，也无法保证用户输入的数据都是合法的数据，一旦输入非法数据，也会导致数据不能正确读入。来看下面的程序。

【例 3.12】 输入两个整型数，并输出。

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a, b;
5      printf("Input a:");
6      scanf("%d", &a);
7      printf("a = %d\n", a);
8      printf("Input b:");
9      scanf("%d", &b);
10     printf("b = %d\n", b);
11     return 0;
12 }
```

第 1 次测试的运行结果如下：

```
Input a:1.2↙
a = 1
Input b:b = 3129
```

第 2 次测试的运行结果如下：

```
Input a:q↙
a = 64
Input b:b = 3129
```

在第 1 次测试时，用户输入的是 1.2，但是第 1 个 scanf()函数调用语句只读入了整数 1，后面的圆点被视为非数字字符导致输入结束。由于这个非数字字符仍然保存在输入缓冲区中，因此第 2 个 scanf()函数调用语句从输入缓冲区中读到的数据仍然是这个非数字字符，所以没等用户输入数据，就打印出了变量 b 中的随机值，和没有执行第 9 行语句的效果是一样的。

在第 2 次测试时，由于用户输入的是非数字字符'q'，而且它一直保存在输入缓冲区中，因此第 6 行和第 9 行的两个 scanf()都试图读取这个数据，然而都没有正确读入数据，相当于变量 a 和 b 没有被赋值，其值是随机不确定的，故打印结果都是随机值。

因为我们无法对用户的实际输入进行控制，所以我们无法保证不会出现各种奇葩的输

入。怎样解决这个问题呢？可以考虑用检验函数 scanf()调用返回值的方法。虽然前面在使用函数 scanf()时，我们并没有使用它的返回值，但事实上，函数 scanf()也是有返回值的。

如果函数 scanf()调用成功（能正常读入输入数据），则其返回值为已成功读入的数据项数。通常非数字字符的输入会导致数据不能成功读入，例如，要求输入的数据是数值型数据，而用户输入的是字符，字符相对于数值型数据而言就是非数字字符，但是反之不然，因为数值型数据可被当作有效字符读入。

如果函数 scanf()调用失败，则返回 EOF（如前所述，EOF 是一个在头文件<stdio.h>中定义的整数型的符号常量，通常被定义为-1，但并不一定是-1，在不同的系统中可能取不同的值）。通常在无数据可读时才会发生这种情况，例如，当标准输入被重定向到一个输入文件时，程序执行函数 scanf()调用就是从该文件中读入数据，当读到文件尾没有数据可读时，函数 scanf()调用就会失败。如果用户按 F6 键强制输入结束，此时测试函数 scanf()的返回值也是 EOF。在 C 语言中，0 和-1 是最常用到的函数调用失败后的返回值。注意，函数 scanf()的返回值是在遇到非数字字符之前已成功读入的数据项数，不一定为-1，也不一定为 0。因此，不能靠检查函数 scanf()的返回值是否为-1 或 0 来判断是否所有数据都已正确读入，应该检查函数 scanf()的返回值是否为应该读入的数据项数。

考虑到以上 scanf()函数返回值的特点，我们可以这样来解决这个问题，即判断函数 scanf()的返回值是否为应该读入的数据项数。如果不是，则清除输入缓冲区中的内容，然后提示用户重新输入数据直到输入正确为止。由于后面这个错误处理操作要用到循环语句，循环语句将在第 4 章中介绍，因此在下面的程序中，我们做了简单处理，只给出了输入错误的提示信息。

```
#include <stdio.h>
int main(void)
{
    int a, b, ret;
    printf("Input a and b:");
    ret = scanf("%d%d", &a, &b);
    if (ret != 2) //包括各种输入错误，如格式错误，输入非数字字符，无数据可读等
    {
       printf("Input error!\n");
    }
    else            //此处可以是正确读入数据后应该执行的操作
    {
       printf("a = %d, b = %d\n", a, b);
    }
    fflush(stdin);//清除输入缓冲区中的错误数据
    return 0;
}
```

在 Visual C++ 6.0 环境下，第 1 次测试的运行结果如下：

```
Input a and b:1.2  3↙
Input error!
```

第 2 次测试的运行结果如下：

```
Input a and b:3  1.2↙
```

```
a = 3, b = 1
```

在第 1 次程序测试时，即输入 1.2 和 3 时，由于程序第 6 行语句要求输入整型数据，因此 1.2 后面的小数点被作为非数字字符看待，使得程序只读入了 1 个整型数 1，第 2 个整型数未能读取成功，此时函数 scanf()的返回值为 1，因此执行第 9 行语句显示输入错误提示信息。在第 2 次程序测试时，由于先输入 3，后输入 1.2，因此，程序将 3 读入赋值给 a，读入 1 赋值给 b，尽管 1.2 后面的小数点仍被作为非数字字符看待，但是它只起到了结束输入的作用，此时函数 scanf()的返回值为 2，所以，程序执行第 13 行语句，显示变量 a 和 b 的值。

那么第 15 行语句的作用是什么呢？为了确保保留在输入缓冲区中的非数字字符不会影响其后的数据输入，程序第 15 行调用函数 fflush()来清除输入缓冲区中的内容。

【编程提示】由于 ANSI C 只规定函数 fflush()处理输出数据流、确保输出缓冲区中的内容写入文件，并未对清理输入缓冲区做出任何规定，只是部分编译器（如 Visual C++）增加了此项功能，因此使用函数 fflush()来清除输入缓冲区中的内容，可能会带来移植性问题。

从这个例子我们还可以看出，由于函数 scanf()不做参数类型匹配检查，因此通过检验函数 scanf()返回值的方法试图发现某些输入数据类型不匹配错误仍然是无效的。例如，本例中要求输入的数据是整型，而输入的数据是实型，那么实数的小数点前面的数字被当作整型数据读入，而后面的小数点则被当作非数字字符处理。

3.5　本章小结

本章介绍了 C 语言的表达式语句的特点及常用标准输入/输出函数的使用方法，函数 getchar()和 putchar()用于字符输入/输出操作，而函数 scanf()和 printf()则用于格式输入/输出操作，可以控制按各种格式进行任意类型数据的读/写操作。最后，在 3.4.3 节中着重介绍了 scanf()函数在使用中容易出现的问题及其解决方法。

本章常见的编程错误如表 3-5 所示。

表 3-5　本章常见编程错误列表

错误描述	错误类型
函数名拼写错误。例如，将 printf()误写为 print()或 Printf()，因为 C 语言编译器只在目标程序中为库函数调用留出空间，并不能识别函数名中的拼写错误，更不知道库函数在何处，寻找库函数并将其插入到目标程序中是链接程序负责的工作，因此这种错误仅在链接时才能被发现	链接错误
将变量定义语句放在可执行语句后面	编译错误
忘记给函数 printf()或 scanf()中的格式控制字符串加上双引号	编译错误
将格式控制字符串和表达式之间的逗号写到了格式控制字符串内	编译错误
忘记给函数 scanf()中的变量加上取地址运算符&	运行时错误
函数 printf()欲输出一个表达式的值，但格式控制字符串中没有与其对应的格式字符	运行时错误
函数 printf()中欲输出一个表达式的值，但输出列表中没有与格式字符相对应的表达式	运行时错误
scanf()或 printf()的格式控制字符串中的格式字符与要输入/输出的数值类型不一致	运行时错误
用户从键盘输入的数据格式与函数 scanf()中格式控制字符串要求的格式不一致。例如，相邻数据项之间应该用逗号分隔，但是用户没有输入逗号，或者不应该用逗号分隔但是用户输入了逗号	运行时错误
函数 scanf()格式控制字符串中含有'\n'等转义序列，导致数据输入不能按正常方式终止	运行时错误
用函数 scanf()输入实型数据时，在格式控制字符串中规定了要输入的实型数据的精度	运行时错误