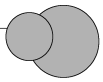
第3章

流程控制



语句(statement)是C++程序的最小单位。程序由一条一条语句组成,程序运行过程就是语句逐条执行的过程,而语句执行的次序则称为流程。有了求解问题的算法,还要用程序实现出来。多数情况下,这种实现表现为一定数量的语句和执行流程。

C++语句分为简单语句、复合语句和控制语句,具有顺序结构、选择结构和循环结构3种基本控制结构。

3.1 语句

3.1.1 简单语句

简单语句包括表达式语句、函数调用语句、空语句和声明语句。

1. 表达式语句

在任何表达式后面加上一个分号(;)就构成了一个表达式语句(expression statement),语句形式为

```
表达式;
```

例如:

```
x=a+b;                          //赋值语句
```

为赋值表达式语句,执行加法和赋值运算。又如:

```
t=a,a=b,b=t;                    //a和b交换
```

为逗号表达式语句,组合了多个赋值运算。

表达式语句用于计算表达式,但下面的语句:

```
a+b+c;                          //能运算但无实际意义
```

却没有任何意义,因为3个变量加起来的结果没有用于赋值或其他用途,计算结果被舍弃了。一般地,表达式语句所包含的表达式应该在计算时对程序的状态或数据有影响,例如赋值、自增自减、输入或输出等操作。

2. 函数调用语句

函数调用语句是由函数调用加分号(;)形成的,语句形式为

```
函数调用(实参);
```

例如:

```
cout<<"a+b="<<a+b<<endl;        //输出流函数调用语句
```

表达式语句和函数调用语句是程序中用得最多的语句,因为程序多数情况下表现为计算和功能执行。

3. 空语句

仅有一个分号就形成了空语句(null statement),它什么也不做,语句形式为

```
;                               //空语句
```

空语句往往用在语法上要求必须有一个语句,而逻辑上不需要做什么的地方,例如循环语句中的循环体。下面语句的功能是从键盘连续输入多个字符直到输入回车符。

```
while(cin.get()!='\n');         //使用空语句
```

这里的循环体就是空语句,因为 while 语句要求必须有循环体,而 cin.get()!='\n'已经完成了语句功能,此时的循环体不需要做什么。

由于空语句是一个语句,因此可用在任何允许使用语句的地方。而意外出现的多余分号(即空语句)不是语法错误,不会由编译器报告出来,因而有时容易让程序员忽视而产生难以消除的程序漏洞(bug)。

4. 声明语句

在 C++ 中对象的定义或类型的声明也是语句,称为声明语句(declaration statements),它可以放在程序中任何允许语句出现的地方,例如:

```
1    int a, b;
2    a=10, b=20;
3    int t;                     //正确,声明语句只要放在使用之前即可
4    t=a, a=b, b=t;
```

第 3 行的变量定义放在了第 2 行程序语句的后面,这种做法在 C 语言中是不合法的,而 C++ 允许这样做,增强了对象定义或类型声明的灵活性,可以很方便地实现定义或声明的局部性,即作用范围从声明语句开始到函数或块的结束。

一般地,声明语句最好放在函数或语句块开头位置。

3.1.2 复合语句

将多个语句组成的语句序列用一对大括号({})括起来组成的语句称为复合语句

(compound statement)，又称语句块，简称块(block)。语句形式为

```
{
    [局部声明部分;]
    语句序列;
}
```

其中，“语句序列”表示任意数目的语句，方括号内的局部声明部分是可选的。例如：

```
{                                          //复合语句
    double s, a=5, b=10, h=8;              //局部声明
    s=(a+b)*h/2.0;
    cout<<"area="<<s<<endl;
}                                          //复合语句不需要分号结尾
```

复合语句内的每条语句必须以分号(;)结尾，但复合语句的右大括号(})已表示结尾，因此其后不需要分号。如果在后面添加分号，意思就变为一个复合语句与一个空语句。

复合语句内部可以进行变量定义或类型声明，这些定义或声明仅在复合语句内部可以使用，称为块的局部作用域，将在第 4 章详细讨论作用域。例如：

```
{
    int t,a=10,b=7;                        //定义局部变量 t、a、b
    t=a,a=b,b=t;                           //仅在这个复合语句中使用
}
```

复合语句允许嵌套，即在复合语句中还可以包含复合语句。例如：

```
{                                          //复合语句
    double v1,r=5;                         //局部声明
    v1=4*3.1415926*r*r*r/3;
    {                                      //嵌套的复合语句
        double v2,h=12;                    //嵌套的局部声明
        v2=3.1415926*r*r*h;
        cout<<v1<<","<<v2<<endl;
    }                                      //嵌套的复合语句结尾
}                                          //复合语句结尾
```

使用复合语句嵌套，程序有了更大能力应付复杂的流程处理。

如果复合语句中没有任何内容，如{}，称为空复合语句，空复合语句与空语句等价，它为空语句提供了一种替代语法。

尽管复合语句内部有许多语句，但从语法角度来看，复合语句是一个语句的意思，因此凡是简单语句能出现的地方都可以使用复合语句。使用复合语句的目的是描述长而复杂的语句序列，利于将复杂的语句形式简单化和结构化。

3.1.3　注释

可以在程序中编写注释(comments)，有两种形式。

/*……*/块注释语法形式：

```
/*
注释内容
*/
```

//行注释语法形式：

```
//注释内容
```

说明：

(1) 注释仅是对源程序的说明文字，它不是程序代码，对程序运行没有任何影响。实际上，在编译程序时所有注释内容将被忽略。

(2) /*……*/块注释允许多行注释，以/*开头，以*/结尾，这中间的任何内容均是注释内容。注释可以是任何来自字符集的字符组合，包括换行符，也允许中文等非ASCII字符。/*……*/不允许嵌套。例如：

```
/* 第1个注释
    ……
    /* 第2个注释
        ……
    */
*/
```

是错误的，因为编译器将第2个注释的*/当作第1个注释的结尾，从而使得后续部分出现编译错误。显而易见，编译器一旦遇见/*开头，就表示注释开始，在遇到*/注释时才会结束。因此只给出/*而没有*/也会产生编译错误。

(3) //行注释是C++标准允许的另一种注释方法，//注释表示从//开始直到本行末尾的所有字符均是注释内容。例如：

```
s=3.1415926*r*r*h/3;                    //计算圆锥体积
```

显而易见，编译器一旦遇见//开头，就表示注释开始，直至本行末尾。

//注释只能注释一行，如果要注释多行就要写多次。一般//注释适用于短小精简的注释，/*……*/注释适用于大段注释。

(4) 编译器将整个注释理解为一个空白字符，相当于一个空格的作用。在编译阶段，所有的注释均被忽略，所以执行程序不包含注释内容；换言之，注释对于程序的执行是没有任何效用的。例如：

```
1    int/*这里有注释*/t, a, b;
2    //t=a, a=b, b=t;
```

第1行在int和t之间的注释起到了空格的作用，第2行实际上没有程序代码。

尽管注释对程序运行没有作用，但编程时还是提倡在适当的地方写注释。这是因为注释出现在程序的源文件中，可以对源程序作出说明，从而增加程序的可读性。而且，从上述程序段第2行中，还可以学到将一段程序代码临时“屏蔽”起来，即让某段程序“暂时

失效”的调试技巧。

注释内容应该是那些能够确切描述程序代码功能、目的、接口、概括算法、确认数据对象含义以及阐明难以理解的代码段的说明性文字，编程时要养成习惯添加注释，但注释也不是越多越好。一般来说，处理复杂的程序注释多些，简单程序甚至可以不写注释，总之从方便程序的阅读和增强对程序的理解出发。

3.1.4　语句的写法

在 C++ 中，对于语句的写法有以下规定或惯例。

(1) 多数情况下，在一个程序行里只写一个语句，这样的程序写法清晰，便于阅读、理解和调试。

(2) 注意使用空格或 Tab 作合理的间隔、缩进或对齐，使程序形成逻辑相关的块状结构，养成优美的程序编写风格。

(3) C++ 允许在一行里写多个语句。例如：

```
a=i/100; b=i/10%10; c=i%10;                    //3 个语句
```

由于行是多数编译器在编译或调试时的基本单位，即使编译器指明了某一行有错误也不能明确判明是哪个语句出错，所以在一行里写多个语句的风格并不好。

(4) C++ 允许将一个语句拆成多行来写。例如：

```
cout<<"a="<<a<<",b="<<b<<",c="<<c<<",d="<<d<<",e="<<e<<",f="
    <<f<<",g="<<g<<",h="<<h<<endl;
```

计算机屏幕宽度有限，过长的语句拆成多行来写是可能的。但需要注意两点：一是 C++ 规定回车换行也是空白符，所以不能在关键字、标识符等中间拆分，否则人为间隔了这些词语，会产生编译错误；二是在 C++ 中字符串常量是不能从中间拆分的，因为编译器会认为字符串没有正确结束。例如：

```
1    cout<<"This is a very long
2    string of examples";
```

第 1 行会产生编译错误。

解决字符串常量拆分的办法是使用反斜杠(\)行连接符，行连接符的作用是用程序的下一行(从第一列开始)替换当前的行连接符。例如：

```
1    "one \
2    two \
3    three"
```

第 2 行会替换第 1 行的行连接符\，第 3 行会替换第 2 行的行连接符\，从而第 1、2、3 行实质上合并为一行，故上述写法与下面的写法等价：

```
"one two three"
```

请注意，如果//注释后面不幸有一个行连接符，那么下一行也依然是注释。例如：

```
1    int t, a=10, b=7;                    //本行的注释\
2    t=a, a=b, b=t;
```

与下面等价：

```
int t, a=10, b=7;                         //本行的注释 t=a, a=b, b=t;
```

3.2 输入与输出

所谓输入是指从外部输入设备(如键盘、鼠标等)向计算机输入数据,输出是指从计算机向外部输出设备(如显示器、打印机等)输出数据。

C++ 的输入输出操作是用流(stream)对象实现的,以标准的终端设备(键盘和显示器)为输入输出设备。cout 是输出流对象,cin 是输入流对象。通过 cout 的<<流插入运算符将需要输出的内容插入到输出流中(即显示出来),通过 cin 的>>流提取运算符从输入设备(键盘)的输入流中读取内容到指定的对象。

程序中使用流对象 cin、cout 和流运算符时,应该用文件包含预处理命令将标准输入输出流库的头文件<iostream>包含到源文件中,即使用输入输出流对象的程序一般在文件开头应有以下命令：

```
#include <iostream>                       //C++标准输入输出流类库
using namespace std;                      //C++标准输入输出流类库要求使用标准命名空间
```

需要注意的是,C++ 标准库头文件一般是不加文件扩展名(.h)的。

C++ 的输入输出操作本质上是类成员函数调用,通常在不引起概念混淆的情况下,将由 cin 和流提取运算符(>>)实现输入功能的函数调用语句称为输入语句,将由 cout 和流插入运算符(<<)实现输出功能的函数调用语句称为输出语句。

C++ 还兼容 C 语言的输入输出操作,由标准库函数来实现。不同的库函数能够处理形式多样的输入输出操作,支持不同的输入输出设备。C 语言标准中定义了“标准输入输出函数”,以标准的终端设备(键盘和显示器)为输入输出设备,可以使用字符输出 putchar、字符输入 getchar、格式输出 printf 和格式输入 scanf 等函数。

程序中使用标准输入输出函数时,应该用文件包含预处理命令将 C 语言头文件<stdio.h>包含到源文件中,即调用输入输出库函数的程序,一般文件开头应有以下命令：

```
#include <stdio.h>                        //C++兼容 C 语言,可以不用标准命名空间
```

或

```
#include <cstdio>                         //C++新方法
using namespace std;                      //C++新方法要求使用标准命名空间
```

* 3.2.1　输入流与输出流

1. cout 和 cin 对象的使用

使用 cout 输出数据的一般形式为

```
cout <<表达式 1 <<表达式 2 <<…;
```

表达式 1、表达式 2……依次按先后顺序插入输出缓冲区中，然后刷新到标准输出设备（显示器）上。

使用 cin 输入数据的一般形式为

```
cin >>变量 1 >>变量 2 >>…;
```

从标准输入设备（键盘）上输入的数据依次按先后顺序提取到变量 1、变量 2……而且只能提取到变量中，不能提取到常量或表达式。

例如：

```
int x, y;
cin>>x>>y;                                    //键盘输入
cout<<"x="<<x<<",y="<<y<<endl;                //输出到显示器上
```

运行时先等待键盘输入

```
12 34↙
```

再输出

```
x=12,y=34
```

cout 输出时，并不是插入一个数据就马上输出，而是把插入的数据顺序存放到一个输出缓冲区中，直到输出缓冲区满或遇到刷新（endl、'\n'、ends 或 flush）为止。此时将缓冲区中已有的数据一起输出，并清空缓冲区。

cin 输入时，缓冲区不为空时，直接从缓冲区中提取数据；否则等待键盘输入。键盘输入必须用 Enter 键结束输入，如图 3.1 所示。

图 3.1　输入输出工作原理

cin 输入时,为了分隔多项数据,默认要求在两个键盘输入数据之间使用空白符。空白符有 3 个:空格、Tab 键、Enter 键,可以任选其一。例如:

```
cin>>a>>b>>c>>d;
```

由键盘输入

```
12␣34  →  56↙
78↙
```

其中,␣表示空格,→表示 Tab 键,↙表示 Enter 键。输入后 a=12、b=34、c=56、d=78。

在输入输出数据时,为简便起见,往往不指定其数据格式,而由系统根据数据的类型采取默认格式。如表 3.1 所示为 cin 所能识别的输入数据类型及默认的输入格式,表 3.2 为 cout 所能识别的输出数据类型及默认的输出格式。

表 3.1　cin 所能识别的输入数据类型及默认的输入格式

输入数据类型	输 入 格 式	输入数据类型	输 入 格 式
bool	1 或 true,0 或 false	short、int、long(signed、unsigned)	整型常量
char(signed、unsigned)	第 1 个非空白字符	float、double、long double	浮点数常量
void * 地址值	十六进制数	char *(signed、unsigned)	空白符之前所有字符

表 3.2　cout 所能识别的输出数据类型及默认的输出格式

输出数据类型	输 出 格 式	输出数据类型	输 出 格 式
bool	1 或 0	short、int、long(signed、unsigned)	整数
char(signed、unsigned)	单个 ASCII 字符	float、double、long double	浮点或指数格式
void * 地址值	十六进制数	char *(signed、unsigned)	字符串

2. 格式控制

有时希望数据按指定的格式输入输出,如要求以十六进制或八进制形式输入一个整数,对输出的小数只保留两位小数,输出的多行数据能够对齐等。有两种方法可以达到这样的要求。

(1) 可以在输入输出流中使用控制符进行格式控制。表 3.3 为常用格式控制符,其中<iomanip>表示若使用了该控制符,除<iostream>外,程序还需要包含头文件<iomanip>,即输入输出操纵器(manipulator)库。

使用格式控制符的形式如下:

```
cout <<…<<格式控制符 <<表达式 <<…;
cin >>…>>格式控制符 >>变量 >>…;
```

格式控制符仅设置其后的输入输出的格式,对它前面的输入输出没有影响。

表 3.3　输入输出格式控制符

格式控制符	含　义
endl	插入换行符且刷新缓冲区输出到设备上
ends	插入空字符('\0')
flush	所有在缓冲区的字符立即输出到设备上
boolalpha	逻辑型用文字值 true 或 false 输入输出
noboolalpha	逻辑型用整数值 1 或 0 输入输出
dec	用十进制形式输入输出整数
oct	用八进制数形式输入输出整数
hex	用十六进制形式输入输出整数
setbase(int base)	设置整数进制基数，base 只能是 8、10 或 16 之一
left	在设定的域宽内左对齐输出，右端用填充字符填满
right	在设定的域宽内右对齐输出，左端用填充字符填满
internal	在设定的域宽内右对齐输出，但若有符号(－或＋)，符号置于最左端
setfill(char c)	＜iomanip＞。设置填充字符为 c 所指定的字符
setw(int n)	＜iomanip＞。设置域宽为 n。输入域宽只对字符串有效，输出时指最小输出宽度。当实际数据宽度小于域宽时，多余的位置用填充字符填满；当实际数据宽度大于域宽，按实际的输出。初始域宽为 0，表示所有数据按实际输出。域宽设置只对一次输入输出有效，它是所有格式中唯一的一次有效的设置。在完成一次输入输出后，域宽自动恢复为 0
fixed	定点形式输出浮点数，未设置时用浮点形式输出浮点数
scientific	指数形式输出浮点数
setprecision(int n)	＜iomanip＞。设置精度为 n。对于默认浮点形式精度决定输出的有效位个数，默认为 6 位，小数点的相对位置随数据的不同而浮动；对于定点或指数形式精度决定输出的小数位数，小数点的相对位置固定不变，必要时进行舍入处理或添加 0
showbase	输出进制整数的前缀，十六进制为 0x，八进制为 0
noshowbase	清除 showbase 的设置
showpoint	强制输出小数点，未设置时只在非整数情况下才输出小数点
noshowpoint	清除 showpoint 的设置
showpos	强制输出数的正负符号，未设置时只在负数情况下才输出符号
noshowpos	清除 showpos 的设置
uppercase	若数据是十六进制或有 0x 前缀时，设置数据中的字母为大写。未设置时为小写
nouppercase	清除 uppercase 的设置
skipws	输入时读取连续多个空白符并丢弃，即跳过空白符，空白符包含空格、Tab 和回车。未设置时空白符当作数据内容的一部分

续表

格式控制符	含　义
noskipws	清除 skipws 的设置
unitbuf	设置每次输出插入操作立即刷新缓冲区内容,未设置时只有在缓冲区满或 flush、endl、ends 时才刷新缓冲区内容
nounitbuf	清除 unitbuf 的设置
setiosflags(mask)	<iomanip>。设置 ios 标志值,mask 为 ios_base::fmtflags 类型,格式标志值如下。 ① 逻辑:ios_base::boolalpha ② 进制:ios_base::dec、ios_base::oct、ios_base::hex、ios_base::basefield ③ 浮点:ios_base::fixed、ios_base::scientific、ios_base::floatfield ④ 对齐:ios_base::left、ios_base::right、ios_base::internal、ios_base::adjustfield ⑤ 显示:ios_base::showbase、ios_base::showpoint、ios_base::showpos、ios_base::uppercase ⑥ 其他:ios_base::skipws、ios_base::unitbuf 可以使用位或运算(|)一次设置多个标志值
resetiosflags(mask)	<iomanip>。清除 ios 标志值

(2) 可以调用输入输出流对象的成员函数进行格式控制。表 3.4 为格式控制的流对象成员函数。

表 3.4　流对象成员函数

函　数	含　义
precision(int n);	等价于 setprecision(int n)
width(int n);	等价于 setw(int n)
setf(mask);	等价于 setiosflags(mask)
unsetf(mask);	等价于 resetiosflags(mask)
fill(char c);	等价于 setfill(char c)

get();
get(char& c);
get(char * s,streamsize n);
 	
get(char * s,streamsize n,char delim);
get(streambuf& sb);
get(streambuf& sb,char delim);
	提取一个字符,函数返回其值
	提取一个字符给 c
	提取 n−1 个字符(遇到 Enter 键结束)到字符数组 s 或字符指针 s 指向区域
	同上,遇到分隔字符 delim 时结束
	提取 n−1 个字符(遇到 Enter 键结束)到字符串流缓冲对象
	同上,遇到分隔字符 delim 时结束

getline(char * s,streamsize n);
 	
getline(char * s,streamsize n,char delim);
	提取 n−1 个字符(遇到 Enter 键结束)到字符数组 s 或字符指针 s 指向区域
	同上,遇到分隔字符 delim 时结束

put(char c);	输出一个字符 c
write(const char * s,streamsize n);	输出 n 个字符数组 s 或字符指针 s 指向区域的字符
flush();	所有在缓冲区的字符立即输出到设备上

【例 3.1】 使用 cin 和 cout 输入输出数据。

程序代码如下：

```
#include <iostream>                          //标准输入输出流类库
#include <iomanip>                           //输入输出操纵器库
using namespace std;                         //标准输入输出需要 std
int main()
{
    bool v; int a, m, n;
    double x,y,z, p, f; float f1;
    cin>>boolalpha>>v;                       //输入文字值逻辑数据
    cin>>oct>>a>>hex>>m>>dec>>n;             //输入进制整数数据
    cin>>p>>f>>f1>>x>>y>>z;                  //输入小数、指数浮点数据
    cout<<v<<'\t'<<boolalpha<<v<<'\t'<<noboolalpha<<v<<endl;    //逻辑型输出
    cout<<a<<'\t'<<p<<endl<<a*p<<endl;                          //整型输出
    cout<<hex<<m<<'\t'<<oct<<m<<'\t'<<dec<<m<<endl;             //进制整型输出
    cout<<showbase<<hex<<m<<'\t'<<oct<<m<<'\t'<<dec<<m<<endl;   //进制前缀
    cout.precision(5); cout<<x<<'\t'<<y<<'\t'<<z<<endl;         //浮点型输出
    cout<<fixed<<x<<'\t'<<y<<'\t'<<z<<endl;                     //定点形式
    cout<<scientific<<x<<'\t'<<y<<'\t'<<z<<endl;                //指数形式
    cout<<left<<setw(6)<<n<<'\t'<<internal<<setw(6)<<n<<'\t';   //对齐
    cout.width(6);cout<<right<<n<<setfill('0')<<endl;           //对齐
    cout<<setw(10)<<77<<'\t'<<setfill('x')<<setw(10)<<77<<endl; //填充
    cout<<setprecision(5)<<f1<<'\t'<<setprecision(9)<<f1<<endl; //精度
    cout<<fixed<<setprecision(5)<<f<<'\t'<<setprecision(9)<<f<<endl;
    cout<<scientific<<setprecision(5)<<f<<'\t'<<setprecision(9)<<f<<endl;
    cout<<setiosflags(ios_base::floatfield|ios_base::showpoint); //标志
    cout<<setprecision(0)<<f<<'\t'<<setprecision(9)<<f<<endl;   //小数位数
    cout<<showpos<<1<<'\t'<<0<<'\t'<<-1<<endl;                  //符号
    cout<<noshowpos<<1<<'\t'<<0<<'\t'<<-1<<endl;
    return 0;
}
```

程序运行情况如下：

```
true↙
144 46 -77↙
3.14 3.14159 3.14159↙
3.1415926534 2006.0 1.0e-10↙
1               true            1
100             3.14
314
46              106             70
0x46            0106            70
3.1416          2006            1e-010
```

```
3.14159          2006.00000          0.00000
3.14159e+000     2.00600e+003        1.00000e-010
-77     -        77                  -77
0000000077       xxxxxxxx77
3.14159e+000     3.141590118e+000
3.14159          3.141590000
3.14159e+000     3.141590000e+000
3.               3.14159000
+1               +0                  -1
1                0                   -1
```

下面集中给出使用 cout 格式化输出数据的例子。

```
//①输出逻辑型数据(默认格式、文字值格式)
cout<<v1<<","<<v2<<","<<boolalpha<<v1<<","<<v2<<","<<noboolalpha<<v2<<
endl;
//输出结果: 1,0,true,false,0
//②输出整型数据
//十进制、十六进制和八进制
cout<<a<<","<<hex<<a<<","<<oct<<a<<dec<<endl;
//输出结果: 123,7b,173
//十进制、十六进制和八进制,负数为补码
cout<<b<<","<<hex<<b<<","<<oct<<b<<dec<<endl;
//输出结果: -1,ffffffff,37777777777
//十六进制大小写数字
cout<<hex<<a<<","<<b<<uppercase<<","<<a<<","<<b<<nouppercase<<endl;
//输出结果: 7b,ffffffff,7B,FFFFFFFF
//填充十六进制、八进制前缀
cout<<showbase<<hex<<a<<","<<oct<<a<<noshowbase<<dec<<endl;
//输出结果: 0x7b,0173
//无符号十进制
cout<<u1<<","<<u2<<endl;
//输出结果: 1,4294967295
//短整型,负数为补码
cout<<i<<","<<hex<<i<<","<<oct<<i<<dec<<endl;
//输出结果: -1,ffff,177777
//③输出带格式的整型数据
//域宽、右对齐、左对齐、实际宽度
cout<<"["<<a<<"],["<<setw(4)<<a<<"],["<<setw(4)<<left<<a<<"]"<<right<<
endl;
//输出结果: [123],[ 123],[123 ]
cout<<"["<<setw(4)<<c<<"],["<<setw(4)<<left<<c<<"]"<<right<<endl;
//输出结果: [12345],[12345]
//显示正负符号
cout<<showpos<<"["<<a<<"],["<<-a<<"]"<<noshowpos<<endl;
```

```
//输出结果: [+123],[-123]
//填充空格
cout<<setfill(' ')<<"["<<setw(5)<<a<<"],["<<setw(5)<<-a<<"]"<<endl;
//输出结果: [  123],[ -123]
//填充 0
cout<<setfill('0')<<"["<<setw(5)<<a<<"],["<<internal<<setw(5)<<-a<<"]"<<endl;
//输出结果: [00123],[-0123]
//④输出字符型数据(字符型数值、ASCII 码)
cout<<k<<","<<int(k)<<setfill(' ')<<endl;
//输出结果: a,97
//⑤输出带格式的字符型数据(宽度、右对齐、左对齐)
cout<<"["<<setw(12)<<k<<"],["<<left<<setw(12)<<k<<"]"<<right<<endl;
//输出结果: [          a],[a          ]
//⑥输出浮点型数据
//默认浮点格式、小数格式、指数格式
cout<<x<<","<<fixed<<x<<","<<scientific<<x<<endl;
//输出结果: 12.3456,12.345600,1.234560e+001
cout<<setiosflags(ios_base::floatfield);          //设置默认浮点格式
cout<<y<<","<<fixed<<y<<","<<scientific<<y<<endl;
//输出结果: 12,12.000000,1.200000e+001
//⑦输出指定精度的浮点型数据
//默认精度、域度
cout<<setiosflags(ios_base::floatfield);          //设置默认浮点格式
cout<<"["<<x<<"],["<<setw(10)<<x<<"]"<<endl;
//输出结果: [12.3456],[   12.3456]
//设置精度
cout<<setprecision(2);
cout<<"["<<x<<"],["<<setw(10)<<x<<"],["<<fixed<<setw(10)<<x<<"]"<<endl;
//输出结果: [12],[        12],[     12.35]
//⑧输出带格式的浮点型数据
cout<<setprecision(6)<<showpos<<"["<<y<<"],["<<-y<<"]"<<noshowpos<<endl;
//输出结果: [+12.000000],[-12.000000]
cout<<setfill(' ')<<"["<<setw(10)<<y<<"],["<<setw(10)<<-y<<"]"<<endl;
//输出结果: [ 12.000000],[-12.000000]
cout<<setfill('0')<<internal<<"["<<setw(15)<<y<<"],["<<setw(15)<<-y<<"]"<<endl;
//输出结果: [00000012.000000],[-0000012.000000]
cout<<setfill(' ')<<"["<<setw(15)<<y<<"],["<<setw(15)<<-y<<"]"<<endl;
//输出结果: [      12.000000],[-     12.000000]
//⑨输出字符串
cout<<"["<<"Java"<<"],["<<setw(6)<<"Java"<<"],[";
cout<<left<<setw(6)<<"Java"<<"]"<<endl;
//输出结果: [Java],[  Java],[Java  ]
```

下面集中给出使用 cin 输入格式化数据的例子。

```
//输入整型、长整型、短整型、浮点型
cin>>a>>h>>i>>x>>y;
//输入: 1 2 3 1.23 3.25          结果 a=1,h=2,i=3,x=1.23,y=3.25
//输入: 1 -1 12345 12.3 12e5     结果 a=1,h=-1,i=12345,x=12.3,y=1.2e6
//连续输入用空格、Tab、Enter 键间隔
cin>>a>>b>>c;
//输入: 1 2 3                    结果 a=1,b=2,c=3
//输入: 1,2,3                    结果 a=1,b,c 不确定(输入逗号不匹配空白符,cin 终止)
cin>>a>>k>>b>>m;
//输入: 12c34a                   结果 a=12,k=c,b=34,m=a
//输入: 12 c 34 a                结果 a=12,k=c,b=34,m=a
cin>>a>>m>>b>>m>>c;
//输入: 12,34,56                 结果 a=12,b=34,c=56
```

3.2.2 字符输入与输出

1. 字符输出 putchar 函数

putchar 函数的作用是向显示终端输出一个字符,一般形式为

```
putchar(c);
```

其中,参数 c 为整型,使用低 8 位的值,输出的字符是 c 值对应的 ASCII 符号。函数调用时,c 可以是常量、变量或表达式,可以是整型数据、字符型数据或转义字符。

putchar 函数可以直接输出附录 A 的 ASCII 码对照表中可显示的字符(ASCII 值为 0x20～0x7f),控制字符(ASCII 值为 0x00～0x1f)的输出有特殊的含义。例如,'\n'输出换行符,使光标移到下一行的开头;'\r'输出回车符,使光标回到本行开头。

2. 字符输入 getchar 函数

getchar 函数的作用是从键盘终端输入一个字符,一般形式为

```
getchar()
```

getchar 函数没有参数,函数返回值为输入的字符。通常将 getchar 的返回值赋给一个字符型变量或整型变量。例如:

```
c=getchar();                  //输入字符保存到 c 中,以便后续能使用它(用 c)
```

或者作为表达式的一部分直接使用。例如:

```
putchar(getchar());           //将输入字符直接输出
putchar(c=getchar());         //将输入字符保存到 c 中,并且输出
```

getchar 函数的输入操作步骤如下。

① 检查键盘缓冲区是否有字符。

② 若有字符则直接从缓冲区中提取一个字符返回,且缓冲区移向下一个字符。

③ 若没有字符则 getchar 等待键盘输入，直到输入 Enter 键结束等待，重复步骤①。例如执行：

```
c=getchar();
```

getchar 将等待键盘输入，如果输入 1↙（本书用↙表示按 Enter 键），则键盘缓冲区有两个字符，c 提取了字符“1”，键盘缓冲区还留有一个字符“↙”。又如执行：

```
1    c1=getchar();
2    c2=getchar();
```

执行第 1 行时 getchar 等待键盘输入，如果输入 1↙，那么 c1 提取了字符“1”；执行第 2 行时由于键盘缓冲区还有一个字符，故第 2 行不用等待键盘输入，直接提取字符“↙”。

由此可见，getchar 函数执行时从键盘上可以连续输入多个字符，直到遇到 Enter 键为止。输入的多个字符放到键盘缓冲区，一次 getchar 函数调用会从缓冲区中提取一个字符，直到缓冲区没有字符时才从键盘输入。

有时，程序员为调试目的在程序中会写出下面的调用：

```
getchar();
```

其含义是让程序执行到这一行时停下来，便于观察运行情况，按 Enter 键继续执行。

3.2.3　格式化输出

1. printf 函数

printf 函数的作用是向标准输出设备（显示终端）输出格式化的数据，一般形式为

```
printf(格式控制,输出项列表);
```

例如：

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

printf 函数的参数包括两部分。

（1）格式控制。

格式控制为字符串形式，称为格式控制串，它主要有两种内容。

① 格式说明。格式说明总是以百分号（%）字符开始，后跟格式控制字符，例如%d、%f等。它的作用是将输出项转换为指定格式输出。

② 一般字符。除格式说明外的其他字符，包含转义字符。一般字符根据从左向右的出现顺序直接输出到显示终端上，ASCII 控制字符的输出有特殊的含义。

（2）输出项列表。

输出项列表为将要输出的数据，可以是常量、变量或表达式。输出项可以是零个或多个，但必须与格式说明一一对应，即一个格式说明决定一个输出项。

下面是没有输出项且无格式说明的 printf 函数调用例子：

```
printf("hello,world\n");                    //没有输出项,且无格式说明
```

2. 格式控制

格式控制串按照从左向右的顺序,当遇到第 1 个格式说明时,那么第 1 个输出项被转换为指定的格式输出,第 2 个格式说明转换第 2 个输出项,以此类推。如果输出项多于格式说明,则多出的输出项被忽略。如果没有足够的输出项对应所有的格式说明,则输出结果无法预料。

(1) 格式说明域。

格式说明由可选(用方括号括起)及必需的域组成,其形式如下:

```
%[flags] [width] [.prec] [h|l|L|F|N] type
```

格式说明域是个表明具体格式选项的单个字符或数字。最简单的格式说明只有百分号和 type 字符(如%s)。可选域出现在 type 字符前,控制格式的其他特征。表 3.5 解释了每个域的含义,如果百分号后的字符作为格式说明域没有意义,则该字符直接输出。

表 3.5 printf 格式说明域含义

域	域选	描述	含 义
type	必需	类型字符	决定输出项转换为字符、字符串还是数值
flags	可选	标志字符	控制输出的对齐、符号、空格及八进制和十六进制前缀。可以出现多个标志
width	可选	宽度说明	指定输出项的最小显示宽度
.prec	可选	精度说明	指定输出项的最大输出字符数或浮点数的小数精度
h/l/L/F/N	可选	大小修饰	指明输出项类型大小或指针的远近

(2) type 类型字符。

类型字符是 printf 函数唯一必需的格式说明域。它出现在任何可选域之后,用来确定输出项的类型。表 3.6 列出了常用类型字符的含义。

表 3.6 printf 类型字符含义

字符	类型	输 出 格 式
d	int	带符号的十进制整数
u	int	无符号十进制整数
o	int	无符号八进制整数
x 或 X	int	无符号十六进制整数(若输出为字母,x 用 abcdef,X 用 ABCDEF)
f	double	具有[-]dddd.dddd 格式的带符号数值,其中,dddd 为一位或多位十进制数字。小数点前的数字个数取决于数的量级;小数点后的数字个数取决于所要求的精度
e 或 E	double	具有[-]d.ddde[+/-]ddd 格式的带符号数值,其中,d 为单个十进制数字,dddd 为一位或多位十进制数字,ddd 为 3 位十进制数,用 e 或 E 表示指数

续表

字符	类型	输 出 格 式
g 或 G	double	以 f 或 e 格式输出的带符号数值，对给出的值及其精度，f 和 e 哪个简洁就用哪个。只有当值的指数小于－4 或大于或等于精度说明时才使用 e 格式。尾部的 0 被截断，只有小数点后跟一位或多位数字时才出现小数点。用 e 或 E 表示指数
c	char	单个字符
s	字符串指针	直到第一个非空字符('\0')或满足精度的字符串
%		输出百分号'%'

(3) 格式。

标志字符是一个字符，可以调整对齐、符号、空格以及八进制和十六进制前缀，格式说明中可以有多个标志字符。宽度说明是非负的十进制整数，它规定输出占位的最小宽度。但输出大于宽度时，按实际值的输出，小的宽度不会引起输出值的截断。精度说明是以圆点(.)开头的非负十进制整数，它规定了输出的最大字符数或有效数字位数。精度说明可以引起输出值的截断，或使浮点数输出值四舍五入。大小修饰指明输出结果的类型大小。

表 3.7 列出了常用标志、宽度、精度和大小修饰的含义。

表 3.7　printf 常用格式

项目	值	含　　义
标志	－	在给定域宽内左对齐输出结果(右边用空格填充)，默认右对齐(左边用空格或 0 填充)
	＋	如果输出值是有符号数，则总是加上符号(＋或－)，默认只在负数前加－
	空格	如果输出值是有符号数或为正数，则以空格作为前缀加到输出值前；如果空格和＋标志同时出现，则忽略空格
	#	指明使用如下的“转换样式”转换输出参数 x 或 X：在任何非 0 输出值前加上 0x 或 0X o：在任何非 0 输出值前加上 0
宽度	n	至少有 n 个字符宽度输出，如果输出值中的宽度小于 n 个，则输出用空格填充直到最小宽度规定(如果 flags 为－，则填充在输出值的右边，否则在左边)
	0n	至少有 n 个字符宽度输出，如果输出值中的宽度小于 n 个，则输出用 0 填充在输出值的左边(对于左对齐无效)
	*	间接设置宽度，此时由输出项列表提供宽度值，且它必须在输出项的前面
精度	.n	精度值指定浮点型小数点后数字的个数(默认 6 位)，四舍五入。或指定可输出字符的最大数目，超出精度值范围的字符不予输出
	.0	浮点型输出不打印小数点(及其后的小数)
	.*	间接设置精度，此时由输出项列表提供精度值，且它必须在输出项的前面。如果宽度说明和精度说明同时使用 *，则先出现宽度值，接着是精度值，然后才是输出项
大小修饰	h	对于 d、o、x、X 类型为 short，u 为 unsigned short
	l	对于 d、o、x、X 类型为 long，u 为 unsigned long，e、E、f、g、G 为 double
	L	对于 e、E、f、g、G 为 long double

请注意,一个浮点类型值若是正无穷大、负无穷大或非 IEEE 浮点数时,printf 函数输出+INF、-INF、+NAN 或-NAN。

下面集中给出调用 printf 函数格式化输出数据的例子。

```
int a=123,b=-1,c=12345; long h=-1; short i=-1,j=32767;
char c1=97; double x=12.3456,y=12,z=12.123456789123;
//①输出整型数据
printf("%d,%u,%x,%X,%o\n",a,a,a,a,a);   //十进制、无符号、十六进制和八进制
//输出结果: 123,123,7b,7B,173
printf("%d,%u,%x,%X,%o\n",b,b,b,b,b);        //十进制、无符号、十六进制和八进制,负数为补码
//输出结果: -1,4294967295,ffffffff,FFFFFFFF,37777777777
printf("%ld,%lu,%lx,%lo\n",h,h,h,h,h);          //长整型,负数为补码
//输出结果: -1,4294967295,ffffffff,37777777777
printf("%hd,%hu,%hx,%ho\n",i,i,i,i,i);          //短整型,负数为补码
//输出结果: -1,65535,ffff,177777
printf("%hd,%hd\n",j,j+1);                      //短整型,数据溢出
//输出结果: 32767,-32768
//②输出带格式的整型数据
printf("[%d],[%4d],[%-4d],[%4d],[%-4d]\n",a,a,a,c,c);
                                                //宽度、右对齐、左对齐、实际宽度
//输出结果: [123],[ 123],[123 ],[12345],[12345]
printf("[%+d],[%+d],[%d],[%d]\n",a,-a,a,-a);    //填充正负符号、填充空格
//输出结果: [+123],[-123],[ 123],[-123]
printf("[%04d],[%04d],[%04d],[%-04d]\n",a,b,c,a);//左边填充 0,右边不影响
//输出结果: [0123],[-001],[12345],[123 ]
printf("%#d,%#x,%#X,%#o\n",a,a,a,a);            //填充十六进制、八进制前缀
//输出结果: 123,0x7b,0X7B,0173
printf("[%*d]\n",5,a);                          //由输出项指定宽度
//输出结果: [  123]
printf("[%8.2d],[%-8.2d]\n",a,a);               //精度对整型无作用
//输出结果: [     123],[123     ]
//③输出字符型数据
printf("%d,%c\n",c1,c1);                        //字符型数值、ASCII 码
//输出结果: 97,a
//④输出带格式的字符型数据
printf("[%12c],[%012c],[%-012c]\n",c1,c1,c1);   //宽度、右对齐、左对齐
//输出结果: [           a],[00000000000a],[a           ]
//⑤输出浮点型数据
printf("%lf,%e,%g\n",x,x,x);                    //小数格式、指数格式、最简格式
//输出结果: 12.345600,1.234560e+001,12.3456
printf("%lf,%e,%g\n",y,y,y);                    //小数格式、指数格式、最简格式
//输出结果: 12.000000,1.200000e+001,12
//⑥输出指定精度的浮点型数据
printf("[%lf],[%10lf],[%10.2lf],[%.2lf]\n",x,x,x,x);    //默认精度、宽度、精度
```

```
//输出结果: [12.345600],[ 12.345600],[     12.35],[12.35]
//⑦输出带格式的浮点型数据
printf("[%+lf],[%+lf],[%lf],[%lf]\n",y,-y,y,-y);   //填充正负符号、填充空格
//输出结果: [+12.000000],[-12.000000],[ 12.000000],[-12.000000]
printf("[%06.1lf],[%-06.1lf]\n",y,y);                  //左边填充 0,右边不影响
//输出结果: [0012.0],[12.0  ]
printf("[%.*f],[%*.*f]\n",6,x,12,3,x);              //由输出项指定宽度、宽度与精度
//输出结果: [12.345600],[      12.346]
//⑧输出字符串
printf("[%s],[%6s],[%-6s]\n","Java","Java","Java");   //宽度对字符串的影响
//输出结果: [Java],[  Java],[Java  ]
printf("[%s],[%.3s],[%6.3s]\n","Basic","Basic","Basic");    //精度对字符串的影响
//输出结果: [Basic],[Bas],[   Bas]
//⑨特殊输出
printf("%%\n",c1);                              //两个%%表示输出一个%,输出项
//输出结果: %
printf("%d,%d\n",a,b,c);                        //格式数目小于输出项数,忽略多余输出项
//输出结果: 123,-1
printf("%d,%d,%d\n",a,b);                       //格式数目大于输出项数,输出结果不确定
//输出结果: 123,-1,2367460
printf("%d,%lf\n",x,a);                         //类型不对应,输出结果不确定
//输出结果: 2075328197,0.000000
```

3.2.4　格式化输入

1. scanf 函数

scanf 函数的作用是从标准输入设备(键盘终端)读取格式化的数据,一般形式为

```
scanf(格式控制,输入项列表,…);
```

例如:

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

scanf 函数将数据读到输入项列表中。每个输入项必须为地址形式(& 变量)。格式控制可以包含下列情况的一种或多种。

(1) 空白符:①空格('␣');②Tab('\t');③换行('\n')。空白符使 scanf 读输入数据中的连续空白符,但不保存它们,直至读到下一个非空白字符为止。格式串中的一个空白字符可以匹配任意数目(包括 0 个)和任意组合的空白符。

(2) 除百分号(%)外的非空白符:非空白符使 scanf 读入一个匹配的非空白字符,但不保存它。如果输入中的字符并不匹配,则立即终止 scanf。

(3) 格式说明符:由百分号%开头。格式说明符使 scanf 读入输入数据中的字符,并将它转换成指定类型的值,该值将保存到输入项中。

格式控制自左向右地处理。遇到第 1 个格式说明符时,读入第 1 个输入数据,并存放

到第 1 个输入项中,以此类推,直到格式控制串结束。其他字符用来匹配输入中的字符序列,输入中的匹配字符被读入但不保存;如果输入中某个字符与格式说明相冲突,则 scanf 终止,该字符将留在输入流中,就像没有读过一样。

输入数据被定义为:下一个空白符之前的所有字符;下一个不能按格式说明转换的字符之前的所有字符(如在八进制下出现 8);到达域宽之前的所有字符。

如果输入项个数比给定的格式说明符多,多余的输入项被忽略;如果输入项个数比格式说明符少,则读入结果是不可预料的。scanf 函数经常会引发灾难性的结果,其原因就是格式说明、输入项与实际输入数据不匹配。

2. 格式控制

(1) 格式说明域。

格式说明由可选(用方括号括起)及必需的域组成,其形式如下:

```
%[*] [width] [h|l|L|F|N] type
```

格式说明域是个表明具体格式选项的单个字符或数字。最简单的格式说明只有百分号和 type 字符(如%s)。如果百分号(%)后面紧跟的字符作为格式控制没有意义,该字符和下一个百分号(%)之前的所有字符被当作必须与输入匹配的字符序列。

(2) type 类型字符。

类型字符是 scanf 函数唯一必需的格式说明域。它出现在任何可选域之后,用来确定输入项的类型。表 3.8 列出了常用类型字符的含义。

表 3.8 scanf 类型字符的含义

类型字符	期望读入(应输入)的类型
d	十进制整数
o	八进制整数
x 或 X	十六进制整数
u	无符号十进制整数
e, E, f, g,G	由下列成分组成的浮点数:可选的符号+或-,包括小数点在内的一个或多个十进制数字序列,可选的指数符('e'或'E')其后的带符号整数。[+/-] ddddddddd [.] dddd [E|e] [+/-] ddd
c	字符。指定 c 后,通常被跳过的空白符将被读入,如果要读下一个非空白符,要使用%1s
s	字符串。默认情况下,输入字符串以空白符作为结束

(3) * 禁止字符。

* 禁止字符的含义是从输入数据中读取类型相当的数据,但跳过这个数据,即不将它保存到输入项中。