第 3 章　表达式和语句

程序是一些按次序执行的语句。执行语句是为了要完成某个操作，修改某个数据。程序中大部分的语句是由表达式构成的，因为表达式直截了当地返回值。正因为如此，表达式是 C++编译器处理的重要环节。学习本章后，要求理解表达式和语句的概念，掌握表达式中各种运算符的功能与特点，明白产生副作用的原因。更好地理解 C++语言的强大与灵活。

3.1　表达式

1. 表达式概述

表达式是操作符、操作数和标点符号组成的序列，其目的是用来说明一个计算过程。

表达式可以嵌套，例如：2＋3＋(5 * sizeof(int))/345。

表达式根据某些约定、求值次序、结合和优先级规则来进行计算。

所谓约定，即类型转换的约定。例如：

```
float a;
a = 5/2;
```

结果 a 得到值为 2。5/2 是整数除法取整，因为 5 和 2 都是整数，不会由于 a 是 float 型而轻易改变运算的性质。

所谓求值次序，是指表达式中各个操作数的求值次序视编译器不同而不同。见 3.9 节。

所谓结合性，是指表达式中出现同等优先级的操作符时，该先做哪个操作的规定。例如：

```
d = a + b - c;          //C++规定，加减法先左后右.先做 a + b,其结果再减去 c
d = a = 3;              //C++规定，等号是先右后左.先做 a = 3,其结果再赋给 d
```

所谓优先级，是指不同优先级的操作符，总是先做优先级高的操作。例如：

```
d = a + b * c;          //乘法优先级比加法高.先做 b * c,其结果再与 a 相加
```

2. 左值和右值

左值(left value，缩写为 lvalue)是能出现在赋值表达式左边的表达式。左值表达式具有存放数据的空间，并且存放是允许的。例如：

```
int a = 3;              //a 是变量,所以 a 是左值
const int b = 4;        //b 是常量,所以 b 不是左值
```

显然常量不是左值，因为 C++规定常量的值一旦确定是不能更改的。

右值(right value，缩写为 rvalue)只能出现在赋值表达式的右边。左值表达式也可以作为右值表达式。例如：

```
int a,b = 6;
a = b;                    //b 是变量,所以是左值,此处作为右值
a = 8;                    //8 是常量,只能作右值,不能作为左值
```

表达式可产生左值、右值或不产生值。例如：

```
int a;
(a = 4) = 28;             //ok:a = 4 是左值表达式,可以被赋以值 28
void f(){return ;}        //此为函数定义,该函数不返回任何值
```

28 是右值表达式,而 a=4 是左值表达式(C++的语法规定),所以可以放在赋值语句的左边。该语句表示 a 的值用 28 替代刚刚赋给的值 4。

函数定义本身不是表达式,它说明了一个不返回值的函数。对函数 f()的调用是语句,它实施了一个没有返回值的操作。

3. 优先级和结合性

表 3-1 对操作符的优先级和结合性作了小结。表中包含了 C++所有的操作符,共有 16 级优先级。表中的操作符如重复出现,则第 1 次出现的是单目运算符,第 2 次出现的双目运算符。每一级的结合性,不是从左到右就是从右到左。表达式中,在没有括号的情况下,这些规则决定了表达式运算的次序。

每一级中的操作符是同优先级的。

表 3-1　C++操作符的优先级与结合性

优先级	操　作　符	结合性
1	() [] -> ::	左→右
2	! ~ + - ++ -- & * (强制转换类型) sizeof new delete	右→左
3	.* ->*	左→右
4	* / %	左→右
5	+-	左→右
6	<< >>	左→右
7	< <= >= >	左→右
8	== !=	左→右
9	&	左→右
10	^	左→右
11	|	左→右
12	&&	左→右
13	||	左→右
14	?:	右→左
15	= *= /= += -= |= <<= >>=	右→左
16	,	左→右

4. 语句与块

C++中所有的操作运算都通过表达式来实现。由表达式组成的语句称为表达式语句，它由一个表达式后接一个分号“;”组成。

通过计算表达式即执行了表达式语句。大多数表达式语句为赋值语句和函数调用。

语句是用来规定程序执行的控制流。在没有跳转和分支(见第 4 章)的情况下，语句将按照其在源程序中出现的次序顺序执行。

语句可以是空语句。空语句是只有一个分号而没有表达式的语句，其形式为：

```
;
```

它不产生任何操作运算，只作为形式上的语句，被填充在控制结构中。例如：

```
if(x>9)
  ;
else
   cout <<"not large than 9\n";
```

例中判断 x 是否大于 9，如果大于 9，不做任何事，否则输出“not large than 9”和回车。

块(或称复合语句)是指括在一对花括号{}里的语句序列。从语法上来说，块可以被认为是单个语句。例如：

```
if(x>9)
{
   cout <<"The number is perfect.\n";
   cout <<"It is large than 9\n";
}
else
{
   cout <<"not large than 9\n";
}
```

x 若大于 9，则执行两条输出语句，否则，输出“not large than 9”和回车。这两条执行语句必须放在花括号中，因为 if 与 else 之间只能容纳一条语句，或一个语句块。而 else 后面的花括号则可以省略。此外，块还可以嵌套。

3.2 算术运算和赋值

1. 操作符种类

C++提供了算术运算符+，-，*，/，%。

+，-，* 是通常意义的加、减、乘法。

/对于整型数则为除法取整操作。例如，5/2 得到结果 2。

/对于浮点数则为通常意义的除法。例如，5.0/2.0 得到结果为 2.5。

由此可见，/操作符可以对不同的数据类型进行不同的操作。事实上，+，-，*，/，%对不同数据类型的操作都不同。如整数加法是将 2 个整型相加，而浮点数加法是将 2 个浮点数相加，相加的具体操作(在机器指令级上)浮点和整数是不同的。

%只能对整型数进行操作。其操作意义为取余。例如,5%2 得到结果为 1。

%不允许对浮点数操作,如果对浮点数操作,则会引起编译错误。

2. 赋值缩写

算术表达式的赋值表示为:

```
int x,y,z;
x = y * z;
x = y/z;
x = y + z;
x = y - z;
x = y % z;
```

当一变量既出现在表达式的左边又出现在右边时,可以缩写。例如:

```
x = x * y;      缩写为:      x * = y;
x = x + y;      缩写为:      x + = y;
x = x - y;      缩写为:      x - = y;
x = x/y;        缩写为:      x/ = y;
x = x % y;      缩写为:      x % = y;
```

赋值以及缩写都要求左边的表达式为左值,即 x 为左值。

赋值构成一个表达式,因而它具有值。赋值表达式的值为赋值符左边表达式的值。例如:

```
cout <<(x = 5) <<endl;
```

将输出 5。同时 x 被赋予值 5。

赋值表达式为左值。例如:

```
(x = max(5,7)) + = 3;
```

该语句先将 max(5,7)函数调用的值赋给 x,然后在此基础上增值 3。它等价于:

```
x = max(5,7) + 3;
```

缩写格式通常更有效,可读性也不差。缩写与不缩写的差别在于缩写式中左边的变量只出现一次,而不缩写的式中出现 2 次。例如:

```
(x = max(5,7)) = (x  =  max(5,7)) + 3;
```

上式是合法的表达式,它与前面讨论的表达式不同之处在于 max(5,7)调用了 2 次。如果函数有副作用(见 3.9 节),则赋值与赋值缩写是不同的。

3. 溢出

进行算术运算时,很可能溢出结果。发生溢出是由于一个变量被赋予一个超出其数据类型表示范围的数值。数值溢出是不会引起编译错误的,只要分母不为 0 也不会引起除 0 运行故障,但会使运行结果发生偏差。

例如,在 16 位机器上进行下面的操作:

```
int weight = 42896;
```

在 16 位机器中将不能得到值 42896，而是－22640。因为有符号整数的表示范围是－32768～32767，所以它只能得到 42896 的补码－22640(42896－65536)。

一个整数类型的变量，用任何一个超过表示范围的整数初始化，得到的值为用该整数范围作模运算后的值。例如：

```
int weight = 142896;
```

则当 weight 是 2 字节整型数时，得到值为 11824。因为 142896＝2×65536 ＋11824。而 142896 －3×65536＝－53712，该数不在有符号整型数表示范围内。

3.3 算术类型转换

C++遇到两种不同数据类型的数值进行运算时，会将两个数作适当的类型转换，然后再进行运算。转换的方向见图 3-1。

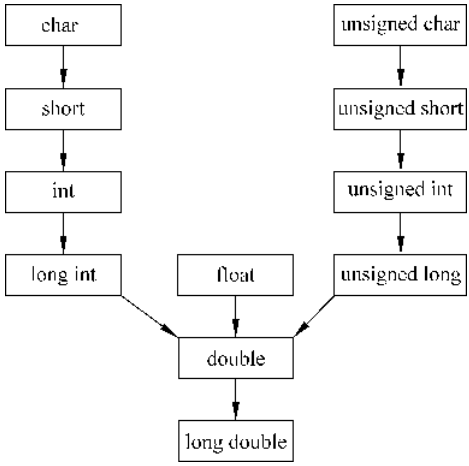

图 3-1　类型转换的方向

如果一个 char 型数和一个 int 型数相加，则将 char 型数转换成 int 型数，然后进行运算。因为在图 3-1 中，char 有向 int 转换的趋势。如果一个 long int 型数和一个 float 型数相加，则先将两个数据类型都转换成 double 型数，然后进行运算。如果一个 int 型数和一个 unsigned long 型数相乘，则先将两数都转换成 double 型数，然后进行运算。

转换总是朝表达数据能力更强的方向，并且转换总是逐个运算符进行的。例如：

```
float f = 3.5;
int n = 6;
long k = 21;
double ss = f * n + k/2;
```

ss 将会得到结果 31。计算 ss 时，首先将 f(float 型)和 n(int 型)转换成 double 型数，算得 21，然后计算 k/2 得整除运算结果 10(long int 型)，再将 long int 型的数字 10 转换成 double 型数。21 和 10 两个数相加，得到最后结果 31。

数据运算过程中自动进行的类型转换称为隐式类型转换。上例的表达式运算过程中进行的数据类型转换就是隐式转换。

有时候,我们会面临下面计算结果不准确的问题:

```
long m = 234 * 456/6;
```

即发现 m 为－4061,而不是 17745。原因是语句先进行 int 型数的乘法运算,结果仍以 int 型数保留起来:234×456＝106704＝2×65536－24368,取模之后得到－24368。该数参加整除运算:－24368/6 得－4061(取整)。由于中间的结果被截断,所以,最后的结果是错的。如果让第 1 次乘法的结果以 long 型数保留下来,就能得到正确的结果。这就要求参加乘法运算的 2 个数至少有一个为 long 型数。

例如,将其中之一标识以 L 或 l(long),则可保证其正确:

```
cout <<234 * 456L/6 <<endl;
```

输出结果为:

```
17784
```

还可以将整型数强制转换为 long 型:

```
cout <<(long)234 * 456/6 <<endl;
```

该语句使 234 成为 long 型数,与整数 456 相乘,先隐式转换,再相乘,得到一个 long 型数 106704。再与 6 相除取整,从而得到正确结果。

强制转换又称显式转换,其语法是在一个数值或变量前加上带括号的类型名。也可以类型名后跟带括号的数值或表达式。如上面语句也可以写成:

```
cout <<long(234) * 456/6 <<endl;
```

如果类型名是带类型修饰的,则要给类型名加括号。例如:

```
cout <<(unsigned long)234 * 456/6 <<endl;
cout <<unsigned long(234) * 456/6 <<endl; //error
```

请注意下面语句不能产生所期望的效果:

```
cout <<long(234 * 456)/6 <<endl;
```

该语句首先执行括号里的乘法,得到一个 int 型整数(已被取模)－4061,然后强制转换为 long 型数,再参加除 6 取整运算,所以得不到正确结果。

3.4 增量和减量

增量和减量操作符表示为:＋＋和－－。

增量操作表示加 1,减量操作表示减 1。例如:

```
a++;    //等价于 a=a+1;
++a;    //等价于 a=a+1;
a--;    //等价于 a=a-1;
--a;    //等价于 a=a-1;
```

增量操作符有前增量与后增量之分。前增量操作 ++ a 的意义为：先修改操作数使之增1，然后将增 1 过的 a 值作为表达式的值。而后增量操作 a ++ 的意义为：先将变量 a 的值作为表达式的值确定下来，再将 a 增 1。对于增量和减量操作符，它要求操作数是左值，因为操作数的值要发生变化。例如：

```
int a = 3;
int b = ++a;          //相当于 a = a + 1; b = a;
cout <<a <<"    "  <<b <<endl;
int c = a++;          //相当于 c = a; a = a + 1;
cout <<a <<"  "  <<c <<endl;
```

输出的结果为：

```
4  4
5  4
```

b 被赋予了 4，因为前增量操作先将 a 自增为 4，然后作为表达式赋值。C 被赋予了 4，因为后增量操作使表达式的值(a)先赋给 c，然后 a 再自增为 5。

由于前增量操作返回的值即修改后的变量值，所以返回的仍是一个左值。例如：

```
int a = 3;
++(++a);              //ok: ++a 是左值,可以接着做 ++ 操作
```

例中得到的 a 的值为 5。

由于后增量操作返回的值是原先 a 的值，尔后 a 的值已经发生变化，故返回的不能是当前 a，只能是过去的 a 值，不能是左值。例如：

```
   int a = 3;
++(a++);              //error: a++ 不是左值
```

相应的，有前减量 --a 和后减量 a-- 。例如：

```
int a = 3;
int b = --a;          //相当于 a = a - 1; b = a;
cout <<a <<"  " <<b <<endl;
int c = a--;          //相当于 c = a; a = a - 1;
cout <<a <<"  " <<c <<endl;
```

输出的结果为：

```
2  2
1  2
```

由于增量与减量操作包含有赋值操作，所以操作数不能是常量，它必须是一个左值表达式。例如：

```
3++;                  //error
```

增量与减量操作符是两个＋或两个－的一个整体，中间不能有空格。如果有多于两个＋或两个－连写的情况，则编译首先识别前面两个＋或－为增量或减量操作符。

例如，对于“int a＝1，b＝5，c；”的变量定义，下面 5 个表达式，有些不允许：

```
c = a + b;          //ok:c = 6
c = a ++ b;         //error:编译接收为 a ++ b
c = a +++ b;        //ok:编译接收为 a ++ + b
c = a ++++ b;       //error:编译接收为 a ++ ++ b
c = a +++++ b;      //error:编译接收为 a ++ ++ + b
```

第 2 行中,编译将其理解为 a ++ b。由于 ++ 操作是单目运算符,所以该表达式语法错误。若要合法,应写成 a+ +b,表示 a 加上正 b。

第 3 行中,编译将其理解为 a ++ ++ b。同样由于 ++ 是单目操作符,引起编译错误。若要合法,应写成 a+++ +b,表示 a++ 加上正 b。

第 4 行中,编译将其理解为 a ++ ++ + b。由于 a++ 是个非左值表达式,所以中间的 ++ 操作符是非法的。若要合法,应写成 a+++ ++b 或者 a++ + ++b,表示 a++ 加上 ++b。

3.5 关系与逻辑运算

1. 运算符

关系操作符有:

```
比较( == )
大于(>)
小于(<)
大于等于(> = )
小于等于(< = )
不等于(!= )
```

逻辑运算符有:

```
非(!)
逻辑与(&&)
逻辑或(||)
```

对于>=,<=,!=,==,&&,||都是一个操作符的整体,所以中间不能有空格,而且前 3 个操作符中的字符次序不能颠倒。例如:下面的写法都是不合法的。

```
=<, =>, =!,<  =,>  =, =  =,!  =
```

2. 比较运算符

比较(==)和赋值(=)是两个不同的操作,所以用的操作符也不同。比较用于测试给定的 2 个操作数是否相等。例如:

```
if(x == 999)
   cout <<"x is 999\n";
```

C++中,表达式都产生值,赋值操作符产生的值正是所赋的值。而比较操作符产生的值是比较的结果,可能是 0 或 1,即假或真。

真和假是逻辑值。在C++中,假意味着0,真意味着非0。所以,任意一个非0数都是真,表示为逻辑值就是1。例如:

```
x = somevalue;
if(x = 9)
   cout <<"x is not 0\n";
```

例中,不管x的初值是什么,总是执行cout语句。因为x=9是赋值表达式,其表达式的值是所赋的值9,而9为非0值,所以if语句的条件为真,所以总是执行cout语句。又例如:

```
x = somevalue;
if(x = 0)
   cout <<"x is 0\n";
```

例中,不管x以前是什么值,总是不会执行cout语句。因为x=0是赋值表达式,并且其值为0,为假。

这一般不是编程的本意,但由于**=与==经常不小心搞错**,使得程序不正确地运行。在BC和VC编译器中,在像if语句这样的条件表达式中,遇到=时都会给予警告。这时,就应该有所警觉,以免程序错误地执行。

3. 不等于运算符

当要测试一些东西不是真时,可以使用不等于操作符。例如,如果要在一些东西是真时在屏幕上显示一则消息,则可以用如下语句:

```
if(x!= 9)
   cout <<"x isn't 9\n";
```

要注意的是,如果颠倒!=,则意义完全不同:

```
if(x = !9)
   cout <<"x isn't 9\n";
```

该if条件表达式是一个赋值语句,!9为非真,即0。所以该条件表达式相当于if(x=0),于是cout语句永远也不会执行。

4. 嵌入赋值

有时候,需要将一个函数值赋给一个变量,然后比较该变量的值与预定值是否相等。例如:

```
x = func();
if(x == somevalue)
   //语句
```

上面的代码与下面的代码等价:

```
if((x = func()) == somevalue)
   //语句
```

因为要给x赋值,然后确定x的值,所以先进行赋值。而赋值表达式的值即x的值可以

作为比较的操作数。这里,由于==操作比=操作优先级高,所以需要额外加一个括号。

这种赋值紧接着比较的表达式称为嵌入赋值,它经常在程序样例中看到。

5. 逻辑非运算符

! 改变条件表达式的真假值,即逻辑运算的"非"。原来是 0,则 !0 为 1;原来为非 0,则 ! 操作使之变为 0。例如:

```
if(!(x == 9))
   cout <<"x is not 9.\n";
```

因为==比 ! 优先级低,所以额外的括号是需要的。包围"x==9"的括号使比较先进行,然后再做非操作。

6. 逻辑运算

&& 和||是两个逻辑运算符,它们的意义为求两个条件表达式的逻辑与和逻辑或。

例如,下面的代码为根据室温打印一则消息:

```
int temp = 90, humi = 80;

if(temp> = 80 && humi> = 50)
   cout <<"wow, it's hot!\n";
if(temp<60)||temp>80)
   cout <<"the room is uncomfortable.\n";
```

输出结果为:

```
wow, it's hot!
the room is uncomfortable.
```

因为 && 比>=优先级低,所以 if 中的条件表达式为先求 temp>=80 和 humi>=50,然后进行逻辑与 && 运算。

7. 短路表达式

如果多个表达式用 && 连接,则一个假表达式将使整个连接都为假(此处需要数理逻辑知识)。例如:

```
int n = 3, m = 6;

if(n>4 && m ++ <10)
   cout <<"m should not changed.\n";

cout <<"m = " <<m <<endl;
```

输出结果为:

```
m = 6
```

由于 n>4 的比较值为 0,所以整个 if 条件表达式的值不用看后面就知道为 0。C++利用这个特点以产生高效的代码。所以,后面的表达式不被执行。这样,m 的值还是 6 而不是

7。知道了短路表达式在C++中的处理方式，就可以在编写程序时，不但避免不必要的错误，而且还可利用它。例如：

```
if(b!= 0 && a/b>2)
   //语句
```

if条件表达式中的b!=0若成立，才会执行后面的关系运算，做分母是b的除法。否则，跳过整个条件语句。

→ 在程序中，如果碰到除0运算，则运行发生异常。如果没有定义异常处理(见第21章)，则整个程序终止运行。

同理，如果多个表达式用||连接，则一个真表达式将使整个连接都为真。例如：

```
if(temp<60||temp>80)
   cout <<"the room is uncomfortable.\n";
```

例中如果temp<60成立，则不会进行temp>80的关系比较，直接执行输出语句。

3.6 if语句

1. if语句

if语句的语法为：

```
if(条件表达式)
   语句;
```

或：

```
if(条件表达式)
{
   语句;
}
```

它的意义为：如果条件表达式进行一次测试，且测试为真，则执行后面的语句。C++中的if语句与其他计算机语言的if语句区别不大。

如果if语句只控制一条语句，则包围该语句的花括号不是必需的。

例如，下面的程序等待键入一字符，如果是'b'，则响铃：

```
#include <iostream.h>
#include <conio.h>

void main()
{
   cout <<"please input the b key to hear a bell.\n";
   char ch = getche();
   if (ch == 'b')
      cout <<'\a';
}
```

2. 空语句

编译器必须在if条件表达式的后面找到一个作为语句结束符的分号“;”，以标志if语