第 5 章　运算的最小单位——表达式

任何合法的变量、常量、运算符，以及函数调用的有机组合就是表达式。它表示了一个概念或模型，是编程语言的基本要素。表达式是程序的核心和灵魂，表达式的重要行为是计算和类型转换。学习本章，读者可以了解 C++语句的构成基础，为后面的进一步学习铺好道路。

5.1　使用表达式

表达式是由变量、常量、函数调用等，由运算符按一定规则连接起来的有意义的式子，类似数学上的公式。它是程序的基本要素。它既可以是单个的变量或数据，也可以是表达式的复合。

5.1.1　表达式的种类

从类型上分，表达式可分为算术表达式、关系表达式、条件表达式、赋值表达式、逗号表达式，以及逻辑表达式等。

- 算数表达式是用算数运算符和常量、变量组成的表达式，用来求数值解；
- 关系表达式的运算符为关系运算符，用来判断表达式元素间的关系；
- 条件表达式是由条件语句构成的表达式；
- 赋值表达式的运算符是赋值运算符，执行右值向左值赋值的功能；
- 逗号表达式是使用逗号连接多个表达式构成的复合表达式；
- 逻辑表达式的运算符为逻辑运算符，用来表示一种逻辑关系。

例如，5+3，x>y，x>0&&y>0，x=6，x>y？x:y 等，这些都是表达式。其中，第 1 个是算数表达式，第 2 个是关系表达式，第 3 个是逻辑表达式，第 4 个是赋值表达式，最后一个是条件表达式。而这一组表达式作为一个表达式来用就行成了逗号表达式。

从复杂程度上分，表达式可分为原子表达式和复合表达式。

- 原子表达式指单个的数字、字符、字符串、函数等表示单一概念的量；
- 复合表达式是原子表达式和运算符（诸如括号、+、–、&&、!等）按一定规则构成的式子。上述的表达式都属于复合表达式。

从上述讨论可知，表达式是可以嵌套的，是一个递归的概念，任何表达式都可以再作为元素去构成更复杂的表达式。

例如，表达式 x+y 和 x-y 可以复合成表达式(x+y)*(x-y)。

5.1.2　表达式到底是什么

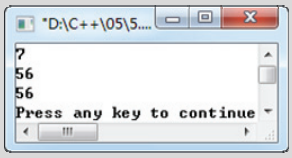
一个表达式也可看做是一个数学函数。它输入的是各元素的定值，而输出则为表达式对这些值计算的结果，即表达式的值。

就表达式本身来讲，它什么都不做，只是计算并返回结果。如果程序不对这个返回的结果做任何进一步的操作，它将不起任何作用。由于它返回的是值，所以也可参与运算。至于能够参与什么样的运算，这依赖于它的返回值是什么样的类型。

如果说两个表达式是相等的，那么意指这表示两个表达式对于任意相同的输入，具有相同的输出。这与数学中函数相等的概念是一致的。

【示例 5-1】演示表达式的运算。

```
01  #include <iostream.h>
02  int main(void)
03  {
04      int num=0;                  //定义变量 num，同时进行赋值
05      cin>>num;                   //从命令行读入数值
06      cout<<((num+num)<<2)<<endl;
                          //由加法和移位构成的复合表达式
07      cout<<(8*num)<<endl;        //乘法表达式
08      return 0;
09  }
```



分析：该示例定义两个表达式“((num+num)<<2)”和“(8*num)”，从命令行读入整数，用这两个表达式计算并输出。这两个表达式具有不同的形式，表面上完全不一样，但它们对于任何 num 的赋值都会输出相同的结果，所以说它们是相等的。实际上可以看出，将第一个表达式简化就得到了第 2 个表达式。

表达式的计算中还包含优先级和结合性两个概念，已在第 4 章“运算符”中讲过，这里不再赘述。

5.1.3　如何写表达式

表达式的书写中需要注意以下几点。

1．表达式必须是“合适的”

这是指表达式中每个运算符都必须是完全的，书写正确的，放在正确的地方，有正确的语义，而且有正确的操作数。

【示例 5-2】下述表达式都是错误的。

```
&&xy,
4*(2+3,
8%,
&8
```

第 1 个表达式中&&放错了地方，应该是 x&&y。第 2 个中小括号不配对，缺少了右括号。第 3 个中操作数不全，原意应该是求 8 和某个数的模。第 4 个&使用错误。如果&作

为按位与，那么缺少了另一个操作数；而如果&作为取地址运算，那么对常量取地址也是不正确的。

2．使用一些符号增加可读性

为了增加可读性，表达式中可以加入任意的 Tab 键、空格符和括号。编译时，Tab 键和空格符会被忽略掉，不会给程序增加负担。括号不仅有助于可读性，更重要的是括号可以明确标出运算符的优先级和结合性，不致引起歧义。

3．结尾不能添加分号

表达式末尾不能带分号“;”。它是语句（参见第 6 章）的结束符，而表达式不是语句。这使得它可以出现在任何需要表达式的地方，扩大了使用范围。如果给表达式加上“;”，它将不再是表达式，而是表达式语句。

技巧：利用表达式具有返回值的特性，常可在一个表达式中出现多个左值。这么做虽然增加了程序的复杂度和理解难度，但可以提高代码运行效率。例如，area=high*(round=_PI_*(r<<1))在计算侧面积的同时，也保留了底面周长。

5.2　数据类型的转换

类型转换指在不同种类型或不同种精度之间的相互转换。表达式能否正确计算，很大程度上取决于操作数的类型是否适当。如果操作数的类型不适当，就要看能否转换为适当的。如果不能转换为需要的类型，那么这个表达式将是无法计算的，或者结果将不是预期的。因此，合理地进行类型转换是表达式成功的要素之一。

5.2.1　隐式转换

隐式类型转换就是自动类型转换。当系统期望得到某种类型的数据，但却得到的是另一种类型的数据时，就会尝试进行类型转换。如果不能自动完成隐式转换，则编译器会提示“不能从 xx 类型转换到 xx 类型”的错误。这个过程由编译器自动完成，无须程序员的介入。本节将从隐式转换规则和转换时机两方面来讲解。

1．发生时机

当下述情况之一发生时，就会发生隐式类型转换。

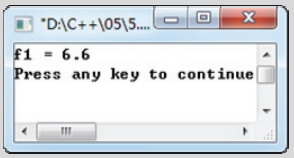
- 在算数表达式中，存在不同类型的操作数时：所有操作数中，精度和级别高的类型将成为目标类型，所有其他类型都向它转换。这种转换也被称为算术转换。
- 在赋值表达式中，左右值类型不同时：左值作为目标类型，将右值自动转换为左值的类型。
- 当表达式作为函数的参数，形参与实参类型不符时：形参的类型作为目标类型，将实参转换为形参的类型。

- 在函数中，将表达式的值作为返回值时：如果返回值类型与表达式的值类型不同时，则目标转换类型是函数的返回类型。
- 当表达式被用做条件时：由于条件语句的结果是 bool 型，所以这种表达式将被转换为 bool 型。

第 1 种转换是向上转换，精度和级别都被提高。而第 2、3、4 种则既可能向上转换，也可能向下转换。向下转换不是四舍五入而是截断。因此可能会丢失精度。此外，有些书上还把向上转换叫做升级，向下转换叫做降级。

【示例 5-3】下述代码中发生了 3 次类型转换，既有类型提升，也有类型降级。

```
#include <iostream.h>
int main(void)
{
    /*定义变量*/
    int i1=2;                  //定义整型变量
    float f1=2.3;              //定义浮点型变量
    double d1=2.3;             //定义双精度型变量
    /*隐式类型转换*/
    f1=i1+f1+d1;               //先类型提升，再类型降级
    cout << "f1 = " << f1 << endl;
    return f1;
}
```



分析：该示例中，语句“f=i1+f1+d1”中 3 个操作数的类型都不一样，以 d1 的类型（double 型）为最高。因此，其他两个操作数将首先转换为 double 型，才继续计算。该语句的计算结果是 double 型，但赋值表达式的左值却是 float 型。因此将再次由 double 转换为 float。main()函数的返回值是 int 型，但 return 却试图返回 float 型数据。因此 f1 会被降级。

注意：当发生降级时，如果源数超出了目标类型的范围，则结果不确定；如果在目标范围内，但不能用目标类型表示，则结果是最接近源数的值；如果源数在目标类型内可以精确表示，则没有精度损失。

2．转换规则

为了正确地执行隐式转换，C++已经为它内置了一套规则。在上述 5 个可能出现类型转换的情况中，第 2、3、4 种情况实际上已经事先决定了转换的目标类型。例如，在转换发生前就已清楚函数的参数和返回值需要什么样的类型，只需将表达式值按规定好的类型转换即可。第 5 种情况中，类型也是事先预知的，但转换结果只能取值 0 和 1。而第 1 种转换情况则比较繁琐，通常称其为算术转换。

当表达式中有不同的数据类型时，就需要进行算术转换。它总是向表达能力强的方向，由低级到高级，由低精度到高精度。它有以下 3 个基本原则需要遵守。

（1）整型提升：所有比 int 范围小的整型，包括 bool、char、signed char、unsigned char、short，以及 unsigned short 等，如果该类型的取值范围包括在 int 型内，就将它们提升为 int 型；否则，提升为 unsigned int 型。bool 型只能转换为 0 或 1。

（2）为了避免精度损失，继续提升为较宽的类型。转换按照下面的方向进行：

int>long int> float>double>long double

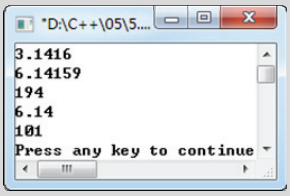
具体描述就是：若有一个操作数是 long double 型，则另一个转换为 long double 型；否则，若有一个操作数是 double 型，则另一个转换为 double 型；否则，若有一个操作数是 float 型，则另一个转换为 float 型；否则，若有一个操作数是 long int，则另一个转换为 long int 型。转换到最后的类型即为表达式的类型。

（3）若表达式中出现无符号型 unsigned，所使用的规则必须保护操作数的精度。

- unsigned short 和 int 型：若 int 型足够表示所有 unsigned short 型的值，则将 unsigned short 转换为 int；否则，均转换为 unsigned int 型。
- unsigned int 和 long 型：若 long 型足够表示 unsigned int 型的范围，则将 unsigned int 转换为 long 型；否则，都转换为 unsigned long 型。
- unsigned 型的操作数的转换依赖于机器中整型相对的大小，所以，这类转换依赖于机器。

【示例 5-4】讲解上述类型转换的规则。

```
01  #include <iostream.h>
02  int main(void)
03  {
04      bool bval=false;
05      char cval='a';
06      short sval=97;                  //短整型
07      unsigned short usval=98;        //无符号短整型
08      int ival=3;                     //整型
09      float fval=3.14;                //浮点型
10      double dval=3.1416;             //双精度型
11      long double ldval=3.1415926;    //长双精度型
12      cout<<bval+dval<<endl;          //bval 提升为 int，然后转换为 double
13      cout<<ldval+ival<<endl;         //ival 转换为 long double
14      cout<<cval+sval<<endl;          //提升到 int
15      cout<<fval+ival<<endl;          //转换为 float
16      cout<<ival+usval<<endl;         //依 unsigned short 和 int 的长度决定提
17                                        类型升到哪种
18      return 0;
19  }
```



分析：语句 bval+dval 中的变量 bval 是 bool，所以首先将它提升为 int 型。然后，dval 是 double 型，故 bval 还需转换为 double 型。语句 ldval+ival 中 ldval 为 long double 型，故 ival 需转换为 long doubel 型。语句 cval+sval 中的变量类型都小于 int 型，故均需提升到 int 型。语句 fval+ival 中 fval 为 float 型，故 ival 需转换为 float 型。最后一条语句 ival+usval 的类型转换则要以依赖于特定机型上 unsigned short 和 int 的长度来决定。

5.2.2　显式转换

显式类型转换也叫强制类型转换，它直接用类型修饰符将操作数转换为需要的数据类型。之所以需要显式类型转换，主要有以下 3 个原因。

- 去掉隐式转换中不必要的中间环节。例如，示例 5-4 中的语句 bval+dval，bval 必须先提升到 int，然后转换为 double 型。
- 隐式转换中，如果从一个高级别类型转换到低级别类型，会导致精度的损失，编

译器将给出警告。若提供显式强制转换，就可以屏蔽掉该警告。

❑ 进行某些特殊的转换。如将浮点型转换为指针型，常量指针转换为非常量指针等。

旧式的显式类型转换格式如下所示。

```
(type) expression
```

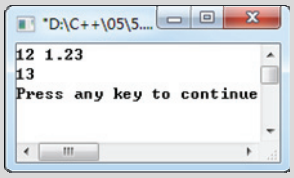
或

```
type( expression)
```

其中，type 是期望的目标类型，expression 是表达式。前者是 C 中转换风格的残留，后者是 C++中函数风格的转换。两者功能是一样的，

【示例 5-5】演示旧式的显式类型转换风格。

```
01  #include <iostream.h>
02  int main(void)
03  {
04      int ival;                //整型
05      float fval;              //浮点型
06      cin>>ival;               //从命令行读入数值
07      cin>>fval;               //从命令行读入数值
08      ival=ival+(int)fval;     //旧式的类型转换
09      cout<<ival<<endl;;       //输出 ival 的值
10      return 0;
11  }
```



分析：如果语句 ival=ival+(int)fval 没有（int）强制转换，则系统会先将 ival 转换成 float 型，计算 ival+fval 的值。然后再将该右值转换为 int 型赋给左值 ival。可见 ival 到 float 型的转换是多余的。如果加上强制转换（int），则明确将 fval 转换为 int 型，只需一次转换即可。

这种旧的转换方式语义不够精确，可视性差，难以追踪错误的转换，不方便系统对转换安全性的控制。为此，C++除了保留这种旧转换方式外，还增加了 4 种新的转换方式——命名的显式转换。它们具有如下统一的格式说明。

```
cast_name<type>(expression)
```

其中，cast_name 是转换类型的修饰符，有 static_cast、dynamic_cast、reinterpret_cast 和 const_cast 等 4 种取值，分别表示 4 种不同目的的转换风格。type 是类型说明符，表明要转换的目标类型。expression 是被转换的表达式。该格式意思是：在 cast_name 方式下，将 expression 的类型转换为 type 型。

4 种转换方式分别对应不同的目的，下面将逐一介绍。

1．static_cast型

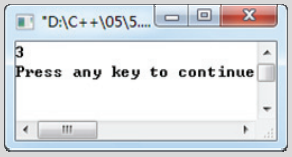
static_cast 表示静态的转换方式，发生时机是在编译时。它可以显式完成编译器隐式执行的任何类型转换，屏蔽潜在精度损失产生的编译器警告，避免不必要的隐式转换。在功能上基本与 C 风格的类型转换一样，含义也一样。但不能用于指针类型转换，不能转换掉 expression 的 const、volitaleconst 属性，也不会执行相应的构造函数。

这种类型转换发生在编译时，将会用到目标类型的类型信息。因此在转换执行时，系

统会进行必要的检测，诸如指针越界、类型检查等。所以其操作是安全的。

【示例 5-6】演示 static_cast 的用法。

```
01  #include <iostream.h>
02  int main(void)
03  {
04      double dVal=3.14;                    //圆周率
05      int iVal=static_cast<int>(dVal);//静态的类型转换
06      cout<<iVal<<endl;                    //输出 iVal 的值
07      return 0;
08  }
```



分析：该示例将 double 型变量静态地转换为 int 型。

2．dynamic_cast型

dynamic_cast 表示动态地转换方式。它依赖于 RTTI（Run-Time Type Information，表示运行时类型信息，指运行时类型检查）。它提供了运行时确定对象类型的方法，支持运行时识别指针或引用所指向的对象。利用 RTTI 功能在运行时检查指针类型，在类的指针向下转换时是安全的，能识别类型。可以用它把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用，而且能知道转换是否成功。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型转换时）。它还能检查转换的源对象是否真的可以转换成目标类型，这种检查不是语法上的，而是真实情况的检查。

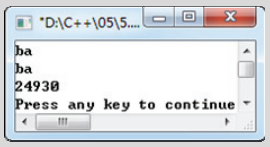
3．reinterpret_cast型

reinterpret_cast 表示重新解释的转换方式，常用于函数指针类型之间进行的转换，能够在非相关的类型之间转换。其操作结果只是简单地从一个指针到别的指针值的二进制复制，仅仅是重新解释了对象的内存表示，对类型之间指向的内容不做任何类型的检查和转换。

警告：不要对动态对象指针使用 static_cast，因为它是编译时进行的，不能检查运行时的非法强制类型转换。可用 dynamic_cast 代替，甚至重新设计。不鼓励使用 reinterpret_cast 强制编译器将某个类型的内存表示重新解释成另一种类型。这违反了类型安全的原则，也不保证一定能达到目的。除非程序员确定了解全部细节，否则不要这么用。

【示例 5-7】演示 reinterpret_cast 转换的典型用法。

```
01  #include <iostream.h>
02  int main(void)
03  {
04      char cval[]="ba";          //定义字符数组变量
05      char *p1=cval;             //定义指针变量
06      short int *p2= reinterpret_cast<short int*>(p1);   //强制类型转换
07      cout<<cval[0]<<cval[1]<<endl;//输出字符数组的原值
08      cout<<p1<<endl;            //用指针输出字符数组的内容
09      cout<<*p2<<endl;           //用指针输出强制转换后的内容
10      return 0;
11  }
```



分析：程序员必须明白 p2 指向的真实对象是字符数组，而不是 short int 型。由于 short int 型占两个字节，因此*p2 的输出将是 cval 前两个字节组成的 short int 数。

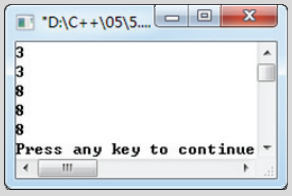
其中，24930 的二进制是 01100001 011000102。011000012 是 97，011000102 是 98，这恰好是字符‘a’和‘b’的 ASCII 码。内存里实际放的是 01100010 011000012，之所以两者顺序相反，是因为对于 short int 来说，高字节在后，低字节在前。

4．const_cast型

const_cast 表示常量的类型转换，它用来转换掉对象的 const 或 volatileness 属性。任何修改 const 或 volatileness 之外的属性的企图都将被拒绝。

【示例 5-8】演示 const_cast 型转换的方法。

```
01  #include <iostream.h>
02  int main(void)
03  {
04      int ival=3;
05      const int * p1=&ival;              //引用
06      cout<<ival<<endl;                  //直接输出
07      cout<<*p1<<endl;                   //间接输出
08      int *p2=const_cast<int *>(p1);     //强制转换
09      *p2=8;                             //给转换后的内容赋值
10      cout<<ival<<endl;                  //直接输出转换前的内容
11      cout<<*p1<<endl;                   //间接输出转换前的内容
12      cout<<*p2<<endl;                   //间接输出强制转换后的内容
13      return 0;
14  }
```



分析：该示例将指针 p1 的 const 属性去掉并赋给了 p2。p1 是常量指针，不能通过它修改 ival 的值。但转换后，可以通过 p2 来修改 ival 的值。

5.3　小　　结

本章主要讲解了表达式的分类、意义和书写。重点是对类型转换的理解，难点是如何正确使用显示类型转换。第 6 章将讲解如何由表达式构成语句。

5.4　习　　题

1．考虑下列情况是否会有类型转换发生？

（1）角度的弧度制表示 π/2（90°）。

（2）double hair_count = 14 0000。

2．编写程序实现摄氏温度向华氏温度的转换。

华氏温度 f，摄氏温度 c，转换表达式：f = c * 9 / 5 + 32;

3．在如下代码中加入合适的强制类型转换。

```
(1) const float _PI_ = 3.14;
    int high = 5;
    double r = 4.3;
    area = _PI_ * r * r * 2.0 + 2.0 * _PI_ * r * high;

(2) int ivalue = 100;
    char c = 'a';
    const double d = ivalue;
    int *i = c;
```