第 3 章 表达式

【学习内容】

本章介绍 C++ 对数据进行处理的最基本的手段——运算操作和表达式。内容包括：

- C++ 表达式的基本概念。
- 各种运算符与表达式。
- 运算优先级和结合律。
- 表达式语句。
- C++ 的类型转换。

【学习目标】

- 熟练运用各种运算符与表达式。
- 理解运算优先级和结合律。
- 掌握类型转换的方法。

在第 2 章中，我们了解了如何在 C++ 程序中表示整数、浮点数等基本数据，以及程序输入输出方式等，但还不清楚如何对数据做进一步的操作。例如，要编写程序求解一个一元二次方程的实数根，如何对系数做运算，这是本章需要解决的问题。

3.1 表达式基础

3.1.1 基本概念

表达式(expression)由一个或多个操作数(operand)及运算符(operator)组成，对表达式求值将得到一个结果。常量和变量是最简单的表达式，其结果就是常量和变量的值。通过运算符把一个或多个操作数组合起来可以构造各种表达式。程序通过计算表达式完成对数据的处理。

C++ 语言提供了丰富的运算符(又称操作符)以实现各种运算功能，主要包括算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符等。除此之外，还有一些用于完成特殊任务的运算符。在第 5 章中，还会看到表达式可以包括函数调用，即把函数调用的返回值作为操作数参与运算。

按照操作数的数量，可以把运算符分为一元运算符、二元运算符、多元运算符。例如，对于取地址符号(&)有一个操作数，所以是一元运算符；小于符号(<)有两个操作数，所以是二元

运算符。函数调用这种特殊的运算符对操作数的数量没有限制。

同一运算符在不同的场景下有不同的含义。例如，对于星号(*)，通常表示乘法运算(二元运算符)，当用于指令运算时又表示解引用运算(一元运算符)。其实，对于这两种情况，可以看作两个完全不同的运算符。

3.1.2 优先级和结合律

表达式与操作数之间可以通过运算符进一步组合，构成更为复杂的表达式，称为复合表达式(compound expression)。复合表达式是指含有两个和多个运算符的表达式。例如：

```
16+9.8*15
```

计算复合表达式时，运算优先级和结合律共同决定了运算对象的组合方式，运算符的优先级规定了不同运算符出现在同一表达式中优先运算的级别，而结合性质则规定了同等优先级的运算符出现在同一表达式中运算的顺序。高优先级运算符的运算对象要比低优先级运算符的运算对象更加紧密地组合在一起。如果优先级相同，则组合规则由结合律确定。其中，右结合是指对于同等优先级的操作符从右至左依次计算，而左结合则表示从左至右依次计算。

附录 A 的“C++ 运算符的优先级和结合性”给出了各种运算符的计算优先级和结合性质，计算机计算表达式时总是按照该表的规定进行的。例如，乘除法优先级高于加减法，则意味着乘除法的运算对象会比加减法先进行运算。加、减、乘、除等算术运算符满足左结合律，这意味着如果运算符的优先级相同，将按照从左到右的顺序组合运算对象。例如：

根据运算符的优先级，表达式 1+2*3 的值是 7，而不是 9；

根据运算符的结合律，表达式 10-5-3 的值是 2，而不是 8。

尤其注意，括号无视优先级和结合律，表达式中括号括起来的部分被当成一个整体先求值，然后再与其他部分按优先级和结合律进行组合。例如：

```
cout << (6+3) * (4/2) << endl;                    //输出为 18
cout << 6+3 * (4/2) << endl;                      //输出为 12
```

编程风格建议：当表达式较为复杂，或者对于运算优先级和结合律不太确定时，应该考虑多使用括号，一方面保证表达式的正确性，另一方面提高程序的可读性，方便他人理解程序的功能。

3.2 运算符和表达式

下面介绍具体的运算符和表达式。根据组成表达式的运算符的不同，表达式可以分为算术表达式、关系表达式、逻辑表达式、位运算表达式、逗号表达式和赋值表达式等。

3.2.1 算术运算

1. 基本的算术运算

基本的算术运算符主要有以下 5 种。

- +：加法，运算结果为两个操作数的和。
- -：减法，运算结果为左操作数减去右操作数的差。

- *：乘法，运算结果为两个操作数的乘积。
- /：除法，运算结果为左操作数除以右操作数的商。
- %：求余，运算结果为左操作数除以右操作数的余数。

这 5 种运算符都是二元运算符，要求有两个操作数，即左操作数和右操作数。单独的常量和变量是表达式，任何表达式又可以成为操作数。表达式和运算符可以构成更复杂的表达式。在表达式中，运算符之间的优先关系与数学上的规定类似。

(1) 括号中的表达式优先级最高，有括号嵌套时，内层括号优先于外层括号，由内层向外层求值。

(2) 乘法、除法和求余运算优先级次之，三者的优先级相同。如果表达式中有连续的乘法、除法和求余运算，则遵循左结合原则，从左算到右。

(3) 加法和减法优先级较低，二者优先级相同。如果表达式中有连续的加法和减法运算，也遵循左结合原则，从左算到右。

数据类型包括两方面的含义：值的集合和数据上的运算。上述运算符中，+、-、* 和/可以施加于整数和浮点类型上，分别代表整数和实数上的加、减、乘、除运算，其结果类型与操作数的类型相同。求余运算%只用于整型数据，其结果也是整型。

注意，运算符/用于整数和实数的含义是不一样的(即整数和实数的除法运算都用同一个运算符号/来表示，这种现象称为重载)。两个实数做/运算，其结果为实数；两个整数做/运算，其结果为整数，其具体结果取决于 C++ 在具体机器上的实现：一般来说，大多数实现都采取“向零取整”，即直接截去商的小数部分。

例如：

```
5/3    结果为 1
-5/3   结果为-1
5/2    结果为 2
-5/2   结果为-2
```

整型数据的求余运算%需注意，其运算结果在不同 C++ 的实现中也不一样。C++ 规定，如果两个操作数都是非负的，那么结果一定是非负的；否则，要看 C++ 的具体实现，C++ 标准对这种情况未进行明确规定，例如在 Microsoft C++ 实现中，求余运算%的结果总是与左操作数的符号一致。这样，整除运算和求余运算的关系是：

```
e1 % e2 = e1 - (e1/e2) * e2
```

下面是一些例子。

```
11/3    结果为 3                    11%3    结果为 2
-11/3   结果为-3                    -11%3   结果为-2
11/-3   结果为-3                    11% -3  结果为 2
-11/-3  结果为 3                    -11% -3 结果为-2
```

对于+、-、* 和/来说，如果两个操作数的类型不同，C++ 会对操作数做隐式类型转换，使得两个操作数具有相同的数据类型，详见后面的 3.3 节。具体做法是，总是将类型较低的操作数转换成较高的数据类型，运算结果也具有较高的数据类型。例如：

```
5/2 结果为整型的 2
5/2.0 先将 5 转换成 double 类型，结果为 double 类型的 2.5
```

【例 3-1】 温标转换。摄氏温标和华氏温标是目前国际使用最为广泛的两种计量温度的标准。摄氏温标是 1740 年瑞典人摄尔修斯(Celsius)提出的,在标准大气压下以水作为测温物质,以 C 为表示符号。华氏温标是 1714 年德国人华伦海特(Fahrenheit)创立的温标,以水银作为测温物质,用 F 为表示符号。摄氏温标转华氏温标的计算方法是:

$$F=9\div5\times C+32$$

由此,如下编写温标转换程序。

```
//ex3_1.cpp:把摄氏温标转换成对应华氏温标
#include <iostream>
using namespace std;

int main()
{
    float C, F;
    cout << "Input the temperature with Centigrade:";
    cin >> C;
    F = 9.0 / 5 * C + 32;                          //温标转换
    cout << "The Fahrenheit value is:" << F << endl;
    return 0;
}
```

程序运行结果:

```
Input the temperature with Centigrade:37.2
The Fahrenheit value is:98.96
```

本程序的实现思路较为直接,C 和 F 是两个浮点类型的变量,分别表示对应的摄氏温度和华氏温度,读入要转换摄氏温度后,参照计算公式通过一个复合表达式实现求解,然后输出结果。

但在这个程序中,需要特别注意的是,如果表达式跟计算公式一致(即 F=9/5 * C+32),由于除法和乘法优先级相同,并且是左结合的,所以会先执行除法。对于表达式"9/5",由于被除数和除数都是整数,所以执行的是整数除法,即结果为整数 1,而不是意料之中的 1.8,所以这种写法是错误的。为了解决这个问题,上述代码使用的是"F=9.0/5 * C+32",由于被除数是浮点数,按照类型转换的原则和结果(见 3.3 节),此时会执行浮点除法,得到期望值 1.8。

2. 其他算术运算

除了上面介绍的二元运算符,C++ 还有一元运算符,这些运算符只有一个操作数。如取正运算+和取负运算-,一元取正运算+和二元加法运算的符号是相同的,取正运算的结果就是操作数本身,而一元取负运算的结果则是操作数的负数。

C++ 还有两个使用起来非常方便、灵活的算术运算符:自增运算符++和自减运算符--。这两个运算符都是一元运算符,只有一个操作数,而且该操作数必须具有左值性质(即该操作数具有对应的内存地址,其值可以被修改)。它们的功能相对复杂一些,而且++和--出现在操作数之前和之后具有不同的功能,具体见表 3-1。

表 3-1　算术运算符++和--的功能运算符名称示例

运算符	名　称	示　例	说　明
++	前自增	++a	将 a 加 1,a 增加后的新值为运算的结果
++	后自增	a++	将 a 加 1,a 增加前的旧值为运算的结果
--	前自减	--a	将 a 减 1,a 减少后的新值为运算的结果
--	后自减	a--	将 a 减 1,a 减少前的旧值为运算的结果

自增运算和自减运算的优先级相同,它们都比括号的优先级低,但比加法、减法、乘法、除法和求余运算要高。下面给出几个示例,假设变量 a 的当前值为 5,那么有

```
++a + 12     结果为 18(运算后 a 为 6)
a++ + 12     结果为 17(运算后 a 为 6)
--a * 12     结果为 48(运算后 a 为 4)
a-- * 12     结果为 60(运算后 a 为 4)
```

以上代码确实令人困惑,实际程序中很少出现这类代码。自增、自减运算一般单独使用,不用在复合表达式中。

自增、自减运算符提示: 虽然传统的 C 语言程序员经常使用后自增、后自减,但除非必要,一般情况下优先考虑使用前置版本,即前自增和前自减。首先是消除不必要的歧义,通常使用自增、自减运算的初衷是把操作数加 1 或减 1,不会使用自增、自减运算之前的结果。其次,编译器为了保存自增、自减运算之前的结果,需要付出一定的开销,对于整数等简单类型来说代价不大,但对于后面介绍的基于面向对象方法实现的迭代器等类型来说,这个额外的消耗成本是很大的。

3.2.2　关系运算

现实世界中通常需要对数据进行比较,为此 C++ 也提供了一组关系运算符,以实现对数据进行关系比较,包括<、>、<=、>=、==和!=等,它们分别计算数据的小于、大于、小于或等于、大于或等于、相等和不相等关系。关系运算的结果反映了操作数的大小关系,为布尔类型,即 true 和 false。下面是一些关系运算表达式示例。

```
1 <= 0     结果为 false
x != x+1   结果总为 true
x > y      结果表示 x 是否大于 y
```

需要注意,"相等比较"运算符是连续两个等号(即==),对于<=、>=、==和!=,这些运算符都是由两个字符组成,书写时两个字符间不能加空格。

3.2.3　逻辑运算

前面的关系运算能够得到布尔型的结果,即 true 和 false。但是,仅有这些简单的条件测试不能满足现实世界中复杂的逻辑计算问题。例如,当需要判断各种复杂的组合条件时,还必须提供逻辑运算的能力。

C++ 提供了 3 种逻辑运算符:二元逻辑与运算符(&&)、二元逻辑或运算符(||)和一元逻辑非运算符(!)。表 3-2 是表示 3 种逻辑运算的真值表。

表 3-2　逻辑运算的真值表

a	b	a && b	a || b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

通过这 3 种逻辑运算符的组合，可以构造出非常复杂的条件，以实现强大的逻辑测试功能。3 种逻辑运算中，逻辑非(!)的优先级最高，逻辑与(&&)次之，逻辑或(||)最低。在结合性质上，逻辑非(!)是右结合，而逻辑与(&&)和逻辑或(||)为左结合。关系表达式是一种基本的逻辑表达式。下面是一组逻辑表达式的例子。

```
(grade >= 60) && (grade <= 70)
```

表示测试 grade 是否在 60 和 70 之间(包括 60 和 70)。

在计算逻辑表达式时，C++ 语言遵循一种所谓的“短路计算法”，即如果按照逻辑运算符的优先顺序和结合性质，只要计算部分表达式就可以确定整个表达式的结果，就不再计算表达式的剩余部分。以上面的表达式为例，计算(grade >= 60) && (grade <= 70)时，如果 grade 是小于 60 的，&& 左边的子表达式(grade >= 60)的结果为 false，此时无须计算右边的子表达式(grade <= 70)，就可以知道整个表达式为 false。对于 a&&b 来说，只有当计算了 a 得到的结果为 true 时，才需要计算 b 的值，以确定整个表达式的值。与此类似，对于逻辑或运算(||)，只有当左操作数的结果为 false 时，才需要计算右操作数的值，该值就是整个表达式的结果。

【例 3-2】　判断闰年：用户输入一个年份，编程判断该年份是否为闰年。

满足闰年的条件是：年份是 4 的倍数，但不是 100 的倍数，或者是 400 的倍数。判断年份是否是某数倍数(即整除)，则可以用年份对该数做求余运算，如果余数为 0，则是倍数关系。然后再通过逻辑运算符把这些条件组合起来进行判断。

由此，如下编写判断闰年程序。

```
//ex3_2.cpp:判断是否为闰年
#include <iostream>
using namespace std;

int main()
{
    int year;
    cout << "请输入一个年份:";
    cin >> year;
    if (((year % 4) == 0) && ((year % 100) != 0) || ((year % 400) == 0))
        cout << "该年份是闰年." << endl;
    else
        cout << "该年份不是闰年." << endl;
    return 0;
}
```

下面是两种可能的运行结果:

```
请输入一个年份:2020
该年份是闰年.
请输入一个年份:2022
该年份不是闰年.
```

测试用户输入的年份保存在变量 year,然后再通过下面的表达式判断是否是闰年。

```
((year % 4) == 0) && ((year % 100) != 0) || ((year % 400) == 0)
```

若表达式结果为 true,则满足闰年条件。其中出现了求余运算、关系运算和逻辑运算等多种运算符。初学 C++ 时,如果不确定运算的优先级,可以参照示例代码,多使用括号以确保正确性。比较熟练之后,有些括号可以省略。本例还使用了 if-else 选择结构,关于选择结构详见 4.3 节。此外,从完整性角度出发,本例应该要判断用户输入年份取值的合法性。

需要说明的是,C 和 C++ 程序在做逻辑判断时,可以把任意非零值当作 true(全零为 false)。所以,如果在 C 和 C++ 程序中看到类似下面的代码,不要奇怪。

```
double x;
…
if (x)                                          //x≠0
    cout << 1 / x << endl;
```

上述程序的判断条件是,如果 x≠0,则条件为“真”,计算并输出 x 的倒数。

从 C++ 11 标准开始,C++ 新引入了关键字 and、or、not,也可以实现逻辑与、或、非运算,即其作用等同于运算符 &&、||、!。例如:

```
(grade >= 60) and (grade <= 70)
```

也是判断 grade 是否在 60 和 70 之间(包括 60 和 70)。

3.2.4 位运算

位运算是 C 语言的重要特色之一,C++ 保留了这一特色。位运算允许在二进制位级别上对数据进行处理。各种位运算符说明如表 3-3 所示。

表 3-3 各种位运算符说明

运 算 符	名 称	示 例	说 明
&	按位与	a & b	a 和 b 的每一位做与运算
|	按位或	a | b	a 和 b 的每一位做或运算
^	按位异或	a ^ b	a 和 b 的每一位做异或运算
~	按位取反	~ a	将 a 的每一位取反
<<	向左移位	a << b	将 a 的每一位向左移 b 位
>>	向右移位	a >> b	将 a 的每一位向右移 b 位

这些运算符中,除了~是一元运算符以外,其余的都是二元运算符,操作数都只能是整型

或字符型数据，不能为浮点型数据，结果也为整型或字符型。

1. 按位与(&)

& 运算结果的每一位是两个操作数二进制表示的对应位进行与运算的结果，即如果两个操作数的对应位都为 1，则结果的对应位也为 1，否则为 0。

例如，3 & 14 的结果为 2。具体计算过程如下。

```
3 的二进制表示：              00000011
14 的二进制表示：             00001110
3 & 14 结果的二进制表示：     00000010
```

2. 按位或(|)

| 运算结果的每一位是两个操作数二进制表示的对应位进行或运算的结果，即如果两个操作数的对应位都为 0，则结果的对应位也为 0，否则为 1。

例如，3 | 14 的结果为 15。具体计算过程如下。

```
3 的二进制表示：              00000011
14 的二进制表示：             00001110
3 | 14 结果的二进制表示：     00001111
```

3. 按位异或(^)

^ 运算结果的每一位是两个操作数二进制表示的对应位进行异或运算的结果，即如果两个操作数的对应位不相同，则结果的对应位为 1，否则为 0。

例如，3 ^ 14 的结果为 13。具体计算过程如下。

```
3 的二进制表示：              00000011
14 的二进制表示：             00001110
3 ^ 14 结果的二进制表示：     00001101
```

4. 按位取反(~)

~ 运算结果的每一位是操作数二进制表示的对应位进行取反运算的结果，即如果操作数的对应位为 0，则结果的对应位为 1，否则为 0。

```
例如，14 的二进制表示：       00001110
~14 结果的二进制表示：        11110001
```

5. 向左移位(<<)

<< 运算将左操作数的二进制表示向左移位，移动的位数就是右操作数的值，右端移出的空位填充 0，移位后的左操作数的值即为运算的结果。

例如，3 <<5 的具体计算过程如下。

```
3 的二进制表示：              00000011
3 << 5 的结果：               01100000
```

6. 向右移位(>>)

>> 运算将左操作数的二进制表示向右移位，移动的位数就是右操作数的值，移位后的左操作数的值即为运算的结果。左端移出的空位填充方式取决于左操作数的类型和具体的值：如果左操作数是无符号类型，或者是有符号类型但其值非负(最高位为 0)，那么高位填充 0；如

果左操作数是有符号类型,并且为负数(最高位为 1),那么高位填充的值取决于所用的计算机系统,有的 C++ 实现填充 0,有的填充 1。

例如,-7 >> 5 的具体计算过程如下。

```
-7 的二进制补码表示:        11111001
-7 >> 5 的结果:             00000111 (填充 0)
-7 >> 5 的结果:             11111111 (填充 1)
```

对于 &、|和^等二元位运算,如果参与位运算的操作数类型不同,系统将二者按右端对齐,并根据操作数的类型和值进行填充,使得两个操作数位数完全一样。例如,如果 a 是 short,占 2 字节,b 是 int,占 4 字节,则系统将按照 a 的符号位扩展成 4 字节,即如果 a 非负,则用 0 扩展;如果 a 为负数,则用 1 扩展。如果需扩展的操作数是无符号类型,则总是用 0 扩展。

位运算符的优先顺序为:按位取反~的优先级最高,向左移位<<和向右移位>>次之,然后依次是 &、^和|。

位运算为程序员提供了非常灵活的、直接对二进制位进行处理的手段。通过各种位运算的合理组合,可以实现很多有趣的功能,位运算的常见组合如表 3-4 所示。

表 3-4 位运算的常见组合

功 能	表 达 式	说 明
逐位清零	a & 0	将 a 的每一位清零
取指定的二进制位	a & 0X00FF	取 a 的低位字节
设置指定位	a | 0X00FF	将 a 的低位字节的每一位设置为 1
指定位翻转	a ^ 0X00FF	将 a 的低位字节的每一位置翻转,0 变 1,1 变 0
乘以 2 的 n 次幂	a << n	a 乘以 2^n
除以 2 的 n 次幂	a >> n	a 除以 2^n

需要注意,一种常见错误是将位运算和逻辑运算搞混。例如,将位与(&)和逻辑与(&&)、位或(|)和逻辑或(||)、位求反(~)和逻辑非(!)搞混。

此外,C++ 11 标准新引入了关键字 bitand、bitor、xor、compl,也可以分别实现按位与、按位或、按位异或、按位取反运算。

下面通过解决一个现实问题来演示 C++ 位运算的应用。常见的数字图像有彩色图像和灰度图像两种。彩色图像一般使用 RGB 颜色模型,其中 R(红色)、G(绿色)、B(蓝色)3 个分量的取值是[0,255]内的一个整数。如果用一个 32 位的无符号整数来表示一个像素的颜色,其二进制格式是 00000000BBBBBBBBGGGGGGGGRRRRRRRR,即 31~24 位取 0,23~16 位为蓝色分量,15~8 位为绿色分量,7~0 为红色分量。例如,RGB 格式的颜色值4 292 863,对应的二进制格式是 00000000 01000001 10000000 11111111,所以该颜色的 B 分量是 65,G 分量是 128,R 分量是 255。而灰度图像是把白色与黑色之间按对数关系分为若干等级,灰度取值是[0,255]内的一个整数。

对于把 RGB 格式的彩色转灰度,有一个著名的心理学公式:

$$Gray = R \times 0.299 + G \times 0.587 + B \times 0.114$$

现在编程实现把 RGB 彩色图像转换成灰度图像功能的程序,用户输入一个由 RGB 颜色

模型的无符号整数，程序输出对应的灰度值(也是无符号整数)。

问题分析：在该问题中，用户输入的颜色值是一个 32 位的无符号整数，关键是如何按照 RGB 颜色模型的格式分别提取出红色分量、绿色分量和蓝色分量。根据二进制位数分配，最直接的方法可通过二进制位运算实现提取 3 个分量值，例如把颜色值与 255(换算成二进制，低 8 位全为 1，其余位为 0)进行位与运算，即可提取出红色分量。对于绿色分量，可以把颜色值向右移动 8 位，即把 15～8 位移动到 7～0 位，再与 255 进行位与运算，得到结果。蓝色分量类似处理。然后，剩下的转换按照公式直接计算即可。

【例 3-3】 将彩色颜色值转换成灰度值。

```
//ex3_3.cpp:把按 RGB 格式输入彩色颜色值转换成灰度值
#include <iostream>
using namespace std;

int main()
{
    unsigned int color;
    unsigned short R, G, B, gray;
    cout << "Input the color using RGB pattern:";
    cin >> color;
    R = color & 255;                              //提取红色分量
    G = (color >> 8) & 255;                       //提取绿色分量
    B = (color >> 16) & 255;                      //提取蓝色分量
    gray = (unsigned short)(R * 0.299 + G * 0.587 + B * 0.114);
    cout << "The corresponding gray value is " << gray << endl;
    return 0;
}
```

程序运行结果：

```
Input the color using RGB pattern:4292863
The corresponding gray value is 158
```

由于红、绿、蓝 3 个分量和灰度值都小于 256，所以第 8 行声明为 unsigned short 类型的变量，第 11～13 行是按上述分析分别计算提取 3 个分量值，第 14 行按心理学公式计算得到的是一个浮点数，所以还需要把结果强制转换为 unsigned short 类型。

需要说明的是，提取红、绿、蓝 3 个分量的方法还有其他做法。例如，以 256 为进制单位来看待 RGB 颜色值，把颜色值对 256 求余可以得到红色分量(color % 256)，把颜色值除以 256 后再对 256 求余可以得到绿色分量(color / 256 % 256)，蓝色分量类似可得。

3.2.5 赋值运算

1. 赋值运算

赋值运算实现了对变量的赋值，即为已声明的变量赋给一个特定值。其功能是：先将赋值运算符右边表达式的值计算出来，将该值赋给赋值运算符左边的变量，并将该值作为赋值运算的结果。因此，赋值表达式 a=3+5 表示将 3 加 5 的和 8 赋给变量 a。

需要特别指出的是，C++ 的赋值运算符除了对变量进行赋值以外，作为一种运算符，还具有运算的结果，这是 C++ 与很多其他程序设计语言不同的地方。对于赋值表达式 a=3+5，整个表达式的计算结果也是 8。所以，可以连续使用赋值运算符。例如：

```
a = b = c = 33 + 67;
```

赋值运算符比大多数运算符的优先级都要低,且具有右结合性质,即相邻的赋值运算符从右算到左。所以,上述示例代码执行结束后,变量 a、b 和 c 的值都是 100。

赋值运算符“=”是二元运算符,也就是说有两个操作数。赋值运算符和这两个操作数一起构成了赋值表达式,其中左边的操作数必须具有左值的表达式。

左值(lvalue)和右值(rvalue)是来源于 C 语言的概念,简单来说,“=”左边的是左值,“=”右边的是右值。当一个数据对象被用作右值时,用的是数据对象的值(内容),当数据对象被用作左值时,用的是数据对象的身份(数据在内存中的位置)。

例如,设 i 是一个整型变量,对于表达式 i=i+1,右边的 i 用的就是其右值,左边 i 用的就是其左值,具体执行过程是:先从内存中读出 i 的取值(右值),把该值复制一份,与整数 1 做加法运算,运算的结果根据 i 的内存地址(左值)存储到内存对应的位置。

新手尤其需要注意上述赋值表达式,它是一种赋值运算,最后的结果就是变量 i 存储的值加一,不要与相等判断运算混淆,相等判断是逻辑运算符“==”。

需要指出的是,如果赋值运算中,赋值运算符两边的变量和表达式的类型不同,C++ 先隐含地将表达式的结果转换成变量的类型,再将转换的结果赋给变量。当然也可以显式地转换表达式的值的类型。因此,赋值运算的结果的类型总是与赋值运算符左边的变量的类型一致。例如,如果 a 为浮点变量,b 和 c 为整型变量,那么 a=b+c 计算过程为:先对整型变量 b 与 c 的值进行整数求和,然后将结果转变成浮点类型,再赋给浮点型的变量 a,同时,该浮点值也是整个表达式的值。下面的表达式具有同样的效果。

```
a = (float)(b + c)
```

3.3 节将会进一步介绍类型转换的功能。

2. 复合赋值运算

程序中经常需要对变量进行某种运算,然后把计算的结果再赋给该变量。这种复合操作可以通过复合赋值运算进行简化。复合赋值运算符包括 *=、/=、%=、+=、-=、>>=、<<=、&=、^=、|=。

注意,复合赋值运算符的两个字符之间不能留空格。复合赋值运算符的使用,都可以概括为下面的模式。

```
变量 操作符= 表达式
```

其功能相当于

```
变量 = 变量 操作符 表达式
```

具体操作是:先计算赋值运算符右边的表达式的值,将作为左操作数的变量的当前值与该值做操作符代表的运算,运算的结果复制给变量并作为整个表达式的结果。例如,假设 a 为整型变量,当前值为 10,那么 a += 18 / 3 的计算过程为:先计算 18 除以 3 的整商,得到整数 6,将变量 a 的当前值 10 与 6 求和,得到 16,再将 16 赋给变量 a,整个表达式的结果也是 16。在这个过程中,a 的值被使用了两次,第一次使用 a 的右值,第二次使用 a 的左值。

此外,C++ 11 标准新引入了关键字 and_eq、or_eq、xor_eq,功能分别与复合赋值运算符 &=、

|=、^=等价。

3. 变量初始化

在2.3.2节介绍过，C++允许在声明变量时通过“=”对变量进行初始化，给变量定初始值。这个初始值可以是一个表达式，但是要求这个表达式的值在编译时是可计算的，即要求该表达式的值是一个常量。下面是一些合法的带初始化的变量声明。

```
int i = 2;
double x = i + 1.0;
double a = i * x;
```

经过声明和初始化后，变量i的值为2，x的值为3.0，a的值为6.0。

在C++新标准中，允许使用花括号括起来的初始化值列表。例如：

```
double pi = {3.14};
int array[5] = {1, 2, 3, 4, 5};
```

其中，array是一个数组，初始化的结果是数组中5个元素值依次取1、2、3、4、5(详见第6章)。

3.2.6 逗号运算

C++提供了一种特殊的运算符——逗号运算符(,)。该运算符将两个表达式连接起来构成逗号表达式。逗号运算符是一个二元运算符，具有左结合性质，其优先级比前面介绍的运算都要低。其计算过程是：先计算逗号左边的表达式，再计算逗号右边的表达式，并且将右边的表达式的计算结果作为整个表达式的结果。例如：

```
12 + 4, 3 * 5 表达式的结果为15。
12 + 4, 3 * 5, 4 - 1 表达式的结果为3。
```

大多数情况，使用逗号表达式的目的是为了顺序计算多个表达式的值，而并非一定要获得逗号运算的结果。第4章将会看到逗号表达式常用于for循环语句中。

3.2.7 条件运算符

条件运算符是一个三目运算符。该运算的一般形式如下：

```
<表达式1> ? <表达式2> : <表达式3> ;
```

条件运算符的含义是：先计算<表达式1>的值，如果为true(非0)，则计算<表达式2>的值，并把该值作为整个表达式的值；如果表达式1的值为false(为0)，则直接计算<表达式3>的值，并把它作为整个表达式的值。

例如，假设浮点型变量grade的值为70，result为字符型变量，表达式

```
result = grade >= 60 ? 'P' : 'F'
```

的计算过程是：计算条件表达式grade >= 60的值，结果为true，所以直接将'P'作为赋值运算符右边表达式的值，并赋给result，该值也是整个表达式的计算结果。可见，利用条件运算符，可以根据条件完成不同的计算。

条件运算符提示：条件运算表达式允许嵌套使用，即<表达式2>和<表达式3>也可以

是一个条件运算表达式,但是随着嵌套层数的增加,代码的可读性极具下降,所以一般建议嵌套不超过两层。更复杂的条件判断嵌套应该使用第 4 章介绍的选择结构。

3.2.8 sizeof 运算符

sizeof 运算符返回一个表示式结果值或者一个类型名字所占的字节数,所得到的值是一个类型为 size_t 的无符号整数。例如:

```
cout << sizeof(3 + 5) << endl;                    //输出结果为 4
cout << sizeof(int) << endl;                      //输出结果也为 4
```

需要说明,sizeof 运算符用于一个表达式时,并不实际计算该表达式的值,只需确定表达式结果的类型。

3.3 类型转换

在表达式求解过程中,参与运算的操作数的类型可能不完全相同,所以常常需要由一种类型转换成另一种类型。C++ 允许不同类型的数据进行转换,即可以将一种数据类型的数据转换成另一种类型的数据。

但是,由于各种数据类型在表示范围和精度上是不同的,所以有的转换不会丢失数据的精度,而有的转换则会丢失数据的精度。例如,将 int 类型数据转换成 double 类型,不会导致数据的改变,而将 double 类型的数据转换成 int 类型,则会截去 double 的小数部分,从而可能改变数据的值;与此类似,将大的整数类型变为较小的整数类型,如 long 转换成 short,也可能改变数据的值。为此,C++ 规定了一个“提升规则”,说明如何保证当一种数据类型转换为另一种数据类型时不会丢失数据的精度。C++ 按照各种数据类型的表示范围和精度,将各种数据类型由高到低进行排序,如图 3-1 所示。

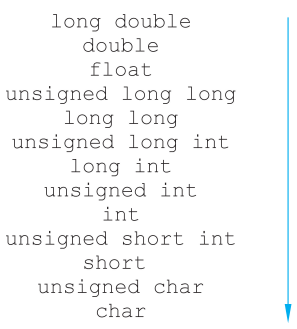

图 3-1 数据类型的排序

类型转换是指把一种类型的数据转换成另一种类型的数据,将数据转换成较低的类型可能导致取值不正确。C++ 的类型转换有两种:隐式类型转换和强制类型转换。C++ 语言要求,如果要将数据转换成较低的类型,必须显式地使用强制类型转换;如果将数据转换成较高的类型则可以通过隐式类型转换。

1. 隐式类型转换

隐式类型转换由系统自动隐含地进行。当表达式中操作数据的类型不同时,要进行隐式类型转换,使表达式中的数据类型相同。例如,在算术表达式和赋值表达式中类型不同时,就进行隐式类型转换。

下面是算术表达式中隐式类型转换规则。

(1) 表达式中如有 char、short 和 enum 类型的数据时,自动将它们转换成 int 类型。

(2) 把表达式中不同类型的数据转换成精度最高、占用内存最多的那个数据的类型。例如,3.14/2,由于被除数是浮点数,其精度比除数要高,所以在执行除法运算之前,首先把除数转换为浮点数,然后再执行浮点数的除法。

注意，正如 3.2.5 节的赋值表达式的介绍，自动将赋值运算符右边表达式值的类型转换成左边变量的类型，这时如果左边变量类型的精度低于右边表达式值的类型时，可能会丢失数据的精度，虽然这种转换是由系统自动进行的。

2. 强制类型转换

强制类型转换又称显式类型转换，它把表达式值的类型强制转换成指定的类型。早期的 C++ 语言使用 C 风格的强制类型转换，其一般形式如下：

```
(<目标数据类型>) <原数据类型的数据>
```

例如：

```
(int) 3.14
```

将 3.14 转换成整数 3。

在这种方式中，类型转换也可以看作一种单目运算符，其优先级比乘、除运算符要高。

C++ 语言也可以使用如下的类型转换形式。

```
<目标数据类型> (<原数据类型的数据>)
```

例如：

```
double (3)
```

将 3 转换成双精度浮点数 3.0。

这种方式把类型转换当成函数使用。两种类型转换方式等价。再看以下示例。

```
(double)3/2
```

或

```
double(3)/2
```

是先把整数 3 强制转换成双精度类型，再把 2 隐式转换成双精度类型，最后得到的值是双精度数 1.5。如果修改括号的位置，把上式写成

```
(double)(3/2)
```

或

```
double(3/2)
```

则先计算 3/2 得到整数值 1，再把 1 转换成双精度类型 1.0。

3. 命名类型转换

在类型转换（主要是强制类型转换）中，由精度高、占用内存多的数据类型转换成精度低、占用内存少的数据类型时，不仅改变数据的类型，也可能改变其值，所以使用类型转换时要非常小心。

C++ 语言为了进一步强调类型转换的风险，使得问题追溯更加方便，从 C++ 11 标准开始，为强制类型转换专门引入了 4 个关键字：static_cast、const_cast、reinterpret_cast 和

dynamic_cast,称为命名类型转换,其基本形式如下:

```
xxx_cast < <目标数据类型> > (<原数据类型的数据>)
```

例如:

```
double score = 95.6;
int n = static_cast<int>(score);
```

这两行代码的作用是,先初始化浮点变量 score 为 95.6,然后读取其值,构造一个副本,转换为整数 95(注意不是四舍五入),用该整数值初始化整型变量 n。所以,static_cast 与之前介绍的强制类型转换功能基本一致。但是 static_cast 要比传统强制类型转换更加严格,使用时编译器会对转换过程的安全性进行检查,尤其是对于后面将介绍的指针转换和对象之间的转换来说。

其他 3 种转换各有其适用场景,基本情况如表 3-5 所示。

表 3-5 命名类型转换说明

关 键 字	说 明
static_cast	用于风险较低的良性转换,一般不会导致意外发生
const_cast	用于 const 与非 const、volatile 与非 volatile 之间的转换
reinterpret_cast	最危险也是最灵活的类型转换,仅仅是对二进制位的重新解释,不会借助已有的转换规则对数据进行调整
dynamic_cast	主要用于 C++ 类层次间之的上行转换和下行转换

3.4 表达式语句

表达式语句是 C++ 最基本的语句。在任何表达式之后加上分号“;”,就是一个表达式语句。表达式语句的形式如下:

```
表达式;
```

表达式语句的功能是计算分号前表达式的结果,但该计算的结果并没有被再利用。需要特别说明的是,分号是表达式语句的组成部分。

常用的表达式语句是赋值语句和具有返回值的函数调用语句,其中赋值语句就是赋值运算表达式构成的语句,而函数调用语句就是由函数调用构成的语句,第 5 章中将介绍。

注意: C++ 允许仅以分号“;”构成一个语句,这种语句称作空语句。空语句仅起标识作用,不做任何事情,一般起到占位作用。

下面是一些表达式语句的示例。

```
a = a + 3;                          //赋值语句
x = y = z = 0;                      //多重赋值语句
t = 2, t + x + a;                   //逗号表达式语句
z = i <j ? x : x + y ;              //条件表达式语句
;                                   //空语句
```

```
f1();                                          //函数调用语句
x1 = exp(x);                                   //函数表达式语句,计算 e^x
x2 = pow(x, y);                                //函数表达式语句,计算 x^y
```

上面的 exp 和 pow 都是标准数学库函数,使用这些函数需要包含头文件 cmath。

【例 3-4】 求解一元二次方程 $ax^2+bx+c=0(a\neq0)$。

现在解决最开始提出的问题,求解一元二次方程 $ax^2+bx+c=0(a\neq0)$。众所周知,如果 b^2-4ac(一般称为 Δ)大于 0,则方程的两个根分别是:$(-b+\sqrt{\Delta})/2a$,$(-b-\sqrt{\Delta})/2a$。所以,首先求出 Δ 值,然后根据求根公式计算两个实根,下面是对应的代码。

```
//ex3_4.cpp:求一元二次方程的根
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double a, b, c;                            //存放 3 个系数
    double root1, root2;                       //存放 2 个实根
    cout << "Please input three coefficients in order: ";
    cin >> a >> b >> c;
    double delta;
    delta = b * b - 4 * a * c;
    root1 = (-b + sqrt(delta)) / (2 * a);
    root2 = (-b - sqrt(delta)) / (2 * a);
    cout << "Two real roots are: " << root1 << "\t" << root2 << endl;
    return 0;
}
```

程序运行结果:

```
Please input three coefficients in order: 1 -5 6
Two real roots are: 3   2
```

因为用到开根号函数,所以程序引入了数学库 cmath。需要说明的是,这里并没有考虑 Δ 小于或等于 0 和系数 a 等于 0 这些异常情况。如果要改进这一问题,需要引入第 4 章的判断选择语句。

习　　题

3.1　将下列数学表达式改写为程序表达式。

(1) $\dfrac{\cos x}{a+b+c}$　　(2) $\sqrt{(x+y)^2+(x-y)^2}$

(3) $\sin\alpha/\cos\beta+\tan\delta$　　(4) $\dfrac{|x-y|}{z+x}$

3.2　假设 a=4,b=10,c=2,计算下面算术表达式的值。

(1) a+b * c/(a+c)%3/a

(2) (float)(a+c)/3+(b-a)%a

(3) a = b = (c = a += 6)

3.3 假设 x=3, y=10, z=12,判断下面关系表达式或者逻辑表达式的真假。

(1) x-y>y-z　　(2) x<=y&&(x>0)||(y>0)

(3) !(x-y)>0||(y-z>0)　　(4) x+z==y||z-x<0

3.4 执行下列语句后,3 个变量 a、b、c 的值各为多少?

```
int  a,  b,  c;
a=20;
b=++a;
c=a++;
```

3.5 若 x=3,y=2,z=1,下列各式的结果是什么?

(1) x|y &z　　(2) x|y-z　　(3) x^y&-z　　(4) x <<=2　　(5) y << 2

3.6 若 x=1,y=-1,下列各式的结果是什么?

(1) !x|x　　(2) ~x|x　　(3) x^x　　(4) x <<=2　　(5) y << 2

3.7 设计一个程序,测试你的计算机是如何处理下面的移位运算的:

(1) 如果向左移位运算>>和向右移位运算<<的右操作数是负数,结果是什么?

(2) 如果向右移位运算<<的左操作数是有符号类型,并且为负数(最高位为 1),高位填充的值是 0 还是 1?

3.8 写出下面表达式运算后 a 的值,设原来 a 的初始值为 12。

(1) a+=a　　(2) a-=2

(3) a*=2+3　　(4) a/=a+a

(5) a%=(n%=2),n 的值等于 5　　(6) a+=a-=a*=a

3.9 先分析下面程序的代码,得出其预期运行结果,然后上机实际运行程序,对比验证与分析的结果是否一致。

```
#include <iostream>
using namespace std;
int main()
{
    int a = 3, b = 7, c, d;
    c = ++a+b++;
    d = (++a)+(++b);
    cout << a << '\t' << b << '\t' << c << '\t' << d << endl;
    return 0;
}
```

3.10 设计一个计算体重指数 BMI(Body Mass Index)的程序,BMI 是体重(kg)除以身高(m)的平方,从键盘输入一个人的体重和身高,计算其 BMI 指数并输出。

3.11 设计一个程序,从键盘输入一个圆的半径,求其周长和面积。

3.12 设计一个程序,从键盘输入一个圆柱底面圆的半径、圆柱的高,求其表面积和体积。

3.13 设计一个程序,从键盘输入一个 3 位正整数,程序逆序输出该整数。例如,若输入 123,则程序输出 321。

3.14 设计一个程序,输入一个整数,判断是否同时满足除 3 余 2、除 5 余 3 和除 7 余 2,如果满足则输出 Yes,否则输出 No。

3.15　设计一个程序，从键盘输入以秒数表示的时间段，然后换算成以天、小时、分钟和秒的组合方式来表示这个时间段，并打印输出。

3.16　美国汽车的油耗量指标是以 1 加仑（等于 3.785L）燃油的行驶里程（以 mile 为单位，100km 等于 62.14mile）来表示，单位为 mpg（miles per gallon），中国的汽车油耗量指标一般是行使 100km 消耗的燃油量，单位为 L/100km。所以，27mpg 约合 8.7L/100km。设计一个程序，用户输入美国标准的汽车油耗量指标，计算转换为中国标准的汽车油耗量，并输出结果。