

第4章 表达式

内容

4.1 基础	120
4.2 算术运算符	124
4.3 逻辑和关系运算符	126
4.4 赋值运算符	129
4.5 递增和递减运算符	131
4.6 成员访问运算符	133
4.7 条件运算符	134
4.8 位运算符	135
4.9 sizeof 运算符	139
4.10 逗号运算符	140
4.11 类型转换	141
4.12 运算符优先级表	147
小结	149
术语表	149

C++语言提供了一套丰富的运算符，并定义了这些运算符作用于内置类型的运算对象时所执行的操作。同时，当运算对象是类类型时，C++语言也允许由用户指定上述运算符的含义。本章主要介绍由语言本身定义、并用于内置类型运算对象的运算符，同时简单介绍几种标准库定义的运算符。第14章会专门介绍用户如何自定义适用于类类型的运算符。

134 表达式由一个或多个运算对象 (operand) 组成, 对表达式求值将得到一个结果 (result)。字面值和变量是最简单的表达式 (expression), 其结果就是字面值和变量的值。把一个运算符 (operator) 和一个或多个运算对象组合起来可以生成较复杂的表达式。

4.1 基础

有几个基础概念对表达式的求值过程有影响, 它们涉及大多数 (甚至全部) 表达式。本节先简要介绍这几个概念, 后面的小节将做更详细的讨论。



4.1.1 基本概念

C++ 定义了一元运算符 (unary operator) 和二元运算符 (binary operator)。作用于一个运算对象的运算符是一元运算符, 如取地址符 (&) 和解引用符 (*); 作用于两个运算对象的运算符是二元运算符, 如相等运算符 (==) 和乘法运算符 (*)。除此之外, 还有一个作用于三个运算对象的三元运算符。函数调用也是一种特殊的运算符, 它对运算对象的数量没有限制。

一些符号既能作为一元运算符也能作为二元运算符。以符号 * 为例, 作为一元运算符时执行解引用操作, 作为二元运算符时执行乘法操作。一个符号到底是一元运算符还是二元运算符由它的上下文决定。对于这类符号来说, 它的两种用法互不相干, 完全可以当成两个不同的符号。

组合运算符和运算对象

对于含有多个运算符的复杂表达式来说, 要想理解它的含义首先要理解运算符的优先级 (precedence)、结合律 (associativity) 以及运算对象的求值顺序 (order of evaluation)。例如, 下面这条表达式的求值结果依赖于表达式中运算符和运算对象的组合方式:

```
5 + 10 * 20/2;
```

乘法运算符 (*) 是一个二元运算符, 它的运算对象有 4 种可能: 10 和 20、10 和 20/2、15 和 20、15 和 20/2。下一节将介绍如何理解这样一条表达式。

运算对象转换

在表达式求值的过程中, 运算对象常常由一种类型转换成另外一种类型。例如, 尽管一般的二元运算符都要求两个运算对象的类型相同, 但是很多时候即使运算对象的类型不相同也没有关系, 只要它们能被转换 (参见 2.1.2 节, 第 32 页) 成同一种类型即可。

类型转换的规则虽然有点复杂, 但大多数都合乎情理、容易理解。例如, 整数能转换成浮点数, 浮点数也能转换成整数, 但是指针不能转换成浮点数。让人稍微有点意外的是, 小整数类型 (如 bool、char、short 等) 通常会被提升 (promoted) 成较大的整数类型, 主要是 int。4.11 节 (第 141 页) 将详细介绍类型转换的细节。

135

重载运算符

C++ 语言定义了运算符作用于内置类型和复合类型的运算对象时所执行的操作。当运算符作用于类类型的运算对象时, 用户可以自行定义其含义。因为这种自定义的过程事实上是为已存在的运算符赋予了另外一层含义, 所以称之为重载运算符 (overloaded operator)。IO 库的 >> 和 << 运算符以及 string 对象、vector 对象和迭代器使用的运算

符都是重载的运算符。

我们使用重载运算符时，其包括运算对象的类型和返回值的类型，都是由该运算符定义的；但是运算对象的个数、运算符的优先级和结合律都是无法改变的。

左值和右值



C++的表达式要不然是右值 (rvalue，读作“are-value”），要不然就是左值 (lvalue，读作“ell-value”）。这两个名词是从 C 语言继承过来的，原本是为了帮助记忆：左值可以位于赋值语句的左侧，右值则不能。

在 C++语言中，二者的区别就没那么简单了。一个左值表达式的求值结果是一个对象或者一个函数，然而以常量对象为代表的某些左值实际上不能作为赋值语句的左侧运算对象。此外，虽然某些表达式的求值结果是对象，但它们是右值而非左值。可以做一个简单的归纳：当一个对象被用作右值的时候，用的是对象的值（内容）；当对象被用作左值的时候，用的是对象的身份（在内存中的位置）。

不同的运算符对运算对象的要求各不相同，有的需要左值运算对象、有的需要右值运算对象；返回值也有差异，有的得到左值结果、有的得到右值结果。一个重要的原则（参见 13.6 节，第 470 页将介绍一种例外的情况）是在需要右值的地方可以用左值来代替，但是不能把右值当成左值（也就是位置）使用。当一个左值被当成右值使用时，实际使用的是它的内容（值）。到目前为止，已经有几种我们熟悉的运算符是要用到左值的。

- 赋值运算符需要一个（非常量）左值作为其左侧运算对象，得到的结果也仍然是一个左值。
- 取地址符（参见 2.3.2 节，第 47 页）作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是一个右值。
- 内置解引用运算符、下标运算符（参见 2.3.2 节，第 48 页；参见 3.5.2 节，第 104 页）、迭代器解引用运算符、`string` 和 `vector` 的下标运算符（参见 3.4.1 节，第 95 页；参见 3.2.3 节，第 83 页；参见 3.3.3 节，第 91 页）的求值结果都是左值。
- 内置类型和迭代器的递增递减运算符（参见 1.4.1 节，第 11 页；参见 3.4.1 节，第 96 页）作用于左值运算对象，其前置版本（本书之前章节所用的形式）所得的结果也是左值。

接下来在介绍运算符的时候，我们将会注明该运算符的运算对象是否必须是左值以及其求值结果是否是左值。

使用关键字 `decltype`（参见 2.5.3 节，第 62 页）的时候，左值和右值也有所不同。如果表达式的求值结果是左值，`decltype` 作用于该表达式（不是变量）得到一个引用类型。举个例子，假定 `p` 的类型是 `int*`，因为解引用运算符生成左值，所以 `decltype(*p)` 的结果是 `int&`。另一方面，因为取地址运算符生成右值，所以 `decltype(&p)` 的结果是 `int**`，也就是说，结果是一个指向整型指针的指针。

136

4.1.2 优先级与结合律



复合表达式 (compound expression) 是指含有两个或多个运算符的表达式。求复合表达式的值需要首先将运算符和运算对象合理地组合在一起，优先级与结合律决定了运算对象组合的方式。也就是说，它们决定了表达式中每个运算符对应的运算对象来自表达式的哪一部分。表达式中的括号无视上述规则，程序员可以使用括号将表达式的某个局部括起来使其得到优先运算。

一般来说，表达式最终的值依赖于其子表达式的组合方式。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为紧密地组合在一起。如果优先级相同，则其组合规则由结合律确定。例如，乘法和除法的优先级相同且都高于加法的优先级。因此，乘法和除法的运算对象会首先组合在一起，然后才能轮到加法和减法的运算对象。算术运算符满足左结合律，意味着如果运算符的优先级相同，将按照从左向右的顺序组合运算对象：

- 根据运算符的优先级，表达式 $3+4*5$ 的值是 23，不是 35。
- 根据运算符的结合律，表达式 $20-15-3$ 的值是 2，不是 8。

举一个稍微复杂一点的例子，如果完全按照从左向右的顺序求值，下面的表达式将得到 20：

```
6 + 3 * 4 / 2 + 2
```

也有一些人会计算得到 9、14 或者 36，然而在 C++ 语言中真实的计算结果应该是 14。这是因为这条表达式事实上与下述表达式等价：

```
// 这条表达式中的括号符合默认的优先级和结合律
((6 + ((3 * 4) / 2)) + 2)
```

括号无视优先级与结合律

括号无视普通的组合规则，表达式中括号括起来的部分被当成一个单元来求值，然后再与其他部分一起按照优先级组合。例如，对上面这条表达式按照不同方式加上括号就能得到 4 种不同的结果：

```
// 不同的括号组合导致不同的组合结果
cout << (6 + 3) * (4 / 2 + 2) << endl;           // 输出 36
cout << ((6 + 3) * 4) / 2 + 2 << endl;           // 输出 20
cout << 6 + 3 * 4 / (2 + 2) << endl;             // 输出 9
```

优先级与结合律有何影响

137 由前面的例子可以看出，优先级会影响程序的正确性，这一点在 3.5.3 节（第 107 页）介绍的解引用和指针运算中也有所体现：

```
int ia[] = {0,2,4,6,8}; // 含有 5 个整数的数组
int last = *(ia + 4); // 把 last 初始化成 8，也就是 ia[4] 的值
last = *ia + 4;        // last = 4，等价于 ia[0] + 4
```

如果想访问 $ia+4$ 位置的元素，那么加法运算两端的括号必不可少。一旦去掉这对括号， $*ia$ 就会首先组合在一起，然后 4 再与 $*ia$ 的值相加。

结合律对表达式产生影响的一个典型示例是输入输出运算，4.8 节（第 138 页）将要介绍 IO 相关的运算符满足左结合律。这一规则意味着我们可以把几个 IO 运算组合在一条表达式当中：

```
cin >> v1 >> v2; // 先读入 v1，再读入 v2
```

4.12 节（第 147 页）罗列出了全部的运算符，并用双横线将它们分割成若干组。同一组内的运算符优先级相同，组的位置越靠前组内的运算符优先级越高。例如，前置递增运算符和解引用运算符的优先级相同并且都比算术运算符的优先级高。表中同样列出了每个运算符在哪一页有详细的描述，有些运算符之前已经使用过了，大多数运算符的细节将在本章剩余部分逐一介绍，还有几个运算符将在后面的内容中提及。

4.1.2 节练习

练习 4.1：表达式 $5+10*20/2$ 的求值结果是多少？

练习 4.2：根据 4.12 节中的表，在下述表达式的合理位置添加括号，使得添加括号后运算对象的组合顺序与添加括号前一致。

- (a) `*vec.begin()` (b) `*vec.begin() + 1`

4.1.3 求值顺序



优先级规定了运算对象的组合方式，但是没有说明运算对象按照什么顺序求值。在大多数情况下，不会明确指定求值的顺序。对于如下的表达式

```
int i = f1() * f2();
```

我们知道 `f1` 和 `f2` 一定会在执行乘法之前被调用，因为毕竟相乘的是这两个函数的返回值。但是我们无法知道到底 `f1` 在 `f2` 之前调用还是 `f2` 在 `f1` 之前调用。

对于那些没有指定执行顺序的运算符来说，如果表达式指向并修改了同一个对象，将会引发错误并产生未定义的行为（参见 2.1.2 节，第 33 页）。举个简单的例子，`<<` 运算符没有明确规定何时以及如何对运算对象求值，因此下面的输出表达式是未定义的：

```
int i = 0;
cout << i << " " << ++i << endl; // 未定义的
```

因为程序是未定义的，所以我们无法推断它的行为。编译器可能先求 `++i` 的值再求 `i` 的值，此时输出结果是 1 1；也可能先求 `i` 的值再求 `++i` 的值，输出结果是 0 1；甚至编译器还可能做完全不同的操作。因为此表达式的行为不可预知，因此不论编译器生成什么样的代码程序都是错误的。

有 4 种运算符明确规定了运算对象的求值顺序。第一种是 3.2.3 节（第 85 页）提到的逻辑与 (`&&`) 运算符，它规定先求左侧运算对象的值，只有当左侧运算对象的值为真时才继续求右侧运算对象的值。另外三种分别是逻辑或 (`||`) 运算符（参见 4.3 节，第 126 页）、条件 (`?:`) 运算符（参见 4.7 节，第 134 页）和逗号 (`,`) 运算符（参见 4.10 节，第 140 页）。

求值顺序、优先级、结合律



运算对象的求值顺序与优先级和结合律无关，在一条形如 `f() + g() * h() + j()` 的表达式中：

- 优先级规定，`g()` 的返回值和 `h()` 的返回值相乘。
- 结合律规定，`f()` 的返回值先与 `g()` 和 `h()` 的乘积相加，所得结果再与 `j()` 的返回值相加。
- 对于这些函数的调用顺序没有明确规定。

如果 `f`、`g`、`h` 和 `j` 是无关函数，它们既不会改变同一对象的状态也不执行 IO 任务，那么函数的调用顺序不受限制。反之，如果其中某几个函数影响同一对象，则它是一条错误的表达式，将产生未定义的行为。

建议：处理复合表达式

139

以下两条经验准则对书写复合表达式有益：

1. 拿不准的时候最好用括号来强制让表达式的组合关系符合程序逻辑的要求。
2. 如果改变了某个运算对象的值，在表达式的其他地方不要再使用这个运算对象。

第2条规则有一个重要例外，当改变运算对象的子表达式本身就是另外一个子表达式的运算对象时该规则无效。例如，在表达式`*++iter`中，递增运算符改变`iter`的值，`iter`（已经改变）的值又是解引用运算符的运算对象。此时（或类似的情况下），求值的顺序不会成为问题，因为递增运算（即改变运算对象的子表达式）必须先求值，然后才轮到解引用运算。显然，这是一种很常见的用法，不会造成什么问题。

4.1.3 节练习

练习 4.3：C++语言没有明确规定大多数二元运算符的求值顺序，给编译器优化留下了余地。这种策略实际上是在代码生成效率和程序潜在缺陷之间进行了权衡，你认为这可以接受吗？请说出你的理由。

4.2 算术运算符

表 4.1：算术运算符（左结合律）

运算符	功能	用法
<code>+</code>	一元正号	<code>+ expr</code>
<code>-</code>	一元负号	<code>- expr</code>
<code>*</code>	乘法	<code>expr * expr</code>
<code>/</code>	除法	<code>expr / expr</code>
<code>%</code>	求余	<code>expr % expr</code>
<code>+</code>	加法	<code>expr + expr</code>
<code>-</code>	减法	<code>expr - expr</code>

表 4.1（以及后面章节的运算符表）按照运算符的优先级将其分组。一元运算符的优先级最高，接下来是乘法和除法，优先级最低的是加法和减法。优先级高的运算符比优先级低的运算符组合得更紧密。上面的所有运算符都满足左结合律，意味着当优先级相同时按照从左向右的顺序进行组合。

除非另做特殊说明，算术运算符都能作用于任意算术类型（参见 2.1.1 节，第 30 页）以及任意能转换为算术类型的类型。算术运算符的运算对象和求值结果都是右值。如 4.11 节（第 141 页）描述的那样，在表达式求值之前，小整数类型的运算对象被提升成较大的整数类型，所有运算对象最终会转换成同一类型。

一元正号运算符、加法运算符和减法运算符都能作用于指针。3.5.3 节（第 106 页）已经介绍过二元加法和减法运算符作用于指针的情况。当一元正号运算符作用于一个指针或者算术值时，返回运算对象值的一个（提升后的）副本。

一元负号运算符对运算对象值取负后，返回其（提升后的）副本：

```
int i = 1024;
int k = -i;      // k 是 -1024
bool b = true;
bool b2 = -b;    // b2 是 true!
```

在 2.1.1 节（第 31 页），我们指出布尔值不应该参与运算，`-b` 就是一个很好的例子。

对大多数运算符来说，布尔类型的运算对象将被提升为 `int` 类型。如上所示，布尔变量 `b` 的值为真，参与运算时将被提升成整数值 1（参见 2.1.2 节，第 32 页），对它求负后的结果是 -1。将 -1 再转换回布尔值并将其作为 `b2` 的初始值，显然这个初始值不等于 0，转换成布尔值后应该为 1。所以，`b2` 的值是真！

提示：溢出和其他算术运算异常

算术表达式有可能产生未定义的结果。一部分原因是数学性质本身：例如除数是 0 的情况；另外一部分则源于计算机的特点：例如溢出，当计算的结果超出该类型所能表示的范围时就会产生溢出。

假设某个机器的 `short` 类型占 16 位，则最大的 `short` 数值是 32767。在这样一台机器上，下面的复合赋值语句将产生溢出：

```
short short_value = 32767; // 如果 short 类型占 16 位，则能表示的最大值是 32767
short_value += 1;          // 该计算导致溢出
cout << "short_value: " << short_value << endl;
```

给 `short_value` 赋值的语句是未定义的，这是因为表示一个带符号数 32768 需要 17 位，但是 `short` 类型只有 16 位。很多系统在编译和运行时都不报溢出错误，像其他未定义的行为一样，溢出的结果是不可预知的。在我们的系统中，程序的输出结果是：

```
short_value: -32768
```

该值发生了“环绕（wrapped around）”，符号位本来是 0，由于溢出被改成了 1，于是结果变成一个负值。在别的系统中也许会有其他结果，程序的行为可能不同甚至直接崩溃。

当作用于算术类型的对象时，算术运算符 +、-、*、/ 的含义分别是加法、减法、乘法和除法。整数相除结果还是整数，也就是说，如果商含有小数部分，直接弃除：

```
int ival1 = 21/6;    // ival1 是 3，结果进行了删节，余数被抛弃掉了
int ival2 = 21/7;    // ival2 是 3，没有余数，结果是整数值
```

运算符 % 俗称“取余”或“取模”运算符，负责计算两个整数相除所得的余数，参与 取余运算的运算对象必须是整数类型：

```
int ival = 42;
double dval = 3.14;
ival % 12;           // 正确：结果是 6
ival % dval;         // 错误：运算对象是浮点类型
```

在除法运算中，如果两个运算对象的符号相同则商为正（如果不为 0 的话），否则商为负。C++ 语言的早期版本允许结果为负值的商向上或向下取整，C++11 新标准则规定商一律向 0 取整（即直接切除小数部分）。

根据取余运算的定义，如果 m 和 n 是整数且 n 非 0，则表达式 $(m/n) * n + m \% n$ 的求值结果与 m 相等。隐含的意思是，如果 $m \% n$ 不等于 0，则它的符号和 m 相同。C++ 语言的早期版本允许 $m \% n$ 的符号匹配 n 的符号，而且商向负无穷一侧取整，这一方式在新标准中已经被禁止使用了。除了 $-m$ 导致溢出的特殊情况，其他时候 $(-m) / n$ 和 $m / (-n)$ 都等于 $-(m/n)$ ， $m \% (-n)$ 等于 $m \% n$ ， $(-m) \% n$ 等于 $- (m \% n)$ 。具体示例如下：

C++
11

```

21 % 6;      /* 结果是 3 */
21 % 7;      /* 结果是 0 */
-21 % -8;    /* 结果是-5 */
21 % -5;     /* 结果是 1 */

21 / 6;       /* 结果是 3 */
21 / 7;       /* 结果是 3 */
-21 / -8;    /* 结果是 2 */
21 / -5;     /* 结果是-4 */

```

142

4.2 节练习

练习 4.4: 在下面的表达式中添加括号，说明其求值的过程及最终结果。编写程序编译该（不加括号的）表达式并输出其结果验证之前的推断。

```
12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
```

练习 4.5: 写出下列表达式的求值结果。

- | | |
|------------------------|-------------------------|
| (a) $-30 * 3 + 21 / 5$ | (b) $-30 + 3 * 21 / 5$ |
| (c) $30 / 3 * 21 \% 5$ | (d) $-30 / 3 * 21 \% 4$ |

练习 4.6: 写出一条表达式用于确定一个整数是奇数还是偶数。

练习 4.7: 溢出是何含义？写出三条将导致溢出的表达式。

4.3 逻辑和关系运算符

关系运算符作用于算术类型或指针类型，逻辑运算符作用于任意能转换成布尔值的类型。逻辑运算符和关系运算符的返回值都是布尔类型。值为 0 的运算对象（算术类型或指针类型）表示假，否则表示真。对于这两类运算符来说，运算对象和求值结果都是右值。

表 4.2：逻辑运算符和关系运算符

结合律	运算符	功能	用法
右	!	逻辑非	<code>!expr</code>
左	<	小于	<code>expr < expr</code>
左	<=	小于等于	<code>expr <= expr</code>
左	>	大于	<code>expr > expr</code>
左	>=	大于等于	<code>expr >= expr</code>
左	==	相等	<code>expr == expr</code>
左	!=	不相等	<code>expr != expr</code>
左	&&	逻辑与	<code>expr && expr</code>
左		逻辑或	<code>expr expr</code>

逻辑与和逻辑或运算符

对于逻辑与运算符 (`&&`) 来说，当且仅当两个运算对象都为真时结果为真；对于逻辑或运算符 (`||`) 来说，只要两个运算对象中的一个为真结果就为真。

逻辑与运算符和逻辑或运算符都是先求左侧运算对象的值再求右侧运算对象的值，当且仅当左侧运算对象无法确定表达式的结果时才会计算右侧运算对象的值。这种策略称为短路求值 (short-circuit evaluation)。

- 对于逻辑与运算符来说，当且仅当左侧运算对象为真时才对右侧运算对象求值。
- 对于逻辑或运算符来说，当且仅当左侧运算对象为假时才对右侧运算对象求值。

第3章中的几个程序用到了逻辑与运算符，它们的左侧运算对象是为了确保右侧运算对象求值过程的正确性和安全性。例如85页的循环条件：

```
index != s.size() && !isspace(s[index])
```

首先检查`index`是否到达`string`对象的末尾，以此确保只有当`index`在合理范围之内时才会计算右侧运算对象的值。

举一个使用逻辑或运算符的例子，假定有一个存储着若干`string`对象的`vector`对象，要求输出`string`对象的内容并且在遇到空字符串或者以句号结束的字符串时进行换行。使用基于范围的`for`循环（参见3.2.3节，第81页）处理`string`对象中的每个元素：

```
// s 是对常量的引用；元素既没有被拷贝也不会被改变
for (const auto &s : text) {           // 对于 text 的每个元素
    cout << s;                      // 输出当前元素
    // 遇到空字符串或者以句号结束的字符串进行换行
    if (s.empty() || s[s.size() - 1] == '.')
        cout << endl;
    else
        cout << " "; // 否则用空格隔开
}
```

输出当前元素后检查是否需要换行。`if`语句的条件部分首先检查`s`是否是一个空`string`，如果是，则不论右侧运算对象的值如何都应该换行。只有当`string`对象非空时才需要求第二个运算对象的值，也就是检查`string`对象是否是以句号结束的。在这条表达式中，利用逻辑或运算符的短路求值策略确保只有当`s`非空时才会用下标运算符去访问它。

值得注意的是，`s`被声明成了对常量的引用（参见2.5.2节，第61页）。因为`text`的元素是`string`对象，可能非常大，所以将`s`声明成引用类型可以避免对元素的拷贝；又因为不需要对`string`对象做写操作，所以`s`被声明成对常量的引用。

逻辑非运算符

逻辑非运算符`(!)`将运算对象的值取反后返回，之前我们曾经在3.2.2节（第79页）使用过这个运算符。下面再举一个例子，假设`vec`是一个整数类型的`vector`对象，可以使用逻辑非运算符将`empty`函数的返回值取反从而检查`vec`是否含有元素：

```
// 输出 vec 的首元素（如果说有的话）
if (!vec.empty())
    cout << vec[0];
```

子表达式

```
!vec.empty()
```

当`empty`函数返回假时结果为真。

关系运算符

顾名思义，关系运算符比较运算对象的大小关系并返回布尔值。关系运算符都满足左结合律。

因为关系运算符的求值结果是布尔值，所以将几个关系运算符连写在一起会产生意想不到的结果：

```
// 哎哟！这个条件居然拿 i < j 的布尔值结果和 k 比较！
if (i < j < k) // 若 k 大于 1 则为真！
```

if 语句的条件部分首先把 i、j 和第一个<运算符组合在一起，其返回的布尔值再作为第二个<运算符的左侧运算对象。也就是说，k 比较的对象是第一次比较得到的那个或真或假的结果！要想实现我们的目的，其实应该使用下面的表达式：

```
// 正确：当 i 小于 j 并且 j 小于 k 时条件为真
if (i < j && j < k) { /* ... */ }
```

相等性测试与布尔字面值

如果想测试一个算术对象或指针对象的真值，最直接的方法就是将其作为 if 语句的条件：

```
if (val) { /* ... */ } // 如果 val 是任意的非 0 值，条件为真
if (!val) { /* ... */ } // 如果 val 是 0，条件为真
```

144 在上面的两个条件中，编译器都将 val 转换成布尔值。如果 val 非 0 则第一个条件为真，如果 val 的值为 0 则第二个条件为真。

有时会试图将上面的真值测试写成如下形式：

```
if (val == true) { /* ... */ } // 只有当 val 等于 1 时条件才为真！
```

但是这种写法存在两个问题：首先，与之前的代码相比，上面这种写法较长而且不太直接（尽管大家都认为缩写的形式对初学者来说有点难理解）；更重要的一点是，如果 val 不是布尔值，这样的比较就失去了原来的意义。

如果 val 不是布尔值，那么进行比较之前会首先把 true 转换成 val 的类型。也就是说，如果 val 不是布尔值，则代码可以改写成如下形式：

```
if (val == 1) { /* ... */ }
```

正如我们已经非常熟悉的那样，当布尔值转换成其他算术类型时，false 转换成 0 而 true 转换成 1（参见 2.1.2 节，第 32 页）。如果真想知道 val 的值是否是 1，应该直接写出 1 这个数值来，而不要与 true 比较。



进行比较运算时除非比较的对象是布尔类型，否则不要使用布尔字面值 true 和 false 作为运算对象。

4.3 节练习

练习 4.8：说明在逻辑与、逻辑或及相等性运算符中运算对象求值的顺序。

练习 4.9：解释在下面的 if 语句中条件部分的判断过程。

```
const char *cp = "Hello World";
if (cp && *cp)
```

练习 4.10：为 while 循环写一个条件，使其从标准输入中读取整数，遇到 42 时停止。

练习 4.11：书写一条表达式用于测试 4 个值 a、b、c、d 的关系，确保 a 大于 b、b 大于 c、c 大于 d。

练习 4.12：假设 i、j 和 k 是三个整数，说明表达式 $i != j < k$ 的含义。

4.4 赋值运算符

赋值运算符的左侧运算对象必须是一个可修改的左值。如果给定

```
int i = 0, j = 0, k = 0;      // 初始化而非赋值
const int ci = i;            // 初始化而非赋值
```

则下面的赋值语句都是非法的：

```
1024 = k;                  // 错误：字面值是右值
i + j = k;                // 错误：算术表达式是右值
ci = k;                   // 错误：ci 是常量（不可修改的）左值
```

赋值运算的结果是它的左侧运算对象，并且是一个左值。相应的，结果的类型就是左侧运算对象的类型。如果赋值运算符的左右两个运算对象类型不同，则右侧运算对象将转换成左侧运算对象的类型：

```
k = 0;                     // 结果：类型是 int，值是 0
k = 3.14159;               // 结果：类型是 int，值是 3
```

C++11 新标准允许使用花括号括起来的初始值列表（参见 2.2.1 节，第 39 页）作为赋值语句的右侧运算对象：

```
k = {3.14};                // 错误：窄化转换
vector<int> vi;           // 初始为空
vi = {0,1,2,3,4,5,6,7,8,9}; // vi 现在含有 10 个元素了，值从 0 到 9
```

如果左侧运算对象是内置类型，那么初始值列表最多只能包含一个值，而且该值即使转换的话其所占空间也不应该大于目标类型的空间（参见 2.2.1 节，第 39 页）。

对于类类型来说，赋值运算的细节由类本身决定。对于 `vector` 来说，`vector` 模板重载了赋值运算符并且可以接收初始值列表，当赋值发生时用右侧运算对象的元素替换左侧运算对象的元素。

无论左侧运算对象的类型是什么，初始值列表都可以为空。此时，编译器创建一个值初始化（参见 3.3.1 节，第 88 页）的临时量并将其赋给左侧运算对象。

赋值运算满足右结合律

赋值运算符满足右结合律，这一点与其他二元运算符不太一样：

```
int ival, jval;
ival = jval = 0;             // 正确：都被赋值为 0
```

因为赋值运算符满足右结合律，所以靠右的赋值运算 `jval=0` 作为靠左的赋值运算符的右侧运算对象。又因为赋值运算返回的是其左侧运算对象，所以靠右的赋值运算的结果（即 `jval`）被赋给了 `ival`。

对于多重赋值语句中的每一个对象，它的类型或者与右边对象的类型相同、或者可由右边对象的类型转换得到（参见 4.11 节，第 141 页）：

```
int ival, *pval;           // ival 的类型是 int；pval 是指向 int 的指针
ival = pval = 0;            // 错误：不能把指针的值赋给 int
string s1, s2;
s1 = s2 = "OK";            // 字符串字面值"OK"转换成 string 对象
```

因为 `ival` 和 `pval` 的类型不同，而且 `pval` 的类型 (`int*`) 无法转换成 `ival` 的类型

(int)，所以尽管 0 这个值能赋给任何对象，但是第一条赋值语句仍然是非法的。

[146] 与之相反，第二条赋值语句是合法的。这是因为字符串字面值可以转换成 string 对象并赋给 s2，而 s2 和 s1 的类型相同，所以 s2 的值可以继续赋给 s1。

赋值运算优先级较低

赋值语句经常会出现条件当中。因为赋值运算的优先级相对较低，所以通常需要给赋值部分加上括号使其符合我们的原意。下面这个循环说明了把赋值语句放在条件当中有什么用处，它的目的是反复调用一个函数直到返回期望的值（比如 42）为止：

```
// 这是一种形式烦琐、容易出错的写法
int i = get_value();           // 得到第一个值
while (i != 42) {
    // 其他处理 .....
    i = get_value();           // 得到剩下的值
}
```

在这段代码中，首先调用 `get_value` 函数得到一个值，然后循环部分使用该值作为条件。在循环体内部，最后一条语句会再次调用 `get_value` 函数并不断重复循环。可以将上述代码以更简单直接的形式表达出来：

```
int i;
// 更好的写法：条件部分表达得更加清晰
while ((i = get_value()) != 42) {
    // 其他处理.....
}
```

这个版本的 `while` 条件更容易表达我们的真实意图：不断循环读取数据直至遇到 42 为止。其处理过程是首先将 `get_value` 函数的返回值赋给 `i`，然后比较 `i` 和 42 是否相等。

如果不加括号的话含义会有很大变化，比较运算符 != 的运算对象将是 `get_value` 函数的返回值及 42，比较的结果不论真假将以布尔值的形式赋值给 `i`，这显然不是我们期望的结果。



因为赋值运算符的优先级低于关系运算符的优先级，所以在条件语句中，赋值部分通常应该加上括号。

切勿混淆相等运算符和赋值运算符

C++语言允许用赋值运算作为条件，但是这一特性可能带来意想不到的后果：

```
if (i = j)
```

此时，`if` 语句的条件部分把 `j` 的值赋给 `i`，然后检查赋值的结果是否为真。如果 `j` 不为 0，条件将为真。然而程序员的初衷很可能是想判断 `i` 和 `j` 是否相等：

```
if (i == j)
```

程序的这种缺陷显然很难被发现，好在一部分编译器会对类似的代码给出警告信息。

[147] **复合赋值运算符**

我们经常需要对对象施以某种运算，然后把计算的结果再赋给该对象。举个例子，考虑 1.4.2 节（第 11 页）的求和程序：

```

int sum = 0;
// 计算从 1 到 10 (包含 10 在内) 的和
for (int val = 1; val <= 10; ++val)
    sum += val;      // 等价于 sum = sum + val

```

这种复合操作不仅对加法来说很常见，而且也常常应用于其他算术运算符或者 4.8 节（第 135 页）将要介绍的位运算符。每种运算符都有相应的复合赋值形式：

$+=$	$-=$	$*=$	$/=$	$\%=$	// 算术运算符
$<<=$	$>>=$	$\&=$	$\^=$	$ =$	// 位运算符，参见 4.8 节（第 135 页）

任意一种复合运算符都完全等价于

```
a = a op b;
```

唯一的区别是左侧运算对象的求值次数：使用复合运算符只求值一次，使用普通的运算符则求值两次。这两次包括：一次是作为右边子表达式的一部分求值，另一次是作为赋值运算的左侧运算对象求值。其实在很多地方，这种区别除了对程序性能有些许影响外几乎可以忽略不计。

4.4 节练习

练习 4.13：在下述语句中，当赋值完成后 *i* 和 *d* 的值分别是多少？

```

int i; double d;
(a) d = i = 3.5;      (b) i = d = 3.5;

```

练习 4.14：执行下述 if 语句后将发生什么情况？

```

if (42 = i) // ...
if (i = 42) // ...

```

练习 4.15：下面的赋值是非法的，为什么？应该如何修改？

```

double dval; int ival; int *pi;
dval = ival = pi = 0;

```

练习 4.16：尽管下面的语句合法，但它们实际执行的行为可能和预期并不一样，为什么？应该如何修改？

(a) if (*p* = getPtr() != 0) (b) if (*i* = 1024)

4.5 递增和递减运算符

递增运算符 (++) 和递减运算符 (--) 为对象的加 1 和减 1 操作提供了一种简洁的书写形式。这两个运算符还可应用于迭代器，因为很多迭代器本身不支持算术运算，所以此时递增和递减运算符除了书写简洁外还是必须的。

递增和递减运算符有两种形式：前置版本和后置版本。到目前为止，本书使用的都是前置版本，这种形式的运算符首先将运算对象加 1 (或减 1)，然后将改变后的对象作为求值结果。后置版本也会将运算对象加 1 (或减 1)，但是求值结果是运算对象改变之前那个值的副本：

```

int i = 0, j;
j = ++i;           // j = 1, i = 1: 前置版本得到递增之后的值
j = i++;          // j = 1, i = 2: 后置版本得到递增之前的值

```

这两种运算符必须作用于左值运算对象。前置版本将对象本身作为左值返回，后置版本则将对象原始值的副本作为右值返回。

建议：除非必须，否则不用递增递减运算符的后置版本

有 C 语言背景的读者可能对优先使用前置版本递增运算符有所疑问，其实原因非常简单：前置版本的递增运算符避免了不必要的工作，它把值加 1 后直接返回改变了的运算对象。与之相比，后置版本需要将原始值存储下来以便于返回这个未修改的内容。如果我们不需要修改前的值，那么后置版本的操作就是一种浪费。

对于整数和指针类型来说，编译器可能对这种额外的工作进行一定的优化；但是对于相对复杂的迭代器类型，这种额外的工作就消耗巨大了。建议养成使用前置版本的习惯，这样不仅不需要担心性能的问题，而且更重要的是写出的代码会更符合编程的初衷。

在一条语句中混用解引用和递增运算符

如果我们想在一条复合表达式中既将变量加 1 或减 1 又能使用它原来的值，这时就可以使用递增和递减运算符的后置版本。

举个例子，可以使用后置的递增运算符来控制循环输出一个 `vector` 对象内容直至遇到（但不包括）第一个负值为止：

```
auto pbeg = v.begin();
// 输出元素直至遇到第一个负值为止
while (pbeg != v.end() && *beg >= 0)
    cout << *pbeg++ << endl; // 输出当前值并将 pbeg 向前移动一个元素
```

对于刚接触 C++ 和 C 的程序员来说，`*pbeg++` 不太容易理解。其实这种写法非常普遍，所以程序员一定要理解其含义。

后置递增运算符的优先级高于解引用运算符，因此`*pbeg++` 等价于`*(pbeg++)`。`pbeg++` 把 `pbeg` 的值加 1，然后返回 `pbeg` 的初始值的副本作为其求值结果，此时解引用运算符的运算对象是 `pbeg` 未增加之前的值。最终，这条语句输出 `pbeg` 开始时指向的那个元素，并将指针向前移动一个位置。

149 这种用法完全是基于一个事实，即后置递增运算符返回初始的未加 1 的值。如果返回的是加 1 之后的值，解引用该值将产生错误的结果。不但无法输出第一个元素，而且糟糕的是如果序列中没有负值，程序将可能试图解引用一个根本不存在的元素。

建议：简洁可以成为一种美德

形如`*pbeg++` 的表达式一开始可能不太容易理解，但其实这是一种被广泛使用的、有效的写法。当对这种形式熟悉之后，书写

```
cout << *iter++ << endl;
```

要比书写下面的等价语句更简洁、也更少出错

```
cout << *iter << endl;
++iter;
```

不断研究这样的例子直到对它们的含义一目了然。大多数 C++ 程序追求简洁、摒弃冗长，因此 C++ 程序员应该习惯于这种写法。而且，一旦熟练掌握了这种写法后，程序出错的可能性也会降低。

运算对象可按任意顺序求值

大多数运算符都没有规定运算对象的求值顺序（参见 4.1.3 节，第 123 页），这在一般情况下不会有什么影响。然而，如果一条子表达式改变了某个运算对象的值，另一条子表达式又要使用该值的话，运算对象的求值顺序就很关键了。因为递增运算符和递减运算符会改变运算对象的值，所以要提防在复合表达式中错用这两个运算符。

为了说明这一问题，我们将重写 3.4.1 节（第 97 页）的程序，该程序使用 `for` 循环将输入的第一个单词改成大写形式：

```
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it);           // 将当前字符改成大写形式
```

在上述程序中，我们把解引用 `it` 和递增 `it` 两项任务分开来完成。如果用一个看似等价的 `while` 循环进行代替：

```
// 该循环的行为是未定义的!
while (beg != s.end() && !isspace(*beg))
    *beg = toupper(*beg++); // 错误：该赋值语句未定义
```

将产生未定义的行为。问题在于：赋值运算符左右两端的运算对象都用到了 `beg`，并且右侧的运算对象还改变了 `beg` 的值，所以该赋值语句是未定义的。编译器可能按照下面的任意一种思路处理该表达式：

```
*beg = toupper(*beg);           // 如果先求左侧的值
*(beg + 1) = toupper(*beg);    // 如果先求右侧的值
```

也可能采取别的什么方式处理它。

4.5 节练习

< 150

练习 4.17：说明前置递增运算符和后置递增运算符的区别。

练习 4.18：如果第 132 页那个输出 `vector` 对象元素的 `while` 循环使用前置递增运算符，将得到什么结果？

练习 4.19：假设 `ptr` 的类型是指向 `int` 的指针、`vec` 的类型是 `vector<int>`、`ival` 的类型是 `int`，说明下面的表达式是何含义？如果有表达式不正确，为什么？应该如何修改？

- | | |
|--|---|
| (a) <code>ptr != 0 && *ptr++</code> | (b) <code>ival++ && ival</code> |
| (c) <code>vec[ival++] <= vec[ival]</code> | |

4.6 成员访问运算符

点运算符（参见 1.5.2 节，第 21 页）和箭头运算符（参见 3.4.1 节，第 98 页）都可用于访问成员，其中，点运算符获取类对象的一个成员；箭头运算符与点运算符有关，表达式 `ptr->mem` 等价于 `(*ptr).mem`：

```
string s1 = "a string", *p = &s1;
auto n = s1.size();           // 运行 string 对象 s1 的 size 成员
n = (*p).size();             // 运行 p 所指对象的 size 成员
n = p->size();              // 等价于 (*p).size()
```

因为解引用运算符的优先级低于点运算符，所以执行解引用运算的子表达式两端必须加上括号。如果没加括号，代码的含义就大不相同了：

```
// 运行 p 的 size 成员，然后解引用 size 的结果
*p.size(); // 错误：p 是一个指针，它没有名为 size 的成员
```

这条表达式试图访问对象 p 的 size 成员，但是 p 本身是一个指针且不包含任何成员，所以上述语句无法通过编译。

箭头运算符作用于一个指针类型的运算对象，结果是一个左值。点运算符分成两种情况：如果成员所属的对象是左值，那么结果是左值；反之，如果成员所属的对象是右值，那么结果是右值。

4.6 节练习

练习 4.20：假设 iter 的类型是 `vector<string>::iterator`，说明下面的表达式是否合法。如果合法，表达式的含义是什么？如果不合法，错在何处？

- | | | |
|------------------------------------|-----------------------------|--------------------------------------|
| (a) <code>*iter++;</code> | (b) <code>(*iter)++;</code> | (c) <code>*iter.empty();</code> |
| (d) <code>iter->empty();</code> | (e) <code>++*iter;</code> | (f) <code>iter++->empty();</code> |

4.7 条件运算符

条件运算符 (`? :`) 允许我们把简单的 if-else 逻辑嵌入到单个表达式当中，条件运算符按照如下形式使用：

```
cond ? expr1 : expr2;
```

其中 `cond` 是判断条件的表达式，而 `expr1` 和 `expr2` 是两个类型相同或可能转换为某个公共类型的表达式。条件运算符的执行过程是：首先求 `cond` 的值，如果条件为真对 `expr1` 求值并返回该值，否则对 `expr2` 求值并返回该值。举个例子，我们可以使用条件运算符判断成绩是否合格：

```
string finalgrade = (grade < 60) ? "fail" : "pass";
```

条件部分判断成绩是否小于 60。如果小于，表达式的结果是"fail"，否则结果是"pass"。有点类似于逻辑与运算符和逻辑或运算符 (`&&` 和 `||`)，条件运算符只对 `expr1` 和 `expr2` 中的一个求值。

当条件运算符的两个表达式都是左值或者能转换成同一种左值类型时，运算的结果是左值；否则运算的结果是右值。

嵌套条件运算符

允许在条件运算符的内部嵌套另外一个条件运算符。也就是说，条件表达式可以作为另外一个条件运算符的 `cond` 或 `expr`。举个例子，使用一对嵌套的条件运算符可以将成绩分成三档：优秀 (high pass)、合格 (pass) 和不合格 (fail)：

```
finalgrade = (grade > 90) ? "high pass"
                      : (grade < 60) ? "fail" : "pass";
```

第一个条件检查成绩是否在 90 分以上，如果是，执行符号?后面的表达式，得到"high pass"；如果否，执行符号:后面的分支。这个分支本身又是一个条件表达式，它检查成绩是否在 60 分以下，如果是，得到"fail"；否则得到"pass"。

条件运算符满足右结合律，意味着运算对象（一般）按照从右向左的顺序组合。因此在上面的代码中，靠右边的条件运算（比较成绩是否小于 60）构成了靠左边的条件运算的分支。



随着条件运算嵌套层数的增加，代码的可读性急剧下降。因此，条件运算的嵌套最好别超过两到三层。

在输出表达式中使用条件运算符

条件运算符的优先级非常低，因此当一条长表达式中嵌套了条件运算子表达式时，通常需要在它两端加上括号。例如，有时需要根据条件值输出两个对象中的一个，如果写这 <152> 条语句时没把括号写全就有可能产生意想不到的结果：

```
cout << ((grade < 60) ? "fail" : "pass"); // 输出 pass 或者 fail  
cout << (grade < 60) ? "fail" : "pass"; // 输出 1 或者 0!  
cout << grade < 60 ? "fail" : "pass"; // 错误：试图比较 cout 和 60
```

在第二条表达式中，`grade` 和 60 的比较结果是 `<<` 运算符的运算对象，因此如果 `grade < 60` 为真输出 1，否则输出 0。`<<` 运算符的返回值是 `cout`，接下来 `cout` 作为条件运算符的条件。也就是说，第二条表达式等价于

```
cout << (grade < 60); // 输出 1 或者 0  
cout ? "fail" : "pass"; // 根据 cout 的值是 true 还是 false 产生对应的字面值
```

因为第三条表达式等价于下面的语句，所以它是错误的：

```
cout << grade; // 小于运算符的优先级低于移位运算符，所以先输出 grade  
cout < 60 ? "fail" : "pass"; // 然后比较 cout 和 60!
```

4.7 节练习

练习 4.21： 编写一段程序，使用条件运算符从 `vector<int>` 中找到哪些元素的值是奇数，然后将这些奇数值翻倍。

练习 4.22： 本节的示例程序将成绩划分成 `high pass`、`pass` 和 `fail` 三种，扩展该程序使其进一步将 60 分到 75 分之间的成绩设定为 `low pass`。要求程序包含两个版本：一个版本只使用条件运算符；另外一个版本使用 1 个或多个 `if` 语句。哪个版本的程序更容易理解呢？为什么？

练习 4.23： 因为运算符的优先级问题，下面这条表达式无法通过编译。根据 4.12 节中的表（第 147 页）指出它的问题在哪里？应该如何修改？

```
string s = "word";  
string p1 = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

练习 4.24： 本节的示例程序将成绩划分成 `high pass`、`pass` 和 `fail` 三种，它的依据是条件运算符满足右结合律。假如条件运算符满足的是左结合律，求值过程将是怎样的？

4.8 位运算符

位运算符作用于整数类型的运算对象，并把运算对象看成是二进制位的集合。位运算符提供检查和设置二进制位的功能，如 17.2 节（第 640 页）将要介绍的，一种名为 `bitset`

的标准库类型也可以表示任意大小的二进制位集合，所以位运算符同样能用于 `bitset` 类型。

153

表 4.3: 位运算符（左结合律）

表 4.3: 位运算符 (左结合律)

运算符	功能	用法
<code>~</code>	位求反	<code>~ expr</code>
<code><<</code>	左移	<code>expr1 << expr2</code>
<code>>></code>	右移	<code>expr1 >> expr2</code>
<code>&</code>	位与	<code>expr & expr</code>
<code>^</code>	位异或	<code>expr ^ expr</code>
<code> </code>	位或	<code>expr expr</code>

一般来说，如果运算对象是“小整型”，则它的值会被自动提升（参见 4.11.1 节，第 142 页）成较大的整数类型。运算对象可以是带符号的，也可以是无符号的。如果运算对象是带符号的且它的值为负，那么位运算符如何处理运算对象的“符号位”依赖于机器。而且，此时的左移操作可能会改变符号位的值，因此是一种未定义的行为。



关于符号位如何处理没有明确的规定，所以强烈建议仅将位运算符用于处理无符号类型。

移位运算符

之前在处理输入和输出操作时，我们已经使用过标准 IO 库定义的<<运算符和>>运算符的重载版本。这两种运算符的内置含义是对其运算对象执行基于二进制位的移动操作，首先令左侧运算对象的内容按照右侧运算对象的要求移动指定位数，然后将经过移动的（可能还进行了提升）左侧运算对象的拷贝作为求值结果。其中，右侧的运算对象一定不能为负，而且值必须严格小于结果的位数，否则就会产生未定义的行为。二进制位或者向左移(<<)或者向右移(>>)，移出边界之外的位就被舍弃掉了：

在下面的图例中右侧为最低位并且假定 char 占 8 位、int 占 32 位

// 0233 是八进制的字面值（参见 2.1.3 节，第 35 页）

```
unsigned char bits = 0233;
```

1 0 0 1 1 0 1 1

`bits << 8` // bits 提升成 int 类型，然后向左移动 8 位

0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 1 0 0 1 1 0 1 1 | 0 0 0 0 0 0 0 0

`bits << 31` // 向左移动 31 位，左边超出边界的位丢弃掉了

1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0

`bits >> 3` // 向右移动 3 位，最右边的 3 位丢弃掉了

0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 1 0 0 1 1

左移运算符 (`<<`) 在右侧插入值为 0 的二进制位。**右移运算符** (`>>`) 的行为则依赖于其左侧运算对象的类型：如果该运算对象是无符号类型，在左侧插入值为 0 的二进制位；如果该运算对象是带符号类型，在左侧插入符号位的副本或值为 0 的二进制位，如何选择要视具体环境而定。

154

位求反运算符

位求反运算符 (`~`) 将运算对象逐位求反后生成一个新值，将 1 置为 0、将 0 置为 1；

unsigned char bits = 0227;	1 0 0 1 0 1 1 1
<code>~bits</code>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0

char 类型的运算对象首先提升成 int 类型，提升时运算对象原来的位保持不变，往高位（high order position）添加 0 即可。因此在本例中，首先将 bits 提升成 int 类型，增加 24 个高位 0，随后将提升后的值逐位求反。

位与、位或、位异或运算符

与（&）、或（|）、异或（^）运算符在两个运算对象上逐位执行相应的逻辑操作：

unsigned char b1 = 0145;	0 1 1 0 0 1 0 1
unsigned char b2 = 0257;	1 0 1 0 1 1 1 1
b1 & b2	24 个高阶位都是 0 0 0 1 0 0 1 0 1
b1 b2	24 个高阶位都是 0 1 1 1 0 1 1 1 1
b1 ^ b2	24 个高阶位都是 0 1 1 0 0 1 0 1 0

对于位与运算符（&）来说，如果两个运算对象的对应位置都是 1 则运算结果中该位为 1，否则为 0。对于位或运算符（|）来说，如果两个运算对象的对应位置至少有一个为 1 则运算结果中该位为 1，否则为 0。对于位异或运算符（^）来说，如果两个运算对象的对应位置有且只有一个为 1 则运算结果中该位为 1，否则为 0。



有一种常见的错误是把位运算符和逻辑运算符（参见 4.3 节，第 126 页）搞混了，比如位与（&）和逻辑与（&&）、位或（|）和逻辑或（||）、位求反（~）和逻辑非（!）。

使用位运算符

我们举一个使用位运算符的例子：假设班级中有 30 个学生，老师每周都会对学生进行一次小测验，测验的结果只有通过和不通过两种。为了更好地追踪测验的结果，我们用一个二进制位代表某个学生在一次测验中是否通过，显然全班的测验结果可以用一个无符号整数来表示：

```
unsigned long quiz1 = 0; // 我们把这个值当成是位的集合来使用
```

定义 quiz1 的类型是 unsigned long，这样，quiz1 在任何机器上都将至少拥有 32 位；给 quiz1 赋一个明确的初始值，使得它的每一位在开始时都有统一且固定的价值。155

教师必须有权设置并检查每一个二进制位。例如，我们需要对序号为 27 的学生对应的位进行设置，以表示他通过了测验。为了达到这一目的，首先创建一个值，该值只有第 27 位是 1 其他位都是 0，然后将这个值与 quiz1 进行位或运算，这样就能强行将 quiz1 的第 27 位设置为 1，其他位都保持不变。

为了实现本例的目的，我们将 quiz1 的低阶位赋值为 0、下一位赋值为 1，以此类推，最后统计 quiz1 各个位的情况。

使用左移运算符和一个 unsigned long 类型的整数字面值 1（参见 2.1.3 节，第 35 页）就能得到一个表示学生 27 通过了测验的数值：

```
1UL << 27 // 生成一个值，该值只有第 27 位为 1
```

`1UL` 的低阶位上有一个 1，除此之外（至少）还有 31 个值为 0 的位。之所以使用 `unsigned long` 类型，是因为 `int` 类型只能确保占用 16 位，而我们至少需要 27 位。上面这条表达式通过在值为 1 的那个二进制位后面添加 0，使得它向左移动了 27 位。

接下来将所得的值与 `quiz1` 进行位或运算。为了同时更新 `quiz1` 的值，使用一条复合赋值语句（参见 4.4 节，第 130 页）：

```
quiz1 |= 1UL << 27; // 表示学生 27 通过了测验
```

`|=` 运算符的工作原理和 `+=` 非常相似，它等价于

```
quiz1 = quiz1 | 1UL << 27; // 等价于 quiz1 |= 1UL << 27;
```

假定教师在重新核对测验结果时发现学生 27 实际上并没有通过测验，他必须要把第 27 位的值置为 0。此时我们需要使用一个特殊的整数，它的第 27 位是 0、其他所有位都是 1。将这个值与 `quiz1` 进行位与运算就能实现目的了：

```
quiz1 &= ~(1UL << 27); // 学生 27 没有通过测验
```

通过将之前的值按位求反得到一个新值，除了第 27 位外都是 1，只有第 27 位的值是 0。随后将该值与 `quiz1` 进行位与运算，所得结果除了第 27 位外都保持不变。

最后，我们试图检查学生 27 测验的情况到底怎么样：

```
bool status = quiz1 & (1UL << 27); // 学生 27 是否通过了测验？
```

我们将 `quiz1` 和一个只有第 27 位是 1 的值按位求与，如果 `quiz1` 的第 27 位是 1，计算的结果就是非 0（真）；否则结果是 0。



移位运算符（又叫 IO 运算符）满足左结合律

尽管很多程序员从未直接用过位运算符，但是几乎所有人都用过它们的重载版本来进行 IO 操作。重载运算符的优先级和结合律都与它的内置版本一样，因此即使程序员用不到移位运算符的内置含义，也仍然有必要理解其优先级和结合律。

因为移位运算符满足左结合律，所以表达式

```
cout << "hi" << " there" << endl;
```

的执行过程实际上等同于

```
( (cout << "hi") << " there" ) << endl;
```

在这条语句中，运算对象 `"hi"` 和第一个 `<<` 组合在一起，它的结果和第二个 `<<` 组合在一起，接下来的结果再和第三个 `<<` 组合在一起。

移位运算符的优先级不高不低，介于中间：比算术运算符的优先级低，但比关系运算符、赋值运算符和条件运算符的优先级高。因此在一次使用多个运算符时，有必要在适当的地方加上括号使其满足我们的要求。

```
cout << 42 + 10; // 正确：+ 的优先级更高，因此输出求和结果
cout << (10 < 42); // 正确：括号使运算对象按照我们的期望组合在一起，输出 1
cout << 10 < 42; // 错误：试图比较 cout 和 42！
```

最后一个 `cout` 的含义其实是

```
(cout << 10) < 42;
```

也就是“把数字 10 写到 `cout`，然后将结果（即 `cout`）与 42 进行比较”。

4.8 节练习

练习 4.25: 如果一台机器上 int 占 32 位、char 占 8 位，用的是 Latin-1 字符集，其中字符‘q’的二进制形式是 01110001，那么表达式~'q'<<6 的值是什么？

练习 4.26: 在本节关于测验成绩的例子中，如果使用 unsigned int 作为 quiz1 的类型会发生什么情况？

练习 4.27: 下列表达式的结果是什么？

```
unsigned long ull = 3, ul2 = 7;
(a) ull & ul2           (b) ull | ul2
(c) ull && ul2         (d) ull || ul2
```

4.9 sizeof 运算符

sizeof 运算符返回一条表达式或一个类型名字所占的字节数。sizeof 运算符满足右结合律，其所得的值是一个 size_t 类型（参见 3.5.2 节，第 103 页）的常量表达式（参见 2.4.4 节，第 58 页）。运算符的运算对象有两种形式：

```
sizeof (type)
sizeof expr
```

在第二种形式中，sizeof 返回的是表达式结果类型的大小。与众不同的一点是，sizeof 并不实际计算其运算对象的值：

```
Sales_data data, *p;
sizeof(Sales_data);          // 存储 Sales_data 类型的对象所占的空间大小
sizeof data;                 // data 的类型的大小，即 sizeof(Sales_data)
sizeof p;                    // 指针所占的空间大小
sizeof *p;                   // p 所指类型的空间大小，即 sizeof(Sales_data)
sizeof data.revenue;         // Sales_data 的 revenue 成员对应类型的大小
sizeof Sales_data::revenue;  // 另一种获取 revenue 大小的方式
```

< 157

这些例子中最有趣的一个是 sizeof *p。首先，因为 sizeof 满足右结合律并且与 * 运算符的优先级一样，所以表达式按照从右向左的顺序组合。也就是说，它等价于 sizeof(*p)。其次，因为 sizeof 不会实际求运算对象的值，所以即使 p 是一个无效（即未初始化）的指针（参见 2.3.2 节，第 47 页）也不会有什么影响。在 sizeof 的运算对象中解引用一个无效指针仍然是一种安全的行为，因为指针实际上并没有被真正使用。sizeof 不需要真的解引用指针也能知道它所指对象的类型。

C++11 新标准允许我们使用作用域运算符来获取类成员的大小。通常情况下只有通过类的对象才能访问到类的成员，但是 sizeof 运算符无须我们提供一个具体的对象，因为要想知道类成员的大小无须真的获取该成员。

C++
11

sizeof 运算符的结果部分地依赖于其作用的类型：

- 对 char 或者类型为 char 的表达式执行 sizeof 运算，结果得 1。
- 对引用类型执行 sizeof 运算得到被引用对象所占空间的大小。
- 对指针执行 sizeof 运算得到指针本身所占空间的大小。
- 对解引用指针执行 sizeof 运算得到指针指向的对象所占空间的大小，指针不需有效。

- 对数组执行 `sizeof` 运算得到整个数组所占空间的大小，等价于对数组中所有的元素各执行一次 `sizeof` 运算并将所得结果求和。注意，`sizeof` 运算不会把数组转换成指针来处理。
- 对 `string` 对象或 `vector` 对象执行 `sizeof` 运算只返回该类型固定部分的大小，不会计算对象中的元素占用了多少空间。

因为执行 `sizeof` 运算能得到整个数组的大小，所以可以用数组的大小除以单个元素的大小得到数组中元素的个数：

```
// sizeof(ia)/sizeof(*ia) 返回 ia 的元素数量
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr2[sz]; // 正确：sizeof 返回一个常量表达式，参见 2.4.4 节（第 58 页）
```

因为 `sizeof` 的返回值是一个常量表达式，所以我们可以用 `sizeof` 的结果声明数组的维度。

4.9 节练习

练习 4.28： 编写一段程序，输出每一种内置类型所占空间的大小。

练习 4.29： 推断下面代码的输出结果并说明理由。实际运行这段程序，结果和你想象的一样吗？如果不一样，为什么？

```
int x[10]; int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

练习 4.30： 根据 4.12 节中的表（第 147 页），在下述表达式的适当位置加上括号，使得加上括号之后表达式的含义与原来的含义相同。

- | | |
|----------------------------------|--------------------------------------|
| (a) <code>sizeof x + y</code> | (b) <code>sizeof p->mem[i]</code> |
| (c) <code>sizeof a < b</code> | (d) <code>sizeof f()</code> |

4.10 逗号运算符

逗号运算符（comma operator）含有两个运算对象，按照从左向右的顺序依次求值。和逻辑与、逻辑或以及条件运算符一样，逗号运算符也规定了运算对象求值的顺序。

对于逗号运算符来说，首先对左侧的表达式求值，然后将求值结果丢弃掉。逗号运算符真正的结果是右侧表达式的值。如果右侧运算对象是左值，那么最终的求值结果也是左值。

逗号运算符经常被用在 `for` 循环当中：

```
vector<int>::size_type cnt = ivec.size();
// 将把从 size 到 1 的值赋给 ivec 的元素
for(vector<int>::size_type ix = 0;
    ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

这个循环在 `for` 语句的表达式中递增 `ix`、递减 `cnt`，每次循环迭代 `ix` 和 `cnt` 相应改变。只要 `ix` 满足条件，我们就把当前元素设成 `cnt` 的当前值。

4.10 节练习

练习 4.31: 本节的程序使用了前置版本的递增运算符和递减运算符，解释为什么要用前置版本而不用后置版本。要想使用后置版本的递增递减运算符需要做哪些改动？使用后置版本重写本节的程序。

练习 4.32: 解释下面这个循环的含义。

```
constexpr int size = 5;
int ia[size] = {1,2,3,4,5};
for (int *ptr = ia, ix = 0;
ix != size && ptr != ia+size;
++ix, ++ptr) { /* ... */ }
```

练习 4.33: 根据 4.12 节中的表（第 147 页）说明下面这条表达式的含义。

```
someValue ? ++x, ++y : --x, --y
```

4.11 类型转换

< 159

在 C++ 语言中，某些类型之间有关联。如果两种类型有关联，那么当程序需要其中一种类型的运算对象时，可以用另一种关联类型的对象或值来替代。换句话说，如果两种类型可以相互转换（conversion），那么它们就是关联的。



举个例子，考虑下面这条表达式，它的目的是将 `ival` 初始化为 6：

```
int ival = 3.541 + 3; // 编译器可能会警告该运算损失了精度
```

加法的两个运算对象类型不同：`3.541` 的类型是 `double`，`3` 的类型是 `int`。C++ 语言不会直接将两个不同类型的值相加，而是先根据类型转换规则设法将运算对象的类型统一后再求值。上述的类型转换是自动执行的，无须程序员的介入，有时甚至不需要程序员了解。因此，它们被称作隐式转换（implicit conversion）。

算术类型之间的隐式转换被设计得尽可能避免损失精度。很多时候，如果表达式中既有整数类型的运算对象也有浮点数类型的运算对象，整型会转换成浮点型。在上面的例子中，`3` 转换成 `double` 类型，然后执行浮点数加法，所得结果的类型是 `double`。

接下来就要完成初始化的任务了。在初始化过程中，因为被初始化的对象的类型无法改变，所以初始值被转换成该对象的类型。仍以这个例子说明，加法运算得到的 `double` 类型的结果转换成 `int` 类型的值，这个值被用来初始化 `ival`。由 `double` 向 `int` 转换时忽略掉了小数部分，上面的表达式中，数值 6 被赋给了 `ival`。

何时发生隐式类型转换

在下面这些情况下，编译器会自动地转换运算对象的类型：

- 在大多数表达式中，比 `int` 类型小的整型值首先提升为较大的整数类型。
- 在条件中，非布尔值转换成布尔类型。
- 初始化过程中，初始值转换成变量的类型；在赋值语句中，右侧运算对象转换成左侧运算对象的类型。
- 如果算术运算或关系运算的运算对象有多种类型，需要转换成同一种类型。
- 如第 6 章将要介绍的，函数调用时也会发生类型转换。



4.11.1 算术转换

算术转换 (arithmetic conversion) 的含义是把一种算术类型转换成另外一种算术类型，这一点在 2.1.2 节（第 32 页）中已有介绍。算术转换的规则定义了一套类型转换的层次，其中运算符的运算对象将转换成最宽的类型。例如，如果一个运算对象的类型是 `long double`，那么不论另外一个运算对象的类型是什么都会转换成 `long double`。还有一种更普遍的情况，当表达式中既有浮点类型也有整数类型时，整数值将转换成相应的浮点类型。

160 整型提升

整型提升 (integral promotion) 负责把小整数类型转换成较大的整数类型。对于 `bool`、`char`、`signed char`、`unsigned char`、`short` 和 `unsigned short` 等类型来说，只要它们所有可能的值都能存在 `int` 里，它们就会提升成 `int` 类型；否则，提升成 `unsigned int` 类型。就如我们所熟知的，布尔值 `false` 提升成 0、`true` 提升成 1。

较大的 `char` 类型 (`wchar_t`、`char16_t`、`char32_t`) 提升成 `int`、`unsigned int`、`long`、`unsigned long`、`long long` 和 `unsigned long long` 中最小的一种类型，前提是转换后的类型要能容纳原类型所有可能的值。

无符号类型的运算对象

如果某个运算符的运算对象类型不一致，这些运算对象将转换成同一种类型。但是如果某个运算对象的类型是无符号类型，那么转换的结果就要依赖于机器中各个整数类型的相对大小了。

像往常一样，首先执行整型提升。如果结果的类型匹配，无须进行进一步的转换。如果两个（提升后的）运算对象的类型要么都是带符号的、要么都是无符号的，则小类型的运算对象转换成较大的类型。

如果一个运算对象是无符号类型、另外一个运算对象是带符号类型，而且其中的无符号类型不小于带符号类型，那么带符号的运算对象转换成无符号的。例如，假设两个类型分别是 `unsigned int` 和 `int`，则 `int` 类型的运算对象转换成 `unsigned int` 类型。需要注意的是，如果 `int` 型的值恰好为负值，其结果将以 2.1.2 节（第 32 页）介绍的方法转换，并带来该节描述的所有副作用。

剩下的一种情况是带符号类型大于无符号类型，此时转换的结果依赖于机器。如果无符号类型的所有值都能存在该带符号类型中，则无符号类型的运算对象转换成带符号类型。如果不能，那么带符号类型的运算对象转换成无符号类型。例如，如果两个运算对象的类型分别是 `long` 和 `unsigned int`，并且 `int` 和 `long` 的大小相同，则 `long` 类型的运算对象转换成 `unsigned int` 类型；如果 `long` 类型占用的空间比 `int` 更多，则 `unsigned int` 类型的运算对象转换成 `long` 类型。

理解算术转换

要想理解算术转换，办法之一就是研究大量的例子：

<code>bool</code>	<code>flag;</code>	<code>char</code>	<code>cval;</code>
<code>short</code>	<code>sval;</code>	<code>unsigned short</code>	<code>usval;</code>
<code>int</code>	<code>ival;</code>	<code>unsigned int</code>	<code>uival;</code>
<code>long</code>	<code>lval;</code>	<code>unsigned long</code>	<code>ulval;</code>
<code>float</code>	<code>fval;</code>	<code>double</code>	<code>dval;</code>

```

3.14159L + 'a';      // 'a' 提升成 int, 然后该 int 值转换成 long double
dval + ival;          // ival 转换成 double
dval + fval;          // fval 转换成 double
ival = dval;          // dval 转换成 (切除小数部分后) int
flag = dval;          // 如果 dval 是 0, 则 flag 是 false, 否则 flag 是 true
cval + fval;          // cval 提升成 int, 然后该 int 值转换成 float
sval + cval;          // sval 和 cval 都提升成 int
cval + lval;          // cval 转换成 long
ival + ulval;         // ival 转换成 unsigned long
usval + ival;         // 根据 unsigned short 和 int 所占空间的大小进行提升
uival + lval;         // 根据 unsigned int 和 long 所占空间的大小进行转换

```

<161

在第一个加法运算中, 小写字母'a'是 char 型的字符常量, 它其实能表示一个数字值(参见 2.1.1 节, 第 30 页)。到底这个数字值是多少完全依赖于机器上的字符集, 在我们的环境中, 'a' 对应的数字值是 97。当把'a'和一个 long double 型的数相加时, char 型的值首先提升成 int 型, 然后 int 型的值再转换成 long double 型。最终我们把这个转换后的值与那个字面值相加。最后的两个含有无符号类型值的表达式也比较有趣, 它们的结果依赖于机器。

4.11.1 节练习

练习 4.34: 根据本节给出的变量定义, 说明在下面的表达式中将发生什么样的类型转换:

- (a) if (fval) (b) dval = fval + ival; (c) dval + ival * cval;

需要注意每种运算符遵循的是左结合律还是右结合律。

练习 4.35: 假设有如下的定义,

```

char cval;      int ival;      unsigned int ui;
float fval;     double dval;

```

请回答在下面的表达式中发生了隐式类型转换吗? 如果有, 指出来。

- (a) cval = 'a' + 3; (b) fval = ui - ival * 1.0;
 (c) dval = ui * fval; (d) cval = ival + fval + dval;

4.11.2 其他隐式类型转换



除了算术转换之外还有几种隐式类型转换, 包括如下几种。

数组转换成指针: 在大多数用到数组的表达式中, 数组自动转换成指向数组首元素的指针:

```

int ia[10];        // 含有 10 个整数的数组
int* ip = ia;       // ia 转换成指向数组首元素的指针

```

当数组被用作 decltype 关键字的参数, 或者作为取地址符(&)、sizeof 及 typeid(第 19.2.2 节, 732 页将介绍)等运算符的运算对象时, 上述转换不会发生。同样的, 如果用一个引用来初始化数组(参见 3.5.1 节, 第 102 页), 上述转换也不会发生。我们将在 6.7 节(第 221 页)看到, 当在表达式中使用函数类型时会发生类似的指针转换。

指针的转换: C++还规定了几种其他的指针转换方式, 包括常量整数值 0 或者字面值 nullptr 能转换成任意指针类型; 指向任意非常量的指针能转换成 void*; 指向任意对象的指针能转换成 const void*。15.2.2 节(第 530 页)将要介绍, 在有继承关系的类

<162

型间还有另外一种指针转换的方式。

转换成布尔类型: 存在一种从算术类型或指针类型向布尔类型自动转换的机制。如果指针或算术类型的值为 0, 转换结果是 `false`; 否则转换结果是 `true`:

```
char *cp = get_string();
if (cp) /* ... */ // 如果指针 cp 不是 0, 条件为真
while (*cp) /* ... */ // 如果*cp 不是空字符, 条件为真
```

转换成常量: 允许将指向非常量类型的指针转换成指向相应的常量类型的指针, 对于引用也是这样。也就是说, 如果 `T` 是一种类型, 我们就能将指向 `T` 的指针或引用分别转换成指向 `const T` 的指针或引用 (参见 2.4.1 节, 第 54 页和 2.4.2 节, 第 56 页):

```
int i;
const int &j = i;           // 非常量转换成 const int 的引用
const int *p = &i;          // 非常量的地址转换成 const 的地址
int &r = j, *q = p;         // 错误: 不允许 const 转换成非常量
```

相反的转换并不存在, 因为它试图删除掉底层 `const`。

类类型定义的转换: 类类型能定义由编译器自动执行的转换, 不过编译器每次只能执行一种类类型的转换。在 7.5.4 节 (第 263 页) 中我们将看到一个例子, 如果同时提出多个转换请求, 这些请求将被拒绝。

我们之前的程序已经使用过类类型转换: 一处是在需要标准库 `string` 类型的地方使用 C 风格字符串 (参见 3.5.5 节, 第 111 页); 另一处是在条件部分读入 `istream`:

```
string s, t = "a value";    // 字符串字面值转换成 string 类型
while (cin >> s)           // while 的条件部分把 cin 转换成布尔值
```

条件 (`cin>>s`) 读入 `cin` 的内容并将 `cin` 作为其求值结果。条件部分本来需要一个布尔类型的值, 但是这里实际检查的是 `istream` 类型的值。幸好, IO 库定义了从 `istream` 向布尔值转换的规则, 根据这一规则, `cin` 自动地转换成布尔值。所得的布尔值到底是什么由输入流的状态决定, 如果最后一次读入成功, 转换得到的布尔值是 `true`; 相反, 如果最后一次读入不成功, 转换得到的布尔值是 `false`。

4.11.3 显式转换

有时我们希望显式地将对象强制转换成另外一种类型。例如, 如果想在下面的代码中执行浮点数除法:

```
int i, j;
double slope = i/j;
```

就要使用某种方法将 `i` 和/或 `j` 显式地转换成 `double`, 这种方法称作**强制类型转换**(cast)。



虽然有时不得不使用强制类型转换, 但这种方法本质上是非常危险的。

163 命名的强制类型转换

一个命名的强制类型转换具有如下形式:

`cast-name<type>(expression)`

其中, `type` 是转换的目标类型而 `expression` 是要转换的值。如果 `type` 是引用类型, 则结果是左值。`cast-name` 是 `static_cast`、`dynamic_cast`、`const_cast` 和

reinterpret_cast 中的一种。**dynamic_cast** 支持运行时类型识别，我们将在 19.2 节（第 730 页）对其做更详细的介绍。*cast-name* 指定了执行的是哪种转换。

static_cast

任何具有明确定义的类型转换，只要不包含底层 `const`，都可以使用 `static_cast`。例如，通过将一个运算对象强制转换成 `double` 类型就能使表达式执行浮点数除法：

```
// 进行强制类型转换以便执行浮点数除法
double slope = static_cast<double>(j) / i;
```

当需要把一个较大的算术类型赋值给较小的类型时，`static_cast` 非常有用。此时，强制类型转换告诉程序的读者和编译器：我们知道并且不在乎潜在的精度损失。一般来说，如果编译器发现一个较大的算术类型试图赋值给较小的类型，就会给出警告信息；但是当我们执行了显式的类型转换后，警告信息就会被关闭了。

`static_cast` 对于编译器无法自动执行的类型转换也非常有用。例如，我们可以使用 `static_cast` 找回存在于 `void*` 指针（参见 2.3.2 节，第 50 页）中的值：

```
void* p = &d;      // 正确：任何非常量对象的地址都能存入 void*
// 正确：将 void* 转换回初始的指针类型
double *dp = static_cast<double*>(p);
```

当我们把指针存放在 `void*` 中，并且使用 `static_cast` 将其强制转换回原来的类型时，应该确保指针的值保持不变。也就是说，强制转换的结果将与原始的地址值相等，因此我们必须确保转换后所得的类型就是指针所指的类型。类型一旦不符，将产生未定义的后果。

const_cast

`const_cast` 只能改变运算对象的底层 `const`（参见 2.4.3 节，第 57 页）：

```
const char *pc;
char *p = const_cast<char*>(pc); // 正确：但是通过 p 写值是未定义的行为
```

对于将常量对象转换成非常量对象的行为，我们一般称其为“去掉 `const` 性质（*cast away the const*）”。一旦我们去掉了某个对象的 `const` 性质，编译器就不再阻止我们对该对象进行写操作了。如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的行为。然而如果对象是一个常量，再使用 `const_cast` 执行写操作就会产生未定义的后果。

只有 `const_cast` 能改变表达式的常量属性，使用其他形式的命名强制类型转换改变表达式的常量属性都将引发编译器错误。同样的，也不能用 `const_cast` 改变表达式的类型：

```
const char *cp;
// 错误：static_cast 不能转换掉 const 性质
char *q = static_cast<char*>(cp);
static_cast<string>(cp);      // 正确：字符串字面值转换成 string 类型
const_cast<string>(cp);      // 错误：const_cast 只改变常量属性
```

`const_cast` 常常用于有函数重载的上下文中，关于函数重载将在 6.4 节（第 208 页）进行详细介绍。

reinterpret_cast

`reinterpret_cast` 通常为运算对象的位模式提供较低层次上的重新解释。举个例

子，假设有如下的转换

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

我们必须牢记 pc 所指的真实对象是一个 int 而非字符，如果把 pc 当成普通的字符指针使用就可能在运行时发生错误。例如：

```
string str(pc);
```

可能导致异常的运行时行为。

使用 `reinterpret_cast` 是非常危险的，用 pc 初始化 `str` 的例子很好地证明了这一点。其中的关键问题是类型改变了，但编译器没有给出任何警告或者错误的提示信息。当我们用一个 int 的地址初始化 pc 时，由于显式地声称这种转换合法，所以编译器不会发出任何警告或错误信息。接下来再使用 pc 时就会认定它的值是 `char*` 类型，编译器没法知道它实际存放的是指向 int 的指针。最终的结果就是，在上面的例子中虽然用 pc 初始化 `str` 没什么实际意义，甚至还可能引发更糟糕的后果，但仅从语法上而言这种操作无可指摘。查找这类问题的原因非常困难，如果将 ip 强制转换成 pc 的语句和用 pc 初始化 `string` 对象的语句分属不同文件就更是如此。



`reinterpret_cast` 本质上依赖于机器。要想安全地使用 `reinterpret_cast` 必须对涉及的类型和编译器实现转换的过程都非常了解。

165 >

建议：避免强制类型转换

强制类型转换干扰了正常的类型检查（参见 2.2.2 节，第 42 页），因此我们强烈建议程序员避免使用强制类型转换。这个建议对于 `reinterpret_cast` 尤其适用，因为此类类型转换总是充满了风险。在有重载函数的上下文中使用 `const_cast` 无可厚非，关于这一点将在 6.4 节（第 208 页）中详细介绍；但是在其他情况下使用 `const_cast` 也就意味着程序存在某种设计缺陷。其他强制类型转换，比如 `static_cast` 和 `dynamic_cast`，都不应该频繁使用。每次书写了一条强制类型转换语句，都应该反复斟酌能否以其他方式实现相同的目标。就算实在无法避免，也应该尽量限制类型转换值的作用域，并且记录对相关类型的所有假定，这样可以减少错误发生的机会。

旧式的强制类型转换

在早期版本的 C++ 语言中，显式地进行强制类型转换包含两种形式：

```
type (expr);           // 函数形式的强制类型转换
(type) expr;          // C 语言风格的强制类型转换
```

根据所涉及的类型不同，旧式的强制类型转换分别具有与 `const_cast`、`static_cast` 或 `reinterpret_cast` 相似的行为。当我们在某处执行旧式的强制类型转换时，如果换成 `const_cast` 和 `static_cast` 也合法，则其行为与对应的命名转换一致。如果替换后不合法，则旧式强制类型转换执行与 `reinterpret_cast` 类似的功能：

```
char *pc = (char*) ip; // ip 是指向整数的指针
```

的效果与使用 `reinterpret_cast` 一样。



与命名的强制类型转换相比，旧式的强制类型转换从表现形式上来说不那么清晰明了，容易被看漏，所以一旦转换过程出现问题，追踪起来也更加困难。

4.11.3 节练习

练习 4.36: 假设 `i` 是 `int` 类型，`d` 是 `double` 类型，书写表达式 `i*=d` 使其执行整数类型的乘法而非浮点类型的乘法。

练习 4.37: 用命名的强制类型转换改写下列旧式的转换语句。

```
int i; double d; const string *ps; char *pc; void *pv;
(a) pv = (void*)ps;   (b) i = int(*pc);
(c) pv = &d;          (d) pc = (char*) pv;
```

练习 4.38: 说明下面这条表达式的含义。

```
double slope = static_cast<double>(j/i);
```

4.12 运算符优先级表

< 166

表 4.4: 运算符优先级

结合律和运算符	功能	用法	参考页码
左 ::	全局作用域	::name	256
左 ::	类作用域	class::name	79
左 ::	命名空间作用域	namespace::name	74
左 .	成员选择	object.member	20
左 ->	成员选择	pointer->member	98
左 []	下标	expr[expr]	104
左 ()	函数调用	name(expr_list)	20
左 ()	类型构造	type(expr_list)	145
右 ++	后置递增运算	lvalue++	131
右 --	后置递减运算	lvalue--	131
右 typeid	类型 ID	typeid(type)	731
右 typeid	运行时类型 ID	typeid(expr)	731
右 explicit cast	类型转换	cast_name<type>(expr)	144
右 ++	前置递增运算	++lvalue	131
右 --	前置递减运算	--lvalue	131
右 ~	位求反	~expr	136
右 !	逻辑非	!expr	126
右 -	一元负号	-expr	124
右 +	一元正号	+expr	124
右 *	解引用	*expr	48
右 &	取地址	&lvalue	47
右 ()	类型转换	(type) expr	145
右 sizeof	对象的大小	sizeof expr	139

续表

结合律和运算符	功能	用法	参考页码
右 sizeof	类型的大小	sizeof(type)	139
右 Sizeof...	参数包的大小	sizeof...(name)	619
右 new	创建对象	new type	407
右 new[]	创建数组	new type[size]	407
右 delete	释放对象	delete expr	409
右 delete[]	释放数组	delete[] expr	409
右 noexcept	能否抛出异常	noexcept(expr)	690
左 ->*	指向成员选择的指针	ptr->*ptr_to_member	740
左 .*	指向成员选择的指针	obj.*ptr_to_member	740
左 *	乘法	expr * expr	124
左 /	除法	expr / expr	124
左 %	取模(取余)	expr % expr	124
左 +	加法	expr + expr	124
左 -	减法	expr - expr	124
左 <<	向左移位	expr << expr	136
左 >>	向右移位	expr >> expr	136
左 <	小于	expr < expr	126
左 <=	小于等于	expr <= expr	126
左 >	大于	expr > expr	126
左 >=	大于等于	expr >= expr	126
左 ==	相等	expr == expr	126
左 !=	不相等	expr != expr	126
左 &	位与	expr & expr	136
左 ^	位异或	expr ^ expr	136
左	位或	expr expr	136
左 &&	逻辑与	expr && expr	126
左	逻辑或	expr expr	126
右 ?: :	条件	expr ? expr : expr	134
右 =	赋值	lvalue = expr	129
右 *=, /=, %=	复合赋值	lvalue += expr 等	129
右 +=, -=			129
右 <<=, >>=			129
右 &=, =, ^=			129
右 throw	抛出异常	throw expr	173
左 ,	逗号	expr, expr	140

167

小结

< 168

C++语言提供了一套丰富的运算符，并定义了这些运算符作用于内置类型的运算对象时所执行的操作。此外，C++语言还支持运算符重载的机制，允许我们自己定义运算符作用于类类型时的含义。第14章将介绍如何定义作用于用户类型的运算符。

对于含有超过一个运算符的表达式，要想理解其含义关键要理解优先级、结合律和求值顺序。每个运算符都有其对应的优先级和结合律，优先级规定了复合表达式中运算符组合的方式，结合律则说明当运算符的优先级一样时应该如何组合。

大多数运算符并不明确规定运算对象的求值顺序：编译器有权自由选择先对左侧运算对象求值还是先对右侧运算对象求值。一般来说，运算对象的求值顺序对表达式的最终结果没有影响。但是，如果两个运算对象指向同一个对象而且其中一个改变了对象的值，就会导致程序出现不易发现的严重缺陷。

最后一点，运算对象经常从原始类型自动转换成某种关联的类型。例如，表达式中的小整型会自动提升成大整型。不论内置类型还是类类型都涉及类型转换的问题。如果需要，我们还可以显式地进行强制类型转换。

术语表

算术转换 (arithmetic conversion) 从一种算术类型转换成另一种算术类型。在二元运算符的上下文中，为了保留精度，算术转换通常把较小的类型转换成较大的类型（例如整型转换成浮点型）。

结合律 (associativity) 规定具有相同优先级的运算符如何组合在一起。结合律分为左结合律（运算符从左向右组合）和右结合律（运算符从右向左组合）。

二元运算符 (binary operator) 有两个运算对象参与运算的运算符。

强制类型转换 (cast) 一种显式的类型转换。

复合表达式 (compound expression) 含有一个以上的运算符的表达式。

const_cast 一种涉及 `const` 的强制类型转换。将底层 `const` 对象转换成对应的非常量类型，或者执行相反的转换。

转换 (conversion) 一种类型的值改变成另一种类型的值的过程。C++语言定义了内置类型的转换规则。类类型同样可以转换。

dynamic_cast 和继承及运行时类型识别一

起使用。参见 19.2 节（第 730 页）。

表达式 (expression) C++程序中最低级别的计算。表达式将运算符作用于一个或多个运算对象，每个表达式都有对应的求值结果。表达式本身也可以作为运算对象，这时就得到了对多个运算符求值的复合表达式。

隐式转换 (implicit conversion) 由编译器自动执行的类型转换。假如表达式需要某种特定的类型而运算对象是另外一种类型，此时只要规则允许，编译器就会自动地将运算对象转换成所需的类型。

整型提升 (integral promotion) 把一种较小的整数类型转换成与之最接近的较大整数类型的过程。不论是否真的需要，小整数类型（即 `short`、`char` 等）总是会得到提升。

左值 (lvalue) 是指那些求值结果为对象或函数的表达式。一个表示对象的非常量左值可以作为赋值运算符的左侧运算对象。

运算对象 (operand) 表达式在某些值上执行运算，这些值就是运算对象。一个运算

< 169

符有一个或多个相关的运算对象。

运算符 (operator) 决定表达式所做操作的符号。C++语言定义了一套运算符并说明了这些运算符作用于内置类型时的含义。C++还定义了运算符的优先级和结合律以及每种运算符处理的运算对象数量。可以重载运算符使其能处理类类型。

求值顺序 (order of evaluation) 是某个运算符的运算对象的求值顺序。大多数情况下，编译器可以任意选择运算对象求值的顺序。不过运算对象一定要在运算符之前得到求值结果。只有`&&`、`||`、条件和逗号四种运算符明确规定了求值顺序。

重载运算符 (overloaded operator) 针对某种运算符重新定义的适用于类类型的版本。第14章将介绍重载运算符的方法。

优先级 (precedence) 规定了复合表达式中不同运算符的执行顺序。与低优先级的运算符相比，高优先级的运算符组合得更紧密。

提升 (promotion) 参见整型提升。

reinterpret_cast 把运算对象的内容解释成另外一种类型。这种强制类型转换本质上依赖于机器而且非常危险。170

结果 (result) 计算表达式得到的值或对象。

右值 (rvalue) 是指一种表达式，其结果是值而非值所在的位置。

短路求值 (short-circuit evaluation) 是一个专有名词，描述逻辑与运算符和逻辑或运算符的执行过程。如果根据运算符的第一个运算对象就能确定整个表达式的结构，求值终止，此时第二个运算对象将不会被求值。

sizeof 是一个运算符，返回存储对象所需的字节数，该对象的类型可能是某个给定的类型名字，也可能由表达式的返回结果确定。

static_cast 显式地执行某种定义明确的类型转换，常用于替代由编译器隐式执行的类型转换。

一元运算符 (unary operators) 只有一个运算对象参与运算的运算符。

, 运算符 (, operator) 逗号运算符，是一种从左向右求值的二元运算符。逗号运算符的结果是右侧运算对象的值，当且仅当右侧运算对象是左值时逗号运算符的结果是左值。

? : 运算符 (?: operator) 条件运算符，以下述形式提供 if-then-else 逻辑的表达式

`cond ? expr1 : expr2;`

如果条件 `cond` 为真，对 `expr1` 求值；否则对 `expr2` 求值。`expr1` 和 `expr2` 的类型应该相同或者能转换成同一种类型。`expr1` 和 `expr2` 中只有一个会被求值。

&&运算符 (&& operator) 逻辑与运算符，如果两个运算对象都是真，结果才为真。只有当左侧运算对象为真时才会检查右侧运算对象。

&运算符 (& operator) 位与运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位都是 1，所得结果中该位为 1；否则所得结果中该位为 0。

^运算符 (^ operator) 位异或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位有且只有一个是 1，所得结果中该位为 1；否则所得结果中该位为 0。

||运算符 (|| operator) 逻辑或运算符，任何一个运算对象是真，结果就为真。只有当左侧运算对象为假时才会检查右侧运算对象。

| 运算符 (| operator) 位或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位至少有一个是 1，所得结果中该位为 1；否则所得结果中该位为 0。

++运算符 (++ operator) 递增运算符。包括两种形式：前置版本和后置版本。前置递增运算符得到一个左值，它给运算符加 1 并得到运算对象改变后的值。后置递增运算符得到一个右值，它给运算符加 1 并得到运算对象原始的、未改变的值的副本。

注意：即使迭代器没有定义+运算符，也会

有++运算符。

-运算符 (— operator) 递减运算符。包括两种形式：前置版本和后置版本。前置递减运算符得到一个左值，它从运算符减 1 并得到运算对象改变后的值。后置递减运算符得到一个右值，它从运算符减 1 并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义-运算符，也会有--运算符。

<<运算符 (<< operator) 左移运算符，将左侧运算对象的值的（可能是提升后的）副本向左移位，移动的位数由右侧运算对象确定。右侧运算对象必须大于等于 0 而且小于结果的位数。左侧运算对象应该是无符号类型，如果它是带符号类型，则一旦移动改变了符号位的值就会产生未定义的结果。

>>运算符 (>> operator) 右移运算符，除了移动方向相反，其他性质都和左移运算符类似。如果左侧运算对象是带符号类型，那么根据实现的不同新移入的内容也不同，新移入的位可能都是 0，也可能都是符号位的副本。

~运算符 (~ operator) 位求反运算符，生成一个新的整型值。该值的每一位恰好与（可能是提升后的）运算对象的对应位相反。

!运算符 (! operator) 逻辑非运算符，将它的运算对象的布尔值取反。如果运算对象是假，则结果为真，如果运算对象是真，则结果为假。

