

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

138

以下两条经验准则对书写复合表达式有益：

139

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 <141> 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。

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)

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

<code>0 0 0 0 0 0 0 0</code>	<code>0 0 0 0 0 0 0 0</code>	<code>1 0 0 1 1 0 1 1</code>	<code>0 0 0 0 0 0 0 0</code>
------------------------------	------------------------------	------------------------------	------------------------------

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

<code>1 0 0 0 0 0 0 0</code>	<code>0 0 0 0 0 0 0 0</code>	<code>0 0 0 0 0 0 0 0</code>	<code>0 0 0 0 0 0 0 0</code>
------------------------------	------------------------------	------------------------------	------------------------------

`bits >> 3` // 向右移动 3 位，最右边的 3 位丢弃掉了

<code>0 0 0 0 0 0 0 0</code>	<code>0 0 0 0 0 0 0 0</code>	<code>0 0 0 0 0 0 0 0</code>	<code>0 0 0 1 0 0 1 1</code>
------------------------------	------------------------------	------------------------------	------------------------------

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。

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：

154 >

```
unsigned char bits = 0227;      1 0 0 1 0 1 1 1
~bits
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 个高位都是 0 0 0 1 0 0 1 0 1
b1 b2	24 个高位都是 0 1 1 1 0 1 1 1 1
b1 ^ b2	24 个高位都是 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 + ival;          // 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 + ival;         // 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类型的数相加时, char类型的值首先提升成int类型, 然后int类型的值再转换成long double类型。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, 如果用一个引用来自初始化数组(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。

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。

第 5 章

语句

内容

5.1 简单语句.....	154
5.2 语句作用域.....	155
5.3 条件语句.....	156
5.4 迭代语句.....	165
5.5 跳转语句.....	170
5.6 try 语句块和异常处理	172
小结	178
术语表.....	178

和大多数语言一样，C++提供了条件执行语句、重复执行相同代码的循环语句和用于中断当前控制流的跳转语句。本章将详细介绍 C++语言所支持的这些语句。

172 通常情况下，语句是顺序执行的。但除非是最简单的程序，否则仅有顺序执行远远不够。因此，C++语言提供了一组控制流（flow-of-control）语句以支持更复杂的执行路径。



5.1 简单语句

C++语言中的大多数语句都以分号结束，一个表达式，比如 `ival + 5`，末尾加上分号就变成了**表达式语句**（expression statement）。表达式语句的作用是执行表达式并丢弃掉求值结果：

```
ival + 5;           // 一条没什么实际用处的表达式语句
cout << ival;      // 一条有用的表达式语句
```

第一条语句没什么用处，因为虽然执行了加法，但是相加的结果没被使用。比较普遍的情况是，表达式语句中的表达式在求值时附带有其他效果，比如给变量赋了新值或者输出了结果。

空语句

最简单的语句是**空语句**（null statement），空语句中只含有一个单独的分号：

```
; // 空语句
```

如果在程序的某个地方，语法上需要一条语句但是逻辑上不需要，此时应该使用空语句。一种常见的情况是，当循环的全部工作在条件部分就可以完成时，我们通常会用到空语句。例如，我们想读取输入流的内容直到遇到一个特定的值为止，除此之外什么事情也不做：

```
// 重复读入数据直至到达文件末尾或某次输入的值等于 sought
while (cin >> s && s != sought)
    ; // 空语句
```

`while` 循环的条件部分首先从标准输入读取一个值并且隐式地检查 `cin`，判断读取是否成功。假定读取成功，条件的后半部分检查读进来的值是否等于 `sought` 的值。如果发现了想要的值，循环终止；否则，从 `cin` 中继续读取另一个值，再一次判断循环的条件。

Best Practices

使用空语句时应该加上注释，从而令读这段代码的人知道该语句是有意省略的。

别漏写分号，也别多写分号

因为空语句是一条语句，所以可用在任何允许使用语句的地方。由于这个原因，某些看起来非法的分号往往只不过是一条空语句而已，从语法上说得过去。下面的片段包含两条语句：表达式语句和空语句。

173 `ival = v1 + v2;; // 正确：第二个分号表示一条多余的空语句`

多余的空语句一般来说是无害的，但是如果在 `if` 或者 `while` 的条件后面跟了一个额外的分号就可能完全改变程序员的初衷。例如，下面的代码将无休止地循环下去：

```
// 出现了糟糕的情况：额外的分号，循环体是那条空语句
while (iter != svec.end()) ;           // while 循环体是那条空语句
    ++iter;                            // 递增运算不属于循环的一部分
```

虽然从形式上来看执行递增运算的语句前面有缩进，但它并不是循环的一部分。循环条件后面跟着的分号构成了一条空语句，它才是真正的循环体。



多余的空语句并非总是无害的。

复合语句（块）

复合语句（compound statement）是指用花括号括起来的（可能为空的）语句和声明的序列，复合语句也被称作块（block）。一个块就是一个作用域（参见 2.2.4 节，第 43 页），在块中引入的名字只能在块内部以及嵌套在块中的子块里访问。通常，名字在有限的区域内可见，该区域从名字定义处开始，到名字所在的（最内层）块的结尾为止。

如果在程序的某个地方，语法上需要一条语句，但是逻辑上需要多条语句，则应该使用复合语句。例如，`while` 或者 `for` 的循环体必须是一条语句，但是我们常常需要在循环体内做很多事情，此时就需要将多条语句用花括号括起来，从而把语句序列转变成块。

举个例子，回忆 1.4.1 节（第 10 页）的 `while` 循环：

```
while (val <= 10) {  
    sum += val;      // 把 sum + val 的值赋给 sum.  
    ++val;          // 给 val 加 1  
}
```

程序从逻辑上来说要执行两条语句，但是 `while` 循环只能容纳一条。此时，把要执行的语句用花括号括起来，就将其转换成了一条（复合）语句。



块不以分号作为结束。

所谓空块，是指内部没有任何语句的一对花括号。空块的作用等价于空语句：

```
while (cin >> s && s != sought)  
{ } // 空块
```

5.1 节练习

174

练习 5.1：什么是空语句？什么时候会用到空语句？

练习 5.2：什么是块？什么时候会用到块？

练习 5.3：使用逗号运算符（参见 4.10 节，第 140 页）重写 1.4.1 节（第 10 页）的 `while` 循环，使它不再需要块，观察改写之后的代码的可读性提高了还是降低了。

5.2 语句作用域

可以在 `if`、`switch`、`while` 和 `for` 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用范围了：

```
while (int i = get_num()) // 每次迭代时创建并初始化 i  
    cout << i << endl;  
i = 0; // 错误：在循环外部无法访问 i
```

如果其他代码也需要访问控制变量，则变量必须定义在语句的外部：

```
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0
```

因为控制结构定义的对象的值马上要由结构本身使用，所以这些变量必须初始化。

5.2 节练习

练习 5.4：说明下列例子的含义，如果存在问题，试着修改它。

```
(a) while (string::iterator iter != s.end()) { /* ... */ }
(b) while (bool status = find(word)) { /* ... */ }
    if (!status) { /* ... */ }
```

5.3 条件语句

C++语言提供了两种按条件执行的语句。一种是 **if** 语句，它根据条件决定控制流；另外一种是 **switch** 语句，它计算一个整型表达式的值，然后根据这个值从几条执行路径中选择一条。



5.3.1 if 语句

175> **if 语句 (if statement)** 的作用是：判断一个指定的条件是否为真，根据判断结果决定是否执行另外一条语句。**if** 语句包括两种形式：一种含有 **else** 分支，另外一种没有。简单 **if** 语句的语法形式是

```
if (condition)
    statement
```

if else 语句的形式是

```
if (condition)
    statement
else
    statement2
```

在这两个版本的 **if** 语句中，*condition* 都必须用圆括号包围起来。*condition* 可以是一个表达式，也可以是一个初始化了的变量声明（参见 5.2 节，第 155 页）。不管是表达式还是变量，其类型都必须能转换成（参见 4.11 节，第 141 页）布尔类型。通常情况下，*statement* 和 *statement2* 是块语句。

如果 *condition* 为真，执行 *statement*。当 *statement* 执行完成后，程序继续执行 **if** 语句后面的其他语句。

如果 *condition* 为假，跳过 *statement*。对于简单 **if** 语句来说，程序继续执行 **if** 语句后面的其他语句；对于 **if else** 语句来说，执行 *statement2*。

使用 if else 语句

我们举个例子来说明 if 语句的功能，程序的目的是把数字形式表示的成绩转换成字母形式。假设数字成绩的范围是从 0 到 100（包括 100 在内），其中 100 分对应的字母形式是“A++”，低于 60 分的成绩对应的字母形式是“F”。其他成绩每 10 个划分成一组：60 到 69（包括 69 在内）对应字母“D”、70 到 79 对应字母“C”，以此类推。使用 vector 对象存放字母成绩所有可能的取值：

```
const vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

我们使用 if else 语句解决该问题，根据成绩是否合格执行不同的操作：

```
// 如果 grade 小于 60，对应的字母是 F；否则计算其下标
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

判断 grade 的值是否小于 60，根据结果选择执行 if 分支还是 else 分支。在 else 分支中，由成绩计算得到一个下标，具体过程是：首先从 grade 中减去 50，然后执行整数除法（参见 4.2 节，在 125 页），去掉余数后所得的商就是数组 scores 对应的下标。

嵌套 if 语句

176

接下来让我们的程序更有趣点儿，试着给那些合格的成绩后面添加一个加号或减号。如果成绩的末位是 8 或者 9，添加一个加号；如果末位是 0、1 或 2，添加一个减号：

```
if (grade % 10 > 7)
    lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
else if (grade % 10 < 3)
    lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

我们使用取模运算符（参见 4.2 节，第 125 页）计算余数，根据余数决定添加哪种符号。

接着把这段添加符号的代码整合到转换成绩形式的代码中去：

```
// 如果成绩不合格，不需要考虑添加加号减号的问题
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10]; // 获得字母形式的成绩
    if (grade != 100) // 只要不是 A++，就考虑添加加号或减号
        if (grade % 10 > 7)
            lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
        else if (grade % 10 < 3)
            lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
}
```

注意，我们使用花括号把第一个 else 后面的两条语句组合成了一个块。如果 grade 不小于 60 要做两件事：从数组 scores 中获取对应的字母成绩，然后根据条件设置加号或减号。

注意使用花括号

有一种常见的错误：本来程序中有几条语句应该作为一个块来执行，但是我们忘了用花括号把这些语句包围。在下面的例子中，添加加号减号的代码将被无条件地执行，这显然违背了我们的初衷：

```
if (grade < 60)
    lettergrade = scores[0];
else // 错误：缺少花括号
    lettergrade = scores[(grade - 50) / 10];
// 虽然下面的语句从形式上看有缩进，但是因为没有花括号，
// 所以无论什么情况都会执行接下来的代码
// 不及格的成绩也会添加上加号或减号，这显然是错误的
if (grade != 100)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else if (grade % 10 < 3)
        lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

要想发现这个错误可能非常困难，毕竟这段代码“看起来”是正确的。

177 为了避免此类问题，有些编码风格要求在 `if` 或 `else` 之后必须写上花括号（对 `while` 和 `for` 语句的循环体两端也有同样的要求）。这么做的好处是可以避免代码混乱不清，以后修改代码时如果想添加别的语句，也可以很容易地找到正确位置。



许多编辑器和开发环境都提供一种辅助工具，它可以自动地缩进代码以匹配其语法结构。善用此类工具益处多多。

悬垂 `else`

当一个 `if` 语句嵌套在另一个 `if` 语句内部时，很可能 `if` 分支会多于 `else` 分支。事实上，之前那个成绩转换的程序就有 4 个 `if` 分支，而只有 2 个 `else` 分支。这时候问题出现了：我们怎么知道某个给定的 `else` 是和哪个 `if` 匹配呢？

这个问题通常称作 **悬垂 `else`**（dangling `else`），在那些既有 `if` 语句又有 `if` `else` 语句的编程语言中是个普遍存在的问题。不同语言解决该问题的思路也不同，就 C++ 而言，它规定 `else` 与离它最近的尚未匹配的 `if` 匹配，从而消除了程序的二义性。

当代码中 `if` 分支多于 `else` 分支时，程序员有时会感觉比较麻烦。举个例子来说明，对于添加加号减号的那个最内层的 `if` `else` 语句，我们用另外一组条件改写它：

```
// 错误：实际的执行过程并非像缩进格式显示的那样；else 分支匹配的是内层 if 语句
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else
        lettergrade += '-'; // 末尾是 3、4、5、6 或者 7 的成绩添加一个减号！
```

从代码的缩进格式来看，程序的初衷应该是希望 `else` 和外层的 `if` 匹配，也就是说，我们希望当 `grade` 的末位小于 3 时执行 `else` 分支。然而，不管我们是什么意图，也不管程序如何缩进，这里的 `else` 分支其实是内层 `if` 语句的一部分。最终，上面的代码将在末位大于 3 小于等于 7 的成绩后面添加减号！它的执行过程实际上等价于如下形式：

```
// 缩进格式与执行过程相符，但不是程序员的意图
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else
        lettergrade += '-'; // 末尾是 3、4、5、6 或者 7 的成绩添加一个减号！
```

使用花括号控制执行路径

要想使 `else` 分支和外层的 `if` 语句匹配起来，可以在内层 `if` 语句的两端加上花括号，使其成为一个块：

```
// 末尾是 8 或者 9 的成绩添加一个加号，末尾是 0、1 或者 2 的成绩添加一个减号
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
} else // 花括号强迫 else 与外层 if 匹配
    lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

语句属于块，意味着语句一定在块的边界之内，因此内层 `if` 语句在关键字 `else` 前面的那个花括号处已经结束了。`else` 不会再作为内层 `if` 的一部分。此时，最近的尚未匹配的 `if` 是外层 `if`，也就是我们希望 `else` 匹配的那个。

< 178

5.3.1 节练习

练习 5.5: 写一段自己的程序，使用 `if else` 语句实现把数字成绩转换成字母成绩的要求。

练习 5.6: 改写上一题的程序，使用条件运算符（参见 4.7 节，第 134 页）代替 `if else` 语句。

练习 5.7: 改正下列代码段中的错误。

- `if (ival1 != ival2)`
 `ival1 = ival2`
`else ival1 = ival2 = 0;`
- `if (ival < minval)`
 `minval = ival;`
 `occurs = 1;`
- `if (int ival = get_value())`
 `cout << "ival = " << ival << endl;`
`if (!ival)`
 `cout << "ival = 0\n";`
- `if (ival = 0)`
 `ival = get_value();`

练习 5.8: 什么是“悬垂 `else`”？C++语言是如何处理 `else` 子句的？

5.3.2 switch 语句

switch 语句 (`switch statement`) 提供了一条便利的途径使得我们能够在若干固定选项中做出选择。举个例子，假如我们想统计五个元音字母在文本中出现的次数，程序逻辑应该如下所示：

- 从输入的内容中读取所有字符。
- 令每一个字符都与元音字母的集合比较。
- 如果字符与某个元音字母匹配，将该字母的数量加 1。
- 显示结果。

例如，以（原书中）本章的文本作为输入内容，程序的输出结果将是：

```
Number of vowel a: 3195
Number of vowel e: 6230
Number of vowel i: 3102
Number of vowel o: 3289
Number of vowel u: 1033
```

179 要想实现这项功能，直接使用 switch 语句即可：

```
// 为每个元音字母初始化其计数值
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // 如果 ch 是元音字母，将其对应的计数值加 1
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// 输出结果
cout << "Number of vowel a: \t" << aCnt << '\n'
    << "Number of vowel e: \t" << eCnt << '\n'
    << "Number of vowel i: \t" << iCnt << '\n'
    << "Number of vowel o: \t" << oCnt << '\n'
    << "Number of vowel u: \t" << uCnt << endl;
```

switch 语句首先对括号里的表达式求值，该表达式紧跟在关键字 switch 的后面，可以是一个初始化的变量声明（参见 5.2 节，第 155 页）。表达式的值转换成整数类型，然后与每个 case 标签的值比较。

如果表达式和某个 case 标签的值匹配成功，程序从该标签之后的第一条语句开始执行，直到到达了 switch 的结尾或者是遇到一条 break 语句为止。

我们将在 5.5.1 节（第 170 页）详细介绍 break 语句，简言之，break 语句的作用是

中断当前的控制流。此例中, `break` 语句将控制权转移到 `switch` 语句外面。因为 `switch` 是 `while` 循环体内唯一的语句, 所以从 `switch` 语句中断出来以后, 程序的控制权将移到 `while` 语句的右花括号处。此时 `while` 语句内部没有其他语句要执行, 所以 `while` 会返回去再一次判断条件是否满足。

如果 `switch` 语句的表达式和所有 `case` 都没有匹配上, 将直接跳转到 `switch` 结构之后的第一条语句。刚刚说过, 在上面的例子中, 退出 `switch` 后控制权回到 `while` 语句的条件部分。

`case` 关键字和它对应的值一起被称为 **case 标签** (`case label`)。`case` 标签必须是整型常量表达式 (参见 2.4.4 节, 第 58 页):

```
char ch = getVal();
int ival = 42;
switch(ch) {
    case 3.14: // 错误: case 标签不是一个整数
    case ival: // 错误: case 标签不是一个常量
    //...
```

< 180

任何两个 `case` 标签的值不能相同, 否则就会引发错误。另外, `default` 也是一种特殊的 `case` 标签, 关于它的知识将在第 162 页介绍。

switch 内部的控制流

理解程序在 `case` 标签之间的执行流程非常重要。如果某个 `case` 标签匹配成功, 将从该标签开始往后顺序执行所有 `case` 分支, 除非程序显式地中断了这一过程, 否则直到 `switch` 的结尾处才会停下来。要想避免执行后续 `case` 分支的代码, 我们必须显式地告诉编译器终止执行过程。大多数情况下, 在下一个 `case` 标签之前应该有一条 `break` 语句。

然而, 也有一些时候默认的 `switch` 行为才是程序真正需要的。每个 `case` 标签只能对应一个值, 但是有时候我们希望两个或更多个值共享同一组操作。此时, 我们就故意省略掉 `break` 语句, 使得程序能够连续执行若干个 `case` 标签。

例如, 也许我们想统计的是所有元音字母出现的总次数:

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    // 出现了 a、e、i、o 或 u 中的任意一个都会将 vowelCnt 的值加 1
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

在上面的代码中, 几个 `case` 标签连写在一起, 中间没有 `break` 语句。因此只要 `ch` 是元音字母, 不管到底是五个中的哪一个都执行相同的代码。

C++程序的形式比较自由, 所以 `case` 标签之后不一定非得换行。把几个 `case` 标签

写在一行里，强调这些 case 代表的是某个范围内的值：

```
switch (ch)
{
    // 另一种合法的书写形式
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

181



一般不要省略 case 分支最后的 break 语句。如果没写 break 语句，最好加一段注释说清楚程序的逻辑。

漏写 break 容易引发缺陷

有一种常见的错觉是程序只执行匹配成功的那个 case 分支的语句。例如，下面程序的统计结果是错误的：

```
// 警告：不正确的程序逻辑！
switch (ch) {
    case 'a':
        ++aCnt; // 此处应该有一条 break 语句
    case 'e':
        ++eCnt; // 此处应该有一条 break 语句
    case 'i':
        ++iCnt; // 此处应该有一条 break 语句
    case 'o':
        ++oCnt; // 此处应该有一条 break 语句
    case 'u':
        ++uCnt;
}
```

要想理解这段程序的执行过程，不妨假设 ch 的值是 'e'。此时，程序直接执行 case 'e' 标签后面的代码，该代码把 eCnt 的值加 1。接下来，程序将跨越 case 标签的边界，接着递增 iCnt、oCnt 和 uCnt。



尽管 switch 语句不是非得在最后一个标签后面写上 break，但是为了安全起见，最好这么做。因为这样的话，即使以后再增加新的 case 分支，也不用再在前面补充 break 语句了。

default 标签

如果没有任何一个 case 标签能匹配上 switch 表达式的值，程序将执行紧跟在 **default 标签 (default label)** 后面的语句。例如，可以增加一个计数值来统计非元音字母的数量，只要在 default 分支内不断递增名为 otherCnt 的变量就可以了：

```
// 如果 ch 是一个元音字母，将相应的计数值加 1
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
```

```

        ++otherCnt;
        break;
    }
}

```

在这个版本的程序中，如果 ch 不是元音字母，就从 default 标签开始执行并把 otherCnt 加 1。 ◀182



即使不准备在 default 标签下做任何工作，定义一个 default 标签也是有用的。其目的在于告诉程序的读者，我们已经考虑到了默认的情况，只是目前什么也没做。

标签不应该孤零零地出现，它后面必须跟上一条语句或者另外一个 case 标签。如果 switch 结构以一个空的 default 标签作为结束，则该 default 标签后面必须跟上一条空语句或一个空块。

switch 内部的变量定义

如前所述，switch 的执行流程有可能会跨过某些 case 标签。如果程序跳转到了某个特定的 case，则 switch 结构中该 case 标签之前的部分会被忽略掉。这种忽略掉一部分代码的行为引出了一个有趣的问题：如果被略过的代码中含有变量的定义该怎么办？

答案是：如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法行为。

```

case true:
    // 因为程序的执行流程可能绕开下面的初始化语句，所以该 switch 语句不合法
    string file_name;      // 错误：控制流绕过一个隐式初始化的变量
    int ival = 0;           // 错误：控制流绕过一个显式初始化的变量
    int jval;               // 正确：因为 jval 没有初始化
    break;
case false:
    // 正确：jval 虽然在作用域内，但是它没有被初始化
    jval = next_num();     // 正确：给 jval 赋一个值
    if (file_name.empty()) // file_name 在作用域内，但是没有被初始化
        ...

```

假设上述代码合法，则一旦控制流直接跳到 false 分支，也就同时略过了变量 file_name 和 ival 的初始化过程。此时这两个变量位于作用域之内，跟在 false 之后的代码试图在尚未初始化的情况下使用它们，这显然是行不通的。因此 C++ 语言规定，不允许跨过变量的初始化语句直接跳转到该变量作用域内的另一个位置。

如果需要为某个 case 分支定义并初始化一个变量，我们应该把变量定义在块内，从而确保后面的所有 case 标签都在变量的作用域之外。

```

case true:
{
    // 正确：声明语句位于语句块内部
    string file_name = get_file_name();
    ...
}
break;
case false:

```

```
if (file_name.empty()) // 错误: file_name 不在作用域之内
```

183>

5.3.2 节练习

练习 5.9: 编写一段程序，使用一系列 if 语句统计从 cin 读入的文本中有多少元音字母。

练习 5.10: 我们之前实现的统计元音字母的程序存在一个问题：如果元音字母以大写形式出现，不会被统计在内。编写一段程序，既统计元音字母的小写形式，也统计大写形式，也就是说，新程序遇到'a' 和'A' 都应该递增 aCnt 的值，以此类推。

练习 5.11: 修改统计元音字母的程序，使其也能统计空格、制表符和换行符的数量。

练习 5.12: 修改统计元音字母的程序，使其能统计以下含有两个字符的字符序列的数量：ff、fl 和 fi。

练习 5.13: 下面显示的每个程序都含有一个常见的编程错误，指出错误在哪里，然后修改它们。

```
(a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
    char ch = next_text();
    switch (ch) {
        case 'a': aCnt++;
        case 'e': eCnt++;
        default: iouCnt++;
    }

(b) unsigned index = some_value();
    switch (index) {
        case 1:
            int ix = get_value();
            ivec[ ix ] = index;
            break;
        default:
            ix = ivec.size()-1;
            ivec[ ix ] = index;
    }

(c) unsigned evenCnt = 0, oddCnt = 0;
    int digit = get_num() % 10;
    switch (digit) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
    }

(d) unsigned ival=512, jval=1024, kval=4096;
    unsigned bufsize;
    unsigned swt = get_bufCnt();
    switch(swt) {
```

```

    case ival:
        bufsize = ival * sizeof(int);
        break;
    case jval:
        bufsize = jval * sizeof(int);
        break;
    case kval:
        bufsize = kval * sizeof(int);
        break;
}

```

5.4 迭代语句

迭代语句通常称为循环，它重复执行操作直到满足某个条件才停下来。`while` 和 `for` 语句在执行循环体之前检查条件，`do while` 语句先执行循环体，然后再检查条件。

5.4.1 while 语句



只要条件为真，`while` 语句（while statement）就重复地执行循环体，它的语法形式是

```
while (condition)
    statement
```

在 `while` 结构中，只要 `condition` 的求值结果为真就一直执行 `statement`（常常是一个块）。`condition` 不能为空，如果 `condition` 第一次求值就得 `false`，`statement` 一次都不执行。

`while` 的条件部分可以是一个表达式或者是一个带初始化的变量声明（参见 5.2 节，第 155 页）。通常来说，应该由条件本身或者是循环体设法改变表达式的值，否则循环可能无法终止。



定义在 `while` 条件部分或者 `while` 循环体内的变量每次迭代都经历从创建到销毁的过程。

使用 while 循环

当不确定到底要迭代多少次时，使用 `while` 循环比较合适，比如读取输入的内容就是如此。还有一种情况也应该使用 `while` 循环，这就是我们想在循环结束后访问循环控制变量。例如：

```

vector<int> v;
int i;
// 重复读入数据，直至到达文件末尾或者遇到其他输入问题
while (cin >> i)
    v.push_back(i);
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0

```

第一个循环从标准输入中读取数据，我们一开始不清楚循环要执行多少次，当 `cin` 读取到无效数据、遇到其他一些输入错误或是到达文件末尾时循环条件失效。第二个循环重复执行直到遇到一个负值为止，循环终止后，`beg` 或者等于 `v.end()`，或者指向 `v` 中一个小于 0 的元素。可以在 `while` 循环外继续使用 `beg` 的状态以进行其他处理。

5.4.1 节练习

练习 5.14：编写一段程序，从标准输入中读取若干 `string` 对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词。如果这样的单词存在，输出重复出现的最大次数；如果不存在，输出一条信息说明任何单词都没有连续出现过。例如，如果输入是

```
how now now now brown cow cow
```

那么输出应该表明单词 `now` 连续出现了 3 次。



5.4.2 传统的 for 语句

for 语句的语法形式是

```
for (init-statement; condition; expression)
    statement
```

关键字 `for` 及括号里的部分称作 `for` 语句头。

`init-statement` 必须是以下三种形式中的一种：声明语句、表达式语句或者空语句，因为这些语句都以分号作为结束，所以 `for` 语句的语法形式也可以看做

```
for (initializer; condition; expression)
    statement
```

一般情况下，`init-statement` 负责初始化一个值，这个值将随着循环的进行而改变。

`condition` 作为循环控制的条件，只要 `condition` 为真，就执行一次 `statement`。如果 `condition` 第一次的求值结果就是 `false`，则 `statement` 一次也不会执行。`expression` 负责修改 `init-statement` 初始化的变量，这个变量正好就是 `condition` 检查的对象，修改发生在每次循环迭代之后。`statement` 可以是一条单独的语句也可以是一条复合语句。

传统 for 循环的执行流程

我们以 3.2.3 节（第 85 页）的 `for` 循环为例：

```
// 重复处理 s 中的字符直至我们处理完全部字符或者遇到了一个表示空白的字符
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
    s[index] = toupper(s[index]); // 将当前字符改成大写形式
```

求值的顺序如下所示：

1. 循环开始时，首先执行一次 `init-statement`。此例中，定义 `index` 并初始化为 0。
2. 接下来判断 `condition`。如果 `index` 不等于 `s.size()` 而且在 `s[index]` 位置的字符不是空白，则执行 `for` 循环体的内容。否则，循环终止。如果第一次迭代时条件就为假，`for` 循环体一次也不会执行。
3. 如果条件为真，执行循环体。此例中，`for` 循环体将 `s[index]` 位置的字符改写

成大写形式。

4. 最后执行 *expression*。此例中，将 *index* 的值加 1。

这 4 步说明了 `for` 循环第一次迭代的过程。其中第 1 步只在循环开始时执行一次，第 2、3、4 步重复执行直到条件为假时终止，也就是在 *s* 中遇到一个空白字符或者 *index* 大于 *s.size()* 时终止。



牢记 `for` 语句头中定义的对象只在 `for` 循环体内可见。因此在上面的例子中，`for` 循环结束后 *index* 就不可用了。

for 语句头中的多重定义

和其他的声明一样，*init-statement* 也可以定义多个对象。但是 *init-statement* 只能有一条声明语句，因此，所有变量的基础类型必须相同（参见 2.3 节，第 45 页）。举个例子，我们用下面的循环把 `vector` 的元素拷贝一份添加到原来的元素后面：

```
// 记录下 v 的大小，当到达原来的最后一个元素后结束循环
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

在这个循环中，我们在 *init-statement* 里同时定义了索引 *i* 和循环控制变量 *sz*。

省略 for 语句头的某些部分

187

`for` 语句头能省略掉 *init-statement*、*condition* 和 *expression* 中的任何一个（或者全部）。

如果无须初始化，则我们可以使用一条空语句作为 *init-statement*。例如，对于在 `vector` 对象中寻找第一个负数的程序，完全能用 `for` 循环改写：

```
auto beg = v.begin();
for (/* 空语句 */; beg != v.end() && *beg >= 0; ++beg)
    ; // 什么也不做
```

注意，分号必须保留以表明我们省略掉了 *init-statement*。说得更准确一点，分号表示的是一个空的 *init-statement*。在这个循环中，因为所有要做的工作都在 `for` 语句头的条件和表达式部分完成了，所以 `for` 循环体也是空的。其中，条件部分决定何时停止查找，表达式部分递增迭代器。

省略 *condition* 的效果等价于在条件部分写了一个 `true`。因为条件的值永远是 `true`，所以在循环体内必须有语句负责退出循环，否则循环就会无休止地执行下去：

```
for (int i = 0; /* 条件为空 */ ; ++i) {
    // 对 i 进行处理，循环内部的代码必须负责终止迭代过程!
}
```

我们也能省略掉 `for` 语句头中的 *expression*，但是在这样的循环中就要求条件部分或者循环体必须改变迭代变量的值。举个例子，之前有一个将整数读入 `vector` 的 `while` 循环，我们使用 `for` 语句改写它：

```
vector<int> v;
for (int i; cin >> i; /* 表达式为空 */ )
    v.push_back(i);
```

因为条件部分能改变 *i* 的值，所以这个循环无须表达式部分。其中，条件部分不断检查输

入流的内容，只要读取完所有的输入或者遇到一个输入错误就终止循环。

5.4.2 节练习

练习 5.15：说明下列循环的含义并改正其中的错误。

```
(a) for (int ix = 0; ix != sz; ++ix) { /* ... */ }
    if (ix != sz)
        // ...
(b) int ix;
    for (ix != sz; ++ix) { /* ... */ }
(c) for (int ix = 0; ix != sz; ++ix, ++sz) { /* ... */ }
```

练习 5.16：while 循环特别适用于那种条件保持不变、反复执行操作的情况，例如，当未达到文件末尾时不断读取下一个值。for 循环则更像是在按步骤迭代，它的索引值在某个范围内依次变化。根据每种循环的习惯用法各自编写一段程序，然后分别用另一种循环改写。如果只能使用一种循环，你倾向于使用哪种呢？为什么？

练习 5.17：假设有两个包含整数的 vector 对象，编写一段程序，检验其中一个 vector 对象是否是另一个的前缀。为了实现这一目标，对于两个不等长的 vector 对象，只需挑出长度较短的那个，把它的所有元素和另一个 vector 对象比较即可。例如，如果两个 vector 对象的元素分别是 0、1、1、2 和 0、1、1、2、3、5、8，则程序的返回结果应该为真。



5.4.3 范围 for 语句

C++11 新标准引入了一种更简单的 for 语句，这种语句可以遍历容器或其他序列的所有元素。范围 for 语句（range for statement）的语法形式是：

```
for (declaration : expression)
    statement
```

expression 表示的必须是一个序列，比如用花括号括起来的初始值列表（参见 3.3.1 节，第 88 页）、数组（参见 3.5 节，第 101 页）或者 vector 或 string 等类型的对象，这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员（参见 3.4 节，第 95 页）。

declaration 定义一个变量，序列中的每个元素都得能转换成该变量的类型（参见 4.11 节，第 141 页）。确保类型相容最简单的办法是使用 auto 类型说明符（参见 2.5.2 节，第 61 页），这个关键字可以令编译器帮助我们指定合适的类型。如果需要对序列中的元素执行写操作，循环变量必须声明成引用类型。

每次迭代都会重新定义循环控制变量，并将其初始化成序列中的下一个值，之后才会执行 *statement*。像往常一样，*statement* 可以是一条单独的语句也可以是一个块。所有元素都处理完毕后循环终止。

之前我们已经接触过几个这样的循环。接下来的例子将把 vector 对象中的每个元素都翻倍，它涵盖了范围 for 语句的几乎所有语法特征：

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// 范围变量必须是引用类型，这样才能对元素执行写操作
for (auto &r : v)      // 对于 v 中的每一个元素
```

```
r *= 2; // 将 v 中每个元素的值翻倍
```

for 语句头声明了循环控制变量 r，并把它和 v 关联在一起，我们使用关键字 auto 令编译器为 r 指定正确的类型。由于准备修改 v 的元素的值，因此将 r 声明成引用类型。此时，在循环体内给 r 赋值，即改变了 r 所绑定的元素的值。

范围 for 语句的定义来源于与之等价的传统 for 语句：

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // r 必须是引用类型，这样才能对元素执行写操作
    r *= 2; // 将 v 中每个元素的值翻倍
}
```

学习了范围 for 语句的原理之后，我们也就不难理解为什么在 3.3.2 节（第 90 页）强调不能通过范围 for 语句增加 vector 对象（或者其他容器）的元素了。在范围 for 语句中，预存了 end() 的值。一旦在序列中添加（删除）元素，end 函数的值就可能变得无效了（参见 3.4.1 节，第 98 页）。关于这一点，将在 9.3.6 节（第 315 页）做更详细的介绍。189

5.4.4 do while 语句

do while 语句（do while statement）和 while 语句非常相似，唯一的区别是，do while 语句先执行循环体后检查条件。不管条件的值如何，我们都至少执行一次循环。do while 语句的语法形式如下所示：

```
do
    statement
    while (condition);
```



do while 语句应该在括号包围起来的条件后面用一个分号表示语句结束。

在 do 语句中，求 condition 的值之前首先执行一次 statement，condition 不能为空。如果 condition 的值为假，循环终止；否则，重复循环过程。condition 使用的变量必须定义在循环体之外。

我们可以使用 do while 循环（不断地）执行加法运算：

```
// 不断提示用户输入一对数，然后求其和
string rsp; // 作为循环的条件，不能定义在 do 的内部
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
        << " = " << val1 + val2 << "\n\n"
        << "More? Enter yes or no: ";
    cin >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

循环首先提示用户输入两个数字，然后输出它们的和并询问用户是否继续。条件部分检查用户做出的回答，如果用户没有回答，或者用户的回答以字母 n 开始，循环都将终止。否则循环继续执行。

因为对于 do while 来说先执行语句或者块，后判断条件，所以不允许在条件部分

定义变量：

```
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // 错误：将变量声明放在了 do 的条件部分
```

如果允许在条件部分定义变量，则变量的使用出现在定义之前，这显然是不合常理的！

5.4.4 节练习

练习 5.18：说明下列循环的含义并改正其中的错误。

```
(a) do
    int v1, v2;
    cout << "Please enter two numbers to sum:" ;
    if (cin >> v1 >> v2)
        cout << "Sum is: " << v1 + v2 << endl;
    while (cin);
(b) do {
    // ...
} while (int ival = get_response());
(c) do {
    int ival = get_response();
} while (ival);
```

练习 5.19：编写一段程序，使用 do while 循环重复地执行下述任务：首先提示用户输入两个 string 对象，然后挑出较短的那个并输出它。

5.5 跳转语句

跳转语句中断当前的执行过程。C++语言提供了 4 种跳转语句：`break`、`continue`、`goto` 和 `return`。本章介绍前三种跳转语句，`return` 语句将在 6.3 节（第 199 页）进行介绍。

5.5.1 break 语句

break 语句（`break statement`）负责终止离它最近的 `while`、`do while`、`for` 或 `switch` 语句，并从这些语句之后的第一条语句开始继续执行。

`break` 语句只能出现在迭代语句或者 `switch` 语句内部（包括嵌套在此类循环里的语句或块的内部）。`break` 语句的作用范围仅限于最近的循环或者 `switch`：

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // 处理到第一个空白为止
        for (auto it = buf.begin() + 1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #1, 离开 for 循环
        }
    }
}
```

```

    }
    // break #1 将控制权转移到这里
    // 剩余的'-'处理:
    break; // #2, 离开 switch 语句
    case '+':
        //...
    } // 结束 switch
    // 结束 switch: break #2 将控制权转移到这里
} // 结束 while

```

< 191

标记为#1 的 break 语句负责终止连字符 case 标签后面的 for 循环。它不但不会终止 switch 语句，甚至连当前的 case 分支也终止不了。接下来，程序继续执行 for 循环之后的第一条语句，这条语句可能接着处理连字符的情况，也可能是另一条用于终止当前分支的 break 语句。

标记为#2 的 break 语句负责终止 switch 语句，但是不能终止 while 循环。执行完这个 break 后，程序继续执行 while 的条件部分。

5.5.1 节练习

练习 5.20: 编写一段程序，从标准输入中读取 string 对象的序列直到连续出现两个相同的单词或者所有单词都读完为止。使用 while 循环一次读取一个单词，当一个单词连续出现两次时使用 break 语句终止循环。输出连续重复出现的单词，或者输出一个消息说明没有任何单词是连续重复出现的。

5.5.2 continue 语句

continue 语句 (continue statement) 终止最近的循环中的当前迭代并立即开始下一次迭代。continue 语句只能出现在 for、while 和 do while 循环的内部，或者嵌套在此类循环里的语句或块的内部。和 break 语句类似的是，出现在嵌套循环中的 continue 语句也仅作用于离它最近的循环。和 break 语句不同的是，只有当 switch 语句嵌套在迭代语句内部时，才能在 switch 里使用 continue。

continue 语句中断当前的迭代，但是仍然继续执行循环。对于 while 或者 do while 语句来说，继续判断条件的值；对于传统的 for 循环来说，继续执行 for 语句头的 expression；而对于范围 for 语句来说，则是用序列中的下一个元素初始化循环控制变量。

例如，下面的程序每次从标准输入中读取一个单词。循环只对那些以下画线开头的单词感兴趣，其他情况下，我们直接终止当前的迭代并获取下一个单词：

```

string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // 接着读取下一个输入
    // 程序执行过程到了这里？说明当前的输入是以下画线开始的；接着处理 buf……
}

```

< 192

5.5.2 节练习

练习 5.21: 修改 5.5.1 节（第 171 页）练习题的程序，使其找到的重复单词必须以大写字母开头。

5.5.3 goto 语句

goto 语句 (goto statement) 的作用是从 **goto** 语句无条件跳转到同一函数内的另一条语句。



不要在程序中使用 **goto** 语句，因为它使得程序既难理解又难修改。

goto 语句的语法形式是

```
goto label;
```

其中，*label* 是用于标识一条语句的标示符。带标签语句 (labeled statement) 是一种特殊的语句，在它之前有一个标示符以及一个冒号：

```
end: return; // 带标签语句，可以作为 goto 的目标
```

标签标示符独立于变量或其他标示符的名字，因此，标签标示符可以和程序中其他实体的标示符使用同一个名字而不会相互干扰。**goto** 语句和控制权转向的那条带标签的语句必须位于同一个函数之内。

和 **switch** 语句类似，**goto** 语句也不能将程序的控制权从变量的作用域之外转移到作用域之内：

```
// ...
goto end;
int ix = 10; // 错误：goto 语句绕过了一个带初始化的变量定义
end:
// 错误：此处的代码需要使用 ix，但是 goto 语句绕过了它的声明
ix = 42;
```

向后跳过一个已经执行的定义是合法的。跳回到变量定义之前意味着系统将销毁该变量，然后重新创建它：

```
// 向后跳过一个带初始化的变量定义是合法的
begin:
int sz = get_size();
if (sz <= 0) {
    goto begin;
}
```

在上面的代码中，**goto** 语句执行后将销毁 *sz*。因为跳回到 *begin* 的动作跨过了 *sz* 的定义语句，所以 *sz* 将重新定义并初始化。

193

5.5.3 节练习

练习 5.22：本节的最后一个例子跳回到 *begin*，其实使用循环能更好地完成该任务。重写这段代码，注意不再使用 **goto** 语句。

5.6 try 语句块和异常处理

异常是指存在于运行时的反常行为，这些行为超出了函数正常功能的范围。典型的异常包括失去数据库连接以及遇到意外输入等。处理反常行为可能是设计所有系统最难的一部分。

当程序的某部分检测到一个它无法处理的问题时，需要用到异常处理。此时，检测出问题的部分应该发出某种信号以表明程序遇到了故障，无法继续下去了，而且信号的发出方无须知道故障将在何处得到解决。一旦发出异常信号，检测出问题的部分也就完成了任务。

如果程序中含有可能引发异常的代码，那么通常也会有专门的代码处理问题。例如，如果程序的问题是输入无效，则异常处理部分可能会要求用户重新输入正确的数据；如果丢失了数据库连接，会发出报警信息。

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持。在 C++ 语言中，异常处理包括：

- **throw 表达式 (throw expression)**，异常检测部分使用 throw 表达式来表示它遇到了无法处理的问题。我们说 throw 引发 (raise) 了异常。
- **try 语句块 (try block)**，异常处理部分使用 try 语句块处理异常。try 语句块以关键字 try 开始，并以一个或多个 catch 子句 (catch clause) 结束。try 语句块中代码抛出的异常通常会被某个 catch 子句处理。因为 catch 子句“处理”异常，所以它们也被称作异常处理代码 (exception handler)。
- 一套异常类 (exception class)，用于在 throw 表达式和相关的 catch 子句之间传递异常的具体信息。

在本节的剩余部分，我们将分别介绍异常处理的这三个组成部分。在 18.1 节（第 684 页）还将介绍更多关于异常的知识。

5.6.1 throw 表达式

程序的异常检测部分使用 throw 表达式引发一个异常。throw 表达式包含关键字 throw 和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。throw 表达式后面通常紧跟一个分号，从而构成一条表达式语句。

举个简单的例子，回忆 1.5.2 节（第 20 页）把两个 Sales_item 对象相加的程序。◀ 194 这个程序检查它读入的记录是否是关于同一种书籍的，如果不是，输出一条信息然后退出。

```
Sales_item item1, item2;
cin >> item1 >> item2;
// 首先检查 item1 和 item2 是否表示同一种书籍
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // 表示成功
} else {
    cerr << "Data must refer to same ISBN" << endl;
    return -1; // 表示失败
}
```

在真实的程序中，应该把对象相加的代码和用户交互的代码分离开来。此例中，我们改写程序使得检查完成后不再直接输出一条信息，而是抛出一个异常：

```
// 首先检查两条数据是否是关于同一种书籍的
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// 如果程序执行到了这里，表示两个 ISBN 是相同的
cout << item1 + item2 << endl;
```

在这段代码中，如果 ISBN 不一样就抛出一个异常，该异常是类型 `runtime_error` 的对象。抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码。

类型 `runtime_error` 是标准库异常类型的一种，定义在 `stdexcept` 头文件中。关于标准库异常类型更多的知识将在 5.6.3 节（第 176 页）介绍。我们必须初始化 `runtime_error` 的对象，方式是给它提供一个 `string` 对象或者一个 C 风格的字符串（参见 3.5.4 节，第 109 页），这个字符串中有一些关于异常的辅助信息。

5.6.2 try 语句块

`try` 语句块的通用语法形式是

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

`try` 语句块一开始是关键字 `try`，随后紧跟着一个块，这个块就像大多数时候那样是花括号括起来的语句序列。

195 跟在 `try` 块之后的是一个或多个 `catch` 子句。`catch` 子句包括三部分：关键字 `catch`、括号内一个（可能未命名的）对象的声明（称作 **异常声明**，`exception declaration`）以及一个块。当选中了某个 `catch` 子句处理异常之后，执行与之对应的块。`catch` 一旦完成，程序跳转到 `try` 语句块最后一个 `catch` 子句之后的那条语句继续执行。

`try` 语句块中的 `program-statements` 组成程序的正常逻辑，像其他任何块一样，`program-statements` 可以有包括声明在内的任意 C++ 语句。一如往常，`try` 语句块内声明的变量在块外部无法访问，特别是在 `catch` 子句内也无法访问。

编写处理代码

在之前的例子里，我们使用了一个 `throw` 表达式以避免把两个代表不同书籍的 `Sales_item` 相加。我们假设执行 `Sales_item` 对象加法的代码是与用户交互的代码分离开来的。其中与用户交互的代码负责处理发生的异常，它的形式可能如下所示：

```
while (cin >> item1 >> item2) {
    try {
        // 执行添加两个 Sales_item 对象的代码
        // 如果添加失败，代码抛出一个 runtime_error 异常
    } catch (runtime_error err) {
        // 提醒用户两个 ISBN 必须一致，询问是否重新输入
        cout << err.what()
            << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
```

```

    if (!cin || c == 'n')
        break; // 跳出 while 循环
    }
}

```

程序本来要执行的任务出现在 `try` 语句块中，这是因为这段代码可能会抛出一个 `runtime_error` 类型的异常。

`try` 语句块对应一个 `catch` 子句，该子句负责处理类型为 `runtime_error` 的异常。如果 `try` 语句块的代码抛出了 `runtime_error` 异常，接下来执行 `catch` 块内的语句。在我们书写的 `catch` 子句中，输出一段提示信息要求用户指定程序是否继续。如果用户输入 '`n`'，执行 `break` 语句并退出 `while` 循环；否则，直接执行 `while` 循环的右侧花括号，意味着程序控制权跳回到 `while` 条件部分准备下一次迭代。

给用户的提示信息中输出了 `err.what()` 的返回值。我们知道 `err` 的类型是 `runtime_error`，因此能推断 `what` 是 `runtime_error` 类的一个成员函数（参见 1.5.2 节，第 20 页）。每个标准库异常类都定义了名为 `what` 的成员函数，这些函数没有参数，返回值是 C 风格字符串（即 `const char*`）。其中，`runtime_error` 的 `what` 成员返回的是初始化一个具体对象时所用的 `string` 对象的副本。如果上一节编写的代码抛出异常，则本节的 `catch` 子句输出

```

Data must refer to same ISBN
Try Again? Enter y or n

```

函数在寻找处理代码的过程中退出

在复杂系统中，程序在遇到抛出异常的代码前，其执行路径可能已经经过了多个 `try` 语句块。例如，一个 `try` 语句块可能调用了包含另一个 `try` 语句块的函数，新的 `try` 语句块可能调用了包含又一个 `try` 语句块的新函数，以此类推。

寻找处理代码的过程与函数调用链刚好相反。当异常被抛出时，首先搜索抛出该异常的函数。如果没找到匹配的 `catch` 子句，终止该函数，并在调用该函数的函数中继续寻找。如果还是没有找到匹配的 `catch` 子句，这个新的函数也被终止，继续搜索调用它的函数。以此类推，沿着程序的执行路径逐层回退，直到找到适当类型的 `catch` 子句为止。

如果最终还是没能找到任何匹配的 `catch` 子句，程序转到名为 `terminate` 的标准库函数。该函数的行为与系统有关，一般情况下，执行该函数将导致程序非正常退出。

对于那些没有任何 `try` 语句块定义的异常，也按照类似的方式处理：毕竟，没有 `try` 语句块也就意味着没有匹配的 `catch` 子句。如果一段程序没有 `try` 语句块且发生了异常，系统会调用 `terminate` 函数并终止当前程序的执行。

提示：编写异常安全的代码非常困难

要好好理解这句话：异常中断了程序的正常流程。异常发生时，调用者请求的一部分计算可能已经完成了，另一部分则尚未完成。通常情况下，略过部分程序意味着某些对象处理到一半就戛然而止，从而导致对象处于无效或未完成的状态，或者资源没有正常释放，等等。那些在异常发生期间正确执行了“清理”工作的程序被称作异常安全（exception safe）的代码。然而经验表明，编写异常安全的代码非常困难，这部分知识也（远远）超出了本书的范围。

对于一些程序来说，当异常发生时只是简单地终止程序。此时，我们不怎么需要担

196

心异常安全的问题。

但是对于那些确实要处理异常并继续执行的程序，就要加倍注意了。我们必须时刻清楚异常何时发生，异常发生后程序应如何确保对象有效、资源无泄漏、程序处于合理状态，等等。

我们会在本书中介绍一些比较常规的提升异常安全性的技术。但是读者需要注意，如果你的程序要求非常鲁棒的异常处理，那么仅有我们介绍的这些技术恐怕还是不够的。

197 5.6.3 标准异常

C++标准库定义了一组类，用于报告标准库函数遇到的问题。这些异常类也可以在用户编写的程序中使用，它们分别定义在 4 个头文件中：

- exception 头文件定义了最通用的异常类 exception。它只报告异常的发生，不提供任何额外信息。
- stdexcept 头文件定义了几种常用的异常类，详细信息在表 5.1 中列出。
- new 头文件定义了 bad_alloc 异常类型，这种类型将在 12.1.2 节（第 407 页）详细介绍。
- type_info 头文件定义了 bad_cast 异常类型，这种类型将在 19.2 节（第 731 页）详细介绍。

表 5.1: <stdexcept> 定义的异常类

exception	最常见的问题
runtime_error	只有在运行时才能检测出的问题
range_error	运行时错误：生成的结果超出了有意义的值域范围
overflow_error	运行时错误：计算上溢
underflow_error	运行时错误：计算下溢
logic_error	程序逻辑错误
domain_error	逻辑错误：参数对应的结果值不存在
invalid_argument	逻辑错误：无效参数
length_error	逻辑错误：试图创建一个超出该类型最大长度的对象
out_of_range	逻辑错误：使用一个超出有效范围的值

标准库异常类只定义了几种运算，包括创建或拷贝异常类型的对象，以及为异常类型的对象赋值。

我们只能以默认初始化（参见 2.2.1 节，第 40 页）的方式初始化 exception、bad_alloc 和 bad_cast 对象，不允许为这些对象提供初始值。

其他异常类型的行为则恰好相反：应该使用 string 对象或者 C 风格字符串初始化这些类型的对象，但是不允许使用默认初始化的方式。当创建此类对象时，必须提供初始值，该初始值含有错误相关的信息。

异常类型只定义了一个名为 what 的成员函数，该函数没有任何参数，返回值是一个指向 C 风格字符串（参见 3.5.4 节，第 109 页）的 const char*。该字符串的目的是提供关于异常的一些文本信息。

what 函数返回的 C 风格字符串的内容与异常对象的类型有关。如果异常类型有一个字符串初始值，则 what 返回该字符串。对于其他无初始值的异常类型来说，what 返回的内容由编译器决定。

5.6.3 节练习

练习 5.23：编写一段程序，从标准输入读取两个整数，输出第一个数除以第二个数的结果。

练习 5.24：修改你的程序，使得当第二个数是 0 时抛出异常。先不要设定 catch 子句，运行程序并真的为除数输入 0，看看会发生什么？

练习 5.25：修改上一题的程序，使用 try 语句块去捕获异常。catch 子句应该为用户输出一条提示信息，询问其是否输入新数并重新执行 try 语句块的内容。

199 小结

C++语言仅提供了有限的语句类型，它们中的大多数会影响程序的控制流程：

- `while`、`for` 和 `do while` 语句，执行迭代操作。
- `if` 和 `switch` 语句，提供条件分支结构。
- `continue` 语句，终止循环的当前一次迭代。
- `break` 语句，退出循环或者 `switch` 语句。
- `goto` 语句，将控制权转移到一条带标签的语句。
- `try` 和 `catch`，将一段可能抛出异常的语句序列括在花括号里构成 `try` 语句块。
`catch` 子句负责处理代码抛出的异常。
- `throw` 表达式语句，存在于代码块中，将控制权转移到相关的 `catch` 子句。
- `return` 语句，终止函数的执行。我们将在第 6 章介绍 `return` 语句。

除此之外还有表达式语句和声明语句。表达式语句用于求解表达式，关于变量的声明和定义在第 2 章已经介绍过了。

术语表

块 (block) 包围在花括号内的由 0 条或多条语句组成的序列。块也是一条语句，所以只要是能使用语句的地方，就可以使用块。

break 语句 (break statement) 终止离它最近的循环或 `switch` 语句。控制权转移到循环或 `switch` 之后的第一条语句。

case 标签 (case label) 在 `switch` 语句中紧跟在 `case` 关键字之后的常量表达式（参见 2.4.4 节，第 58 页）。在同一个 `switch` 语句中任意两个 `case` 标签的值不能相同。

catch 子句 (catch clause) 由三部分组成：
catch 关键字、括号里的异常声明以及一个语句块。`catch` 子句的代码负责处理在异常声明中定义的异常。

复合语句 (compound statement) 和块是同义词。

continue 语句 (continue statement) 终止离它最近的循环的当前迭代。控制权转移到 `while` 或 `do while` 语句的条件部分、或者范围 `for` 循环的下一次迭代、或者传统 `for` 循环头部的表达式。

悬垂 else (dangling else) 是一个俗语，指的是如何处理嵌套 `if` 语句中 `if` 分支多于 `else` 分支的情况。C++语言规定，`else` 应该与前一个未匹配的 `if` 匹配在一起。使用花括号可以把位于内层的 `if` 语句隐藏起来，这样程序员就能更好地控制 `else` 该与哪个 `if` 匹配。

default 标签 (default label) 是一种特殊的 `case` 标签，当 `switch` 表达式的值与所有 `case` 标签都无法匹配时，程序执行 `default` 标签下的内容。

200 do while 语句 (do while statement) 与 `while` 语句类似，区别是 `do while` 语句先执行循环体，再判断条件。循环体代码至少会执行一次。

异常类 (exception class) 是标准库定义的一组类，用于表示程序发生的错误。表 5.1（第 176 页）列出了不同用途的异常类。

异常声明 (exception declaration) 位于 `catch` 子句中的声明，指定了该 `catch` 子句能处理的异常类型。

异常处理代码 (exception handler) 程序某处引发异常后，用于处理该异常的另一

处代码。和 catch 子句是同义词。

异常安全 (exception safe) 是一个术语，表示的含义是当抛出异常后，程序能执行正确的行为。

表达式语句 (expression statement) 即一条表达式后面跟上一个分号，令表达式执行求值过程。

控制流 (flow of control) 程序的执行路径。

for 语句 (for statement) 提供迭代执行的迭代语句。常用于遍历一个容器或者重复计算若干次。

goto 语句 (goto statement) 令控制权无条件转移到同一函数中一个指定的带标签语句。goto 语句容易造成程序的控制流混乱，应禁止使用。

if else 语句 (if else statement) 判断条件，根据其结果分别执行 if 分支或 else 分支的语句。

if 语句 (if statement) 判断条件，根据其结果有选择地执行语句。如果条件为真，执行 if 分支的代码；如果条件为假，控制权转移到 if 结构之后的第一条语句。

带标签语句 (labeled statement) 前面带有标签的语句。所谓标签是指一个标识符以及紧跟着的一个冒号。对于同一个标识符来说，用作标签的同时还能用于其他目的，互不干扰。

空语句 (null statement) 只含有一个分号的语句。

引发 (raise) 含义类似于 throw。在 C++ 语言中既可以说抛出异常，也可以说引发异常。

范围 for 语句 (range for statement) 在一个序列中进行迭代的语句。

switch 语句 (switch statement) 一种条件语句，首先求 switch 关键字后面表达式的值，如果某个 case 标签的值与表达式的值相等，程序直接跨过之前的代码从这个 case 标签开始执行。当所有 case 标签都无法匹配时，如果有 default 标签，从 default 标签继续执行；如果没有，结束 switch 语句。

terminate 是一个标准库函数，当异常没有被捕捉到时调用。terminate 终止当前程序的执行。

throw 表达式 (throw expression) 一种中断当前执行路径的表达式。throw 表达式抛出一个异常并把控制权转移到能处理该异常的最近的 catch 子句。

try 语句块 (try block) 跟在 try 关键字后面的块，以及一个或多个 catch 子句。如果 try 语句块的代码引发异常并且其中一个 catch 子句匹配该异常类型，则异常被该 catch 子句处理。否则，异常将由外围 try 语句块处理，或者程序终止。

while 语句 (while statement) 只要指定的条件为真，就一直迭代执行目标语句。随着条件真值的不同，循环可能执行多次，也可能一次也不执行。

第 6 章

函数

内容

6.1 函数基础.....	182
6.2 参数传递.....	187
6.3 返回类型和 return 语句.....	199
6.4 函数重载.....	206
6.5 特殊用途语言特性	211
6.6 函数匹配.....	217
6.7 函数指针.....	221
小结	225
术语表.....	225

本章首先介绍函数的定义和声明，包括参数如何传入函数以及函数如何返回结果。在 C++ 语言中允许重载函数，也就是几个不同的函数可以使用同一个名字。所以接下来我们介绍重载函数的方法，以及编译器如何从函数的若干重载形式中选取一个与调用匹配的版本。最后，我们将介绍一些关于函数指针的知识。

202 函数是一个命名了的代码块，我们通过调用函数执行相应的代码。函数可以有 0 个或多个参数，而且（通常）会产生一个结果。可以重载函数，也就是说，同一个名字可以对应几个不同的函数。



6.1 函数基础

一个典型的函数（function）定义包括以下部分：返回类型（return type）、函数名字、由 0 个或多个形参（parameter）组成的列表以及函数体。其中，形参以逗号隔开，形参的列表位于一对圆括号之内。函数执行的操作在语句块（参见 5.1 节，第 155 页）中说明，该语句块称为函数体（function body）。

我们通过调用运算符（call operator）来执行函数。调用运算符的形式是一对圆括号，它作用于一个表达式，该表达式是函数或者指向函数的指针；圆括号之内是一个用逗号隔开的实参（argument）列表，我们用实参初始化函数的形参。调用表达式的类型就是函数的返回类型。

编写函数

举个例子，我们准备编写一个求数的阶乘的程序。 n 的阶乘是从 1 到 n 所有数字的乘积，例如 5 的阶乘是 120。

```
1 * 2 * 3 * 4 * 5 = 120
```

程序如下所示：

```
// val 的阶乘是 val*(val - 1)*(val - 2) ...*((val - (val - 1))* 1)
int fact(int val)
{
    int ret = 1;           // 局部变量，用于保存计算结果
    while (val > 1)
        ret *= val--;
    return ret;           // 返回结果
}
```

函数的名字是 `fact`，它作用于一个整型参数，返回一个整型值。在 `while` 循环内部，在每次迭代时用后置递减运算符（参见 4.5 节，第 131 页）将 `val` 的值减 1。`return` 语句负责结束 `fact` 并返回 `ret` 的值。

调用函数

要调用 `fact` 函数，必须提供一个整数值，调用得到的结果也是一个整数：

```
int main()
{
    int j = fact(5);      // j 等于 120，即 fact(5) 的结果
    cout << "5! is " << j << endl;
    return 0;
}
```

203 函数的调用完成两项工作：一是用实参初始化函数对应的形参，二是将控制权转移给被调用函数。此时，主调函数（calling function）的执行被暂时中断，被调函数（called function）开始执行。

执行函数的第一步是（隐式地）定义并初始化它的形参。因此，当调用 fact 函数时，首先创建一个名为 val 的 int 变量，然后将它初始化为调用时所用的实参 5。

当遇到一条 return 语句时函数结束执行过程。和函数调用一样，return 语句也完成两项工作：一是返回 return 语句中的值（如果有的话），二是将控制权从被调函数转移回主调函数。函数的返回值用于初始化调用表达式的结果，之后继续完成调用所在的表达式的剩余部分。因此，我们对 fact 函数的调用等价于如下形式：

```
int val = 5;           // 用字面值 5 初始化 val
int ret = 1;           // fact 函数体内的代码
while (val > 1)
    ret *= val--;
int j = ret;           // 用 ret 的副本初始化 j
```

形参和实参

实参是形参的初始值。第一个实参初始化第一个形参，第二个实参初始化第二个形参，以此类推。尽管实参与形参存在对应关系，但是并没有规定实参的求值顺序（参见 4.1.3 节，第 123 页）。编译器能以任意可行的顺序对实参数求值。

实参的类型必须与对应的形参类型匹配，这一点与之前的规则是一致的，我们知道在初始化过程中初始值的类型也必须与初始化对象的类型匹配。函数有几个形参，我们就必须提供相同数量的实参。因为函数的调用规定实参数量应与形参数量一致，所以形参一定会被初始化。

在上面的例子中，fact 函数只有一个 int 类型的形参，所以每次我们调用它的时候，都必须提供一个能转换（参见 4.11 节，第 141 页）成 int 的实参：

```
fact("hello");          // 错误：实参类型不正确
fact();                 // 错误：实参数量不足
fact(42, 10, 0);        // 错误：实参数量过多
fact(3.14);             // 正确：该实参能转换成 int 类型
```

因为不能将 const char* 转换成 int，所以第一个调用失败。第二个和第三个调用也会失败，不过错误的原因与第一个不同，它们是因为传入的实参数量不对。要想调用 fact 函数只能使用一个实参，只要实参数量不是一个，调用都将失败。最后一个调用是合法的，因为 double 可以转换成 int。执行调用时，实参隐式地转换成 int 类型（截去小数部分），调用等价于

```
fact(3);
```

函数的形参列表

204

函数的形参列表可以为空，但是不能省略。要想定义一个不带形参的函数，最常用的办法是书写一个空的形参列表。不过为了与 C 语言兼容，也可以使用关键字 void 表示函数没有形参：

```
void f1() { /* ... */ }           // 隐式地定义空形参列表
void f2(void) { /* ... */ }         // 显式地定义空形参列表
```

形参列表中的形参通常用逗号隔开，其中每个形参都是含有一个声明符的声明。即使两个形参的类型一样，也必须把两个类型都写出来：

```
int f3(int v1, v2) { /* ... */ }      // 错误
int f4(int v1, int v2) { /* ... */ }    // 正确
```

任意两个形参都不能同名，而且函数最外层作用域中的局部变量也不能使用与函数形参一样的名字。

形参名是可选的，但是由于我们无法使用未命名的形参，所以形参一般都应该有个名字。偶尔，函数确实有个别形参不会被用到，则此类形参通常不命名以表示在函数体内不会使用它。不管怎样，是否设置未命名的形参并不影响调用时提供的实参数量。即使某个形参不被函数使用，也必须为它提供一个实参。

函数返回类型

大多数类型都能用作函数的返回类型。一种特殊的返回类型是 `void`，它表示函数不返回任何值。函数的返回类型不能是数组（参见 3.5 节，第 101 页）类型或函数类型，但可以是指向数组或函数的指针。我们将在 6.3.3 节（第 205 页）介绍如何定义一种特殊的函数，它的返回值是数组的指针（或引用），在 6.7 节（第 221 页）将介绍如何返回指向函数的指针。

6.1 节练习

练习 6.1： 实参和形参的区别是什么？

练习 6.2： 请指出下列函数哪个有错误，为什么？应该如何修改这些错误呢？

- (a) `int f() {
 string s;
 //...
 return s;
}`
- (b) `f2(int i) { /* ... */ }`
- (c) `int calc(int v1, int v1) /* ... */`
- (d) `double square(double x) return x * x;`

练习 6.3： 编写你自己的 `fact` 函数，上机检查是否正确。

练习 6.4： 编写一个与用户交互的函数，要求用户输入一个数字，计算生成该数字的阶乘。在 `main` 函数中调用该函数。

练习 6.5： 编写一个函数输出其实参的绝对值。



6.1.1 局部对象

在 C++ 语言中，名字有作用域（参见 2.2.4 节，第 43 页），对象有生命周期（lifetime）。理解这两个概念非常重要。

- 名字的作用域是程序文本的一部分，名字在其中可见。
- 对象的生命周期是程序执行过程中该对象存在的一段时间。

如我们所知，函数体是一个语句块。块构成一个新的作用域，我们可以在其中定义变量。形参和函数体内部定义的变量统称为 **局部变量**（local variable）。它们对函数而言是“局部”的，仅在函数的作用域内可见，同时局部变量还会隐藏（hide）在外层作用域中同名的其他所有声明中。

在所有函数体之外定义的对象存在于程序的整个执行过程中。此类对象在程序启动时被创建，直到程序结束才会销毁。局部变量的生命周期依赖于定义的方式。

自动对象

对于普通局部变量对应的对象来说，当函数的控制路径经过变量定义语句时创建该对象，当到达定义所在的块末尾时销毁它。我们把只存在于块执行期间的对象称为**自动对象**（automatic object）。当块的执行结束后，块中创建的自动对象的值就变成未定义的了。

形参是一种自动对象。函数开始时为形参申请存储空间，因为形参定义在函数体作用域之内，所以一旦函数终止，形参也就被销毁。

我们用传递给函数的实参初始化形参对应的自动对象。对于局部变量对应的自动对象来说，则分为两种情况：如果变量定义本身含有初始值，就用这个初始值进行初始化；否则，如果变量定义本身不含初始值，执行默认初始化（参见 2.2.1 节，第 40 页）。这意味着内置类型的未初始化局部变量将产生未定义的值。

局部静态对象

某些时候，有必要令局部变量的生命周期贯穿函数调用及之后的时间。可以将局部变量定义成 static 类型从而获得这样的对象。**局部静态对象**（local static object）在程序的执行路径第一次经过对象定义语句时初始化，并且直到程序终止才被销毁，在此期间即使对象所在的函数结束执行也不会对它有影响。

<206

举个例子，下面的函数统计它自己被调用了多少次，这样的函数也许没什么实际意义，但是足够说明问题：

```
size_t count_calls()
{
    static size_t ctr = 0; // 调用结束后，这个值仍然有效
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

这段程序将输出从 1 到 10（包括 10 在内）的数字。

在控制流第一次经过 ctr 的定义之前，ctr 被创建并初始化为 0。每次调用将 ctr 加 1 并返回新值。每次执行 count_calls 函数时，变量 ctr 的值都已经存在并且等于函数上一次退出时 ctr 的值。因此，第二次调用时 ctr 的值是 1，第三次调用时 ctr 的值是 2，以此类推。

如果局部静态变量没有显式的初始值，它将执行值初始化（参见 3.3.1 节，第 88 页），内置类型的局部静态变量初始化为 0。

6.1.1 节练习

练习 6.6：说明形参、局部变量以及局部静态变量的区别。编写一个函数，同时用到这三种形式。

练习 6.7：编写一个函数，当它第一次被调用时返回 0，以后每次被调用返回值加 1。



6.1.2 函数声明

和其他名字一样，函数的名字也必须在使用之前声明。类似于变量（参见 2.2.2 节，第 41 页），函数只能定义一次，但可以声明多次。唯一的例外是如 15.3 节（第 535 页）将要介绍的，如果一个函数永远也不会被我们用到，那么它可以只有声明没有定义。

函数的声明和函数的定义非常类似，唯一的区别是函数声明无须函数体，用一个分号替代即可。

因为函数的声明不包含函数体，所以也就无须形参的名字。事实上，在函数的声明中经常省略形参的名字。尽管如此，写上形参的名字还是有用处的，它可以帮助使用者更好地理解函数的功能：

207 // 我们选择 beg 和 end 作为形参的名字以表示这两个迭代器划定了输出值的范围
void print(vector<int>::const_iterator beg,
 vector<int>::const_iterator end);

函数的三要素（返回类型、函数名、形参类型）描述了函数的接口，说明了调用该函数所需的全部信息。函数声明也称作 **函数原型**（function prototype）。

在头文件中进行函数声明

回忆之前所学的知识，我们建议变量在头文件（参见 2.6.3 节，第 68 页）中声明，在源文件中定义。与之类似，函数也应该在头文件中声明而在源文件中定义。

看起来把函数的声明直接放在使用该函数的源文件中是合法的，也比较容易被人接受；但是这么做可能会很烦琐而且容易出错。相反，如果把函数声明放在头文件中，就能确保同一函数的所有声明保持一致。而且一旦我们想改变函数的接口，只需改变一条声明即可。

定义函数的源文件应该把含有函数声明的头文件包含进来，编译器负责验证函数的定义和声明是否匹配。



含有函数声明的头文件应该被包含到定义函数的源文件中。

6.1.2 节练习

练习 6.8：编写一个名为 Chapter6.h 的头文件，令其包含 6.1 节练习（第 184 页）中的函数声明。



6.1.3 分离式编译

随着程序越来越复杂，我们希望把程序的各个部分分别存储在不同文件中。例如，可以把 6.1 节练习（第 184 页）的函数存在一个文件里，把使用这些函数的代码存在其他源文件中。为了允许编写程序时按照逻辑关系将其划分开来，C++语言支持所谓的分离式编译（separate compilation）。分离式编译允许我们把程序分割到几个文件中去，每个文件独立编译。

编译和链接多个源文件

举个例子，假设 fact 函数的定义位于一个名为 fact.cc 的文件中，它的声明位于

名为 Chapter6.h 的头文件中。显然与其他所有用到 fact 函数的文件一样, fact.cc 应该包含 Chapter6.h 头文件。另外, 我们在名为 factMain.cc 的文件中创建 main 函数, main 函数将调用 fact 函数。要生成可执行文件 (executable file), 必须告诉编译器我们用到的代码在哪里。对于上述几个文件来说, 编译的过程如下所示:

```
$ CC factMain.cc fact.cc # generates factMain.exe or a.out  
$ CC factMain.cc fact.cc -o main # generates main or main.exe
```

其中, CC 是编译器的名字, \$ 是系统提示符, # 后面是命令行下的注释语句。接下来运行可执行文件, 就会执行我们定义的 main 函数。

如果我们修改了其中一个源文件, 那么只需重新编译那个改动了的文件。大多数编译器提供了分离式编译每个文件的机制, 这一过程通常会产生一个后缀名是 .obj (Windows) 或 .o (UNIX) 的文件, 后缀名的含义是该文件包含对象代码 (object code)。

接下来编译器负责把对象文件链接在一起形成可执行文件。在我们的系统中, 编译的过程如下所示:

```
$ CC -c factMain.cc # generates factMain.o  
$ CC -c fact.cc # generates fact.o  
$ CC factMain.o fact.o # generates factMain.exe or a.out  
$ CC factMain.o fact.o -o main # generates main or main.exe
```

你可以仔细阅读编译器的用户手册, 弄清楚由多个文件组成的程序是如何编译并执行的。

6.1.3 节练习

练习 6.9: 编写你自己的 fact.cc 和 factMain.cc, 这两个文件都应该包含上一小节的练习中编写的 Chapter6.h 头文件。通过这些文件, 理解你的编译器是如何支持分离式编译的。

6.2 参数传递



如前所述, 每次调用函数时都会重新创建它的形参, 并用传入的实参对形参进行初始化。

Note

形参初始化的机理与变量初始化一样。

和其他变量一样, 形参的类型决定了形参和实参交互的方式。如果形参是引用类型 (参见 2.3.1 节, 第 45 页), 它将绑定到对应的实参上; 否则, 将实参的值拷贝后赋给形参。

当形参是引用类型时, 我们说它对应的实参被引用传递 (passed by reference) 或者函数被传引用调用 (called by reference)。和其他引用一样, 引用形参也是它绑定的对象的别名; 也就是说, 引用形参是它对应的实参的别名。

当实参的值被拷贝给形参时, 形参和实参是两个相互独立的对象。我们说这样的实参被值传递 (passed by value) 或者函数被传值调用 (called by value)。

209

6.2.1 传值参数



当初始化一个非引用类型的变量时, 初始值被拷贝给变量。此时, 对变量的改动不会

影响初始值：

```
int n = 0;           // int 类型的初始变量
int i = n;           // i 是 n 的副本
i = 42;             // i 的值改变；n 的值不变
```

传值参数的机理完全一样，函数对形参做的所有操作都不会影响实参。例如，在 fact 函数（参见 6.1 节，第 182 页）内对变量 val 执行递减操作：

```
ret *= val--;      // 将 val 的值减 1
```

尽管 fact 函数改变了 val 的值，但是这个改动不会影响传入 fact 的实参。调用 fact(i) 不会改变 i 的值。

指针形参

指针的行为和其他非引用类型一样。当执行指针拷贝操作时，拷贝的是指针的值。拷贝之后，两个指针是不同的指针。因为指针使我们可以间接地访问它所指的对象，所以通过指针可以修改它所指对象的值：

```
int n = 0, i = 42;
int *p = &n, *q = &i;    // p 指向 n; q 指向 i
*p = 42;                // n 的值改变；p 不变
p = q;                  // p 现在指向了 i；但是 i 和 n 的值都不变
```

指针形参的行为与之类似：

```
// 该函数接受一个指针，然后将指针所指的值置为 0
void reset(int *ip)
{
    *ip = 0;        // 改变指针 ip 所指对象的值
    ip = 0;         // 只改变了 ip 的局部拷贝，实参未被改变
}
```

调用 reset 函数之后，实参所指的对象被置为 0，但是实参本身并没有改变：

```
int i = 42;
reset(&i);          // 改变 i 的值而非 i 的地址
cout << "i = " << i << endl;    // 输出 i = 0
```

210

Best Practices

熟悉 C 的程序员常常使用指针类型的形参访问函数外部的对象。在 C++ 语言中，建议使用引用类型的形参替代指针。

6.2.1 节练习

练习 6.10：编写一个函数，使用指针形参交换两个整数的值。在代码中调用该函数并输出交换后的结果，以此验证函数的正确性。



6.2.2 传引用参数

回忆过去所学的知识，我们知道对于引用的操作实际上是作用在引用所引的对象上（参见 2.3.1 节，第 45 页）：

```
int n = 0, i = 42;
int &r = n;      // r 绑定了 n（即 r 是 n 的另一个名字）
```

```
r = 42;           // 现在 n 的值是 42
r = i;           // 现在 n 的值和 i 相同
i = r;           // i 的值和 n 相同
```

引用形参的行为与之类似。通过使用引用形参，允许函数改变一个或多个实参的值。

举个例子，我们可以改写上一小节的 `reset` 程序，使其接受的参数是引用类型而非指针：

```
// 该函数接受一个 int 对象的引用，然后将对象的值置为 0
void reset(int &i) // i 是传给 reset 函数的对象的另一个名字
{
    i = 0;          // 改变了 i 所引对象的值
}
```

和其他引用一样，引用形参绑定初始化它的对象。当调用这一版本的 `reset` 函数时，`i` 绑定我们传给函数的 `int` 对象，此时改变 `i` 也就是改变 `i` 所引对象的值。此例中，被改变的对象是传入 `reset` 的实参。

调用这一版本的 `reset` 函数时，我们直接传入对象而无须传递对象的地址：

```
int j = 42;
reset(j);           // j 采用传引用方式，它的值被改变
cout << "j = " << j << endl;      // 输出 j = 0
```

在上述调用过程中，形参 `i` 仅仅是 `j` 的又一个名字。在 `reset` 内部对 `i` 的使用即是对 `j` 的使用。

使用引用避免拷贝

<211

拷贝大的类类型对象或者容器对象比较低效，甚至有的类类型（包括 IO 类型在内）根本就不支持拷贝操作。当某种类型不支持拷贝操作时，函数只能通过引用形参访问该类型的对象。

举个例子，我们准备编写一个函数比较两个 `string` 对象的长度。因为 `string` 对象可能会非常长，所以应该尽量避免直接拷贝它们，这时使用引用形参是比较明智的选择。又因为比较长度无须改变 `string` 对象的内容，所以把形参定义成对常量的引用（参见 2.4.1 节，第 54 页）：

```
// 比较两个 string 对象的长度
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

如 6.2.3 节（第 191 页）将要介绍的，当函数无须修改引用形参的值时最好使用常量引用。



如果函数无须改变引用形参的值，最好将其声明为常量引用。

使用引用形参返回额外信息

一个函数只能返回一个值，然而有时函数需要同时返回多个值，引用形参为我们一次返回多个结果提供了有效的途径。举个例子，我们定义一个名为 `find_char` 的函数，它返回在 `string` 对象中某个指定字符第一次出现的位置。同时，我们也希望函数能返回该

字符出现的总次数。

该如何定义函数使得它能够既返回位置也返回出现次数呢？一种方法是定义一个新的数据类型，让它包含位置和数量两个成员。还有另一种更简单的方法，我们可以给函数传入一个额外的引用实参，令其保存字符出现的次数：

```
// 返回 s 中 c 第一次出现的位置索引
// 引用形参 occurs 负责统计 c 出现的总次数
string::size_type find_char(const string &s, char c,
                             string::size_type &occurs)
{
    auto ret = s.size();           // 第一次出现的位置（如果有的话）
    occurs = 0;                  // 设置表示出现次数的形参的值
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;          // 记录 c 第一次出现的位置
            ++occurs;           // 将出现的次数加 1
        }
    }
    return ret;                  // 出现次数通过 occurs 隐式地返回
}
```

212> 当我们调用 `find_char` 函数时，必须传入三个实参：作为查找范围的一个 `string` 对象、要找的字符以及一个用于保存字符出现次数的 `size_type`（参见 3.2.2 节，第 79 页）对象。假设 `s` 是一个 `string` 对象，`ctr` 是一个 `size_type` 对象，则我们通过如下形式调用 `find_char` 函数：

```
auto index = find_char(s, 'o', ctr);
```

调用完成后，如果 `string` 对象中确实存在 `o`，那么 `ctr` 的值就是 `o` 出现的次数，`index` 指向 `o` 第一次出现的位置；否则如果 `string` 对象中没有 `o`，`index` 等于 `s.size()` 而 `ctr` 等于 0。

6.2.2 节练习

练习 6.11： 编写并验证你自己的 `reset` 函数，使其作用于引用类型的参数。

练习 6.12： 改写 6.2.1 节中练习 6.10（第 188 页）的程序，使用引用而非指针交换两个整数的值。你觉得哪种方法更易于使用呢？为什么？

练习 6.13： 假设 `T` 是某种类型的名字，说明以下两个函数声明的区别：一个是 `void f(T)`，另一个是 `void f(&T)`。

练习 6.14： 举一个形参应该是引用类型的例子，再举一个形参不能是引用类型的例子。

练习 6.15： 说明 `find_char` 函数中的三个形参为什么是现在的类型，特别说明为什么 `s` 是常量引用而 `occurs` 是普通引用？为什么 `s` 和 `occurs` 是引用类型而 `c` 不是？如果令 `s` 是普通引用会发生什么情况？如果令 `occurs` 是常量引用会发生什么情况？



6.2.3 const 形参和实参

当形参是 `const` 时，必须要注意 2.4.3 节（第 57 页）关于顶层 `const` 的讨论。如前

所述，顶层 const 作用于对象本身：

```
const int ci = 42; // 不能改变 ci, const 是顶层的
int i = ci; // 正确：当拷贝 ci 时，忽略了它的顶层 const
int * const p = &i; // const 是顶层的，不能给 p 赋值
*p = 0; // 正确：通过 p 改变对象的内容是允许的，现在 i 变成了 0
```

和其他初始化过程一样，当用实参初始化形参时会忽略掉顶层 const。换句话说，形参的顶层 const 被忽略掉了。当形参有顶层 const 时，传给它常量对象或者非常量对象都是可以的：

```
void fcn(const int i) { /* fcn 能够读取 i，但是不能向 i 写值 */ }
```

调用 fcn 函数时，既可以传入 const int 也可以传入 int。忽略掉形参的顶层 const 可能产生意想不到的结果：

```
void fcn(const int i) { /* fcn 能够读取 i，但是不能向 i 写值 */ } ◀ 213
void fcn(int i) { /* ... */ } // 错误：重复定义了 fcn(int)
```

在 C++ 语言中，允许我们定义若干具有相同名字的函数，不过前提是不同函数的形参列表应该有明显的区别。因为顶层 const 被忽略掉了，所以在上面的代码中传入两个 fcn 函数的参数可以完全一样。因此第二个 fcn 是错误的，尽管形式上有差异，但实际上它的形参和第一个 fcn 的形参没什么不同。

指针或引用形参与 const

形参的初始化方式和变量的初始化方式是一样的，所以回顾通用的初始化规则有助于理解本节知识。我们可以使用非常量初始化一个底层 const 对象，但是反过来不行；同时一个普通的引用必须用同类型的对象初始化。

```
int i = 42;
const int *cp = &i; // 正确：但是 cp 不能改变 i (参见 2.4.2 节, 第 56 页)
const int &r = i; // 正确：但是 r 不能改变 i (参见 2.4.1 节, 第 55 页)
const int &r2 = 42; // 正确：(参见 2.4.1 节, 第 55 页)
int *p = cp; // 错误：p 的类型和 cp 的类型不匹配 (参见 2.4.2 节, 第 56 页)
int &r3 = r; // 错误：r3 的类型和 r 的类型不匹配 (参见 2.4.1 节, 第 55 页)
int &r4 = 42; // 错误：不能用字面值初始化一个非常量引用 (参见 2.3.1 节, 第 45 页)
```

将同样的初始化规则应用到参数传递上可得如下形式：

```
int i = 0;
const int ci = i;
string::size_type ctr = 0;
reset(&i); // 调用形参类型是 int* 的 reset 函数
reset(&ci); // 错误：不能用指向 const int 对象的指针初始化 int*
reset(i); // 调用形参类型是 int& 的 reset 函数
reset(ci); // 错误：不能把普通引用绑定到 const 对象 ci 上
reset(42); // 错误：不能把普通应用绑定到字面值上
reset(ctr); // 错误：类型不匹配，ctr 是无符号类型
// 正确：find_char 的第一个形参是对常量的引用
find_char("Hello World!", 'o', ctr);
```

要想调用引用版本的 reset（参见 6.2.2 节，第 189 页），只能使用 int 类型的对象，而不能使用字面值、求值结果为 int 的表达式、需要转换的对象或者 const int 类型的对象。类似的，要想调用指针版本的 reset（参见 6.2.1 节，第 188 页）只能使用 int*。