3.5.5 与旧代码的接口

很多C++程序在标准库出现之前就已经写成了，它们肯定没用到string和vector类型。而且，有一些C++程序实际上是与C语言或其他语言的接口程序，当然也无法使用C++标准库。因此，现代的C++程序不得不与那些充满了数组和/或C风格字符串的代码衔接，为了使这一工作简单易行，C++专门提供了一组功能。

混用string对象和C风格字符串

3.2.1节（第76页）介绍过允许使用字符串字面值来初始化string对象：

```
string s("Hello World"); // s的内容是Hello World
```

更一般的情况是，任何出现字符串字面值的地方都可以用以空字符结束的字符数组来替代：

- 允许使用以空字符结束的字符数组来初始化string对象或为string对象赋值。
- 在string对象的加法运算中允许使用以空字符结束的字符数组作为其中一个运算对象（不能两个运算对象都是）；在string对象的复合赋值运算中允许使用以空字符结束的字符数组作为右侧的运算对象。

上述性质反过来就不成立了：如果程序的某处需要一个C风格字符串，无法直接用string对象来代替它。例如，不能用string对象直接初始化指向字符的指针。为了完成该功能，string专门提供了一个名为c_str的成员函数：

```
char *str = s; // 错误：不能用string对象初始化char*
const char *str = s.c_str(); // 正确
```

顾名思义，c_str函数的返回值是一个C风格的字符串。也就是说，函数的返回结果是一个指针，该指针指向一个以空字符结束的字符数组，而这个数组所存的数据恰好与那个string对象的一样。结果指针的类型是const char*，从而确保我们不会改变字符数组的内容。 125

我们无法保证c_str函数返回的数组一直有效，事实上，如果后续的操作改变了s的值就可能让之前返回的数组失去效用。

如果执行完c_str()函数后程序想一直都能使用其返回的数组，最好将该数组重新拷贝一份。

使用数组初始化vector对象

3.5.1节（第102页）介绍过不允许使用一个数组为另一个内置类型的数组赋初值，也不允许使用vector对象初始化数组。相反的，允许使用数组来初始化vector对象。要实现这一目的，只需指明要拷贝区域的首元素地址和尾后地址就可以了：

```
int int_arr[] = {0, 1, 2, 3, 4, 5};
// ivec有6个元素，分别是int_arr中对应元素的副本
vector<int> ivec(begin(int_arr), end(int_arr));
```

在上述代码中，用于创建ivec的两个指针实际上指明了用来初始化的值在数组int_arr中的位置，其中第二个指针应指向待拷贝区域尾元素的下一位置。此例中，使用标准库函数begin和end（参见3.5.3节，第106页）来分别计算int_arr的首指针和尾后指针。在最终的结果中，ivec将包含6个元素，它们的次序和值都与数组int_arr完全

一样。

用于初始化 vector 对象的值也可能仅是数组的一部分：

```
// 拷贝三个元素：int_arr[1]、int_arr[2]、int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
```

这条初始化语句用 3 个元素创建了对象 subVec，3 个元素的值分别来自 int_arr[1]、int_arr[2]和 int_arr[3]。

建议：尽量使用标准库类型而非数组

使用指针和数组很容易出错。一部分原因是概念上的问题：指针常用于底层操作，因此容易引发一些与烦琐细节有关的错误。其他问题则源于语法错误，特别是声明指针时的语法错误。

现代的 C++程序应当尽量使用 vector 和迭代器，避免使用内置数组和指针；应该尽量使用 string，避免使用 C 风格的基于数组的字符串。

3.5.5 节练习

练习 3.41：编写一段程序，用整型数组初始化一个 vector 对象。

练习 3.42：编写一段程序，将含有整数元素的 vector 对象拷贝给一个整型数组。

3.6 多维数组

严格来说，C++语言中没有多维数组，通常所说的多维数组其实是数组的数组。谨记
126 这一点，对今后理解和使用多维数组大有益处。

当一个数组的元素仍然是数组时，通常使用两个维度来定义它：一个维度表示数组本身大小，另外一个维度表示其元素（也是数组）大小：

```
int ia[3][4]; // 大小为 3 的数组，每个元素是含有 4 个整数的数组
// 大小为 10 的数组，它的每个元素都是大小为 20 的数组，
// 这些数组的元素是含有 30 个整数的数组
int arr[10][20][30] = {0}; // 将所有元素初始化为 0
```

如 3.5.1 节（第 103 页）所介绍的，按照由内而外的顺序阅读此类定义有助于更好地理解其真实含义。在第一条语句中，我们定义的名字是 ia，显然 ia 是一个含有 3 个元素的数组。接着观察右边发现，ia 的元素也有自己的维度，所以 ia 的元素本身又都是含有 4 个元素的数组。再观察左边知道，真正存储的元素是整数。因此最后可以明确第一条语句的含义：它定义了一个大小为 3 的数组，该数组的每个元素都是含有 4 个整数的数组。

使用同样的方式理解 arr 的定义。首先 arr 是一个大小为 10 的数组，它的每个元素都是大小为 20 的数组，这些数组的元素又都是含有 30 个整数的数组。实际上，定义数组时对下标运算符的数量并没有限制，因此只要愿意就可以定义这样一个数组：它的元素还是数组，下一级数组的元素还是数组，再下一级数组的元素还是数组，以此类推。

对于二维数组来说，常把第一个维度称作行，第二个维度称作列。

多维数组的初始化

允许使用花括号括起来的一组值初始化多维数组，这点和普通的数组一样。下面的初始化形式中，多维数组的每一行分别用花括号括了起来：

```
int ia[3][4] = {          // 三个元素，每个元素都是大小为 4 的数组
    {0, 1, 2, 3},         // 第 1 行的初始值
    {4, 5, 6, 7},         // 第 2 行的初始值
    {8, 9, 10, 11}        // 第 3 行的初始值
};
```

其中内层嵌套着的花括号并非必需的，例如下面的初始化语句，形式上更为简洁，完成的功能和上面这段代码完全一样：

```
// 没有标识每行的花括号，与之前的初始化语句是等价的
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

类似于一维数组，在初始化多维数组时也并非所有元素的值都必须包含在初始化列表之内。如果仅仅想初始化每一行的第一个元素，通过如下的语句即可：

```
// 显式地初始化每行的首元素
int ia[3][4] = {{ 0 }, { 4 }, { 8 }};
```

127

其他未列出的元素执行默认值初始化，这个过程和一维数组（参见 3.5.1 节，第 102 页）一样。在这种情况下如果再省略掉内层的花括号，结果就大不一样了。下面的代码

```
// 显式地初始化第 1 行，其他元素执行值初始化
int ix[3][4] = {0, 3, 6, 9};
```

含义发生了变化，它初始化的是第一行的 4 个元素，其他元素被初始化为 0。

多维数组的下标引用

可以使用下标运算符来访问多维数组的元素，此时数组的每个维度对应一个下标运算符。

如果表达式含有的下标运算符数量和数组的维度一样多，该表达式的结果将是给定类型的元素；反之，如果表达式含有的下标运算符数量比数组的维度小，则表达式的结果将是给定索引处的一个内层数组：

```
// 用 arr 的首元素为 ia 最后一行的最后一个元素赋值
ia[2][3] = arr[0][0][0];
int (&row)[4] = ia[1]; // 把 row 绑定到 ia 的第二个 4 元素数组上
```

在第一个例子中，对于用到的两个数组来说，表达式提供的下标运算符数量都和它们各自的维度相同。在等号左侧，ia[2]得到数组 ia 的最后一行，此时返回的是表示 ia 最后一行的那个一维数组而非任何实际元素；对这个一维数组再取下标，得到编号为[3]的元素，也就是这一行的最后一个元素。

类似的，等号右侧的运算对象包含 3 个维度。首先通过索引 0 得到最外层的数组，它是一个大小为 20 的（多维）数组；接着获取这 20 个元素数组的第一个元素，得到一个大小为 30 的一维数组；最后再取出其中的第一个元素。

在第二个例子中，把 row 定义成一个含有 4 个整数的数组的引用，然后将其绑定到 ia 的第 2 行。

再举一个例子，程序中经常会用到两层嵌套的 for 循环来处理多维数组的元素：

```
constexpr size_t rowCnt = 3, colCnt = 4;
```

```
int ia[rowCnt][colCnt]; // 12 个未初始化的元素
// 对于每一行
for (size_t i = 0; i != rowCnt; ++i) {
    // 对于行内的每一列
    for (size_t j = 0; j != colCnt; ++j) {
        // 将元素的位置索引作为它的值
        ia[i][j] = i * colCnt + j;
    }
}
```

外层的 for 循环遍历 ia 的所有元素，注意这里的元素是一维数组；内层的 for 循环则遍历那些一维数组的整数元素。此例中，我们将元素的值设为该元素在整个数组中的序号。

使用范围 for 语句处理多维数组

128

由于在 C++11 新标准中新增了范围 for 语句，所以前一个程序可以简化为如下形式：

```
size_t cnt = 0;
for (auto &row : ia)              // 对于外层数组的每一个元素
    for (auto &col : row) {       // 对于内层数组的每一个元素
        col = cnt;                // 将下一个值赋给该元素
        ++cnt;                    // 将 cnt 加 1
}
```

这个循环赋给 ia 元素的值和之前那个循环是完全相同的，区别之处是通过使用范围 for 语句把管理数组索引的任务交给了系统来完成。因为要改变元素的值，所以得把控制变量 row 和 col 声明成引用类型（参见 3.2.3 节，第 83 页）。第一个 for 循环遍历 ia 的所有元素，这些元素是大小为 4 的数组，因此 row 的类型就应该是含有 4 个整数的数组的引用。第二个 for 循环遍历那些 4 元素数组中的某一个，因此 col 的类型是整数的引用。每次迭代把 cnt 的值赋给 ia 的当前元素，然后将 cnt 加 1。

在上面的例子中，因为要改变数组元素的值，所以我们选用引用类型作为循环控制变量，但其实还有一个深层次的原因促使我们这么做。举一个例子，考虑如下的循环：

```
for (const auto &row : ia)    // 对于外层数组的每一个元素
    for (auto col : row)      // 对于内层数组的每一个元素
        cout << col << endl;
```

这个循环中并没有任何写操作，可是我们还是将外层循环的控制变量声明成了引用类型，这是为了避免数组被自动转成指针（参见 3.5.3 节，第 105 页）。假设不用引用类型，则循环如下述形式：

```
for (auto row : ia)
    for (auto col : row)
```

程序将无法通过编译。这是因为，像之前一样第一个循环遍历 ia 的所有元素，注意这些元素实际上是大小为 4 的数组。因为 row 不是引用类型，所以编译器初始化 row 时会自动将这些数组形式的元素（和其他类型的数组一样）转换成指向该数组内首元素的指针。这样得到的 row 的类型就是 int*，显然内层的循环就不合法了，编译器将试图在一个 int*内遍历，这显然和程序的初衷相去甚远。

要使用范围 for 语句处理多维数组，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型。

指针和多维数组

当程序使用多维数组的名字时，也会自动将其转换成指向数组首元素的指针。

定义指向多维数组的指针时，千万别忘了这个多维数组实际上是数组的数组。 129

因为多维数组实际上是数组的数组，所以由多维数组名转换得来的指针实际上是指向第一个内层数组的指针：

```
int ia[3][4];          // 大小为 3 的数组，每个元素是含有 4 个整数的数组
int (*p)[4] = ia;      // p 指向含有 4 个整数的数组
p = &ia[2];            // p 指向 ia 的尾元素
```

根据 3.5.1 节（第 103 页）提出的策略，我们首先明确 (*p) 意味着 p 是一个指针。接着观察右边发现，指针 p 所指的是一个维度为 4 的数组；再观察左边知道，数组中的元素是整数。因此，p 就是指向含有 4 个整数的数组的指针。

在上述声明中，圆括号必不可少：

```
int *ip[4];        // 整型指针的数组
int (*ip)[4];      // 指向含有 4 个整数的数组
```

随着 C++11 新标准的提出，通过使用 auto 或者 decltype（参见 2.5.2 节，第 61 页）就能尽可能地避免在数组前面加上一个指针类型了：

C++ 11

```
// 输出 ia 中每个元素的值，每个内层数组各占一行
// p 指向含有 4 个整数的数组
for (auto p = ia; p != ia + 3; ++p) {
    // q 指向 4 个整数数组的首元素，也就是说，q 指向一个整数
    for (auto q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
```

外层的 for 循环首先声明一个指针 p 并令其指向 ia 的第一个内层数组，然后依次迭代直到 ia 的全部 3 行都处理完为止。其中递增运算++p 负责将指针 p 移动到 ia 的下一行。

内层的 for 循环负责输出内层数组所包含的值。它首先令指针 q 指向 p 当前所在行的第一个元素。*p 是一个含有 4 个整数的数组，像往常一样，数组名被自动地转换成指向该数组首元素的指针。内层 for 循环不断迭代直到我们处理完了当前内层数组的所有元素为止。为了获取内层 for 循环的终止条件，再一次解引用 p 得到指向内层数组首元素的指针，给它加上 4 就得到了终止条件。

当然，使用标准库函数 begin 和 end（参见 3.5.3 节，第 106 页）也能实现同样的功能，而且看起来更简洁一些：

```
// p 指向 ia 的第一个数组
for (auto p = begin(ia); p != end(ia); ++p) {
    // q 指向内层数组的首元素
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' '; // 输出 q 所指的整数值
    cout << endl;
}
```

130 在这一版本的程序中，循环终止条件由 end 函数负责判断。虽然我们也能推断出 p 的类型是指向含有 4 个整数的数组的指针，q 的类型是指向整数的指针，但是使用 auto 关键字我们就不必再烦心这些类型到底是什么了。

类型别名简化多维数组的指针

读、写和理解一个指向多维数组的指针是一个让人不胜其烦的工作，使用类型别名（参见 2.5.1 节，第 60 页）能让这项工作变得简单一点儿，例如：

```
using int_array = int[4]; // 新标准下类型别名的声明，参见 2.5.1 节（第 60 页）
typedef int int_array[4]; // 等价的 typedef 声明，参见 2.5.1 节（第 60 页）

// 输出 ia 中每个元素的值，每个内层数组各占一行
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
```

程序将类型“4 个整数组成的数组”命名为 int_array，用类型名 int_array 定义外层循环的控制变量让程序显得简洁明了。

3.6 节练习

练习 3.43：编写 3 个不同版本的程序，令其均能输出 ia 的元素。版本 1 使用范围 for 语句管理迭代过程；版本 2 和版本 3 都使用普通的 for 语句，其中版本 2 要求用下标运算符，版本 3 要求用指针。此外，在所有 3 个版本的程序中都要直接写出数据类型，而不能使用类型别名、auto 关键字或 decltype 关键字。

练习 3.44：改写上一个练习中的程序，使用类型别名来代替循环控制变量的类型。

练习 3.45：再一次改写程序，这次使用 auto 关键字。

小结

131

string 和 vector 是两种最重要的标准库类型。string 对象是一个可变长的字符序列，vector 对象是一组同类型对象的容器。

迭代器允许对容器中的对象进行间接访问，对于 string 对象和 vector 对象来说，可以通过迭代器访问元素或者在元素间移动。

数组和指向数组元素的指针在一个较低的层次上实现了与标准库类型 string 和 vector 类似的功能。一般来说，应该优先选用标准库提供的类型，之后再考虑 C++语言内置的低层的替代品数组或指针。

术语表

begin 是 string 和 vector 的成员，返回指向第一个元素的迭代器。也是一个标准库函数，输入一个数组，返回指向该数组首元素的指针。

缓冲区溢出（buffer overflow） 一种严重的程序故障，主要的原因是试图通过一个越界的索引访问容器内容，容器类型包括 string、vector 和数组等。

C 风格字符串（C-style string） 以空字符结束的字符数组。字符串字面值是 C 风格字符串，C 风格字符串容易出错。

类模板（class template） 用于创建具体类类型的模板。要想使用类模板，必须提供关于类型的辅助信息。例如，要定义一个 vector 对象需要指定元素的类型：vector<int>包含 int 类型的元素。

编译器扩展（compiler extension） 某个特定的编译器为 C++语言额外增加的特性。基于编译器扩展编写的程序不易移植到其他编译器上。

容器（container） 是一种类型，其对象容纳了一组给定类型的对象。vector 是一种容器类型。

拷贝初始化（copy initialization） 使用赋值号（=）的初始化形式。新创建的对象是初始值的一个副本。

difference_type 由 string 和 vector 定义的一种带符号整数类型，表示两个迭代器之间的距离。

直接初始化（direct initialization） 不使用赋值号（=）的初始化形式。

empty 是 string 和 vector 的成员，返回一个布尔值。当对象的大小为 0 时返回真，否则返回假。

end 是 string 和 vector 的成员，返回一个尾后迭代器。也是一个标准库函数，输入一个数组，返回指向该数组尾元素的下一位置的指针。

getline 在 string 头文件中定义的一个函数，以一个 istream 对象和一个 string 对象为输入参数。该函数首先读取输入流的内容直到遇到换行符停止，然后将读入的数据存入 string 对象，最后返回 istream 对象。其中换行符读入但是不保留。

索引（index） 是下标运算符使用的值。表示要在 string 对象、vector 对象或者数组中访问的一个位置。

实例化（instantiation） 编译器生成一个指定的模板类或函数的过程。

迭代器（iterator） 是一种类型，用于访问容器中的元素或者在元素之间移动。

迭代器运算（iterator arithmetic） 是 string 或 vector 的迭代器的运算：迭代器与整数相加或相减得到一个新的迭代器，与原来的迭代器相比，新迭代器向前

或向后移动了若干个位置。两个迭代器相减得到它们之间的距离，此时它们必须指向同一个容器的元素或该容器尾元素的下一位置。

132 **以空字符结束的字符串**（null-terminated string） 是一个字符串，它的最后一个字符后面还跟着一个空字符（'\0'）。

尾后迭代器（off-the-end iterator） end函数返回的迭代器，指向一个并不存在的元素，该元素位于容器尾元素的下一位置。

指针运算（pointer arithmetic） 是指针类型支持的算术运算。指向数组的指针所支持的运算种类与迭代器运算一样。

prtdiff_t 是cstddef头文件中定义的一种与机器实现有关的带符号整数类型，它的空间足够大，能够表示数组中任意两个指针之间的距离。

push_back 是 vector 的成员，向vector对象的末尾添加元素。

范围 for 语句（range for） 一种控制语句，可以在值的一个特定集合内迭代。

size 是 string 和 vector 的成员，分别返回字符的数量或元素的数量。返回值的类型是 size_type。

size_t 是 cstddef 头文件中定义的一种与机器实现有关的无符号整数类型，它的空间足够大，能够表示任意数组的大小。

size_type 是 string 和 vector 定义的类型的名字，能存放下任意 string 对象或 vector 对象的大小。在标准库中，size_type 被定义为无符号类型。

string 是一种标准库类型，表示字符的序列。

using 声明（using declaration） 令命名空间中的某个名字可被程序直接使用。

```
using 命名空间 :: 名字;
```

上述语句的作用是令程序可以直接使用*名字*，而无须写它的前缀部分*命名空间*::。

值初始化（value initialization） 是一种初始化过程。内置类型初始化为0，类类型由类的默认构造函数初始化。只有当类包含默认构造函数时，该类的对象才会被值初始化。对于容器的初始化来说，如果只说明了容器的大小而没有指定初始值的话，就会执行值初始化。此时编译器会生成一个值，而容器的元素被初始化为该值。

vector 是一种标准库类型，容纳某指定类型的一组元素。

++运算符（++ operator）是迭代器和指针定义的递增运算符。执行“加1”操作使得迭代器指向下一个元素。

[ ]运算符（[ ] operator）下标运算符。obj[j]得到容器对象 obj 中位置 j 的那个元素。索引从 0 开始，第一个元素的索引是 0，尾元素的索引是 obj.size()-1。下标运算符的返回值是一个对象。如果 p 是指针、n 是整数，则 p[n] 与 *(p+n) 等价。

->运算符（-> operator）箭头运算符，该运算符综合了解引用操作和点操作。a->b 等价于 (*a).b。

<<运算符（<< operator）标准库类型 string 定义的输出运算符，负责输出 string 对象中的字符。

>>运算符（>> operator）标准库类型 string 定义的输入运算符，负责读入一组字符，遇到空白停止，读入的内容赋给运算符右侧的运算对象，该运算对象应该是一个 string 对象。

!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。

&&运算符（&& operator）逻辑与运算符，如果两个运算对象都是真，结果为真。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）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。

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：

138

```
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
+	一元正号	`+ expr`
-	一元负号	`- expr`
*	乘法	`expr * expr`
/	除法	`expr / expr`
%	求余	`expr % expr`
+	加法	`expr + expr`
-	减法	`expr - expr`

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
140 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：

141

```
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）。

C++11

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：

```
21 % 6;      /* 结果是 3 */      21 / 6;      /* 结果是 3 */
21 % 7;      /* 结果是 0 */      21 / 7;      /* 结果是 3 */
-21 % -8;    /* 结果是-5 */     -21 / -8;    /* 结果是 2 */
21 % -5;     /* 结果是 1 */      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	`!expr`
左	<	小于	`expr < expr`
左	<=	小于等于	`expr <= expr`
左	>	大于	`expr > expr`
左	>=	大于等于	`expr >= expr`
左	==	相等	`expr == expr`
左	!=	不相等	`expr != expr`
左	&&	逻辑与	`expr && expr`
左	||	逻辑或	`expr || expr`

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。 143

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 145

```
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： C++11

```
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在条件当中。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。 148

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，而且更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) ptr != 0 && *ptr++　　(b) ival++ && ival
(c) vec[ival++] <= vec[ival]

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) *iter++;　　(b) (*iter)++;　　(c) *iter.empty()
(d) iter->empty();　　(e) ++*iter;　　(f) iter++->empty();

151 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，如果写这条语句时没把括号写全就有可能产生意想不到的结果：

152

```
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 pl = 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
~	位求反	~ expr
<<	左移	expr1 << expr2
>>	右移	expr1 >> expr2
&	位与	expr & expr
^	位异或	expr ^ expr
|	位或	expr | expr

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
```

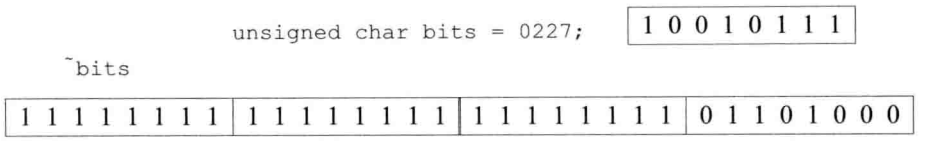
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 1 1

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；

154 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

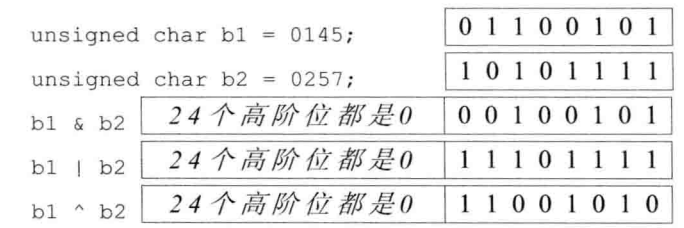
位求反运算符（~）将运算对象逐位求反后生成一个新值，将 1 置为 0、将 0 置为 1：



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：



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

156

```
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 ul1 = 3, ul2 = 7;
(a) ul1 & ul2                    (b) ul1 | ul2
(c) ul1 && ul2                   (d) ul1 || 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：

157

```
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
```

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

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。

158 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：

```
bool          flag;       char              cval;
short         sval;       unsignedshort     usval;
int           ival;       unsigned int      uival;
long          lval;       unsigned long     ulval;
float         fval;       double            dval;
```

```
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 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