

```
int i = 0, &r = i;
auto a = r;           // a 是一个整数 (r 是 i 的别名, 而 i 是一个整数)
```

其次, `auto` 一般会忽略掉顶层 `const` (参见 2.4.3 节, 第 57 页), 同时底层 `const` 则会保留下来, 比如当初始值是一个指向常量的指针时:

```
const int ci = i, &cr = ci;
auto b = ci;           // b 是一个整数 (ci 的顶层 const 特性被忽略掉了)
auto c = cr;           // c 是一个整数 (cr 是 ci 的别名, ci 本身是一个顶层 const)
auto d = &i;            // d 是一个整型指针 (整数的地址就是指向整数的指针)
auto e = &ci;           // e 是一个指向整数常量的指针 (对常量对象取地址是一种底层 const)
```

如果希望推断出的 `auto` 类型是一个顶层 `const`, 需要明确指出:

```
const auto f = ci;       // ci 的推演类型是 int, f 是 const int
```

还可以将引用的类型设为 `auto`, 此时原来的初始化规则仍然适用:

```
auto &g = ci;           // g 是一个整型常量引用, 绑定到 ci
auto &h = 42;             // 错误: 不能为非常量引用绑定字面值
const auto &j = 42;        // 正确: 可以为常量引用绑定字面值
```

**70** 设置一个类型为 `auto` 的引用时, 初始值中的顶层常量属性仍然保留。和往常一样, 如果我们给初始值绑定一个引用, 则此时的常量就不是顶层常量了。

要在一条语句中定义多个变量, 切记, 符号`&`和`*`只从属于某个声明符, 而非基本数据类型的一部分, 因此初始值必须是同一种类型:

```
auto k = ci, &l = i;      // k 是整数, l 是整型引用
auto &m = ci, *p = &ci;    // m 是对整型常量的引用, p 是指向整型常量的指针
// 错误: i 的类型是 int 而 &ci 的类型是 const int
auto &n = i, *p2 = &ci;
```

## 2.5.2 节练习

**练习 2.33:** 利用本节定义的变量, 判断下列语句的运行结果。

```
a = 42; b = 42; c = 42;
d = 42; e = 42; g = 42;
```

**练习 2.34:** 基于上一个练习中的变量和语句编写一段程序, 输出赋值前后变量的内容, 你刚才的推断正确吗? 如果不对, 请反复研读本节的示例直到你明白错在何处为止。

**练习 2.35:** 判断下列定义推断出的类型是什么, 然后编写程序进行验证。

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```



## 2.5.3 decltype 类型指示符

有时会遇到这种情况: 希望从表达式的类型推断出要定义的变量的类型, 但是不想用该表达式的值初始化变量。为了满足这一要求, C++11 新标准引入了第二种类型说明符 `decltype`, 它的作用是选择并返回操作数的数据类型。在此过程中, 编译器分析表达式并得到它的类型, 却不实际计算表达式的值:

```
decltype(f()) sum = x; // sum 的类型就是函数 f 的返回类型
```



编译器并不实际调用函数 `f`, 而是使用当调用发生时 `f` 的返回值类型作为 `sum` 的类型。换句话说, 编译器为 `sum` 指定的类型是什么呢? 就是假如 `f` 被调用的话将会返回的那个类型。

`decltype` 处理顶层 `const` 和引用的方式与 `auto` 有些许不同。如果 `decltype` 使用的表达式是一个变量, 则 `decltype` 返回该变量的类型(包括顶层 `const` 和引用在内):

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0;           // x 的类型是 const int
decltype(cj) y = x;          // y 的类型是 const int&, y 绑定到变量 x
decltype(cj) z;              // 错误: z 是一个引用, 必须初始化
```

因为 `cj` 是一个引用, `decltype(cj)` 的结果就是引用类型, 因此作为引用的 `z` 必须被初始化。

需要指出的是, 引用从来都作为其所指对象的同义词出现, 只有用在 `decltype` 处是一个例外。

## decltype 和引用



如果 `decltype` 使用的表达式不是一个变量, 则 `decltype` 返回表达式结果对应的类型。如 4.1.1 节(第 120 页)将要介绍的, 有些表达式将向 `decltype` 返回一个引用类型。一般来说当这种情况发生时, 意味着该表达式的结果对象能作为一条赋值语句的左值:

```
// decltype 的结果可以是引用类型
int i = 42, *p = &i, &r = i;
decltype(r + 0) b;      // 正确: 加法的结果是 int, 因此 b 是一个(未初始化的) int
decltype(*p) c;        // 错误: c 是 int&, 必须初始化
```

因为 `r` 是一个引用, 因此 `decltype(r)` 的结果是引用类型。如果想让结果类型是 `r` 所指的类型, 可以把 `r` 作为表达式的一部分, 如 `r+0`, 显然这个表达式的结果将是一个具体值而非一个引用。

另一方面, 如果表达式的内容是解引用操作, 则 `decltype` 将得到引用类型。正如我们所熟悉的那样, 解引用指针可以得到指针所指的对象, 而且还能给这个对象赋值。因此, `decltype(*p)` 的结果类型就是 `int&`, 而非 `int`。



`decltype` 和 `auto` 的另一处重要区别是, `decltype` 的结果类型与表达式形式密切相关。有一种情况需要特别注意: 对于 `decltype` 所用的表达式来说, 如果变量名加上了一对括号, 则得到的类型与不加括号时会有不同。如果 `decltype` 使用的是一个不加括号的变量, 则得到的结果就是该变量的类型; 如果给变量加上了一层或多层括号, 编译器就会把它当成是一个表达式。变量是一种可以作为赋值语句左值的特殊表达式, 所以这样的 `decltype` 就会得到引用类型:

```
// decltype 的表达式如果是加上了括号的变量, 结果将是引用
decltype((i)) d;      // 错误: d 是 int&, 必须初始化
decltype(i) e;         // 正确: e 是一个(未初始化的) int
```



**切记:** `decltype((variable))`(注意是双层括号)的结果永远是引用, 而 `decltype(variable)` 结果只有当 `variable` 本身就是一个引用时才是引用。

72

### 2.5.3 节练习

**练习 2.36:** 关于下面的代码，请指出每一个变量的类型以及程序结束时它们各自的值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

**练习 2.37:** 赋值是会产生引用的一类典型表达式，引用的类型就是左值的类型。也就是说，如果 *i* 是 int，则表达式 *i=x* 的类型是 int&。根据这一特点，请指出下面的代码中每一个变量的类型和值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

**练习 2.38:** 说明由 decltype 指定类型和由 auto 指定类型有何区别。请举出一个例子， decltype 指定的类型与 auto 指定的类型一样；再举一个例子， decltype 指定的类型与 auto 指定的类型不一样。



## 2.6 自定义数据结构

从最基本的层面理解，数据结构是把一组相关的数据元素组织起来然后使用它们的策略和方法。举一个例子，我们的 Sales\_item 类把书本的 ISBN 编号、售出量及销售收入等数据组织在了一起，并且提供诸如 isbn 函数、>>、<<、+、+= 等运算在内的一系列操作，Sales\_item 类就是一个数据结构。

C++语言允许用户以类的形式自定义数据类型，而库类型 string、istream、ostream 等也都是以类的形式定义的，就像第 1 章的 Sales\_item 类型一样。C++语言对类的支持甚多，事实上本书的第 III 部分和第 IV 部分都将大篇幅地介绍与类有关的知识。尽管 Sales\_item 类非常简单，但是要想给出它的完整定义可在第 14 章介绍自定义运算符之后。



### 2.6.1 定义 Sales\_data 类型

尽管我们还写不出完整的 Sales\_item 类，但是可以尝试着把那些数据元素组织到一起形成一个简单点儿的类。初步的想法是用户能直接访问其中的数据元素，也能实现一些基本的操作。

既然我们筹划的这个数据结构不带有任何运算功能，不妨把它命名为 Sales\_data 以示与 Sales\_item 的区别。Sales\_data 初步定义如下：

73

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

我们的类以关键字 struct 开始，紧跟着类名和类体（其中类体部分可以为空）。类体由

花括号包围形成了一个新的作用域（参见 2.2.4 节，第 43 页）。类内部定义的名字必须唯一，但是可以与类外部定义的名字重复。

类体右侧的表示结束的花括号后必须写一个分号，这是因为类体后面可以紧跟变量名以示对该类型对象的定义，所以分号必不可少：

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;  
// 与上一条语句等价，但可能更好一些  
struct Sales_data { /* ... */ };  
Sales_data accum, trans, *salesptr;
```

分号表示声明符（通常为空）的结束。一般来说，最好不要把对象的定义和类的定义放在一起。这么做无异于把两种不同实体的定义混在了一条语句里，一会儿定义类，一会儿又定义变量，显然这是一种不被建议的行为。



WARNING

很多新手程序员经常忘了在类定义的最后加上分号。

## 类数据成员

类体定义类的成员，我们的类只有数据成员（data member）。类的数据成员定义了类的对象的具体内容，每个对象都有自己的一份数据成员拷贝。修改一个对象的数据成员，不会影响其他 `Sales_data` 的对象。

定义数据成员的方法和定义普通变量一样：首先说明一个基本类型，随后紧跟一个或多个声明符。我们的类有 3 个数据成员：一个名为 `bookNo` 的 `string` 成员、一个名为 `units_sold` 的 `unsigned` 成员和一个名为 `revenue` 的 `double` 成员。每个 `Sales_data` 的对象都将包括这 3 个数据成员。

C++11 新标准规定，可以为数据成员提供一个类内初始值（in-class initializer）。创建对象时，类内初始值将用于初始化数据成员。没有初始值的成员将被默认初始化（参见 2.2.1 节，第 40 页）。因此当定义 `Sales_data` 的对象时，`units_sold` 和 `revenue` 都将初始化为 0，`bookNo` 将初始化为空字符串。

C++  
11

对类内初始值的限制与之前（参见 2.2.1 节，第 39 页）介绍的类似：或者放在花括号里，或者放在等号右边，记住不能使用圆括号。

7.2 节（第 240 页）将要介绍，用户可以使用 C++ 语言提供的另外一个关键字 `class` 来定义自己的数据结构，到时也将说明现在我们使用 `struct` 的原因。在第 7 章学习与 `class` 有关的知识之前，建议读者继续使用 `struct` 定义自己的数据类型。

### 2.6.1 节练习

&lt; 74

**练习 2.39：**编译下面的程序观察其运行结果，注意，如果忘记写类定义体后面的分号会发生什么情况？记录下相关信息，以后可能会有用。

```
struct Foo { /* 此处为空 */ } // 注意：没有分号  
int main()  
{  
    return 0;  
}
```

**练习 2.40：**根据自己的理解写出 `Sales_data` 类，最好与书中的例子有所区别。



## 2.6.2 使用 Sales\_data 类

和 Sales\_item 类不同的是，我们自定义的 Sales\_data 类没有提供任何操作，Sales\_data 类的使用者如果想执行什么操作就必须自己动手实现。例如，我们将参照 1.5.2 节（第 20 页）的例子写一段程序实现求两次交易相加结果的功能。程序的输入是下面这两条交易记录：

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

每笔交易记录着图书的 ISBN 编号、售出数量和售出单价。

### 添加两个 Sales\_data 对象

因为 Sales\_data 类没有提供任何操作，所以我们必须自己编码实现输入、输出和相加的功能。假设已知 Sales\_data 类定义于 Sales\_data.h 文件内，2.6.3 节（第 67 页）将详细介绍定义头文件的方法。

因为程序比较长，所以接下来分成几部分介绍。总的来说，程序的结构如下：

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // 读入 data1 和 data2 的代码
    // 检查 data1 和 data2 的 ISBN 是否相同的代码
    // 如果相同，求 data1 和 data2 的总和
}
```

和原来的程序一样，先把所需的头文件包含进来并且定义变量用于接受输入。和 Sales\_item 类不同的是，新程序还包含了 string 头文件，因为我们的代码中将用到 string 类型的成员变量 bookNo。

75

### Sales\_data 对象读入数据

第 3 章和第 10 章将详细介绍 string 类型的细节，在此之前，我们先了解一点儿关于 string 的知识以便定义和使用我们的 ISBN 成员。string 类型其实就是字符的序列，它的操作有`>>`、`<<`和`==`等，功能分别是读入字符串、写出字符串和比较字符串。这样我们就能书写代码读入第一笔交易了：

```
double price = 0; // 书的单价，用于计算销售收入
// 读入第 1 笔交易： ISBN、销售数量、单价
std::cin >> data1.bookNo >> data1.units_sold >> price;
// 计算销售收入
data1.revenue = data1.units_sold * price;
```

交易信息记录的是书售出的单价，而数据结构存储的是一次交易的销售收入，因此需要将单价读入到 double 变量 price，然后再计算销售收入 revenue。输入语句

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

使用点操作符（参见 1.5.2 节，第 20 页）读入对象 data1 的 bookNo 成员和 units\_sold 成员。

最后一条语句把 data1.units\_sold 和 price 的乘积赋值给 data1 的 revenue 成员。

接下来程序重复上述过程读入对象 data2 的数据:

```
// 读入第 2 笔交易
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

### 输出两个 Sales\_data 对象的和

剩下的工作就是检查两笔交易涉及的 ISBN 编号是否相同了。如果相同输出它们的和，否则输出一条报错信息:

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // 输出: ISBN、总销售量、总销售额、平均价格
    std::cout << data1.bookNo << " " << totalCnt
        << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0;           // 标示成功
} else {               // 两笔交易的 ISBN 不一样
    std::cerr << "Data must refer to the same ISBN"
        << std::endl;
    return -1;          // 标示失败
}
```

在第一个 if 语句中比较了 data1 和 data2 的 bookNo 成员是否相同。如果相同则执行第一个 if 语句花括号内的操作，首先计算 units\_sold 的和并赋给变量 totalCnt，然后计算 revenue 的和并赋给变量 totalRevenue，输出这些值。接下来检查是否确实售出了书籍，如果是，计算并输出每本书的平均价格；如果售量为零，输出一条相应的信息。

&lt; 76

### 2.6.2 节练习

**练习 2.41：**使用你自己的 Sales\_data 类重写 1.5.1 节（第 20 页）、1.5.2 节（第 21 页）和 1.6 节（第 22 页）的练习。眼下先把 Sales\_data 类的定义和 main 函数放在同一个文件里。

### 2.6.3 编写自己的头文件



尽管如 19.7 节（第 754 页）所讲可以在函数体内定义类，但是这样的类毕竟受到了一些限制。所以，类一般都不定义在函数体内。当在函数体外部定义类时，在各个指定的源文件中可能只有一处该类的定义。而且，如果要在不同文件中使用同一个类，类的定义就必须保持一致。

为了确保各个文件中类的定义一致，类通常被定义在头文件中，而且类所在头文件的名字应与类的名字一样。例如，库类型 string 在名为 string 的头文件中定义。又如，我们应该把 Sales\_data 类定义在名为 Sales\_data.h 的头文件中。

头文件通常包含那些只能被定义一次的实体，如类、const 和 constexpr 变量（参见 2.4 节，第 54 页）等。头文件也经常用到其他头文件的功能。例如，我们的 Sales\_data 类包含有一个 string 成员，所以 Sales\_data.h 必须包含 string.h 头文件。同时，使用 Sales\_data 类的程序为了能操作 bookNo 成员需要再一次包含 string.h 头文件。

这样，事实上使用 `Sales_data` 类的程序就先后两次包含了 `string.h` 头文件：一次是直接包含的，另有一次是随着包含 `Sales_data.h` 被隐式地包含进来的。有必要在书写头文件时做适当处理，使其遇到多次包含的情况也能安全和正常地工作。



头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

## 预处理器概述

确保头文件多次包含仍能安全工作的常用技术是预处理器（preprocessor），它由 C++ 语言从 C 语言继承而来。预处理器是在编译之前执行的一段程序，可以部分地改变我们所写的程序。之前已经用到了一项预处理功能`#include`，当预处理器看到`#include` 标记时就会用指定的头文件的内容代替`#include`。

C++ 程序还会用到的一项预处理功能是头文件保护符（header guard），头文件保护符依赖于预处理变量（参见 2.3.2 节，第 48 页）。预处理变量有两种状态：已定义和未定义。`#define` 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义：`#ifdef` 当且仅当变量已定义时为真，`#ifndef` 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直至遇到`#endif` 指令为止。

使用这些功能就能有效地防止重复包含的发生：

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

第一次包含 `Sales_data.h` 时，`#ifndef` 的检查结果为真，预处理器将顺序执行后面的操作直至遇到`#endif` 为止。此时，预处理变量 `SALES_DATA_H` 的值将变为已定义，而且 `Sales_data.h` 也会被拷贝到我们的程序中来。后面如果再一次包含 `Sales_data.h`，则`#ifndef` 的检查结果将为假，编译器将忽略`#ifndef` 到`#endif` 之间的部分。



预处理变量无视 C++ 语言中关于作用域的规则。



头文件即使（目前还）没有被包含在任何其他头文件中，也应该设置保护符。

头文件保护符很简单，程序员只要习惯性地加上就可以了，没必要太在乎你的程序到底需不需要。

### 2.6.3 节练习

**练习 2.42：**根据你自己的理解重写一个 `Sales_data.h` 头文件，并以此为基础重做 2.6.2 节（第 67 页）的练习。

## 小结

78

类型是 C++ 编程的基础。

类型规定了其对象的存储要求和所能执行的操作。C++ 语言提供了一套基础内置类型，如 int 和 char 等，这些类型与实现它们的机器硬件密切相关。类型分为非常量和常量，一个常量对象必须初始化，而且一旦初始化其值就不能再改变。此外，还可以定义复合类型，如指针和引用等。复合类型的定义以其他类型为基础。

C++ 语言允许用户以类的形式自定义类型。C++ 库通过类提供了一套高级抽象类型，如输入输出和 string 等。

## 术语表

**地址 (address)** 是一个数字，根据它你可以找到内存中的一个字节。

**别名声明 (alias declaration)** 为另外一种类型定义一个同义词：使用“名字=类型”的格式将名字作为该类型的同义词。

**算术类型 (arithmetic type)** 布尔值、字符、整数、浮点数等内置类型。

**数组 (array)** 是一种数据结构，存放着一组未命名的对象，可以通过索引来访问这些对象。3.5 节将详细介绍数组的知识。

**auto** 是一个类型说明符，通过变量的初始值来推断变量的类型。

**基本类型 (base type)** 是类型说明符，可用 const 修饰，在声明语句中位于声明符之前。基本类型提供了最常见的数据类型，以此为基础构建声明符。

**绑定 (bind)** 令某个名字与给定的实体关联在一起，使用该名字也就是使用该实体。例如，引用就是将某个名字与某个对象绑定在一起。

**字节 (byte)** 内存中可寻址的最小单元，大多数机器的字节占 8 位。

**类成员 (class member)** 类的组成部分。

**复合类型 (compound type)** 是一种类型，它的定义以其他类型为基础。

**const** 是一种类型修饰符，用于说明永不改变的对象。**const** 对象一旦定义就无法再

赋新值，所以必须初始化。

**常量指针 (const pointer)** 是一种指针，它的值永不改变。

**常量引用 (const reference)** 是一种习惯叫法，含义是指向常量的引用。

**常量表达式 (const expression)** 能在编译时计算并获取结果的表达式。

**constexpr** 是一种函数，用于代表一条常量表达式。6.5.2 节（第 214 页）将介绍 constexpr 函数。

**转换 (conversion)** 一种类型的值转变成另外一种类型值的过程。C++ 语言支持内置类型之间的转换。

**数据成员 (data member)** 组成对象的数据元素，类的每个对象都有类的数据成员的一份拷贝。数据成员可以在类内部声明的同时初始化。

**声明 (declaration)** 声称存在一个变量、函数或是别处定义的类型。名字必须在定义或声明之后才能使用。

**声明符 (declarator)** 是声明的一部分，包括被定义的名字和类型修饰符，其中类型修饰符可以有也可以没有。

**decltype** 是一个类型说明符，从变量或表达式推断得到类型。

**默认初始化 (default initialization)** 当对象未被显式地赋予初始值时执行的初始化行

79

为。由类本身负责执行的类对象的初始化行为。全局作用域的内置类型对象初始化为 0；局部作用域的对象未被初始化即拥有未定义的值。

**定义 (definition)** 为某一特定类型的变量申请存储空间，可以选择初始化该变量。名字必须在定义或声明之后才能使用。

**转义序列 (escape sequence)** 字符特别是那些不可打印字符的替代形式。转义以反斜线开头，后面紧跟一个字符，或者不多于 3 个八进制数字，或者字母 x 加上 1 个十六进制数。

**全局作用域 (global scope)** 位于其他所有作用域之外的作用域。

**头文件保护符 (header guard)** 使用预处理变量以防止头文件被某个文件重复包含。

**标识符 (identifier)** 组成名字的字符序列，标识符对大小写敏感。

**类内初始值 (in-class initializer)** 在声明类的数据成员时同时提供的初始值，必须置于等号右侧或花括号内。

**在作用域内 (in scope)** 名字在当前作用域内可见。

**被初始化 (initialized)** 变量在定义的同时被赋予初始值，变量一般都应该被初始化。

**内层作用域 (inner scope)** 嵌套在其他作用域之内的作用域。

**整型 (integral type)** 参见算术类型。

**列表初始化 (list initialization)** 利用花括号把一个或多个初始值放在一起的初始化形式。

**字面值 (literal)** 是一个不能改变的值，如数字、字符、字符串等。单引号内的是字符字面值，双引号内的是字符串字面值。

**局部作用域 (local scope)** 是块作用域的习惯叫法。

**底层 const (low-level const)** 一个不属于顶层的 const，类型如果由底层常量定义，

则不能被忽略。

**成员 (member)** 类的组成部分。

**不可打印字符 (nonprintable character)** 不具有可见形式的字符，如控制符、退格、换行符等。

**空指针 (null pointer)** 值为 0 的指针，空指针合法但是不指向任何对象。

**nullptr** 是表示空指针的字面值常量。

**对象 (object)** 是内存的一块区域，具有某种类型，变量是命名了的对象。

**外层作用域 (outer scope)** 嵌套着别的作用域的作用域。

**指针 (pointer)** 是一个对象，存放着某个对象的地址，或者某个对象存储区域之后的下一地址，或者 0。

**指向常量的指针 (pointer to const)** 是一个指针，存放着某个常量对象的地址。指向常量的指针不能用来改变它所指对象的值。

**预处理器 (preprocessor)** 在 C++ 编译过程中执行的一段程序。

**预处理变量 (preprocessor variable)** 由预处理器管理的变量。在程序编译之前，预处理器负责将程序中的预处理变量替换成它的真实值。

**引用 (reference)** 是某个对象的别名。

80 对常量的引用 (reference to const) 是一个引用，不能用来改变它所绑定对象的值。对常量的引用可以绑定常量对象，或者非常量对象，或者表达式的结果。

**作用域 (scope)** 是程序的一部分，在其中某些名字有意义。C++ 有几级作用域：

**全局 (global)** ——名字定义在所有其他作用域之外。

**类 (class)** ——名字定义在类内部。

**命名空间 (namespace)** ——名字定义在命名空间内部。

**块 (block)** ——名字定义在块内部。

名字从声明位置开始直至声明语句所在的作用域末端为止都是可用的。

**分离式编译 (separate compilation)** 把程序分割为多个单独文件的能力。

**带符号类型 (signed)** 保存正数、负数或 0 的整型。

**字符串 (string)** 是一种库类型，表示可变长字符序列。

**struct** 是一个关键字，用于定义类。

**临时值 (temporary)** 编译器在计算表达式结果时创建的无名对象。为某表达式创建了一个临时值，则此临时值将一直存在直到包含有该表达式的最大的表达式计算完成为止。

**顶层 const (top-level const)** 是一个 **const**，规定某对象的值不能改变。

**类型别名 (type alias)** 是一个名字，是另外一个类型的同义词，通过关键字 **typedef** 或别名声明语句来定义。

**类型检查 (type checking)** 是一个过程，编译器检查程序使用某给定类型对象的方式与该类型的定义是否一致。

**类型说明符 (type specifier)** 类型的名字。

**typedef** 为某类型定义一个别名。当关键字 **typedef** 作为声明的基本类型出现时，声明中定义的名字就是类型名。

**未定义 (undefined)** 即 C++ 语言没有明确规定的情况。不论是否有意为之，未定义行为都可能引发难以追踪的运行时错误、安全问题和可移植性问题。

**未初始化 (uninitialized)** 变量已定义但未被赋予初始值。一般来说，试图访问未初始化变量的值将引发未定义行为。

**无符号类型 (unsigned)** 保存大于等于 0 的整型。

**变量 (variable)** 命名的对象或引用。C++ 语言要求变量要先声明后使用。

**void\*** 可以指向任意非常量的指针类型，不能执行解引用操作。

**void 类型** 是一种有特殊用处的类型，既无操作也无值。不能定义一个 **void** 类型的变量。

**字 (word)** 在指定机器上进行整数运算的自然单位。一般来说，字的空间足够存放地址。32 位机器上的字通常占据 4 个字节。

**& 运算符 (& operator)** 取地址运算符。

**\* 运算符 (\* operator)** 解引用运算符。解引用一个指针将返回该指针所指的对象，为解引用的结果赋值也就是为指针所指的对象赋值。

**#define** 是一条预处理指令，用于定义一个预处理变量。

**#endif** 是一条预处理指令，用于结束一个 **#ifdef** 或 **#ifndef** 区域。

**#ifdef** 是一条预处理指令，用于判断给定的变量是否已经定义。

**#ifndef** 是一条预处理指令，用于判断给定的变量是否尚未定义。



# 第3章 字符串、向量和数组

## 内容

---

3.1 命名空间的 using 声明 .....	74
3.2 标准库类型 string .....	75
3.3 标准库类型 vector .....	86
3.4 迭代器介绍 .....	95
3.5 数组 .....	101
3.6 多维数组 .....	112
小结 .....	117
术语表 .....	117

除了第 2 章介绍的内置类型之外, C++ 语言还定义了一个内容丰富的抽象数据类型库。其中, `string` 和 `vector` 是两种最重要的标准库类型, 前者支持可变长字符串, 后者则表示可变长的集合。还有一种标准库类型是迭代器, 它是 `string` 和 `vector` 的配套类型, 常被用于访问 `string` 中的字符或 `vector` 中的元素。

内置数组是一种更基础的类型, `string` 和 `vector` 都是对它的某种抽象。本章将分别介绍数组以及标准库类型 `string` 和 `vector`。

82 第2章介绍的内置类型是由C++语言直接定义的。这些类型，比如数字和字符，体现了大多数计算机硬件本身具备的能力。标准库定义了另外一组具有更高级性质的类型，它们尚未直接实现到计算机硬件中。

本章将介绍两种最重要的标准库类型：`string` 和 `vector`。`string` 表示可变长的字符串序列，`vector` 存放的是某种给定类型对象的可变长序列。本章还将介绍内置数组类型，和其他内置类型一样，数组的实现与硬件密切相关。因此相较于标准库类型 `string` 和 `vector`，数组在灵活性上稍显不足。

在开始介绍标准库类型之前，先来学习一种访问库中名字的简单方法。

## 3.1 命名空间的 `using` 声明

目前为止，我们用到的库函数基本上都属于命名空间 `std`，而程序也显式地将这一点标示了出来。例如，`std::cin` 表示从标准输入中读取内容。此处使用作用域操作符 (`::`) (参见 1.2 节，第 7 页) 的含义是：编译器应从操作符左侧名字所示的作用域中寻找右侧那个名字。因此，`std::cin` 的意思就是要使用命名空间 `std` 中的名字 `cin`。

上面的方法显得比较烦琐，然而幸运的是，通过更简单的途径也能使用到命名空间中的成员。本节将学习其中一种最安全的方法，也就是使用 **using 声明** (using declaration)，18.2.2 节 (第 702 页) 会介绍另一种方法。

有了 `using` 声明就无须专门的前缀 (形如命名空间`::`) 也能使用所需的名字了。`using` 声明具有如下的形式：

```
using namespace ::name;
```

一旦声明了上述语句，就可以直接访问命名空间中的名字：

```
#include <iostream>
// using 声明，当我们使用名字 cin 时，从命名空间 std 中获取它
using std::cin;

int main()
{
    int i;
    cin >> i;           // 正确：cin 和 std::cin 含义相同
    cout << i;          // 错误：没有对应的 using 声明，必须使用完整的名字
    std::cout << i;    // 正确：显式地从 std 中使用 cout
    return 0;
}
```

### 每个名字都需要独立的 `using` 声明

按照规定，每个 `using` 声明引入命名空间中的一个成员。例如，可以把要用到的标准库中的名字都以 `using` 声明的形式表示出来，重写 1.2 节 (第 5 页) 的程序如下：

83

```
#include <iostream>
// 通过下列 using 声明，我们可以使用标准库中的名字
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
```

```
int v1, v2;
cin >> v1 >> v2;
cout << "The sum of " << v1 << " and " << v2
    << " is " << v1 + v2 << endl;
return 0;
}
```

在上述程序中，一开始就有对 `cin`、`cout` 和 `endl` 的 `using` 声明，这意味着我们不用再添加 `std::` 形式的前缀就能直接使用它们。C++语言的形式比较自由，因此既可以一行只放一条 `using` 声明语句，也可以一行放上多条。不过要注意，用到的每个名字都必须有自己的声明语句，而且每句话都得以分号结束。

### 头文件不应包含 `using` 声明

位于头文件的代码（参见 2.6.3 节，第 67 页）一般来说不应该使用 `using` 声明。这是因为头文件的内容会拷贝到所有引用它的文件中去，如果头文件里有某个 `using` 声明，那么每个使用了该头文件的文件就都会有这个声明。对于某些程序来说，由于不经意间包含了一些名字，反而可能产生始料未及的名字冲突。

### 一点注意事项

经本节所述，后面的所有例子将假设，但凡用到的标准库中的名字都已经使用 `using` 语句声明过了。例如，我们将在代码中直接使用 `cin`，而不再使用 `std::cin`。

为了让书中的代码尽量简洁，今后将不会再把所有 `using` 声明和 `#include` 指令一一标出。附录 A 中的表 A.1（第 766 页）列出了本书涉及的所有标准库中的名字及对应的头文件。



读者请注意：在编译及运行本书的示例前请为代码添加必要的 `#include` 指令和 `using` 声明。

## 3.1 节练习

练习 3.1：使用恰当的 `using` 声明重做 1.4.1 节（第 11 页）和 2.6.2 节（第 67 页）的练习。

## 3.2 标准库类型 `string`



标准库类型 `string` 表示可变长的字符序列，使用 `string` 类型必须首先包含 `string` 头文件。作为标准库的一部分，`string` 定义在命名空间 `std` 中。接下来的示例都假定已包含了下述代码：

```
#include <string>
using std::string;
```

本节描述最常用的 `string` 操作，9.5 节（第 320 页）还将介绍另外一些。



C++ 标准一方面对库类型所提供的操作做了详细规定，另一方面也对库的实现者做出一些性能上的需求。因此，标准库类型对于一般应用场合来说有足够的效率。



### 3.2.1 定义和初始化 string 对象

如何初始化类的对象是由类本身决定的。一个类可以定义很多种初始化对象的方式，只不过这些方式之间必须有所区别：或者是初始值的数量不同，或者是初始值的类型不同。

表 3.1 列出了初始化 `string` 对象最常用的一些方式，下面是几个例子：

```
string s1;                      // 默认初始化，s1 是一个空字符串
string s2 = s1;                  // s2 是 s1 的副本
string s3 = "hiya";              // s3 是该字符串字面值的副本
string s4(10, 'c');              // s4 的内容是 ccccccccccc
```

可以通过默认的方式（参见 2.2.1 节，第 40 页）初始化一个 `string` 对象，这样就会得到一个空的 `string`，也就是说，该 `string` 对象中没有任何字符。如果提供了一个字符串字面值（参见 2.1.3 节，第 36 页），则该字面值中除了最后那个空字符外其他所有的字符都被拷贝到新创建的 `string` 对象中去。如果提供的是一个数字和一个字符，则 `string` 对象的内容是给定字符连续重复若干次后得到的序列。

表 3.1：初始化 `string` 对象的方式

<code>string s1</code>	默认初始化，s1 是一个空串
<code>string s2(s1)</code>	s2 是 s1 的副本
<code>string s2 = s1</code>	等价于 <code>s2(s1)</code> ，s2 是 s1 的副本
<code>string s3("value")</code>	s3 是字面值" value "的副本，除了字面值最后的那个空字符外
<code>string s3 = "value"</code>	等价于 <code>s3("value")</code> ，s3 是字面值" value "的副本
<code>string s4(n, 'c')</code>	把 s4 初始化为由连续 n 个字符 c 组成的串

#### 直接初始化和拷贝初始化

由 2.2.1 节（第 39 页）的学习可知，C++语言有几种不同的初始化方式，通过 `string` 我们可以清楚地看到在这些初始化方式之间到底有什么区别和联系。如果使用等号 (=) 初始化一个变量，实际上执行的是拷贝初始化（copy initialization），编译器把等号右侧的初始值拷贝到新创建的对象中去。与之相反，如果不使用等号，则执行的是直接初始化（direct initialization）。

当初始值只有一个时，使用直接初始化或拷贝初始化都行。如果像上面的 `s4` 那样初始化要用到的值有多个，一般来说只能使用直接初始化的方式：

```
string s5 = "hiya";          // 拷贝初始化
string s6("hiya");          // 直接初始化
string s7(10, 'c');          // 直接初始化，s7 的内容是 ccccccccccc
```

对于用多个值进行初始化的情况，非要用拷贝初始化的方式来处理也不是不可以，不过需要显式地创建一个（临时）对象用于拷贝：

```
string s8 = string(10, 'c'); // 拷贝初始化，s8 的内容是 ccccccccccc
```

`s8` 的初始值是 `string(10, 'c')`，它实际上是用数字 10 和字符 c 两个参数创建出来的一个 `string` 对象，然后这个 `string` 对象又拷贝给了 `s8`。这条语句本质上等价于下面的两条语句：

```
string temp(10, 'c');        // temp 的内容是 ccccccccccc
string s8 = temp;            // 将 temp 拷贝给 s8
```

其实我们可以看到，尽管初始化 s8 的语句合法，但和初始化 s7 的方式比较起来可读性较差，也没有任何补偿优势。

### 3.2.2 string 对象上的操作



一个类除了要规定初始化其对象的方式外，还要定义对象上所能执行的操作。其中，类既能定义通过函数名调用的操作，就像 Sales\_item 类的 isbn 函数那样（参见 1.5.2 节，第 20 页），也能定义 <<、+ 等各种运算符在该类对象上的新含义。表 3.2 中列举了 string 的大多数操作。

表 3.2: string 的操作

os<<s	将 s 写到输出流 os 当中，返回 os
is>>s	从 is 中读取字符串赋给 s，字符串以空白分隔，返回 is
getline(is, s)	从 is 中读取一行赋给 s，返回 is
s.empty()	s 为空返回 true，否则返回 false
s.size()	返回 s 中字符的个数
s[n]	返回 s 中第 n 个字符的引用，位置 n 从 0 计起
s1+s2	返回 s1 和 s2 连接后的结果
s1=s2	用 s2 的副本代替 s1 中原来的字符
s1==s2	如果 s1 和 s2 中所含的字符完全一样，则它们相等；string 对象的相等性判断对字母的大小写敏感
s1!=s2	等性判断对字母的大小写敏感
<, <=, >, >=	利用字符在字典中的顺序进行比较，且对字母的大小写敏感

### 读写 string 对象

第 1 章曾经介绍过，使用标准库中的 `iostream` 来读写 `int`、`double` 等内置类型的值。同样，也可以使用 IO 操作符读写 `string` 对象：

```
// 注意：要想编译下面的代码还需要适当的#include 语句和 using 声明
int main()
{
    string s;           // 空字符串
    cin >> s;          // 将 string 对象读入 s，遇到空白停止
    cout << s << endl; // 输出 s
    return 0;
}
```

这段程序首先定义一个名为 s 的空 `string`，然后将标准输入的内容读取到 s 中。在执行读取操作时，`string` 对象会自动忽略开头的空白（即空格符、换行符、制表符等）并从第一个真正的字符开始读起，直到遇见下一处空白为止。 ◀ 86

如上所述，如果程序的输入是“Hello World!”（注意开头和结尾处的空格），则输出将是“Hello”，输出结果中没有任何空格。

和内置类型的输入输出操作一样，`string` 对象的此类操作也是返回运算符左侧的运算对象作为其结果。因此，多个输入或者多个输出可以连写在一起：

```
string s1, s2;
cin >> s1 >> s2;           // 把第一个输入读到 s1 中，第二个输入读到 s2 中
cout << s1 << s2 << endl; // 输出两个 string 对象
```

假设给上面这段程序输入与之前一样的内容“**Hello World!**”，输出将是“**HelloWorld!**”。

### 读取未知数量的 string 对象

1.4.3 节（第 13 页）的程序可以读入数量未知的整数，下面编写一个类似的程序用于读取 string 对象：

```
int main()
{
    string word;
    while (cin >> word)           // 反复读取，直至到达文件末尾
        cout << word << endl;      // 逐个输出单词，每个单词后面紧跟一个换行
    return 0;
}
```

在该程序中，读取的对象是 string 而非 int，但是 while 语句的条件部分和之前版本的程序是一样的。该条件负责在读取时检测流的情况，如果流有效，也就是说没遇到文件结束标记或非法输入，那么执行 while 语句内部的操作。此时，循环体将输出刚刚从标准输入读取的内容。重复若干次之后，一旦遇到文件结束标记或非法输入循环也就结束了。

### 使用 `getline` 读取一整行

有时我们希望能在最终得到的字符串中保留输入时的空白符，这时应该用 `getline` 函数代替原来的`>>`运算符。`getline` 函数的参数是一个输入流和一个 string 对象，函数从给定的输入流中读入内容，直到遇到换行符为止（注意换行符也被读进来了），然后把所读的内容存入到那个 string 对象中去（注意不存换行符）。`getline` 只要一遇到换行符就结束读取操作并返回结果，哪怕输入的一开始就是换行符也是如此。如果输入真的一开始就是换行符，那么所得的结果是个空 string。

和输入运算符一样，`getline` 也会返回它的流参数。因此既然输入运算符能作为判断的条件（参见 1.4.3 节，第 13 页），我们也能用 `getline` 的结果作为条件。例如，可以通过改写之前的程序让它一次输出一整行，而不再是每行输出一个词：

```
int main()
{
    string line;
    // 每次读入一整行，直至到达文件末尾
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

因为 `line` 中不包含换行符，所以我们手动地加上换行操作符。和往常一样，使用 `endl` 结束当前行并刷新显示缓冲区。



触发 `getline` 函数返回的那个换行符实际上被丢弃掉了，得到的 string 对象中并不包含该换行符。

### string 的 `empty` 和 `size` 操作

顾名思义，`empty` 函数根据 string 对象是否为空返回一个对应的布尔值（参见第

2.1 节, 30 页)。和 Sales\_item 类(参见 1.5.2 节, 第 20 页)的 isbn 成员一样, empty 也是 string 的一个成员函数。调用该函数的方法很简单, 只要使用点操作符指明是哪个对象执行了 empty 函数就可以了。

通过改写之前的程序, 可以做到只输出非空的行:

```
// 每次读入一整行, 遇到空行直接跳过
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

在上面的程序中, if 语句的条件部分使用了逻辑非运算符 (!), 它返回与其运算对象相反的结果。此例中, 如果 str 不为空则返回真。

**size** 函数返回 string 对象的长度(即 string 对象中字符的个数), 可以使用 size 88 函数只输出长度超过 80 个字符的行:

```
string line;
// 每次读入一整行, 输出其中超过 80 个字符的行
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

### string::size\_type 类型

对于 size 函数来说, 返回一个 int 或者如前面 2.1.1 节(第 31 页)所述的那样返回一个 unsigned 似乎都是合情合理的。但其实 size 函数返回的是一个 string::size\_type 类型的值, 下面就对这种新的类型稍作解释。

string 类及其他大多数标准库类型都定义了几种配套的类型。这些配套类型体现了标准库类型与机器无关的特性, 类型 **size\_type** 即是其中的一种。在具体使用的时候, 通过作用域操作符来表明名字 size\_type 是在类 string 中定义的。

尽管我们不太清楚 string::size\_type 类型的细节, 但有一点是肯定的: 它是一个无符号类型的值(参见 2.1.1 节, 第 30 页)而且能足够存放下任何 string 对象的大小。所有用于存放 string 类的 size 函数返回值的变量, 都应该是 string::size\_type 类型的。

过去, string::size\_type 这种类型有点儿神秘, 不太容易理解和使用。在 C++11 新标准中, 允许编译器通过 auto 或者 decltype(参见 2.5.2 节, 第 61 页)来推断变量的类型: C++ 11

```
auto len = line.size(); // len 的类型是 string::size_type
```

由于 size 函数返回的是一个无符号整型数, 因此切记, 如果在表达式中混用了带符号数和无符号数将可能产生意想不到的结果(参见 2.1.2 节, 第 33 页)。例如, 假设 n 是一个具有负值的 int, 则表达式 s.size()<n 的判断结果几乎肯定是 true。这是因为负值 n 会自动地转换成一个比较大的无符号值。



如果一条表达式中已经有了 size() 函数就不要再使用 int 了, 这样可以避免混用 int 和 unsigned 可能带来的问题。

### 比较 string 对象

string 类定义了几种用于比较字符串的运算符。这些比较运算符逐一比较 string

对象中的字符，并且对大小写敏感，也就是说，在比较时同一个字母的大写形式和小写形式是不同的。

相等性运算符（`==`和`!=`）分别检验两个 `string` 对象相等或不相等，`string` 对象相等意味着它们的长度相同而且所包含的字符也全都相同。关系运算符`<`、`<=`、`>`、`>=`分别检验一个 `string` 对象是否小于、小于等于、大于、大于等于另外—个 `string` 对象。上述这些运算符都依照（大小写敏感的）字典顺序：

- 89> 1. 如果两个 `string` 对象的长度不同，而且较短 `string` 对象的每个字符都与较长 `string` 对象对应位置上的字符相同，就说较短 `string` 对象小于较长 `string` 对象。  
 2. 如果两个 `string` 对象在某些对应的位置上不一致，则 `string` 对象比较的结果其实是 `string` 对象中第一对相异字符比较的结果。

下面是 `string` 对象比较的一个示例：

```
string str = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

根据规则 1 可判断，对象 `str` 小于对象 `phrase`；根据规则 2 可判断，对象 `slang` 既大于 `str` 也大于 `phrase`。

### 为 `string` 对象赋值

一般来说，在设计标准库类型时都力求在易用性上向内置类型看齐，因此大多数库类型都支持赋值操作。对于 `string` 类而言，允许把一个对象的值赋给另外一个对象：

```
string st1(10, 'c'), st2;// st1 的内容是 cccccccccc; st2 是一个空字符串
st1 = st2;                // 赋值：用 st2 的副本替换 st1 的内容
                           // 此时 st1 和 st2 都是空字符串
```

### 两个 `string` 对象相加

两个 `string` 对象相加得到一个新的 `string` 对象，其内容是把左侧的运算对象与右侧的运算对象串接而成。也就是说，对 `string` 对象使用加法运算符（`+`）的结果是一个新的 `string` 对象，它所包含的字符由两部分组成：前半部分是加号左侧 `string` 对象所含的字符、后半部分是加号右侧 `string` 对象所含的字符。另外，复合赋值运算符（`+=`）（参见 1.4.1 节，第 10 页）负责把右侧 `string` 对象的内容追加到左侧 `string` 对象的后面：

```
string s1 = "hello, ", s2 = "world\n";
string s3 = s1 + s2;      // s3 的内容是 hello, world\n
s1 += s2;                // 等价于 s1 = s1 + s2
```

### 字面值和 `string` 对象相加

如 2.1.2 节（第 33 页）所讲的，即使一种类型并非所需，我们也可以使用它，不过前提是该种类型可以自动转换成所需的类型。因为标准库允许把字符字面值和字符串字面值（参见 2.1.3 节，第 36 页）转换成 `string` 对象，所以在需要 `string` 对象的地方就可以使用这两种字面值来替代。利用这一点将之前的程序改写为如下形式：

```
string s1 = "hello", s2 = "world"; // 在 s1 和 s2 中都没有标点符号
string s3 = s1 + ", " + s2 + '\n';
```

当把 `string` 对象和字符串字面值及字符串字面值混在一条语句中使用时，必须确保每个加法运算符（`+`）的两侧的运算对象至少有一个是 `string`：

```
string s4 = s1 + ", " ; // 正确：把一个 string 对象和一个字面值相加
string s5 = "hello" + ", " ; // 错误：两个运算对象都不是 string
// 正确：每个加法运算符都有一个运算对象是 string
string s6 = s1 + ", " + "world";
string s7 = "hello" + ", " + s2; // 错误：不能把字面值直接相加
```

90

`s4` 和 `s5` 初始化时只用到了一个加法运算符，因此很容易判断是否合法。`s6` 的初始化形式之前没有出现过，但其实它的工作机理和连续输入连续输出（参见 1.2 节，第 6 页）是一样的，可以用如下的形式分组：

```
string s6 = (s1 + ", ") + "world";
```

其中子表达式 `s1 + ", "` 的结果是一个 `string` 对象，它同时作为第二个加法运算符的左侧运算对象，因此上述语句和下面的两个语句是等价的：

```
string tmp = s1 + ", " ; // 正确：加法运算符有一个运算对象是 string
s6 = tmp + "world"; // 正确：加法运算符有一个运算对象是 string
```

另一方面，`s7` 的初始化是非法的，根据其语义加上括号后就成了下面的形式：

```
string s7 = ("hello" + ", ") + s2; // 错误：不能把字面值直接相加
```

很容易看到，括号内的子表达式试图把两个字符串字面值加在一起，而编译器根本没法做到这一点，所以这条语句是错误的。



因为某些历史原因，也为了与 C 兼容，所以 C++ 语言中的字符串字面值并不是标准库类型 `string` 的对象。切记，字符串字面值与 `string` 是不同的类型。

### 3.2.2 节练习

**练习 3.2：** 编写一段程序从标准输入中一次读入一整行，然后修改该程序使其一次读入一个词。

**练习 3.3：** 请说明 `string` 类的输入运算符和 `getline` 函数分别是如何处理空白字符的。

**练习 3.4：** 编写一段程序读入两个字符串，比较其是否相等并输出结果。如果不相等，输出较大的那个字符串。改写上述程序，比较输入的两个字符串是否等长，如果不等长，输出长度较大的那个字符串。

**练习 3.5：** 编写一段程序从标准输入中读入多个字符串并将它们连接在一起，输出连接成的大字符串。然后修改上述程序，用空格把输入的多个字符串分隔开来。

### 3.2.3 处理 `string` 对象中的字符



我们经常需要单独处理 `string` 对象中的字符，比如检查一个 `string` 对象是否包含空白，或者把 `string` 对象中的字母改成小写，再或者查看某个特定的字符是否出现等。

这类处理的一个关键问题是获取字符本身。有时需要处理 `string` 对象中的每一个字符，另外一些时候则只需处理某个特定的字符，还有些时候遇到某个条件处理就要停

91

下来。以往的经验告诉我们，处理这些情况常常要涉及语言和库的很多方面。

另一个关键问题是想知道能改变某个字符的特性。在 `cctype` 头文件中定义了一组标准库函数处理这部分工作，表 3.3 列出了主要的函数名及其含义。

表 3.3: `cctype` 头文件中的函数

<code>isalnum(c)</code>	当 c 是字母或数字时为真
<code>isalpha(c)</code>	当 c 是字母时为真
<code>iscntrl(c)</code>	当 c 是控制字符时为真
<code>isdigit(c)</code>	当 c 是数字时为真
<code>isgraph(c)</code>	当 c 不是空格但可打印时为真
<code>islower(c)</code>	当 c 是小写字母时为真
<code>isprint(c)</code>	当 c 是可打印字符时为真（即 c 是空格或 c 具有可视形式）
<code>ispunct(c)</code>	当 c 是标点符号时为真（即 c 不是控制字符、数字、字母、可打印空白中的一种）
<code>isspace(c)</code>	当 c 是空白时为真（即 c 是空格、横向制表符、纵向制表符、回车符、换行符、进纸符中的一种）
<code>isupper(c)</code>	当 c 是大写字母时为真
<code>isxdigit(c)</code>	当 c 是十六进制数字时为真
<code>tolower(c)</code>	如果 c 是大写字母，输出对应的小写字母；否则原样输出 c
<code>toupper(c)</code>	如果 c 是小写字母，输出对应的大写字母；否则原样输出 c

### 建议：使用 C++ 版本的 C 标准库头文件

C++ 标准库中除了定义 C++ 语言特有的功能外，也兼容了 C 语言的标准库。C 语言的头文件形如 `name.h`，C++ 则将这些文件命名为 `cname`。也就是去掉了 `.h` 后缀，而在文件名 `name` 之前添加了字母 `c`，这里的 `c` 表示这是一个属于 C 语言标准库的头文件。

因此，`cctype` 头文件和 `ctype.h` 头文件的内容是一样的，只不过从命名规范上来讲更符合 C++ 语言的要求。特别的，在名为 `cname` 的头文件中定义的名字从属于命名空间 `std`，而定义在名为 `.h` 的头文件中的则不然。

一般来说，C++ 程序应该使用名为 `cname` 的头文件而不使用 `name.h` 的形式，标准库中的名字总能在命名空间 `std` 中找到。如果使用 `.h` 形式的头文件，程序员就不得不时刻牢记哪些是从 C 语言那儿继承过来的，哪些又是 C++ 语言所独有的。

### 处理每个字符？使用基于范围的 for 语句

C++ 11 如果想对 `string` 对象中的每个字符做点儿什么操作，目前最好的办法是使用 C++11 新标准提供的一种语句：范围 `for`（range `for`）语句。这种语句遍历给定序列中的每个元素并对序列中的每个值执行某种操作，其语法形式是：

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

其中，`expression` 部分是一个对象，用于表示一个序列。`declaration` 部分负责定义一个变量，该变量将被用于访问序列中的基础元素。每次迭代，`declaration` 部分的变量会被初始化为 `expression` 部分的下一个元素值。

一个 `string` 对象表示一个字符的序列，因此 `string` 对象可以作为范围 `for` 语句

中的 *expression* 部分。举一个简单的例子，我们可以使用范围 for 语句把 string 对象中的字符每行一个输出出来：

```
string str("some string");
// 每行输出 str 中的一个字符。
for (auto c : str)           // 对于 str 中的每个字符
    cout << c << endl;       // 输出当前字符，后面紧跟一个换行符
```

for 循环把变量 c 和 str 联系了起来，其中我们定义循环控制变量的方式与定义任意一个普通变量是一样的。此例中，通过使用 auto 关键字（参见 2.5.2 节，第 61 页）让编译器来决定变量 c 的类型，这里 c 的类型是 char。每次迭代，str 的下一个字符被拷贝给 c，因此该循环可以读作“对于字符串 str 中的每个字符 c，”执行某某操作。此例中的“某某操作”即输出一个字符，然后换行。

&lt; 92

举个稍微复杂一点的例子，使用范围 for 语句和 ispunct 函数来统计 string 对象中标点符号的个数：

```
string s("Hello World!!!");
// punct_cnt 的类型和 s.size 的返回类型一样；参见 2.5.3 节（第 62 页）
decltype(s.size()) punct_cnt = 0;
// 统计 s 中标点符号的数量
for (auto c : s)           // 对于 s 中的每个字符
    if (ispunct(c))        // 如果该字符是标点符号
        ++punct_cnt;         // 将标点符号的计数值加 1
cout << punct_cnt
<< " punctuation characters in " << s << endl;
```

程序的输出结果将是：

```
3 punctuation characters in Hello World!!!
```

这里我们使用 decltype 关键字（参见 2.5.3 节，第 62 页）声明计数变量 punct\_cnt，它的类型是 s.size 函数返回值的类型，也就是 string::size\_type。使用范围 for 语句处理 string 对象中的每个字符并检查其是否是标点符号。如果是，使用递增运算符（参见 1.4.1 节，第 10 页）给计数变量加 1。最后，待范围 for 语句结束后输出统计结果。

&lt; 93

### 使用范围 for 语句改变字符串中的字符

如果想要改变 string 对象中字符的值，必须把循环变量定义成引用类型（参见 2.3.1 节，第 45 页）。记住，所谓引用只是给定对象的一个别名，因此当使用引用作为循环控制变量时，这个变量实际上被依次绑定到了序列的每个元素上。使用这个引用，我们就能改变它绑定的字符。

新的例子不再是统计标点符号的个数了，假设我们想要把字符串改写为大写字母的形式。为了做到这一点可以使用标准库函数 toupper，该函数接收一个字符，然后输出其对应的大写形式。这样，为了把整个 string 对象转换成大写，只要对其中的每个字符调用 toupper 函数并将结果再赋给原字符就可以了：

```
string s("Hello World!!!");
// 转换成大写形式。
for (auto &c : s)           // 对于 s 中的每个字符（注意：c 是引用）
    c = toupper(c);          // c 是一个引用，因此赋值语句将改变 s 中字符的值
cout << s << endl;
```

上述代码的输出结果将是：

```
HELLO WORLD!!!
```

每次迭代时，变量 c 引用 string 对象 s 的下一个字符，赋值给 c 也就是在改变 s 中对应字符的值。因此当执行下面的语句时，

```
c = toupper(c); // c 是一个引用，因此赋值语句将改变 s 中字符的值
```

实际上改变了 c 绑定的字符的值。整个循环结束后，str 中的所有字符都变成了大写形式。

### 只处理一部分字符？

如果要处理 string 对象中的每一个字符，使用范围 for 语句是个好主意。然而，有时我们需要访问的只是其中一个字符，或者访问多个字符但遇到某个条件就要停下来。例如，同样是将字符改为大写形式，不过新的要求不再是对整个字符串都这样做，而仅仅把 string 对象中的第一个字母或第一个单词大写化。

要想访问 string 对象中的单个字符有两种方式：一种是使用下标，另外一种是使用迭代器，其中关于迭代器的内容将在 3.4 节（第 95 页）和第 9 章中介绍。

下标运算符 ([ ]) 接收的输入参数是 string::size\_type 类型的值（参见 3.2.2 节，第 79 页），这个参数表示要访问的字符的位置；返回值是该位置上字符的引用。

string 对象的下标从 0 计起。如果 string 对象 s 至少包含两个字符，则 s[0] 是第 1 个字符、s[1] 是第 2 个字符、s[s.size()-1] 是最后一个字符。

string 对象的下标必须大于等于 0 而小于 s.size()。



使用超出此范围的下标将引发不可预知的结果，以此推断，使用下标访问空 string 也会引发不可预知的结果。

94

下标的值称作“下标”或“索引”，任何表达式只要它的值是一个整型值就能作为索引。不过，如果某个索引是带符号类型的值将自动转换成由 string::size\_type（参见 2.1.2 节，第 33 页）表达的无符号类型。

下面的程序使用下标运算符输出 string 对象中的第一个字符：

```
if (!s.empty())           // 确保确实有字符需要输出
    cout << s[0] << endl;   // 输出 s 的第一个字符
```

在访问指定字符之前，首先检查 s 是否为空。其实不管什么时候只要对 string 对象使用了下标，都要确认在那个位置上确实有值。如果 s 为空，则 s[0] 的结果将是未定义的。

只要字符串不是常量（参见 2.4 节，第 53 页），就能为下标运算符返回的字符赋新值。例如，下面的程序将字符串的首字符改成了大写形式：

```
string s("some string");
if (!s.empty())           // 确保 s[0] 的位置确实有字符
    s[0] = toupper(s[0]);  // 为 s 的第一个字符赋一个新值
```

程序的输出结果将是：

```
Some string
```

## 使用下标执行迭代

另一个例子是把 s 的第一个词改成大写形式：

```
// 依次处理 s 中的字符直至我们处理完全部字符或者遇到一个空白  
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
```

程序的输出结果将是：

**SOME string**

在上述程序中，`for` 循环使用变量 `index` 作为 `s` 的下标，`index` 的类型是由 `decltype` 关键字决定的。首先把 `index` 初始化为 0，这样第一次迭代就会从 `s` 的首字符开始；之后每次迭代将 `index` 加 1 以得到 `s` 的下一个字符。循环体负责将当前的字母改写为大写形式。

`for` 语句的条件部分涉及一点新知识，该条件使用了逻辑与运算符 (`&&`)。如果参与运算的两个运算对象都为真，则逻辑与结果为真；否则结果为假。对这个运算符来说最重要的一点是，C++语言规定只有当左侧运算对象为真时才会检查右侧运算对象的情况。如上例所示，这条规定确保了只有当下标取值在合理范围之内时才会真的用此下标去访问字符串。也就是说，只有在 `index` 达到 `s.size()` 之前才会执行 `s[index]`。随着 `index` 的增加，它永远也不可能超过 `s.size()` 的值，所以可以确保 `index` 比 `s.size()` 小。

### 提示：注意检查下标的合法性

95

使用下标时必须确保其在合理范围之内，也就是说，下标必须大于等于 0 而小于字符串的 `size()` 的值。一种简便易行的方法是，总是设下标的类型为 `string::size_type`，因为此类型是无符号数，可以确保下标不会小于 0。此时，代码只需保证下标小于 `size()` 的值就可以了。



C++标准并不要求标准库检测下标是否合法。一旦使用了一个超出范围的下标，就会产生不可预知的结果。

## 使用下标执行随机访问

在之前的示例中，我们让字符串的下标每次加 1 从而按顺序把所有字符改写成了大写形式。其实也能通过计算得到某个下标值，然后直接获取对应位置的字符，并不是每次都得从前往后依次访问。

例如，想要编写一个程序把 0 到 15 之间的十进制数转换成对应的十六进制形式，只需初始化一个字符串令其存放 16 个十六进制“数字”：

```
const string hexdigits = "0123456789ABCDEF"; // 可能的十六进制数字  
cout << "Enter a series of numbers between 0 and 15"  
     << " separated by spaces. Hit ENTER when finished: "  
     << endl;  
string result; // 用于保存十六进制的字符串  
string::size_type n; // 用于保存从输入流读取的数  
while (cin >> n)  
    if (n < hexdigits.size()) // 忽略无效输入  
        result += hexdigits[n]; // 得到对应的十六进制数字
```

```
cout << "Your hex number is: " << result << endl;
```

假设输入的内容如下：

```
12 0 5 15 8 15
```

程序的输出结果将是：

```
Your hex number is: C05F8F
```

上述程序的执行过程是这样的：首先初始化变量 `hexdigits` 令其存放从 0 到 F 的十六进制数字，注意我们把 `hexdigits` 声明成了常量（参见 2.4 节，第 53 页），这是因为在后面的程序中不打算再改变它的值。在循环内部使用输入值 `n` 作为 `hexdigits` 的下标，`hexdigits[n]` 的值就是 `hexdigits` 内位置 `n` 处的字符。例如，如果 `n` 是 15，则结果是 F；如果 `n` 是 12，则结果是 C，以此类推。把得到的十六进制数字添加到 `result` 内，最后一并输出。

无论何时用到字符串的下标，都应该注意检查其合法性。在上面的程序中，下标 `n` 是 `string::size_type` 类型，也就是无符号类型，所以 `n` 可以确保大于或等于 0。在实际使用时，还需检查 `n` 是否小于 `hexdigits` 的长度。

96

### 3.2.3 节练习

**练习 3.6：**编写一段程序，使用范围 `for` 语句将字符串内的所有字符用 X 替换。

**练习 3.7：**就上一题完成的程序而言，如果将循环控制变量的类型设为 `char` 将发生什么？先估计一下结果，然后实际编程进行验证。

**练习 3.8：**分别用 `while` 循环和传统的 `for` 循环重写第一题的程序，你觉得哪种形式更好呢？为什么？

**练习 3.9：**下面的程序有何作用？它合法吗？如果不合法，为什么？

```
string s;
cout << s[0] << endl;
```

**练习 3.10：**编写一段程序，读入一个包含标点符号的字符串，将标点符号去除后输出字符串剩余的部分。

**练习 3.11：**下面的范围 `for` 语句合法吗？如果合法，`c` 的类型是什么？

```
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
```



## 3.3 标准库类型 `vector`

标准库类型 `vector` 表示对象的集合，其中所有对象的类型都相同。集合中的每个对象都有一个与之对应的索引，索引用于访问对象。因为 `vector` “容纳着”其他对象，所以它也常被称作容器（container）。第 II 部将对容器进行更为详细的介绍。

要想使用 `vector`，必须包含适当的头文件。在后续的例子中，都将假定做了如下 `using` 声明：

```
#include <vector>
using std::vector;
```

C++语言既有类模板（class template），也有函数模板，其中 `vector` 是一个类模板。只有对 C++有了相当深入的理解才能写出模板，事实上，我们直到第 16 章才会学习如何自定义模板。幸运的是，即使还不会创建模板，我们也可以先试着用用它。

模板本身不是类或函数，相反可以将模板看作为编译器生成类或函数编写的一份说明。编译器根据模板创建类或函数的过程称为实例化（instantiation），当使用模板时，需要指出编译器应把类或函数实例化成何种类型。

对于类模板来说，我们通过提供一些额外信息来指定模板到底实例化成什么样的类，需要提供哪些信息由模板决定。提供信息的方式总是这样：即在模板名字后面跟一对尖括号，在括号内放上信息。

以 `vector` 为例，提供的额外信息是 `vector` 内所存放对象的类型：

```
vector<int> ivec;           // ivec 保存 int 类型的对象
vector<Sales_item> Sales_vec; // 保存 Sales_item 类型的对象
vector<vector<string>> file; // 该向量的元素是 vector 对象
```

97

在上面的例子中，编译器根据模板 `vector` 生成了三种不同的类型：`vector<int>`、`vector<Sales_item>` 和 `vector<vector<string>>`。



`vector` 是模板而非类型，由 `vector` 生成的类型必须包含 `vector` 中元素的类型，例如 `vector<int>`。

C++  
11

`vector` 能容纳绝大多数类型的对象作为其元素，但是因为引用不是对象（参见 2.3.1 节，第 45 页），所以不存在包含引用的 `vector`。除此之外，其他大多数（非引用）内置类型和类类型都可以构成 `vector` 对象，甚至组成 `vector` 的元素也可以是 `vector`。

需要指出的是，在早期版本的 C++ 标准中如果 `vector` 的元素还是 `vector`（或者其他模板类型），则其定义的形式与现在的 C++11 新标准略有不同。过去，必须在外层 `vector` 对象的右尖括号和其元素类型之间添加一个空格，如应该写成 `vector<vector<int>>` 而非 `vector<vector<int>>`。



某些编译器可能仍需以老式的声明语句来处理元素为 `vector` 的 `vector` 对象，如 `vector<vector<int> >`。

### 3.3.1 定义和初始化 vector 对象



和任何一种类类型一样，`vector` 模板控制着定义和初始化向量的方法。表 3.4 列出了定义 `vector` 对象的常用方法。

表 3.4：初始化 `vector` 对象的方法

<code>vector&lt;T&gt; v1</code>	<code>v1</code> 是一个空 <code>vector</code> ，它潜在的元素是 <code>T</code> 类型的，执行默认初始化
<code>vector&lt;T&gt; v2(v1)</code>	<code>v2</code> 中包含有 <code>v1</code> 所有元素的副本
<code>vector&lt;T&gt; v2 = v1</code>	等价于 <code>v2(v1)</code> ， <code>v2</code> 中包含有 <code>v1</code> 所有元素的副本
<code>vector&lt;T&gt; v3(n, val)</code>	<code>v3</code> 包含了 <code>n</code> 个重复的元素，每个元素的值都是 <code>val</code>
<code>vector&lt;T&gt; v4(n)</code>	<code>v4</code> 包含了 <code>n</code> 个重复地执行了值初始化的对象
<code>vector&lt;T&gt; v5{a, b, c...}</code>	<code>v5</code> 包含了初始值个数的元素，每个元素被赋予相应的初始值
<code>vector&lt;T&gt; v5={a, b, c...}</code>	等价于 <code>v5(a, b, c...)</code>

可以默认初始化 `vector` 对象（参见 2.2.1 节，第 40 页），从而创建一个指定类型的空 `vector`：

```
vector<string> svec; // 默认初始化，svec 不含任何元素
```

看起来空 `vector` 好像没什么用，但是很快我们就会知道程序在运行时可以很高效地往 `vector` 对象中添加元素。事实上，最常见的方式就是先定义一个空 `vector`，然后当运行时获取到元素的值后再逐一添加。

当然也可以在定义 `vector` 对象时指定元素的初始值。例如，允许把一个 `vector` 对象的元素拷贝给另外一个 `vector` 对象。此时，新 `vector` 对象的元素就是原 `vector` 对象对应元素的副本。注意两个 `vector` 对象的类型必须相同：

```
vector<int> ivec;           // 初始状态为空
// 在此处给 ivec 添加一些值
vector<int> ivec2(ivec);    // 把 ivec 的元素拷贝给 ivec2
vector<int> ivec3 = ivec;    // 把 ivec 的元素拷贝给 ivec3
vector<string> svec(ivec2); // 错误：svec 的元素是 string 对象，不是 int
```

## 98 ◀ 列表初始化 `vector` 对象

C++  
11

C++11 新标准还提供了另外一种为 `vector` 对象的元素赋初值的方法，即列表初始化（参见 2.2.1 节，第 39 页）。此时，用花括号括起来的 0 个或多个初始元素值被赋给 `vector` 对象：

```
vector<string> articles = {"a", "an", "the"};
```

上述 `vector` 对象包含三个元素：第一个是字符串“a”，第二个是字符串“an”，最后一个是字符串“the”。

之前已经讲过，C++语言提供了几种不同的初始化方式（参见 2.2.1 节，第 39 页）。在大多数情况下这些初始化方式可以相互等价地使用，不过也并非一直如此。目前已经介绍过的两种例外情况是：其一，使用拷贝初始化时（即使用=时）（参见 3.2.1 节，第 76 页），只能提供一个初始值；其二，如果提供的是一个类内初始值（参见 2.6.1 节，第 64 页），则只能使用拷贝初始化或使用花括号的形式初始化。第三种特殊的要求是，如果提供的是初始元素值的列表，则只能把初始值都放在花括号里进行列表初始化，而不能放在圆括号里：

```
vector<string> v1{"a", "an", "the"}; // 列表初始化
vector<string> v2("a", "an", "the"); // 错误
```

## 创建指定数量的元素

还可以用 `vector` 对象容纳的元素数量和所有元素的统一初始值来初始化 `vector` 对象：

```
vector<int> ivec(10, -1); // 10 个 int 类型的元素，每个都被初始化为 -1
vector<string> svec(10, "hi!"); // 10 个 string 类型的元素，
                                // 每个都被初始化为 "hi!"
```

## 值初始化

通常情况下，可以只提供 `vector` 对象容纳的元素数量而不用略去初始值。此时库会创建一个值初始化的（value-initialized）元素初值，并把它赋给容器中的所有元素。这个初值由 `vector` 对象中元素的类型决定。

如果 `vector` 对象的元素是内置类型，比如 `int`，则元素初始值自动设为 0。如果元素是某种类类型，比如 `string`，则元素由类默认初始化：

```
vector<int> ivec(10);           // 10 个元素，每个都初始化为 0
vector<string> svec(10);        // 10 个元素，每个都是空 string 对象
```

对这种初始化的方式有两个特殊限制：其一，有些类要求必须明确地提供初始值（参见 2.2.1 节，第 40 页），如果 `vector` 对象中元素的类型不支持默认初始化，我们就必须提供初始的元素值。对这种类型的对象来说，只提供元素的数量而不设定初始值无法完成初始化工作。

其二，如果只提供了元素的数量而没有设定初始值，只能使用直接初始化：

```
vector<int> vi = 10; // 错误：必须使用直接初始化的形式指定向量大小
```

99

这里的 10 是用来说明如何初始化 `vector` 对象的，我们用它的本意是想创建含有 10 个值初始化了的元素的 `vector` 对象，而非把数字 10 “拷贝”到 `vector` 中。因此，此时不宜使用拷贝初始化，7.5.4 节（第 265 页）将对这一点做更详细的介绍。

### 列表初始值还是元素数量？



在某些情况下，初始化的真实含义依赖于传递初始值时用的是花括号还是圆括号。例如，用一个整数来初始化 `vector<int>` 时，整数的含义可能是 `vector` 对象的容量也可能元素的值。类似的，用两个整数来初始化 `vector<int>` 时，这两个整数可能一个是 `vector` 对象的容量，另一个是元素的初值，也可能它们是容量为 2 的 `vector` 对象中两个元素的初值。通过使用花括号或圆括号可以区分上述这些含义：

```
vector<int> v1(10);           // v1 有 10 个元素，每个的值都是 0
vector<int> v2{10};            // v2 有 1 个元素，该元素的值是 10

vector<int> v3(10, 1);         // v3 有 10 个元素，每个的值都是 1
vector<int> v4{10, 1};          // v4 有 2 个元素，值分别是 10 和 1
```

如果用的是圆括号，可以说提供的值是用来构造（construct）`vector` 对象的。例如，`v1` 的初始值说明了 `vector` 对象的容量；`v3` 的两个初始值则分别说明了 `vector` 对象的容量和元素的初值。

如果用的是花括号，可以表述成我们想列表初始化（list initialize）该 `vector` 对象。也就是说，初始化过程会尽可能地把花括号内的值当成是元素初始值的列表来处理，只有在无法执行列表初始化时才会考虑其他初始化方式。在上例中，给 `v2` 和 `v4` 提供的初始值都能作为元素的值，所以它们都会执行列表初始化，`vector` 对象 `v2` 包含一个元素而 `vector` 对象 `v4` 包含两个元素。

另一方面，如果初始化时使用了花括号的形式但是提供的值又不能用来列表初始化，就要考虑用这样的值来构造 `vector` 对象了。例如，要想列表初始化一个含有 `string` 对象的 `vector` 对象，应该提供能赋给 `string` 对象的初值。此时不难区分到底是要列表初始化 `vector` 对象的元素还是用给定的容量值来构造 `vector` 对象：

```
vector<string> v5{"hi"}; // 列表初始化：v5 有一个元素
vector<string> v6("hi"); // 错误：不能使用字符串字面值构建 vector 对象
vector<string> v7{10};      // v7 有 10 个默认初始化的元素
vector<string> v8{10, "hi"}; // v8 有 10 个值为"hi"的元素
```

100

尽管在上面的例子中除了第二条语句之外都用了花括号，但其实只有 `v5` 是列表初始化。要想列表初始化 `vector` 对象，花括号里的值必须与元素类型相同。显然不能用 `int` 初始化 `string` 对象，所以 `v7` 和 `v8` 提供的值不能作为元素的初始值。确认无法执行列表初始化后，编译器会尝试用默认值初始化 `vector` 对象。

### 3.3.1 节练习

**练习 3.12:** 下列 `vector` 对象的定义有不正确的吗？如果有，请指出来。对于正确的，描述其执行结果；对于不正确的，说明其错误的原因。

- (a) `vector<vector<int>> ivect;`
- (b) `vector<string> svec = ivect;`
- (c) `vector<string> svec(10, "null");`

**练习 3.13:** 下列的 `vector` 对象各包含多少个元素？这些元素的值分别是多少？

- |   |   |
|---|---|
| (a) <code>vector&lt;int&gt; v1;</code>              | (b) <code>vector&lt;int&gt; v2(10);</code>    |
| (c) <code>vector&lt;int&gt; v3(10, 42);</code>      | (d) <code>vector&lt;int&gt; v4{10};</code>    |
| (e) <code>vector&lt;int&gt; v5{10, 42};</code>      | (f) <code>vector&lt;string&gt; v6{10};</code> |
| (g) <code>vector&lt;string&gt; v7{10, "hi"};</code> |   |



### 3.3.2 向 `vector` 对象中添加元素

对 `vector` 对象来说，直接初始化的方式适用于三种情况：初始值已知且数量较少、初始值是另一个 `vector` 对象的副本、所有元素的初始值都一样。然而更常见的情况是：创建一个 `vector` 对象时并不清楚实际所需的元素个数，元素的值也经常无法确定。还有些时候即使元素的初值已知，但如果这些值总量较大而各不相同，那么在创建 `vector` 对象的时候执行初始化操作也会显得过于烦琐。

举个例子，如果想创建一个 `vector` 对象令其包含从 0 到 9 共 10 个元素，使用列表初始化的方法很容易做到这一点；但如果 `vector` 对象包含的元素是从 0 到 99 或者从 0 到 999 呢？这时通过列表初始化把所有元素都一一罗列出来就不太合适了。对此例来说，更好的处理方法是先创建一个空 `vector`，然后在运行时再利用 `vector` 的成员函数 `push_back` 向其中添加元素。`push_back` 负责把一个值当成 `vector` 对象的尾元素“压到（push）”`vector` 对象的“尾端（back）”。例如：

```
101> vector<int> v2;           // 空 vector 对象
    for (int i = 0; i != 100; ++i)
        v2.push_back(i); // 依次把整数值放到 v2 尾端
    // 循环结束后 v2 有 100 个元素，值从 0 到 99
```

在上例中，尽管知道 `vector` 对象最后会包含 100 个元素，但在一开始还是把它声明成空 `vector`，在每次迭代时才顺序地把下一个整数作为 `v2` 的新元素添加给它。

同样的，如果直到运行时才能知道 `vector` 对象中元素的确切个数，也应该使用刚刚这种方法创建 `vector` 对象并为其赋值。例如，有时需要实时读入数据然后将其赋予 `vector` 对象：

```
// 从标准输入中读取单词，将其作为 vector 对象的元素存储
string word;
vector<string> text;           // 空 vector 对象
while (cin >> word) {
    text.push_back(word);     // 把 word 添加到 text 后面
}
```

和之前的例子一样，本例也是先创建一个空 `vector`，之后依次读入未知数量的值并保存到 `text` 中。

### 关键概念：vector 对象能高效增长

C++标准要求 `vector` 应该能在运行时高效快速地添加元素。因此既然 `vector` 对象能高效地增长，那么在定义 `vector` 对象的时候设定其大小也就没什么必要了，事实上如果这么做性能可能更差。只有一种例外情况，就是所有 (all) 元素的值都一样。一旦元素的值有所不同，更有效的办法是先定义一个空的 `vector` 对象，再在运行时向其中添加具体值。此外，9.4 节（第 317 页）将介绍，`vector` 还提供了方法，允许我们进一步提升动态添加元素的性能。

开始的时候创建空的 `vector` 对象，在运行时再动态添加元素，这一做法与 C 语言及其他大多数语言中内置数组类型的用法不同。特别是如果用惯了 C 或者 Java，可以预计在创建 `vector` 对象时顺便指定其容量是最好的。然而事实上，通常的情况是恰恰相反。

### 向 `vector` 对象添加元素蕴含的编程假定

由于能高效便捷地向 `vector` 对象中添加元素，很多编程工作被极大简化了。然而，这种简便性也伴随着一些对编写程序更高的要求：其中一条就是必须要确保所写的循环正确无误，特别是在循环有可能改变 `vector` 对象容量的时候。

随着对 `vector` 的更多使用，我们还会逐渐了解到其他一些隐含的要求，其中一条是现在就要指出的：如果循环体内部包含有向 `vector` 对象添加元素的语句，则不能使用范围 `for` 循环，具体原因将在 5.4.3 节（第 168 页）详细解释。



范围 `for` 语句体内不应改变其所遍历序列的大小。

### 3.3.2 节练习

102

**练习 3.14：**编写一段程序，用 `cin` 读入一组整数并把它们存入一个 `vector` 对象。

**练习 3.15：**改写上题的程序，不过这次读入的是字符串。

### 3.3.3 其他 `vector` 操作



除了 `push_back` 之外，`vector` 还提供了几种其他操作，大多数都和 `string` 的相关操作类似，表 3.5 列出了其中比较重要的一些。

表 3.5: `vector` 支持的操作

<code>v.empty()</code>	如果 <code>v</code> 不含有任何元素，返回真；否则返回假
<code>v.size()</code>	返回 <code>v</code> 中元素的个数
<code>v.push_back(t)</code>	向 <code>v</code> 的尾端添加一个值为 <code>t</code> 的元素
<code>v[n]</code>	返回 <code>v</code> 中第 <code>n</code> 个位置上元素的引用
<code>v1 = v2</code>	用 <code>v2</code> 中元素的拷贝替换 <code>v1</code> 中的元素
<code>v1 = {a, b, c...}</code>	用列表中元素的拷贝替换 <code>v1</code> 中的元素
<code>v1 == v2</code>	<code>v1</code> 和 <code>v2</code> 相等当且仅当它们的元素数量相同且对应位置的元素值都相同
<code>v1 != v2</code>	
<code>&lt;, &lt;=, &gt;, &gt;=</code>	顾名思义，以字典顺序进行比较

访问 `vector` 对象中元素的方法和访问 `string` 对象中字符的方法差不多，也是通过元素在 `vector` 对象中的位置。例如，可以使用范围 `for` 语句处理 `vector` 对象中的所有元素：

```
vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto &i : v)           // 对于 v 中的每个元素（注意：i 是一个引用）
    i *= i;                 // 求元素值的平方
for (auto i : v)            // 对于 v 中的每个元素
    cout << i << " ";      // 输出该元素
cout << endl;
```

第一个循环把控制变量 `i` 定义成引用类型，这样就能通过 `i` 给 `v` 的元素赋值，其中 `i` 的类型由 `auto` 关键字指定。这里用到了一种新的复合赋值运算符（参见 1.4.1 节，第 10 页）。如我们所知，`+=` 把左侧运算对象和右侧运算对象相加，结果存入左侧运算对象；类似的，`*=` 把左侧运算对象和右侧运算对象相乘，结果存入左侧运算对象。最后，第二个循环输出所有元素。

`vector` 的 `empty` 和 `size` 两个成员与 `string` 的同名成员（参见 3.2.2 节，第 78 页）功能完全一致：`empty` 检查 `vector` 对象是否包含元素然后返回一个布尔值；`size` 则返回 `vector` 对象中元素的个数，返回值的类型是由 `vector` 定义的 `size_type` 类型。



要使用 `size_type`，需首先指定它是由哪种类型定义的。`vector` 对象的类型总是包含着元素的类型（参见 3.3 节，第 87 页）：

<code>vector&lt;int&gt;::size_type</code>	<code>// 正确</code>
<code>vector::size_type</code>	<code>// 错误</code>

各个相等性运算符和关系运算符也与 `string` 的相应运算符（参见 3.2.2 节，第 79 页）功能一致。两个 `vector` 对象相等当且仅当它们所含的元素个数相同，而且对应位置的元素值也相同。关系运算符依照字典顺序进行比较：如果两个 `vector` 对象的容量不同，但是在相同位置上的元素值都一样，则元素较少的 `vector` 对象小于元素较多的 `vector` 对象；若元素的值有区别，则 `vector` 对象的大小关系由第一对相异的元素值的大小关系决定。

103

只有当元素的值可比较时，`vector` 对象才能被比较。一些类，如 `string` 等，确实定义了自己的相等性运算符和关系运算符；另外一些，如 `Sales_item` 类支持的运算已经全都罗列在 1.5.1 节（第 17 页）中了，显然并不支持相等性判断和关系运算等操作。因此，不能比较两个 `vector<Sales_item>` 对象。

### 计算 `vector` 内对象的索引

使用下标运算符（参见 3.2.3 节，第 84 页）能获取到指定的元素。和 `string` 一样，`vector` 对象的下标也是从 0 开始计起，下标的类型是相应的 `size_type` 类型。只要 `vector` 对象不是一个常量，就能向下标运算符返回的元素赋值。此外，如 3.2.3 节（第 85 页）所述的那样，也能通过计算得到 `vector` 内对象的索引，然后直接获取索引位置上的元素。

举个例子，假设有一组成绩的集合，其中成绩的取值是从 0 到 100。以 10 分为一个分数段，要求统计各个分数段各有多少个成绩。显然，从 0 到 100 总共有 101 种可能的成绩取值，这些成绩分布在 11 个分数段上：每 10 个分数构成一个分数段，这样的分数段有 10 个，额外还有一个分数段表示满分 100 分。这样第一个分数段将统计成绩在 0 到 9 之间的数量；第二个分数段将统计成绩在 10 到 19 之间的数量，以此类推。最后一个分数段统计满分 100 分的数量。

按照上面的描述，如果输入的成绩如下：

```
42 65 95 100 39 67 95 76 88 76 83 92 76 93
```

则输出的结果应该是：

```
0 0 0 1 1 0 2 3 2 4 1
```

结果显示：成绩在 30 分以下的没有、30 分至 39 分有 1 个、40 分至 49 分有 1 个、50 分至 59 分没有、60 分至 69 分有 2 个、70 分至 79 分有 3 个、80 分至 89 分有 2 个、90 分至 99 分有 4 个，还有 1 个是满分。

在具体实现时使用一个含有 11 个元素的 `vector` 对象，每个元素分别用于统计各个分数段上出现的成绩个数。对于某个成绩来说，将其除以 10 就能得到对应的分数段索引。  
注意：两个整数相除，结果还是整数，余数部分被自动忽略掉了。例如， $42/10=4$ 、 $65/10=6$ 、 $100/10=10$  等。一旦计算得到了分数段索引，就能用它作为 `vector` 对象的下标，进而获取该分数段的计数值并加 1；

```
// 以 10 分为一个分数段统计成绩的数量：0~9, 10~19, ..., 90~99, 100
vector<unsigned> scores(11, 0); // 11 个分数段，全都初始化为 0
unsigned grade;
while (cin >> grade) { // 读取成绩
    if (grade <= 100) // 只处理有效的成绩
        ++scores[grade/10]; // 将对应分数段的计数值加 1
}
```

在上面的程序中，首先定义了一个 `vector` 对象存放各个分数段上成绩的数量。此例中，由于初始状态下每个元素的值都相同，所以我们为 `vector` 对象申请了 11 个元素，并把所有元素的初始值都设为 0。`while` 语句的条件部分负责读入成绩，在循环体内部首先检查读入的成绩是否合法（即是否小于等于 100 分），如果合法，将成绩对应的分数段的计数值加 1。

执行计数值累加的那条语句很好地体现了 C++ 程序代码的简洁性。表达式

```
++scores[grade/10]; // 将当前分数段的计数值加 1
```

等价于

```
auto ind = grade/10; // 得到分数段索引
scores[ind] = scores[ind] + 1; // 将计数值加 1
```

上述语句的含义是：用 `grade` 除以 10 来计算成绩所在的分数段，然后将所得的结果作为变量 `scores` 的下标。通过运行下标运算获取该分数段对应的计数值，因为新出现了一个属于该分数段的成绩，所以将计数值加 1。

如前所述，使用下标的时候必须清楚地知道它是否在合理范围之内（参见 3.2.3 节，第 85 页）。在这个程序里，我们事先确认了输入的成绩确实在 0 到 100 之间，这样计算所得的下标就一定在 0 到 10 之间，属于 0 到 `scores.size()-1` 规定的有效范围，一定是合法的。

### 不能用下标形式添加元素

刚接触 C++ 语言的程序员也许会认为可以通过 `vector` 对象的下标形式来添加元素，事实并非如此。下面的代码试图为 `vector` 对象 `ivec` 添加 10 个元素：

```
vector<int> ivec; // 空 vector 对象
```

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec[ix] = ix; // 严重错误：ivec 不包含任何元素
```

然而，这段代码是错误的：ivec 是一个空 vector，根本不包含任何元素，当然也就不能通过下标去访问任何元素！如前所述，正确的方法是使用 push\_back：

105 &gt;

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // 正确：添加一个新元素，该元素的值是 ix
```



WARNING

vector 对象（以及 string 对象）的下标运算符可用于访问已存在的元素，而不能用于添加元素。

### 提示：只能对可知已存在的元素执行下标操作！

关于下标必须明确的一点是：只能对可知已存在的元素执行下标操作。例如，

```
vector<int> ivec;           // 空 vector 对象
cout << ivec[0];           // 错误：ivec 不包含任何元素

vector<int> ivec2(10);     // 含有 10 个元素的 vector 对象
cout << ivec2[10];         // 错误：ivec2 元素的合法索引是从 0 到 9
```

试图用下标的形式去访问一个不存在的元素将引发错误，不过这种错误不会被编译器发现，而是在运行时产生一个不可预知的值。

不幸的是，这种通过下标访问不存在的元素的行为非常常见，而且会产生很严重的后果。所谓的缓冲区溢出（buffer overflow）指的就是这类错误，这也是导致 PC 及其他设备上应用程序出现安全问题的一个重要原因。



Tip

确保下标合法的一种有效手段就是尽可能使用范围 for 语句。

### 3.3.3 节练习

**练习 3.16：**编写一段程序，把练习 3.13 中 vector 对象的容量和具体内容输出出来。检验你之前的回答是否正确，如果不对，回过头重新学习 3.3.1 节（第 87 页）直到弄明白错在何处为止。

**练习 3.17：**从 cin 读入一组词并把它们存入一个 vector 对象，然后设法把所有词都改写为大写形式。输出改变后的结果，每个词占一行。

**练习 3.18：**下面的程序合法吗？如果不合法，你准备如何修改？

```
vector<int> ivec;
ivec[0] = 42;
```

**练习 3.19：**如果想定义一个含有 10 个元素的 vector 对象，所有元素的值都是 42，请列举出三种不同的实现方法。哪种方法更好呢？为什么？

**练习 3.20：**读入一组整数并把它们存入一个 vector 对象，将每对相邻整数的和输出出来。改写你的程序，这次要求先输出第 1 个和最后 1 个元素的和，接着输出第 2 个和倒数第 2 个元素的和，以此类推。

## 3.4 迭代器介绍



&lt; 106

我们已经知道可以使用下标运算符来访问 `string` 对象的字符或 `vector` 对象的元素，还有另外一种更通用的机制也可以实现同样的目的，这就是 **迭代器**（iterator）。在第 II 部分中将要介绍，除了 `vector` 之外，标准库还定义了其他几种容器。所有标准库容器都可以使用迭代器，但是其中只有少数几种才同时支持下标运算符。严格来说，`string` 对象不属于容器类型，但是 `string` 支持很多与容器类型类似的操作。`vector` 支持下标运算符，这点和 `string` 一样；`string` 支持迭代器，这也和 `vector` 是一样的。

类似于指针类型（参见 2.3.2 节，第 47 页），迭代器也提供了对对象的间接访问。就迭代器而言，其对象是容器中的元素或者 `string` 对象中的字符。使用迭代器可以访问某个元素，迭代器也能从一个元素移动到另外一个元素。迭代器有有效和无效之分，这一点和指针差不多。有效的迭代器或者指向某个元素，或者指向容器中尾元素的下一位置；其他所有情况都属于无效。

### 3.4.1 使用迭代器



和指针不一样的是，获取迭代器不是使用取地址符，有迭代器的类型同时拥有返回迭代器的成员。比如，这些类型都拥有名为 `begin` 和 `end` 的成员，其中 `begin` 成员负责返回指向第一个元素（或第一个字符）的迭代器。如有下述语句：

```
// 由编译器决定 b 和 e 的类型；参见 2.5.2 节（第 61 页）
// b 表示 v 的第一个元素，e 表示 v 尾元素的下一位置
auto b = v.begin(), e = v.end(); // b 和 e 的类型相同
```

`end` 成员则负责返回指向容器（或 `string` 对象）“尾元素的下一位置（one past the end）”的迭代器，也就是说，该迭代器指示的是容器的一个本不存在的“尾后（off the end）”元素。这样的迭代器没什么实际含义，仅是个标记而已，表示我们已经处理完了容器中的所有元素。`end` 成员返回的迭代器常被称作 **尾后迭代器**（off-the-end iterator）或者简称为尾迭代器（end iterator）。特殊情况下如果容器为空，则 `begin` 和 `end` 返回的是同一个迭代器。

**Note**

如果容器为空，则 `begin` 和 `end` 返回的是同一个迭代器，都是尾后迭代器。

一般来说，我们不清楚（不在意）迭代器准确的类型到底是什么。在上面的例子中，使用 `auto` 关键字定义变量 `b` 和 `e`（参见 2.5.2 节，第 61 页），这两个变量的类型也就是 `begin` 和 `end` 的返回值类型，第 97 页将对相关内容做更详细的介绍。

#### 迭代器运算符

表 3.6 列举了迭代器支持的一些运算。使用 `==` 和 `!=` 来比较两个合法的迭代器是否相等，如果两个迭代器指向的元素相同或者都是同一个容器的尾后迭代器，则它们相等；否则就说这两个迭代器不相等。

表 3.6: 标准容器迭代器的运算符

<code>*iter</code>	返回迭代器 <code>iter</code> 所指元素的引用
<code>iter-&gt;mem</code>	解引用 <code>iter</code> 并获取该元素的名为 <code>mem</code> 的成员，等价于 <code>(*iter).mem</code>
<code>++iter</code>	令 <code>iter</code> 指示容器中的下一个元素
<code>--iter</code>	令 <code>iter</code> 指示容器中的上一个元素
<code>iter1 == iter2</code>	判断两个迭代器是否相等（不相等），如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器，则相等；反之，不相等
<code>iter1 != iter2</code>	

107 和指针类似，也能通过解引用迭代器来获取它所指示的元素，执行解引用的迭代器必须合法并确实指示着某个元素（参见 2.3.2 节，第 48 页）。试图解引用一个非法迭代器或者尾后迭代器都是未被定义的行为。

举个例子，3.2.3 节（第 84 页）中的程序利用下标运算符把 `string` 对象的第一个字母改为了大写形式，下面利用迭代器实现同样的功能：

```
string s("some string");
if (s.begin() != s.end()) { // 确保 s 非空
    auto it = s.begin(); // it 表示 s 的第一个字符
    *it = toupper(*it); // 将当前字符改成大写形式
}
```

本例和原来的程序一样，首先检查 `s` 是否为空，显然通过检查 `begin` 和 `end` 返回的结果是否一致就能做到这一点。如果返回的结果一样，说明 `s` 为空；如果返回的结果不一样，说明 `s` 不为空，此时 `s` 中至少包含一个字符。

我们在 `if` 内部，声明了一个迭代器变量 `it` 并把 `begin` 返回的结果赋给它，这样就得到了指示 `s` 中第一个字符的迭代器，接下来通过解引用运算符将第一个字符更改为大写形式。和原来的程序一样，输出结果将是：

**Some string**

### 将迭代器从一个元素移动到另外一个元素

迭代器使用递增 (`++`) 运算符（参见 1.4.1 节，第 11 页）来从一个元素移动到下一个元素。从逻辑上来说，迭代器的递增和整数的递增类似，整数的递增是在整数值上“加 1”，迭代器的递增则是将迭代器“向前移动一个位置”。



因为 `end` 返回的迭代器并不实际指示某个元素，所以不能对其进行递增或解引用的操作。

之前有一个程序把 `string` 对象中第一个单词改写为大写形式，现在利用迭代器及其递增运算符可以实现相同的功能：

108 // 依次处理 s 的字符直至我们处理完全部字符或者遇到空白  
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

和 3.2.3 节（第 84 页）的那个程序一样，上面的循环也是遍历 `s` 的字符直到遇到空白字符为止，只不过之前的程序用的是下标运算符，现在这个程序用的是迭代器。

循环首先用 `s.begin` 的返回值来初始化 `it`，意味着 `it` 指示的是 `s` 中的第一个字符（如果有的话）。条件部分检查是否已到达 `s` 的尾部，如果尚未到达，则将 `it` 解引用的结

果传入 `isspace` 函数检查是否遇到了空白。每次迭代的最后，执行 `++it` 令迭代器前移一个位置以访问 `s` 的下一个字符。

循环体内部和上一个程序 `if` 语句内的最后一句话一样，先解引用 `it`，然后将结果传入 `toupper` 函数得到该字母对应的大写形式，再把这个大写字母重新赋值给 `it` 所指示的字符。

### 关键概念：泛型编程

原来使用 C 或 Java 的程序员在转而使用 C++ 语言之后，会对 `for` 循环中使用 `!=` 而非 `<` 进行判断有点儿奇怪，比如上面的这个程序以及 85 页的那个。C++ 程序员习惯性地使用 `!=`，其原因和他们更愿意使用迭代器而非下标的原因一样：因为这种编程风格在标准库提供的所有容器上都有效。

之前已经说过，只有 `string` 和 `vector` 等一些标准库类型有下标运算符，而并非全都如此。与之类似，所有标准库容器的迭代器都定义了 `==` 和 `!=`，但是它们中的大多数都没有定义 `<` 运算符。因此，只要我们养成使用迭代器和 `!=` 的习惯，就不用太在意用的到底是哪种容器类型。

## 迭代器类型

就像不知道 `string` 和 `vector` 的 `size_type` 成员（参见 3.2.2 节，第 79 页）到底是什么类型一样，一般来说我们也不知道（其实是无须知道）迭代器的精确类型。而实际上，那些拥有迭代器的标准库类型使用 `iterator` 和 `const_iterator` 来表示迭代器的类型：

```
vector<int>::iterator it;      // it 能读写 vector<int> 的元素  
string::iterator it2;         // it2 能读写 string 对象中的字符  
  
vector<int>::const_iterator it3; // it3 只能读元素，不能写元素  
string::const_iterator it4;    // it4 只能读字符，不能写字符  
  
const_iterator 和常量指针（参见 2.4.2 节，第 56 页）差不多，能读取但不能修改它所指的元素值。相反，iterator 的对象可读可写。如果 vector 对象或 string 对象是一个常量，只能使用 const_iterator；如果 vector 对象或 string 对象不是常量，那么既能使用 iterator 也能使用 const_iterator。
```

### 术语：迭代器和迭代器类型

109

迭代器这个名词有三种不同的含义：可能是迭代器概念本身，也可能是指容器定义的迭代器类型，还可能是指某个迭代器对象。

重点是理解存在一组概念上相关的类型，我们认定某个类型是迭代器当且仅当它支持一套操作，这套操作使得我们能访问容器的元素或者从某个元素移动到另外一个元素。

每个容器类定义了一个名为 `iterator` 的类型，该类型支持迭代器概念所规定的一套操作。

## begin 和 end 运算符

`begin` 和 `end` 返回的具体类型由对象是否是常量决定，如果对象是常量，`begin` 和 `end` 返回 `const_iterator`；如果对象不是常量，返回 `iterator`：

```
vector<int> v;
```

```
const vector<int> cv;
auto it1 = v.begin();      // it1 的类型是 vector<int>::iterator
auto it2 = cv.begin();    // it2 的类型是 vector<int>::const_iterator
```

有时候这种默认的行为并非我们所要。在 6.2.3 节（第 191 页）中将会看到，如果对象只需读操作而无须写操作的话最好使用常量类型（比如 `const_iterator`）。为了便于专门得到 `const_iterator` 类型的返回值，C++11 新标准引入了两个新函数，分别是 `cbegin` 和 `cend`：

```
auto it3 = v.cbegin(); // it3 的类型是 vector<int>::const_iterator
```

类似于 `begin` 和 `end`，上述两个新函数也分别返回指示容器第一个元素或最后元素下一位置的迭代器。有所不同的是，不论 `vector` 对象（或 `string` 对象）本身是否是常量，返回值都是 `const_iterator`。

### 结合解引用和成员访问操作

解引用迭代器可获得迭代器所指的对象，如果该对象的类型恰好是类，就有可能希望进一步访问它的成员。例如，对于一个由字符串组成的 `vector` 对象来说，要想检查其元素是否为空，令 `it` 是该 `vector` 对象的迭代器，只需检查 `it` 所指字符串是否为空就可以了，其代码如下所示：

```
(*it).empty()
```

注意，`(*it).empty()` 中的圆括号必不可少，具体原因将在 4.1.2 节（第 121 页）介绍，该表达式的含义是先对 `it` 解引用，然后解引用的结果再执行点运算符（参见 1.5.2 节，第 20 页）。如果不加圆括号，点运算符将由 `it` 来执行，而非 `it` 解引用的结果：

```
(*it).empty()    // 解引用 it，然后调用结果对象的 empty 成员
*it.empty()       // 错误：试图访问 it 的名为 empty 的成员，但 it 是个迭代器，
                  // 没有 empty 成员
```

**110** 上面第二个表达式的含义是从名为 `it` 的对象中寻找其 `empty` 成员，显然 `it` 是一个迭代器，它没有哪个成员是叫 `empty` 的，所以第二个表达式将发生错误。

为了简化上述表达式，C++语言定义了箭头运算符（`->`）。箭头运算符把解引用和成员访问两个操作结合在一起，也就是说，`it->mem` 和 `(*it).mem` 表达的意思相同。

例如，假设用一个名为 `text` 的字符串向量存放文本文件中的数据，其中的元素或者是一句话或者是一个用于表示段落分隔的空字符串。如果要输出 `text` 中第一段的内容，可以利用迭代器写一个循环令其遍历 `text`，直到遇到空字符串的元素为止：

```
// 依次输出 text 的每一行直至遇到第一个空白行为止
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

我们首先初始化 `it` 令其指向 `text` 的第一个元素，循环重复执行直至处理完了 `text` 的所有元素或者发现某个元素为空。每次迭代时只要发现还有元素并且尚未遇到空元素，就输出当前正在处理的元素。值得注意的是，因为循环从头到尾只是读取 `text` 的元素而未向其中写值，所以使用了 `cbegin` 和 `cend` 来控制整个迭代过程。

### 某些对 `vector` 对象的操作会使迭代器失效

3.3.2 节（第 90 页）曾经介绍过，虽然 `vector` 对象可以动态地增长，但是也会有一

些副作用。已知的一个限制是不能在范围 `for` 循环中向 `vector` 对象添加元素。另外一个限制是任何一种可能改变 `vector` 对象容量的操作，比如 `push_back`，都会使该 `vector` 对象的迭代器失效。9.3.6 节（第 315 页）将详细解释迭代器是如何失效的。



**谨记，但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。**

WARNING

### 3.4.1 节练习

**练习 3.21：**请使用迭代器重做 3.3.3 节（第 94 页）的第一个练习。

**练习 3.22：**修改之前那个输出 `text` 第一段的程序，首先把 `text` 的第一段全都改成大写形式，然后再输出它。

**练习 3.23：**编写一段程序，创建一个含有 10 个整数的 `vector` 对象，然后使用迭代器将所有元素的值都变成原来的两倍。输出 `vector` 对象的内容，检验程序是否正确。

### 3.4.2 迭代器运算



迭代器的递增运算令迭代器每次移动一个元素，所有的标准库容器都有支持递增运算的迭代器。类似的，也能用`==`和`!=`对任意标准库类型的两个有效迭代器（参见 3.4 节，第 95 页）进行比较。

111

`string` 和 `vector` 的迭代器提供了更多额外的运算符，一方面可使得迭代器的每次移动跨过多个元素，另外也支持迭代器进行关系运算。所有这些运算被称作**迭代器运算** (iterator arithmetic)，其细节由表 3.7 列出。

表 3.7：`vector` 和 `string` 迭代器支持的运算

<code>iter + n</code>	迭代器加上一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向前移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置
<code>iter - n</code>	迭代器减去一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向后移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置
<code>iter1 += n</code>	迭代器加法的复合赋值语句，将 <code>iter1</code> 加 <code>n</code> 的结果赋给 <code>iter1</code>
<code>iter1 -= n</code>	迭代器减法的复合赋值语句，将 <code>iter1</code> 减 <code>n</code> 的结果赋给 <code>iter1</code>
<code>iter1 - iter2</code>	两个迭代器相减的结果是它们之间的距离，也就是说，将运算符右侧的迭代器向前移动差值个元素后将得到左侧的迭代器。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置
<code>&gt;、&gt;=、&lt;、&lt;=</code>	迭代器的关系运算符，如果某迭代器指向的容器位置在另一个迭代器所指位置之前，则说前者小于后者。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置

### 迭代器的算术运算

可以令迭代器和一个整数值相加（或相减），其返回值是向前（或向后）移动了若干个位置的迭代器。执行这样的操作时，结果迭代器或者指示原 `vector` 对象（或 `string` 对象）内的一个元素，或者指示原 `vector` 对象（或 `string` 对象）尾元素的下一位置。

举个例子，下面的代码得到一个迭代器，它指向某 `vector` 对象中间位置的元素：

```
// 计算得到最接近 vi 中间元素的一个迭代器
auto mid = vi.begin() + vi.size() / 2;
```

如果 `vi` 有 20 个元素，`vi.size() / 2` 得 10，此例中即令 `mid` 等于 `vi.begin() + 10`。已知下标从 0 开始，则迭代器所指的元素是 `vi[10]`，也就是从首元素开始向前相隔 10 个位置的那个元素。

对于 `string` 或 `vector` 的迭代器来说，除了判断是否相等，还能使用关系运算符(`<`、`<=`、`>`、`>=`) 对其进行比较。参与比较的两个迭代器必须合法而且指向的是同一个容器的元素（或者尾元素的下一位置）。例如，假设 `it` 和 `mid` 是同一个 `vector` 对象的两个迭代器，可以用下面的代码来比较它们所指的位置孰前孰后：

```
if (it < mid)
    // 处理 vi 前半部分的元素
```

112

只要两个迭代器指向的是同一个容器中的元素或者尾元素的下一位置，就能将其相减，所得结果是两个迭代器的距离。所谓距离指的是右侧的迭代器向前移动多少位置就能追上左侧的迭代器，其类型是名为 `difference_type` 的带符号整型数。`string` 和 `vector` 都定义了 `difference_type`，因为这个距离可正可负，所以 `difference_type` 是带符号类型的。

## 使用迭代器运算

使用迭代器运算的一个经典算法是二分搜索。二分搜索从有序序列中寻找某个给定的值。二分搜索从序列中间的位置开始搜索，如果中间位置的元素正好就是要找的元素，搜索完成；如果不是，假如该元素小于要找的元素，则在序列的后半部分继续搜索；假如该元素大于要找的元素，则在序列的前半部分继续搜索。在缩小的范围内计算一个新的中间元素并重复之前的过程，直至最终找到目标或者没有元素可供继续搜索。

下面的程序使用迭代器完成了二分搜索：

```
// text 必须是有序的
// beg 和 end 表示我们搜索的范围
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg) / 2; // 初始状态下的中间点
// 当还有元素尚未检查并且我们还没有找到 sought 时执行循环
while (mid != end && *mid != sought) {
    if (sought < *mid) // 我们要找的元素在前半部分吗？
        end = mid; // 如果是，调整搜索范围使得忽略掉后半部分
    else
        beg = mid + 1; // 我们要找的元素在后半部分
    mid = beg + (end - beg) / 2; // 新的中间点
}
```

程序一开始定义了三个迭代器：`beg` 指向搜索范围内的第一个元素、`end` 指向尾元素的下一位置、`mid` 指向中间的那个元素。初始状态下，搜索范围是名为 `text` 的 `vector<string>` 的全部范围。

循环部分先检查搜索范围是否为空，如果 `mid` 和 `end` 的当前值相等，说明已经找遍了所有元素。此时条件不满足，循环终止。当搜索范围不为空时，可知 `mid` 指向了某个元素，检查该元素是否就是我们所要搜索的，如果是，也终止循环。

当进入到循环体内部后，程序通过某种规则移动 `beg` 或者 `end` 来缩小搜索的范围。如果 `mid` 所指的元素比要找的元素 `sought` 大，可推测若 `text` 含有 `sought`，则必出现在 `mid` 所指元素的前面。此时，可以忽略 `mid` 后面的元素不再查找，并把 `mid` 赋给 `end` 即可。另一种情况，如果 `*mid` 比 `sought` 小，则要找的元素必出现在 `mid` 所指元素的后面。此时，通过令 `beg` 指向 `mid` 的下一个位置即可改变搜索范围。因为已经验证过 `mid` 不是我们要找的对象，所以在接下来的搜索中不必考虑它。

循环过程终止时，`mid` 或者等于 `end` 或者指向要找的元素。如果 `mid` 等于 `end`，说明 `text` 中没有我们要找的元素。

### 3.4.2 节练习

113

**练习 3.24：**请使用迭代器重做 3.3.3 节（第 94 页）的最后一个练习。

**练习 3.25：**3.3.3 节（第 93 页）划分分数段的程序是使用下标运算符实现的，请利用迭代器改写该程序并实现完全相同的功能。

**练习 3.26：**在 100 页的二分搜索程序中，为什么用的是 `mid = beg + (end - beg) / 2,` 而非 `mid = (beg + end) / 2;`？

## 3.5 数组

数组是一种类似于标准库类型 `vector`（参见 3.3 节，第 86 页）的数据结构，但是在性能和灵活性的权衡上又与 `vector` 有所不同。与 `vector` 相似的地方是，数组也是存放类型相同的对象的容器，这些对象本身没有名字，需要通过其所在位置访问。与 `vector` 不同的地方是，数组的大小确定不变，不能随意向数组中增加元素。因为数组的大小固定，因此对某些特殊的应用来说程序的运行时性能较好，但是相应地也损失了一些灵活性。



如果不清楚元素的确切个数，请使用 `vector`。

Tip

### 3.5.1 定义和初始化内置数组

数组是一种复合类型（参见 2.3 节，第 45 页）。数组的声明形如 `a[d]`，其中 `a` 是数组的名字，`d` 是数组的维度。维度说明了数组中元素的个数，因此必须大于 0。数组中元素的个数也属于数组类型的一部分，编译的时候维度应该是已知的。也就是说，维度必须是一个常量表达式（参见 2.4.4 节，第 58 页）：

```
unsigned cnt = 42;           // 不是常量表达式
constexpr unsigned sz = 42;   // 常量表达式，关于 constexpr，参见 2.4.4 节（第 59 页）
int arr[10];                 // 含有 10 个整数的数组
int *parr[sz];               // 含有 42 个整型指针的数组
string bad[cnt];             // 错误：cnt 不是常量表达式
string strs[get_size()];     // 当 get_size 是 constexpr 时正确；否则错误
```

默认情况下，数组的元素被默认初始化（参见 2.2.1 节，第 40 页）。



和内置类型的变量一样，如果在函数内部定义了某种内置类型的数组，那么默认初始化会令数组含有未定义的值。

定义数组的时候必须指定数组的类型，不允许用 auto 关键字由初始值的列表推断类型。另外和 vector 一样，数组的元素应为对象，因此不存在引用的数组。

### 114 显式初始化数组元素

可以对数组的元素进行列表初始化（参见 3.3.1 节，第 88 页），此时允许忽略数组的维度。如果在声明时没有指明维度，编译器会根据初始值的数量计算并推测出来；相反，如果指明了维度，那么初始值的总数量不应该超出指定的大小。如果维度比提供的初始值数量大，则用提供的初始值初始化靠前的元素，剩下的元素被初始化成默认值（参见 3.3.1 节，第 88 页）：

```
const unsigned sz = 3;
int ia1[sz] = {0, 1, 2};           // 含有 3 个元素的数组，元素值分别是 0, 1, 2
int a2[] = {0, 1, 2};              // 维度是 3 的数组
int a3[5] = {0, 1, 2};             // 等价于 a3[] = {0, 1, 2, 0, 0}
string a4[3] = {"hi", "bye"};       // 等价于 a4[] = {"hi", "bye", ""}
int a5[2] = {0, 1, 2};             // 错误：初始值过多
```

### 字符数组的特殊性

字符数组有一种额外的初始化形式，我们可以用字符串字面值（参见 2.1.3 节，第 36 页）对此类数组初始化。当使用这种方式时，一定要注意字符串字面值的结尾处还有一个空字符，这个空字符也会像字符串的其他字符一样被拷贝到字符数组中去：

```
char a1[] = {'C', '+', '+'};        // 列表初始化，没有空字符
char a2[] = {'C', '+', '+', '\0'};    // 列表初始化，含有显式的空字符
char a3[] = "C++";                  // 自动添加表示字符串结束的空字符
const char a4[6] = "Daniel";        // 错误：没有空间可存放空字符！
```

a1 的维度是 3，a2 和 a3 的维度都是 4，a4 的定义是错误的。尽管字符串字面值"Daniel"看起来只有 6 个字符，但是数组的大小必须至少是 7，其中 6 个位置存放字面值的内容，另外 1 个存放结尾处的空字符。

### 不允许拷贝和赋值

不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值：

```
int a[] = {0, 1, 2};               // 含有 3 个整数的数组
int a2[] = a;                      // 错误：不允许使用一个数组初始化另一个数组
a2 = a;                            // 错误：不能把一个数组直接赋值给另一个数组
```



一些编译器支持数组的赋值，这就是所谓的编译器扩展（compiler extension）。但一般来说，最好避免使用非标准特性，因为含有非标准特性的程序很可能在其他编译器上无法正常工作。

### 理解复杂的数组声明

和 vector 一样，数组能存放大多数类型的对象。例如，可以定义一个存放指针的数组。又因为数组本身就是对象，所以允许定义数组的指针及数组的引用。在这几种情况中，定义存放指针的数组比较简单和直接，但是定义数组的指针或数组的引用就稍微复杂一点了：

```
int *ptrs[10];                    // ptrs 是含有 10 个整型指针的数组
int &refs[10] = /* ? */;          // 错误：不存在引用的数组
int (*Parray)[10] = &arr;         // Parray 指向一个含有 10 个整数的数组
int (&arrRef)[10] = arr;          // arrRef 引用一个含有 10 个整数的数组
```

默认情况下，类型修饰符从右向左依次绑定。对于 `ptrs` 来说，从右向左（参见 2.3.3 节，第 52 页）理解其含义比较简单：首先知道我们定义的是一个大小为 10 的数组，它的名字是 `ptrs`，然后知道数组中存放的是指向 `int` 的指针。

但是对于 `Parray` 来说，从右向左理解就不太合理了。因为数组的维度是紧跟着被声明的名字的，所以就数组而言，由内向外阅读要比从右向左好多了。由内向外的顺序可帮助我们更好地理解 `Parray` 的含义：首先是圆括号括起来的部分，`*Parray` 意味着 `Parray` 是个指针，接下来观察右边，可知 `Parray` 是个指向大小为 10 的数组的指针，最后观察左边，知道数组中的元素是 `int`。这样最终的含义就明白无误了，`Parray` 是一个指针，它指向一个 `int` 数组，数组中包含 10 个元素。同理，`(&arrRef)` 表示 `arrRef` 是一个引用，它引用的对象是一个大小为 10 的数组，数组中元素的类型是 `int`。

当然，对修饰符的数量并没有特殊限制：

```
int *(&arry)[10] = ptrs; // arry 是数组的引用，该数组含有 10 个指针
```

按照由内向外的顺序阅读上述语句，首先知道 `arry` 是一个引用，然后观察右边知道，`arry` 引用的对象是一个大小为 10 的数组，最后观察左边知道，数组的元素类型是指向 `int` 的指针。这样，`arry` 就是一个含有 10 个 `int` 型指针的数组的引用。



要想理解数组声明的含义，最好的办法是从数组的名字开始按照由内向外的顺序阅读。

### 3.5.1 节练习

**练习 3.27：**假设 `txt_size` 是一个无参数的函数，它的返回值是 `int`。请回答下列哪个定义是非法的？为什么？

```
unsigned buf_size = 1024;
(a) int ia[buf_size];           (b) int ia[4 * 7 - 14];
(c) int ia[txt_size()];         (d) char st[11] = "fundamental";
```

**练习 3.28：**下列数组中元素的值是什么？

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int ia2[10];
}
```

**练习 3.29：**相比于 `vector` 来说，数组有哪些缺点，请列举一些。

### 3.5.2 访问数组元素

116

与标准库类型 `vector` 和 `string` 一样，数组的元素也能使用范围 `for` 语句或下标运算符来访问。数组的索引从 0 开始，以一个包含 10 个元素的数组为例，它的索引从 0 到 9，而非从 1 到 10。

在使用数组下标的时候，通常将其定义为 `size_t` 类型。`size_t` 是一种机器相关的无符号类型，它被设计得足够大以便能表示内存中任意对象的大小。在 `cstdint` 头文件中定义了 `size_t` 类型，这个文件是 C 标准库 `stddef.h` 头文件的 C++ 语言版本。

数组除了大小固定这一特点外，其他用法与 `vector` 基本类似。例如，可以用数组来记录各分数段的成绩个数，从而实现与 3.3.3 节（第 93 页）的程序一样的功能：

```
// 以 10 分为一个分数段统计成绩的数量：0~9, 10~19, ..., 90~99, 100
unsigned scores[11] = {}； // 11 个分数段，全部初始化为 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10]; // 将当前分数段的计数值加 1
}
```

与 93 页的程序相比，上面程序最大的不同是 `scores` 的声明。这里 `scores` 是一个含有 11 个无符号元素的数组。另外一处不太明显的区别是，本例所用的下标运算符是由 C++ 语言直接定义的，这个运算符能用在数组类型的运算对象上。93 页的那个程序所用的下标运算符是库模板 `vector` 定义的，只能用于 `vector` 类型的运算对象。

与 `vector` 和 `string` 一样，当需要遍历数组的所有元素时，最好的办法也是使用范围 `for` 语句。例如，下面的程序输出所有的 `scores`：

```
for (auto i : scores) // 对于 scores 中的每个计数值
    cout << i << " "; // 输出当前的计数值
cout << endl;
```

因为维度是数组类型的一部分，所以系统知道数组 `scores` 中有多少个元素，使用范围 `for` 语句可以减轻人为控制遍历过程的负担。

### 检查下标的值

与 `vector` 和 `string` 一样，数组的下标是否在合理范围之内由程序员负责检查，所谓合理就是说下标应该大于等于 0 而且小于数组的大小。要想防止数组下标越界，除了小心谨慎注意细节以及对代码进行彻底的测试之外，没有其他好办法。对于一个程序来说，即使顺利通过编译并执行，也不能肯定它不包含此类致命的错误。



大多数常见的安全问题都源于缓冲区溢出错误。当数组或其他类似数据结构的下标越界并试图访问非法内存区域时，就会产生此类错误。

117

## 3.5.2 节练习

**练习 3.30：**指出下面代码中的索引错误。

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
    ia[ix] = ix;
```

**练习 3.31：**编写一段程序，定义一个含有 10 个 `int` 的数组，令每个元素的值就是其下标值。

**练习 3.32：**将上一题刚刚创建的数组拷贝给另外一个数组。利用 `vector` 重写程序，实现类似的功能。

**练习 3.33：**对于 104 页的程序来说，如果不初始化 `scores` 将发生什么？

### 3.5.3 指针和数组

在 C++ 语言中，指针和数组有非常紧密的联系。就如即将介绍的，使用数组的时候编译器一般会把它转换成指针。

通常情况下，使用取地址符（参见 2.3.2 节，第 47 页）来获取指向某个对象的指针，取地址符可以用于任何对象。数组的元素也是对象，对数组使用下标运算符得到该数组指定位置的元素。因此像其他对象一样，对数组的元素使用取地址符就能得到指向该元素的指针：

```
string nums[] = {"one", "two", "three"}; // 数组的元素是 string 对象
string *p = &nums[0]; // p 指向 nums 的第一个元素
```

然而，数组还有一个特性：在很多用到数组名字的地方，编译器都会自动地将其替换为一个指向数组首元素的指针：

```
string *p2 = nums; // 等价于 p2 = &nums[0]
```



在大多数表达式中，使用数组类型的对象其实是使用一个指向该数组首元素的指针。

由上可知，在一些情况下数组的操作实际上是指针的操作，这一结论有很多隐含的意思。其中一层意思是当使用数组作为一个 auto（参见 2.5.2 节，第 61 页）变量的初始值时，推断得到的类型是指针而非数组：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia 是一个含有 10 个整数的数组
auto ia2(ia); // ia2 是一个整型指针，指向 ia 的第一个元素
ia2 = 42; // 错误：ia2 是一个指针，不能用 int 值给指针赋值
```

尽管 ia 是由 10 个整数构成的数组，但当使用 ia 作为初始值时，编译器实际执行的初始化过程类似于下面的形式：

```
auto ia2(&ia[0]); // 显然 ia2 的类型是 int*
```

118

必须指出的是，当使用 decltype 关键字（参见 2.5.3 节，第 62 页）时上述转换不会发生，decltype(ia) 返回的类型是由 10 个整数构成的数组：

```
// ia3 是一个含有 10 个整数的数组
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p; // 错误：不能用整型指针给数组赋值
ia3[4] = i; // 正确：把 i 的值赋给 ia3 的一个元素
```

#### 指针也是迭代器

与 2.3.2 节（第 47 页）介绍的内容相比，指向数组元素的指针拥有更多功能。vector 和 string 的迭代器（参见 3.4 节，第 95 页）支持的运算，数组的指针全都支持。例如，允许使用递增运算符将指向数组元素的指针向前移动到下一个位置上：

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p 指向 arr 的第一个元素
++p; // p 指向 arr[1]
```

就像使用迭代器遍历 vector 对象中的元素一样，使用指针也能遍历数组中的元素。当然，这样做的前提是先得获取到指向数组第一个元素的指针和指向数组尾元素的下一个位置的指针。之前已经介绍过，通过数组名字或者数组中首元素的地址都能得到指向首元素的指针；不过获取尾后指针就要用到数组的另外一个特殊性质了。我们可以设法获取数组

尾元素之后的那个并不存在的元素的地址：

```
int *e = &arr[10]; // 指向 arr 尾元素的下一位置的指针
```

这里显然使用下标运算符索引了一个不存在的元素，arr 有 10 个元素，尾元素所在位置的索引是 9，接下来那个不存在的元素唯一的用处就是提供其地址用于初始化 e。就像尾后迭代器（参见 3.4.1 节，第 95 页）一样，尾后指针也不指向具体的元素。因此，不能对尾后指针执行解引用或递增的操作。

利用上面得到的指针能重写之前的循环，令其输出 arr 的全部元素：

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // 输出 arr 的元素
```

### 标准库函数 begin 和 end

尽管能计算得到尾后指针，但这种用法极易出错。为了让指针的使用更简单、更安全，C++11 新标准引入了两个名为 begin 和 end 的函数。这两个函数与容器中的两个同名成员（参见 3.4.1 节，第 95 页）功能类似，不过数组毕竟不是类类型，因此这两个函数不是成员函数。正确的使用形式是将数组作为它们的参数：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia 是一个含有 10 个整数的数组
int *beg = begin(ia);           // 指向 ia 首元素的指针
int *last = end(ia);           // 指向 arr 尾元素的下一位置的指针
```

**119** begin 函数返回指向 ia 首元素的指针，end 函数返回指向 ia 尾元素下一位置的指针，这两个函数定义在 iterator 头文件中。

使用 begin 和 end 可以很容易地写出一个循环并处理数组中的元素。例如，假设 arr 是一个整型数组，下面的程序负责找到 arr 中的第一个负数：

```
// pbeg 指向 arr 的首元素，pend 指向 arr 尾元素的下一位置
int *pbeg = begin(arr), *pend = end(arr);
// 寻找第一个负值元素，如果已经检查完全部元素则结束循环
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

首先定义了两个名为 pbeg 和 pend 的整型指针，其中 pbeg 指向 arr 的第一个元素，pend 指向 arr 尾元素的下一位置。while 语句的条件部分通过比较 pbeg 和 pend 来确保可以安全地对 pbeg 解引用，如果 pbeg 确实指向了一个元素，将其解引用并检查元素值是否为负值。如果是，条件失效、退出循环；如果不是，将指针向前移动一位继续考查下一个元素。



一个指针如果指向了某种内置类型数组的尾元素的“下一位置”，则其具备与 vector 的 end 函数返回的与迭代器类似的功能。特别要注意，尾后指针不能执行解引用和递增操作。

### 指针运算

指向数组元素的指针可以执行表 3.6（第 96 页）和表 3.7（第 99 页）列出的所有迭代器运算。这些运算，包括解引用、递增、比较、与整数相加、两个指针相减等，用在指针和用在迭代器上意义完全一致。

给（从）一个指针加上（减去）某整数值，结果仍是指针。新指针指向的元素与原来

的指针相比前进了（后退了）该整数值个位置：

```
constexpr size_t sz = 5;
int arr[sz] = {1, 2, 3, 4, 5};
int *ip = arr;           // 等价于 int *ip = &arr[0]
int *ip2 = ip + 4;      // ip2 指向 arr 的尾元素 arr[4]
```

ip 加上 4 所得的结果仍是一个指针，该指针所指的元素与 ip 原来所指的元素相比前进了 4 个位置。

给指针加上一个整数，得到的新指针仍需指向同一数组的其他元素，或者指向同一数组的尾元素的下一位置：

```
// 正确：arr 转换成指向它首元素的指针；p 指向 arr 尾元素的下一位置
int *p = arr + sz;        // 使用警告：不要解引用！
int *p2 = arr + 10;       // 错误：arr 只有 5 个元素，p2 的值未定义
```

当给 arr 加上 sz 时，编译器自动地将 arr 转换成指向数组 arr 中首元素的指针。执行加法后，指针从首元素开始向前移动了 sz（这里是 5）个位置，指向新位置的元素。也就是说，它指向了数组 arr 尾元素的下一位置。如果计算所得的指针超出了上述范围就将产生错误，而且这种错误编译器一般发现不了。◀ [120]

和迭代器一样，两个指针相减的结果是它们之间的距离。参与运算的两个指针必须指向同一个数组当中的元素：

```
auto n = end(arr) - begin(arr); // n 的值是 5，也就是 arr 中元素的数量
```

两个指针相减的结果的类型是一种名为 **ptrdiff\_t** 的标准库类型，和 **size\_t** 一样，**ptrdiff\_t** 也是一种定义在 **cstddef** 头文件中的机器相关的类型。因为差值可能为负值，所以 **ptrdiff\_t** 是一种带符号类型。

只要两个指针指向同一个数组的元素，或者指向该数组的尾元素的下一位置，就能利用关系运算符对其进行比较。例如，可以按照如下的方式遍历数组中的元素：

```
int *b = arr, *e = arr + sz;
while (b < e) {
    // 使用 *b
    ++b;
}
```

如果两个指针分别指向不相关的对象，则不能比较它们：

```
int i = 0, sz = 42;
int *p = &i, *e = &sz;
// 未定义的：p 和 e 无关，因此比较毫无意义！
while (p < e)
```

尽管作用可能不是特别明显，但必须说明的是，上述指针运算同样适用于空指针（参见 2.3.2 节，第 48 页）和所指对象并非数组的指针。在后一种情况下，两个指针必须指向同一个对象或该对象的下一位置。如果 p 是空指针，允许给 p 加上或减去一个值为 0 的整型常量表达式（参见 2.4.4 节，第 58 页）。两个空指针也允许彼此相减，结果当然是 0。

## 解引用和指针运算的交互

指针加上一个整数所得的结果还是一个指针。假设结果指针指向了一个元素，则允许解引用该结果指针：

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

表达式`* (ia+4)`计算 ia 前进 4 个元素后的新地址，解引用该结果指针的效果等价于表达式`ia[4]`。

回忆一下在 3.4.1 节（第 98 页）中介绍过如果表达式含有解引用运算符和点运算符，最好在必要的地方加上圆括号。类似的，此例中指针加法的圆括号也不可缺少。如果写成下面的形式：

```
last = *ia + 4; // 正确：last = 4 等价于 ia[0] + 4
```

含义就与之前完全不同了，此时先解引用 ia，然后给解引用的结果再加上 4。4.1.2 节（第 121 页）将对这一问题做进一步分析。



## 下标和指针

121

如前所述，在很多情况下使用数组的名字其实用的是一个指向数组首元素的指针。一个典型的例子是当对数组使用下标运算符时，编译器会自动执行上述转换操作。给定

```
int ia[] = {0,2,4,6,8}; // 含有 5 个整数的数组
```

此时，`ia[0]`是一个使用了数组名字的表达式，对数组执行下标运算其实是对指向数组元素的指针执行下标运算：

```
int i = ia[2];           // ia 转换成指向数组首元素的指针
                         // ia[2] 得到 (ia + 2) 所指的元素
int *p = ia;             // p 指向 ia 的首元素
i = *(p + 2);           // 等价于 i = ia[2]
```

只要指针指向的是数组中的元素（或者数组中尾元素的下一位置），都可以执行下标运算：

```
int *p = &ia[2];        // p 指向索引为 2 的元素
int j = p[1];           // p[1] 等价于 *(p + 1)，就是 ia[3] 表示的那个元素
int k = p[-2];          // p[-2] 是 ia[0] 表示的那个元素
```

虽然标准库类型 `string` 和 `vector` 也能执行下标运算，但是数组与它们相比还是有所不同。标准库类型限定使用的下标必须是无符号类型，而内置的下标运算无此要求，上面的最后一个例子很好地说明了这一点。内置的下标运算符可以处理负值，当然，结果地址必须指向原来的指针所指同一数组中的元素（或是同一数组尾元素的下一位置）。



内置的下标运算符所用的索引值不是无符号类型，这一点与 `vector` 和 `string` 不一样。

### 3.5.3 节练习

**练习 3.34：**假定 `p1` 和 `p2` 指向同一个数组中的元素，则下面程序的功能是什么？什么情况下该程序是非法的？

```
p1 += p2 - p1;
```

**练习 3.35：**编写一段程序，利用指针将数组中的元素置为 0。

**练习 3.36：**编写一段程序，比较两个数组是否相等。再写一段程序，比较两个 `vector` 对象是否相等。

### 3.5.4 C 风格字符串



尽管 C++ 支持 C 风格字符串，但在 C++ 程序中最好还是不要使用它们。这是因为 C 风格字符串不仅使用起来不太方便，而且极易引发程序漏洞，是诸多安全问题的根本原因。

&lt;122

字符串字面值是一种通用结构的实例，这种结构即是 C++ 由 C 继承而来的 C 风格字符串 (C-style character string)。C 风格字符串不是一种类型，而是为了表达和使用字符串而形成的一种约定俗成的写法。按此习惯书写的字符串存放在字符数组中并以空字符结束 (null terminated)。以空字符结束的意思是在字符串最后一个字符后面跟着一个空字符 ('\0')。一般利用指针来操作这些字符串。

#### C 标准库 String 函数

表 3.8 列举了 C 语言标准库提供的一组函数，这些函数可用于操作 C 风格字符串，它们定义在 `cstring` 头文件中，`cstring` 是 C 语言头文件 `string.h` 的 C++ 版本。

表 3.8: C 风格字符串的函数

<code>strlen(p)</code>	返回 p 的长度，空字符不计算在内
<code>strcmp(p1, p2)</code>	比较 p1 和 p2 的相等性。如果 $p1==p2$ ，返回 0；如果 $p1>p2$ ，返回一个正值；如果 $p1<p2$ ，返回一个负值
<code>strcat(p1, p2)</code>	将 p2 附加到 p1 之后，返回 p1
<code>strcpy(p1, p2)</code>	将 p2 拷贝给 p1，返回 p1



表 3.8 所列的函数不负责验证其字符串参数。

WARNING

传入此类函数的指针必须指向以空字符作为结束的数组：

```
char ca[] = {'C', '+', '+'};      // 不以空字符结束
cout << strlen(ca) << endl;      // 严重错误：ca 没有以空字符结束
```

此例中，ca 虽然也是一个字符数组但它不是以空字符作为结束的，因此上述程序将产生未定义的结果。`strlen` 函数将有可能沿着 ca 在内存中的位置不断向前寻找，直到遇到空字符才停下来。

#### 比较字符串

比较两个 C 风格字符串的方法和之前学习过的比较标准库 `string` 对象的方法大相径庭。比较标准库 `string` 对象的时候，用的是普通的关系运算符和相等性运算符：

```
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2) // false: s2 小于 s1
```

如果把这些运算符用在两个 C 风格字符串上，实际比较的将是指针而非字符串本身：

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2) // 未定义的：试图比较两个无关地址
```

&lt;123

谨记之前介绍过的，当使用数组的时候其实真正用的是指向数组首元素的指针（参见 3.5.3 节，第 105 页）。因此，上面的 if 条件实际上比较的是两个 `const char*` 的值。这两个

指针指向的并非同一对象，所以将得到未定义的结果。

要想比较两个 C 风格字符串需要调用 `strcmp` 函数，此时比较的就不再是指针了。如果两个字符串相等，`strcmp` 返回 0；如果前面的字符串较大，返回正值；如果后面的字符串较大，返回负值：

```
if (strcmp(ca1, ca2) < 0) // 和两个 string 对象的比较 s1 < s2 效果一样
```

### 目标字符串的大小由调用者指定

连接或拷贝 C 风格字符串也与标准库 `string` 对象的同类操作差别很大。例如，要想把刚刚定义的那两个 `string` 对象 `s1` 和 `s2` 连接起来，可以直接写成下面的形式：

```
// 将 largeStr 初始化成 s1、一个空格和 s2 的连接
string largeStr = s1 + " " + s2;
```

同样的操作如果放到 `ca1` 和 `ca2` 这两个数组身上就会产生错误了。表达式 `ca1 + ca2` 试图将两个指针相加，显然这样的操作没什么意义，也肯定非法的。

正确的方法是使用 `strcat` 函数和 `strcpy` 函数。不过要想使用这两个函数，还必须提供一个用于存放结果字符串的数组，该数组必须足够大以便容纳下结果字符串及末尾的空字符。下面的代码虽然很常见，但是充满了安全风险，极易引发严重错误：

```
// 如果我们计算错了 largeStr 的大小将引发严重错误
strcpy(largeStr, ca1);           // 把 ca1 拷贝给 largeStr
strcat(largeStr, " ");           // 在 largeStr 的末尾加上一个空格
strcat(largeStr, ca2);           // 把 ca2 连接到 largeStr 后面
```

一个潜在的问题是，我们在估算 `largeStr` 所需的空间时不容易估准，而且 `largeStr` 所存的内容一旦改变，就必须重新检查其空间是否足够。不幸的是，这样的代码到处都是，程序员根本没法照顾周全。这类代码充满了风险而且经常导致严重的安全泄漏。



对大多数应用来说，使用标准库 `string` 要比使用 C 风格字符串更安全、更高效。

124 &gt;

### 3.5.4 节练习

**练习 3.37：**下面的程序是何含义，程序的输出结果是什么？

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

**练习 3.38：**在本节中我们提到，将两个指针相加不但是非法的，而且也没什么意义。请问为什么两个指针相加没什么意义？

**练习 3.39：**编写一段程序，比较两个 `string` 对象。再编写一段程序，比较两个 C 风格字符串的内容。

**练习 3.40：**编写一段程序，定义两个字符数组并用字符串字面值初始化它们；接着再定义一个字符数组存放前两个数组连接后的结果。使用 `strcpy` 和 `strcat` 把前两个数组的内容拷贝到第三个数组中。

### 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。

&lt; 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。



**WARNING** 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}};
```

&lt;127&gt;

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 rowCount = 3, colCount = 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

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：

```
// 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;
}
```

C++  
11

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个位置。

**[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个位置。

**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。

`ptrdiff_t` 是 `cstdint` 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 `cstdint` 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：

```
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

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 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 &lt; expr</code>
左	<=	小于等于	<code>expr &lt;= expr</code>
左	>	大于	<code>expr &gt; expr</code>
左	>=	大于等于	<code>expr &gt;= expr</code>
左	==	相等	<code>expr == expr</code>
左	!=	不相等	<code>expr != expr</code>
左	&&	逻辑与	<code>expr &amp;&amp; expr</code>
左		逻辑或	<code>expr    expr</code>

### 逻辑与和逻辑或运算符

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

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

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

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

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

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

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

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

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

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

## 逻辑非运算符

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

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

子表达式

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

当`empty`函数返回假时结果为真。

## 关系运算符

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

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

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

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

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

### 相等性测试与布尔字面值

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

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

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

有时会试图将上面的真值测试写成如下形式：

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

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

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

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

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



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

## 4.3 节练习

**练习 4.8：**说明在逻辑与、逻辑或及相等性运算符中运算对象求值的顺序。

**练习 4.9：**解释在下面的 if 语句中条件部分的判断过程。

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

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

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

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

## 4.4 赋值运算符

赋值运算符的左侧运算对象必须是一个可修改的左值。如果给定

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

则下面的赋值语句都是非法的：

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

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

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

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

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

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

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

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

### 赋值运算满足右结合律

赋值运算符满足右结合律，这一点与其他二元运算符不太一样：

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

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

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

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

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

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

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

### 赋值运算优先级较低

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

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

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

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

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

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



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

### 切勿混淆相等运算符和赋值运算符

C++语言允许用赋值运算作为条件，但是这一特性可能带来意想不到的后果：

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

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

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

程序的这种缺陷显然很难被发现，好在一部分编译器会对类似的代码给出警告信息。

**147** **复合赋值运算符**

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

```

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

```

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

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

任意一种复合运算符都完全等价于

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

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

## 4.4 节练习

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

```

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

```

**练习 4.14：**执行下述 if 语句后将发生什么情况？

```

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

```

**练习 4.15：**下面的赋值是非法的，为什么？应该如何修改？

```

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

```

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

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

## 4.5 递增和递减运算符

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

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

```

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

```

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

### 建议：除非必须，否则不用递增递减运算符的后置版本

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

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

### 在一条语句中混用解引用和递增运算符

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

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

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

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

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

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

### 建议：简洁可以成为一种美德

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

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

要比书写下面的等价语句更简洁、也更少出错

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

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

## 运算对象可按任意顺序求值

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

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

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

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

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

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

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

也可能采取别的什么方式处理它。

## 4.5 节练习

&lt; 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 &amp;&amp; *ptr++</code>  | (b) <code>ival++ &amp;&amp; ival</code> |
| (c) <code>vec[ival++] &lt;= 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-&gt;empty();</code> | (e) <code>++*iter;</code>   | (f) <code>iter++-&gt;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 &gt;

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>&lt;&lt;</code>	左移	<code>expr1 &lt;&lt; expr2</code>
<code>&gt;&gt;</code>	右移	<code>expr1 &gt;&gt; expr2</code>
<code>&amp;</code>	位与	<code>expr &amp; expr</code>
<code>^</code>	位异或	<code>expr ^ expr</code>
<code> </code>	位或	<code>expr   expr</code>

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



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

## 移位运算符

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

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

// 0233 是八进制的字面值（参见 2.1.3 节，第 35 页）

`unsigned char bits = 0233;`

1	0	0	1	1	0	1	1
---	---	---	---	---	---	---	---

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

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

`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	1	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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

154 &gt;

## 位求反运算符

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

unsigned char bits = 0227;	1 0 0 1 0 1 1 1
~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 个高阶位都是 0   0 0 1 0 0 1 0 1
b1   b2	24 个高阶位都是 0   1 1 1 0 1 1 1 1
b1 ^ b2	24 个高阶位都是 0   1 1 0 0 1 0 1 0

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



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

## 使用位运算符

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

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

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

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

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

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

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

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

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

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

`|=` 运算符的工作原理和 `+=` 非常相似，它等价于

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

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

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

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

最后，我们试图检查学生 27 测验的情况到底怎么样：

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

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



## 移位运算符（又叫 IO 运算符）满足左结合律

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

因为移位运算符满足左结合律，所以表达式

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

的执行过程实际上等同于

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

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

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

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

最后一个 `cout` 的含义其实是

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

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

## 4.8 节练习

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

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

**练习 4.27:** 下列表达式的结果是什么？

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

## 4.9 sizeof 运算符

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

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

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

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

&lt; 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-&gt;mem[i]</code> |
| (c) <code>sizeof a &lt; 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

&lt; 159

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



举个例子，考虑下面这条表达式，它的目的是将 `ival` 初始化为 6：

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

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

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

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

### 何时发生隐式类型转换

在下面这些情况下，编译器会自动地转换运算对象的类型：

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



## 4.11.1 算术转换

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

### 160 整型提升

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

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

### 无符号类型的运算对象

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

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

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

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

### 理解算术转换

要想理解算术转换，办法之一就是研究大量的例子：

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

```

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

```

&lt;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

&lt;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

&lt; 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